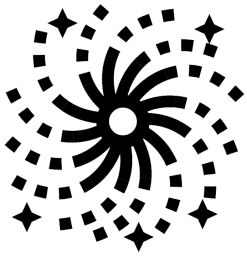
Reference Document

Perri van den Berg, Amber van den Broek, Dirck van den Ende
Wormholes - Team 62



0.1. Program Template

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

typedef long long ll;
typedef long double ld;
typedef vector<ll> vi;
typedef vector<vi> vvi;
typedef pair<ll, ll> ii;
typedef vector<ii> vii;

#define x first
#define y second
#define pb push_back
#define sz(v) ((int)(v).size())
#define all(v) (v).begin(), (v).end()
#define rep(i, a, b) for (auto i = (a); i < (b); i++)
#define REP(i, b) rep(i, 0LL, b)
#ifdef DEBUG
    #define DBG(x) cerr << __LINE__ << ": " << #x \
    << " = " << (x) << endl
#else
    #define DBG(x) ((void)(x))
#endif

template <class T> ostream &operator<<(ostream &os,
const vector<T> &v) {
    os << '[';
    for (const T &i : v) os << i << ", ";
    return os << ']' << endl;
}
template <class T, class S> ostream &operator<<(ostream
&os, const pair<T, S> &v) {
    return os << '{' << v.x << ", " << v.y << '}';
}

void run() {
}

int main() {
    // No automatic flushing before cin
    cin.tie(NULL);
    // Mixed C/C++ IO
    ios_base::sync_with_stdio(false);
    // Enough for double precision
    cout.precision(20);
    run();
    return 0;
}
```

Compilation

```
alias gg="g++ -std=gnu++20 -x c++ -Wall -O2 -static
-pipe -lm"
```

(NWERC 2024)

0.2. Strategy

Solving strategy

- Go over all problems that haven't been solved.
- Small conditions → Brute force/less efficient algorithm.
- Do we need to take a break? Drink enough water!
- Should a solution be possible for *all cases*? Think of special cases.
  - Is it $<$ or $\leq$?
  - Why would a special condition be given?
- Sketch on paper.
- Did we read the problem correctly? Look at images on problem sheet.
- Use Python when working with very large numbers.

Problem solving strategies

- **Greedy**: Pick best looking on each step.
- **Dynamic Programming**: Use table to remember results.
- **Backtracking**: Check intermediate results.
- **Divide and conquer**: Split up into smaller problems.
- **BFS/DFS/floodfill**.
- **Prefix/Suffix tree**: For string problems.
- **Create/Extend a graph**: Minimizing problems → use Dijkstra/Kruskal.
- **Hash table**: Up to around $10^8 \approx 2^{26.5}$ entries.
  - Don't make too many hash tables!
- **Matrix form**: Gain more insight in how parts interact.
- **Randomization**: Chance of failure is small enough.
- **Bitset**: Encode binary data. Useful for bitshifting.
- Use **polynomials** to model the problem.
- Consider the problem in **reverse**.

Wrong answer

- Check boundary values: Minimum, maximum, powers of 2.
- Division by 0 → NaN.
- Off-by-one error?
- Is it $<$ or $\leq$?
- Input all 0.
- Integer overflow or NaN.
- Can precision be influenced by transformation function? i.e. log, exp, etc.
- Does output have to be given modulo M?

Timelimit

- Make sum/minimum calculation efficient.
- `DBG` statements can cause slowdowns.
- Use fast IO and flush output manually.
- Use hash tables or rolling hashes.
- Don't create too many sets or queues.

Run error

- Forgot to compile?
- Check typos.
- Division by 0?
- Accessing index -1 of vector?

Memory limit

- Replace `long long` with `int`.
- Reduce DP table dimensions.

0.3. Useful built-ins

```cpp
// Standard input/output: reads until whitespace!
cin >> var; cout << expr << endl;
// Read up to next line ending
getline(cin, str);
// Enable/disable fixed point display
cout << fixed << val; cout << defaultfloat << val;
// Throws exception when condition is not met
assert(expr);

// Sort using iterators from low to high
bool lt(T &a, T &b);
sort(begin, end, lt);
// Reverse using iterators
reverse(begin, end);
// Iterator operations
prev(it, n = 1), next(it, n = 1);
// Swaps values
swap(a, b);
// Unordered uses hashtable, otherwise binary tree
(unordered_)(multi)[set|map] obj;
obj.insert(item), obj.erase(item),
obj.lower_bound(item), obj.equal_range(item);
// Amortized constant time insert
vector<T> v;
v.pb(), v.pop_back();
// Reference to a string, no copies performed
string_view sv;
// constants
LLONG_MAX, INT_MAX;

// Lambda expressions + examples
[captures](params) specs {body}
[](int x) -> int { return -x; }
[&]() -> string { return s + "!"; }

// Complex numbers
complex<ld> num;
```

Python

```python
import io, os
# Fast input
inp = io.BytesIO(os.read(0, os.fstat(0).st_size)) \
.readline
s = inp().decode()
# Fast output
sys.stdout.write(s)
# Normal input and output
input(), print()

# Sort a list, with key returning a number
list.sort(reverse, key)
# Unordered set and map
set([a, b, c]), {"a": 5, "g": 34}
set.add(), set.remove()
lambda a : a + 10
# Floor division
a = b // c
# Parallel assignment (example: swap)
a, b = b, a
```

1. Math

1.1. Factorization, multiples and primes

Greatest common denominator

This algorithm uses the formula $\gcd(a, b) = \gcd(b, r)$ if $a = bq + r$, where $q \neq 0$, $r \geq 0$ are integers.

```cpp
ll gcd(ll a, ll b) {
    while (b != 0)
        a %= b, swap(a, b);
    return a;
}
```

The gcd also has the following properties:

$$\gcd(a,b,c) = \gcd(a,\gcd(b,c)),$$
$$\gcd(a,b) = \gcd(b,a),$$
$$\gcd(a,0) = a.$$

Extended Euclidean algorithm

We can also determine the (lowest) numbers that form the gcd with the following algorithm:

```
// Euclidean algorithm to find numbers t, s such
// that as + bt = gcd(a, b). Here s and t are the
// smallest such numbers possible. The function
// returns the gcd(a, b). The s and t may have the
// wrong sign!
ll euclidGcd(ll a, ll b, ll &s, ll &t) {
    ll r = b, rr = a;
    s = 0, t = 1;
    ll ss = 1, tt = 0;
    while (rr != 0) {
        ll q = r / rr;
        r -= q * rr; swap(r, rr);
        s -= q * ss; swap(s, ss);
        t -= q * tt; swap(t, tt);
    }
    return r;
}
```

Least common multiple

$$\operatorname{lcm}(a,b) = \frac{|ab|}{\gcd(a,b)}$$

Prime factorization $\mathcal{O}(\log n)$

```
vi fact(ll n) {
    vi r;
    for (ll k = 2; n > 1 && k * k <= n; k++)
        while (n % k == 0)
            n /= k, r.pb(k);
    if (n > 1)
        r.pb(n);
    return r;
}
```

Miller-Rabin primality test $\mathcal{O}(\log n)$

This primality test works for $n < 2^{64}$. For the `modPower` function see the "Modular arithmetic" section.

```
// Check if the number n is prime
bool isPrime(ll n) {
    // Miller-Rabin only works for odd numbers
    if (n < 2 || n % 2 == 0)
        return n == 2;
    // Determine s > 0 and d odd > 0 such that
    // n - 1 = 2^s * d
    ll d = n - 1, s = 0;
    while (d % 2 == 0)
        d /= 2, s++;
    // This list needs to be extended for numbers >2^64
    for (ll a : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
    31, 37}) {
        if (a >= n - 1 || a > 2 * log(n) * log(n))
            return true;
        ll x = modPower(a, d, n);
        REP(i, s) {
            // For large n, use modMul
            ll y = (x * x) % n;
            if (y == 1 && x != 1 && x != n - 1)
                return false;
            x = y;
        }
        if (x != 1)
            return false;
    }
    return true;
}
```

Wilson's theorem

A natural number $p > 1$ is a prime if and only if

$$(p-1)! \equiv -1 \mod p.$$

Euler totient function

The Euler totient function $\varphi(n)$ returns the number of positive integers smaller than n that are coprime with n. An algorithm to determine $\varphi(n)$ can be optimized by using the formula

$$\varphi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right).$$

```
ll phi(ll n) {
    ll res = n;
    for (ll p = 2; p * p <= n; p++) {
        if (n % p == 0) {
            while (n % p == 0)
                n /= p;
            res -= res / p;
        }
    }
    // If n is prime itself
    if (n > 1)
        res -= res / n;
    return res;
}
```

Sieve of Eratosthenes $\mathcal{O}(n \log(\log n))$

This algorithm finds all primes lower than a number n, by keeping track of a table and marking multiples of primes as non-prime.

```
// Returns the list of all primes that are lower than n
// using a sieve of Eratosthenes.
vi primeSieve(ll n) {
    vi primes(n + 1, 1);
    for (ll p = 2; p * p <= n; p++)
        if (primes[p])
            for (ll i = p * p; i <= n; i += p)
                primes[i] = 0;
    vi primeList;
    rep(i, 2, n + 1)
        if (primes[i])
            primeList.pb(i);
    return primeList;
}
```

Number of primes below n

The number of primes $\pi(n)$ smaller than or equal to n has the property

$$\lim_{n\to\infty} \frac{\pi(n)\log n}{n} = 1.$$

1.2. Modular arithmetic

Basic properties

$$\left.\begin{array}{r} a_1 \equiv b_1 \mod n \\ a_2 \equiv b_2 \mod n \end{array}\right\} \Rightarrow a_1 a_2 \equiv b_1 b_2 \mod n$$

$$\left.\begin{array}{r} a_1 \equiv b_1 \mod n \\ a_2 \equiv b_2 \mod n \end{array}\right\} \Rightarrow a_1 + a_2 \equiv b_1 + b_2 \mod n$$

$$\left.\begin{array}{r} \gcd(a,n) = 1 \\ c \equiv d \mod \varphi(n) \end{array}\right\} \Rightarrow a^c \equiv a^d \mod n$$

Modulo multiplication $\mathcal{O}(\log y)$

```
// Determines x * y % n without risking overflow issues
// x, y and n need to be in the range [0, n)
ll modMul(ll x, ll y, ll n) {
    ll step = (1ll << 61) / n;
    ll res = 0;
    while (y > 0) {
        res = (res + x * (y % step)) % n;
        x = (x * step) % n;
        y /= step;
    }
    return res;
}
```

Modulo power $\mathcal{O}(\log p)$

```
// Used to determine the power of a number x^p,
// modulo another number n, without doing all of
// the powers
ll modPower(ll x, ll p, ll n) {
    ll res = 1;
    x %= n;
    while (p > 0) {
        // For large modulo (>2^31) use modMul instead
        if (p % 2 == 1)
            res = res * x % n;
        x = x * x % n;
        p /= 2;
    }
    return res;
}
```

Equal modulo

```
// Since C++ can return negative numbers, for checking
// if two numbers are congruent mod n, we need this
bool modEq(ll a, ll b, ll n) {
    ll aa = a % n, bb = b % n;
    return aa == bb || aa == bb + n || aa + n == b;
}
```

Chinese remainder theorem

The Chinese remainder theorem states that for any $n, m > 1$ coprime and a number $x < nm$ there exist unique $a < n$ and $b < m$ such that

$$x \equiv a \mod n,$$
$$x \equiv b \mod m.$$

The solution x can be determined using the extended Euclidean algorithm.

```
// Returns the number generated by the Chinese
// remainder theorem, using the extended euclidean
// algorithm. Inputs are numbers a and b with modulo
// constants n and m. Returns x
ll crt(ll a, ll b, ll n, ll m) {
    ll s, t;
    euclidGcd(n, m, s, t);
    // Might have to change this if the numbers get too
    // large
    ll res = (a * m * t + b * n * s) % (n * m);
    if (res < 0)
        res += n * m;
    return res;
}
```

Inverse elements

The inverse of an integer a modulo n can be found using the extended Euclidean algorithm. An inverse exists if and only if $\gcd(a, n) = 1$. Likewise the system $a \cdot x \equiv b \mod n$ can be solved if $\gcd(a, n)$ divides b.

```
// Find x such that a * x = b (mod n)
// Solution exists iff gcd(a, n) divides b
ll modSys(ll a, ll b, ll n) {
    ll s, t;
    ll g = euclidGcd(a, n, s, t);
    assert(b % g == 0);
    ll res = s * (b / g);
    return res < 0 ? res + n : res;
}
// Determines the inverse of a mod n
ll modInv(ll a, ll n) {
    return modSys(a, 1, n);
}
```

1.3. Geometry

Area of a triangle

$$A = \sqrt{x(x-a)(x-b)(x-c)}$$

where a, b and c are the side lengths and $x = (a+b+c)/2$.

Area of a quadrilateral

$$A = \sqrt{(a-x)(b-x)(c-x)(d-x) - \cos^2\left(\frac{\theta_1+\theta_2}{2}\right)}$$

where θ_1 and θ_2 are opposing corners and $x = (a+b+c+d)/2$.

Area of a simple polygon

Given a simple polygon (no intersecting edges) with n vertices $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ where $(x_0, y_0) = (x_n, y_n)$. The area of the polygon is

$$A = \frac{1}{2}\left|\sum_{k=0}^{n-1}(x_k y_{k+1} - x_{k+1} y_k)\right|.$$

Area of a regular polyhedron

Given the number of side edges n, and either the distance from the center to a vertex r or the distance from the center to an edge a, the area of a regular polyhedron is

$$A = nr^2 \sin\frac{\pi}{n}\cos\frac{\pi}{n} = a^2 n \tan\frac{\pi}{n}.$$

Maximal quadrilateral problem

The maximum area of a quadrilateral is given by $\sqrt{(a-x)(b-x)(c-x)(d-x)}$ where $x = (a+b+c+d)/2$. This way the opposing corners add up to π and all points lie on a circle.

Circles and spheres

- Circumference of a circle: $2\pi r$.
- Area of a circle: πr^2.
- Area of a 3-D sphere: $4\pi r^2$.

Circle-line intersection

The intersection point(s) of a circle with center $(0, 0)$ and radius r, and a line given by points (x_1, y_1) and (x_2, y_3) are given by $(X_\pm, Y_\pm)$:

$$d_x = x_2 - x_1,$$
$$d_y = y_2 - y_1,$$
$$d_r = \sqrt{d_x^2 + d_y^2},$$
$$D = x_1 y_2 - x_2 y_1,$$
$$X_\pm = \frac{D d_y \pm \operatorname{sgn}(d_y) d_x \sqrt{r^2 d_r^2 - D^2}}{d_r^2},$$

$$Y_{\pm} = \frac{-Dd_x \pm |d_y|\sqrt{r^2 d_r^2 - D^2}}{d_r^2}.$$

where $\text{sgn}(x)$ is -1 if $x < 0$ and 1 otherwise.

Circle-circle intersection

The intersection point(s) of two circles with centers (x_1, y_1) and (x_2, y_2) and radii r_1 and r_2 are given by $(X_{\pm}, Y_{\pm})$:

$$\begin{aligned} d &= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}, \\ l &= \frac{r_1^2 - r_2^2 + d^2}{2d}, \\ h &= \sqrt{r_1^2 - l^2}, \\ X_{\pm} &= \frac{l}{d}(x_2 - x_1) \pm \frac{h}{d}(y_2 - y_1) + x_1, \\ Y_{\pm} &= \frac{l}{d}(y_2 - y_1) \mp \frac{h}{d}(x_2 - x_1) + y_1. \end{aligned}$$

Volume of an n-ball

An n-ball with radius r has the following volume:

$$V_n(r) = \begin{cases} 1, & \text{if } n = 0, \\ 2r, & \text{if } n = 1, \\ \frac{2\pi}{n} r^2 V_{n-2}(r), & \text{if } n > 1. \end{cases}$$

Angle between two vectors

Given two vectors $a, b \in \mathbb{R}^n$, the angle between them is given by

$$\theta = \cos^{-1} \frac{a \cdot b}{||a|| \cdot ||b||}.$$

Distance from a point to a line

Given line $\ell(t) = a + bt$ with $a, b \in \mathbb{R}^n$ and point $p \in \mathbb{R}^n$, the distance is

$$d = \frac{(p - a) \times (p - b)}{||b - a||}.$$

In 2-D the line is given by $ax + by + c = 0$, then the distance to point $p = (x_p, y_p)$ is given by

$$d = \frac{|ax_p + by_p + c|}{\sqrt{a^2 + b^2}}.$$

Note that we can remove the absolute value to check if two points are on the same side of a line!

Intersection of lines

Let the lines ℓ_1 and ℓ_2 be given by the points (x_1, y_1) and (x_2, y_2) respectively (x_3, y_3) and (x_4, y_4). Then the $c = x, y$ coordinate of the intersection point P is given by

$$P_c = \frac{(x_1 y_2 - x_2 y_1)(c_3 - c_4) - (c_1 - c_2)(x_3 y_4 - x_4 y_3)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}.$$

Angle sum property

The sum of the angles on of an n-lateral is $\pi(n - 2)$. The sum of the exterior angles of a polygon is 2π.

Maximum points on a circle

Given the angle α, the maximum number of points on a circle is $M = \lfloor \frac{2\pi}{\alpha} \rfloor$. Given a minimum distance d between the points:

$$\alpha = 2 \sin^{-1} \frac{d}{2r}.$$

3-D shapes

Shape	Area	Volume
Cylinder	$2\pi rh + 2\pi r^2$	$\pi r^2 h$
Cone	$\pi r\sqrt{r^2 + h^2} + \pi r^2$	$\frac{1}{3}\pi r^2 h$
Hemisphere (half sphere)	$3\pi r^2$	$\frac{2}{3}\pi r^3$
Pyramid	$lw + l\sqrt{\frac{w^2}{4} + h^2} + \sqrt{\frac{l^2}{4} + h^2}$	$\frac{1}{3}lwh$
Torus	$4\pi^2 Rr$	$2\pi^2 Rr^2$
Tetrahedron	$\sqrt{3}a^2$	$\frac{\sqrt{2}}{12}a^3$
Octahedron	$2\sqrt{3}a^2$	$\frac{\sqrt{2}}{3}a^3$
Dodecahedron	$3\sqrt{25 + 10\sqrt{5}}a^2$	$\frac{1}{4}(15 + 7\sqrt{5})a^3$

Distance on a sphere

Given latitudes $\varphi_1, \varphi_2 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ and longitudes $\lambda_1, \lambda_2 \in [-\pi, \pi]$ the distance "as the crow flies" on a sphere with radius $R > 0$ between (φ_1, λ_1) and (φ_2, λ_2) is given by d:

$$\begin{aligned} a &= \sin^2 \frac{\varphi_2 - \varphi_1}{2} + \cos\varphi_1 \cos\varphi_2 \sin^2 \frac{\lambda_2 - \lambda_1}{2}, \\ c &= 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{1 - a}), \\ d &= Rc. \end{aligned}$$

The function atan2 is available in `C++` STL.

Weighted average of triangle vertices

Given a (proper) triangle in 2-D space with vertices $v_1, v_2, v_3 \in \mathbb{R}^2$ and a point $p \in \mathbb{R}^2$ inside the triangle. The following calculates weights $w_1, w_2, w_3 \in [0, 1]$ such that $p = w_1 v_1 + w_2 v_2 + w_3 v_3$ with $w_1 + w_2 + w_3 = 1$.

Note that this implementation uses the "Distance from a point to a line" formula.

```
typedef pair<ld, ld> dd;

// Find weights of vertices of a triangle to average
// point p inside the triangle, returns a boolean
// indicating if the point is in the triangle. If the
// point is not inside the triangle, the weights are
// not calculated correctly
bool triangleWeights(dd p, dd v1, dd v2, dd v3, ld &w1,
ld &w2, ld &w3) {
    // This lambda will return the weight of v1
    auto f = [](dd p, dd v1, dd v2, dd v3) -> ld {
        dd v = {v2.x - v3.x, v2.y - v3.y};
        // Calculate distance to line v2,v3 from both
        // v1 and p
        ld a = v.y, b = -v.x;
        ld c = v.x * v2.y - v.y * v2.x;
        ld dv = a * v1.x + b * v1.y + c;
        ld dp = a * p.x + b * p.y + c;
        if (dv == 0.0)
            return 0.0;
        return dp / dv;
    };
    // Calculate weight of every vertex
    w1 = f(p, v1, v2, v3);
    w2 = f(p, v2, v1, v3);
    w3 = f(p, v3, v1, v2);
    return w1 >= 0.0 && w2 >= 0.0 && w3 >= 0.0;
}
```

Convex hull $\mathcal{O}(n \log n)$

The convex hull of a set A is defined as

$$\text{conv}(A) = \{\lambda x + (1 - \lambda)y \mid x, y \in A, \lambda \in [0, 1]\}.$$

Given a finite set of points, their convex hull is a convex polygon with vertices that are in the original set of points. Using the Graham scan algorithm this subset of points can be determined.

```
// Square distance between two points
ll sqDist(ii a, ii b) {
    return (a.x - b.x) * (a.x - b.x) +
    (a.y - b.y) * (a.y - b.y);
}

// Determine orientation of two points w.r.t. another
int orient(ii p, ii q, ii r) {
    ll val = (q.y - p.y) * (r.x - q.x) -
    (q.x - p.x) * (r.y - q.y);
    if (val == 0)
        return 0;
    return (val > 0) ? 1 : 2;
}

// Determines the points that make up the convex hull.
// Returns empty list if convex hull doesn't exist.
// WARNING: Makes changes to the points list!
vii convexHull(vii &points) {
    // Find bottom-most point (left in case of a tie)
    ii ym = points[0]; ll yi = 0;
    REP(i, sz(points)) {
        ii point = points[i];
        if (ii{point.y, point.x} < ii{ym.y, ym.x})
            ym = point, yi = i;
    }
    swap(points[0], points[yi]);
    ii p0 = points[0];
    // Sort by orientation w.r.t. first point, is case
    // of tie pick closest first
    sort(all(points), [&](ii &a, ii &b) -> int {
        int o = orient(p0, a, b);
        if (o == 0)
            return sqDist(p0, a) < sqDist(p0, b);
        return o == 2;
    });
    // Remove some points that are on the same line
    ll m = 1;
    rep(i, 1, sz(points)) {
        while (i < sz(points) - 1 && orient(p0,
        points[i], points[i + 1]) == 0)
            i++;
        points[m] = points[i], m++;
    }
    if (m < 3)
        return {};
    // Process points and keep track of stack
    vii s = {points[0], points[1], points[2]};
    rep(i, 3, m) {
        while (sz(s) > 1 && orient(s[sz(s) - 2],
        s.back(), points[i]) != 2)
            s.pop_back();
        s.pb(points[i]);
    }
    return s;
}
```

1.4. Useful numbers

- Prime numbers: 31, 1031, 32771, 1048583, 8125343, 33554467, 9982451653, 1073741827, 34359738421, 1099511627791, 35184372088891, 1125899906842679, 36028797018963971, $10^3 + \{-9, -3, 9, 13\}$, $10^6 + \{-17, 3, 33\}$, $10^9 + \{7, 9, 21, 33, 87\}$.
- $\pi = \cos^{-1}(-1) = 3.14159265358979323846264\ldots$.
- $\varphi = \frac{1+\sqrt{5}}{2}$.
- $\log_{10}(2^{32}) \approx 9.632$, $\log_{10}(2^{64}) \approx 19.266$, $10^9 \approx 2^{30}$.
- Fibonacci numbers: $F_{n+2} = F_{n+1} + F_n$, starts with 0, 1. Lucas numbers start with 2, 1.
  - $\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \lim_{n\to\infty} \frac{L_{n+1}}{L_n} = \varphi$.
  - $F_{n-1}F_{n+1} - F_n^2 = (-1)^n$.
  - $F_{n+k} = F_k F_{n+1} + F_{k-1} F_n$.
- Catalan numbers: $C_n = \frac{1}{n+1}\binom{2n}{n}$, $C_0 = 1$.
  - $C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}$.
  - $C_{n+1} = \frac{4n+2}{n+2} C_n$

Modular inverses

p	M	$p^{-1} \mod M$
31	33554467	6494413
1031	33554467	28802816
32771	33554467	24638282
8125343	33554467	14214340
31	36028797018963971	26731042949553914
1031	36028797018963971	11776629093492588
32771	36028797018963971	30736232591595488
8125343	36028797018963971	34789068517540942

1.5. Matrices

Gaussian elimination $\mathcal{O}(nm \min\{n, m\})$

Gaussian elimination is used to reduce a matrix to reduced row echelon form. This form is useful for calculating the determinant, finding the inverse, etc.

```
const ld EPS = 1e-12;
typedef vector<ld> vd;
typedef vector<vd> vvd;

// Perform gaussian elimination on a matrix, which does
// not have to be square. The input matrix is modified!
// Returns the determinant of the matrix if it's square
ld gauss(vvd &a) {
    ll n = sz(a), m = sz(a[0]); // n rows, m columns
    ld det = 1.0;
    // Loop over pivot index
    REP(i, min(n, m)) {
        // Search for row where i'th column is non-zero
        ll j = i;
        for (; j < n; j++)
            if (abs(a[j][i]) > EPS) {
                if (i != j)
                    swap(a[i], a[j]), det *= -1.0;
                break;
            }
        // No pivot in this column
        if (j == n) {
            det = 0.0;
            continue;
        }
        // Reduce such that 1 is in the pivot column
        ld d = a[i][i];
        det *= d;
        REP(j, m)
            a[i][j] /= d;
        // Subtract row from all other rows
        REP(j, n)
            if (i != j) {
                ld d = a[j][i];
                REP(k, m)
                    a[j][k] -= d * a[i][k];
            }
    }
    return det;
}
```

Determinant of a matrix $\mathcal{O}(n^3)$

Defined recursively as $|A| = a_{11}$ if $A \in \mathrm{Mat}(1 \times 1)$, and for *any* j and $A \in \mathrm{Mat}(n \times n)$:

$$|A| := \sum_{i=1}^{n} (-1)^{i+j} a_{ij} \cdot |\tilde{A}_{ij}|.$$

where $\tilde{A}_{ij}$ is the matrix A with row i and column j removed. The determinant can be determined more quickly by reducing the matrix to reduced row-echelon form and keeping track of row multiplications (see above).

Multiplying two matrices $\mathcal{O}(Nnm)$

Multiplication of matrices $A \in \mathrm{Mat}(n \times N)$ and $B \in \mathrm{Mat}(N \times m)$ is defined as

$$(AB)_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

so that $AB \in \mathrm{Mat}(n \times m)$.

Inverse of a matrix $\mathcal{O}(n^3)$

The inverse of a matrix can be determined with the formula

$$A^{-1} = \frac{1}{|A|} \mathrm{adj}(A),$$

where $(\mathrm{adj}(A))_{ij} := (-1)^{i+j} C_{ji}$ with C_{ji} the determinant of A with row j and column i removed. This makes it possible to determine the inverse without division, but is $\mathcal{O}(n^5)$. The following implementation uses Gaussian elimination, which is $\mathcal{O}(n^3)$.

```
// Calculates the inverse of a matrix. The input matrix
// is modified to store the result! Assumes matrix is
// square and invertable (det != 0)!
void inv(vvd &a) {
    ll n = sz(a);
    REP(i, n) REP(j, n)
        a[i].pb(i == j ? 1.0 : 0.0);
    gauss(a);
    REP(i, n)
        a[i] = vd(a[i].begin() + n, a[i].end());
}
```

Miscellaneous

- $\begin{vmatrix} A & B \\ 0 & D \end{vmatrix} = |A| \cdot |D|$.
- If A is invertable, then $\begin{vmatrix} A & B \\ C & D \end{vmatrix} = |A| \cdot |D - CA^{-1}B|$.
- For $A, B \in \mathrm{Mat}(n \times n)$, $\begin{vmatrix} A & B \\ B & A \end{vmatrix} = |A - B| \cdot |A + B|$.
- For $A, B \in \mathrm{Mat}(2 \times 2)$, $|A+B| = |A| + |B| + \mathrm{tr}(A)\mathrm{tr}(B) - \mathrm{tr}(AB)$.
- $|A + B| \geq |A| + |B|$.
- $|cA| = c^n |A|$.
- $|A^\top| = |A|$.
- $|AB| = |A| \cdot |B|$.
- $|A^{-1}| = |A|^{-1}$.
- $|A| = \prod_{i=1}^{n} \lambda_i$.
- $\frac{n}{\mathrm{tr}(A^{-1})} \leq |A|^{1/n} \leq \frac{1}{n}\mathrm{tr}(A)$.
- $\begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad-bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$.
- $\mathrm{tr}(A) := \sum_{i=1}^{n} a_i$.
- For $A, B \in \mathrm{Mat}(n \times m)$ we have $\mathrm{tr}(A^\top B) = \mathrm{tr}(AB^\top) = \mathrm{tr}(B^\top A) = \mathrm{tr}(BA^\top) = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} b_{ij}$.
- $\mathrm{tr}(ABCD) = \mathrm{tr}(BCDA)$.
- $\mathrm{tr}(P^{-1}AP) = \mathrm{tr}(A)$.
- $\mathrm{tr}(A) = \sum_{i=1}^{n} \lambda_i$.

1.6. Linear equation solver $\mathcal{O}(nm \min\{n, m\})$

```
// Solve a linear system of equations Ax = b. If there
// no or multiple solutions, an empty vector is
// returned. The input matrix is modified!
vd linsys(vvd &a, const vd &b) {
    // Put vector b on the right side of matrix A
    REP(i, sz(b))
        a[i].pb(b[i]);
    gauss(a);
    // No solutions:
    REP(i, sz(a)) {
        bool allZero = true;
        REP(j, sz(a[i]) - 1)
            if (abs(a[i][j]) > EPS) {
                allZero = false;
                break;
            }
        if (allZero && abs(a[i].back()) > EPS)
            return {};
    }
    // Multiple solutions: (independent coordinates can
    // still be found)
    if (sz(a) < sz(a[0]) - 1)
        return {};
    REP(i, sz(a[0]) - 1)
        if (abs(a[i][i] - 1.0) > EPS)
            return {};
    // Return right column (modified b)
    vd out;
    REP(i, sz(a))
        out.pb(a[i].back());
    return out;
}
```

1.7. Roots of polynomials

Linear

The system $ax + b = 0$ has one solution if $a \neq 0$, given by

$$x = -\frac{b}{a}.$$

Quadratic

The system $ax^2 + bx + c = 0$ has at most two (possibly complex) solutions if $a \neq 0$, given by

$$x_\pm = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Cubic

Given is a cubic equation $ax^3 + bx^2 + cx + d = 0$ with $a \neq 0$. Define

$$\Delta_0 = b^2 - 3ac,$$
$$\Delta_1 = 2b^3 - 9abc + 27a^2 d,$$
$$C = \sqrt[3]{\frac{\Delta_1 \pm \sqrt{\Delta_1^2 - 4\Delta_0^3}}{2}},$$
$$\xi = \frac{-1 + \sqrt{-3}}{2}.$$

In the case of a complex number, any square root or cube root in the formula for C can be taken (there are multiple). The sign in the formula for C should be chosen such that $C \neq 0$. The three solutions are given by

$$x_k = -\frac{1}{3a}\left(b + \xi^k C + \frac{\Delta_0}{\xi^k C}\right), \qquad k \in \{0, 1, 2\}.$$

1.8. Series and sums

- Zeta constants: $\zeta(2n) = \sum_{k=1}^{\infty} \frac{1}{k^{2n}}$.
  - $\zeta(2) = \sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$.

  - $\zeta(4) = \sum_{n=1}^{\infty} \frac{1}{n^4} = \frac{\pi^4}{90}$.
  - $\zeta(6) = \sum_{n=1}^{\infty} \frac{1}{n^6} = \frac{\pi^6}{945}$.
- $\sum_{k=1}^{m} k = \frac{m(m+1)}{2}$.
- $\sum_{k=1}^{m} k^2 = \frac{m(m+1)(2m+1)}{6} = \frac{m^3}{3} + \frac{m^2}{2} + \frac{m}{6}$.
- $\sum_{k=1}^{m} k^3 = \left(\frac{m(m+1)}{2}\right)^2 = \frac{m^4}{4} + \frac{m^3}{2} + \frac{m^2}{4}$.
- $\sum_{k=m}^{n} z^k = \frac{z^m - z^{n+1}}{1-z}$.
- Polylogarithm:
  - $\sum_{k=1}^{\infty} \frac{z^k}{k} = -\ln(1-z)$.
  - $\sum_{k=1}^{\infty} z^k = \frac{z}{1-z}$.
  - $\sum_{k=1}^{\infty} k z^k = \frac{z}{(1-z)^2}$.
  - $\sum_{k=1}^{\infty} k^2 z^k = \frac{z(1+z)}{(1-z)^3}$.
- Exponential function:
  - $\sum_{k=0}^{\infty} \frac{z^k}{k!} = e^z$.
  - $\sum_{k=0}^{\infty} k \frac{z^k}{k!} = ze^z$.
  - $\sum_{k=0}^{\infty} k^2 \frac{z^k}{k!} = (z+z^2)e^z$.
- Trigonometry:
  - $\sin z = \sum_{k=0}^{\infty} \frac{(-1)^k z^{2k+1}}{(2k+1)!}$.
  - $\sinh z = \sum_{k=0}^{\infty} \frac{z^{2k+1}}{(2k+1)!}$.
  - $\cos z = \sum_{k=0}^{\infty} \frac{(-1)^k z^{2k}}{(2k)!}$.
  - $\cosh z = \sum_{k=0}^{\infty} \frac{z^{2k}}{(2k)!}$.
- Binomial coefficients:
  - $\sum_{k=0}^{n} \binom{n}{k} = 2^n$.
  - $\sum_{k=0}^{n} (-1)^k \binom{n}{k} = 0$ for $n \geq 1$.
  - $\sum_{k=0}^{n} \binom{k}{m} = \binom{n+1}{m+1}$.
  - $\sum_{k=0}^{n} \binom{m+k-1}{k} = \binom{n+m}{n}$.
  - $\sum_{k=0}^{n} \binom{\alpha}{k}\binom{\beta}{n-k} = \binom{\alpha+\beta}{n}$.
- $\sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} = \log 2$.
- $\sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{2k-1} = \frac{\pi}{4}$.

Minimum squared difference

$$\arg\min_{x \in \mathbb{R}} \sum_{k=1}^{n} (x_k - x)^2 = \frac{1}{n} \sum_{k=1}^{n} x_k$$

Binomials $\mathcal{O}(n)$

The binomial "n choose k" (typically $k \leq n$) is defined as

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

Then Newton's binomial theorem says that for $z \in \mathbb{C}$, $|z| < 1$ and $r \geq 0$ we have

$$(1+z)^r = \sum_{k=0}^{\infty} \binom{r}{k} z^k$$

Calculating a high power of a sum of $x, y \in \mathbb{C}$ is done with

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k.$$

```
// Calculates n choose k. Returns double because of
// large numbers, see function below for long longs
ld binom(ll n, ll k) {
    if (k > n)
        return 0.0;
    ld r = 1.0;
    REP(i, n - k)
        r *= ld(n - i) / ld(n - k - i);
    return r;
}
// This variant is slower than the above
ll binom(ll n, ll k) {
    if (k > n)
        return 0;
    ll r = 1;
    REP(i, n - k)
        r *= i + k + 1;
    REP(i, n - k)
        r /= i + 1;
    return r;
}
```

1.9. Probability

Basic properties

- $\mathbb{E}[X] = \sum_{x \in \Omega} x\mathbb{P}(X = x)$.
- $\mathbb{E}[X] = \mathbb{E}[X \mid A]\mathbb{P}(X \in A) + \mathbb{E}[X \mid B]\mathbb{P}(X \in B)$, when $\Omega = A \cup B$ and $A \cap B = \varnothing$.
- $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$.
- $\mathbb{P}(X \in A) = 1 - \mathbb{P}(X \notin A)$.
- $\mathbb{P}(X \leq x) = \mathbb{P}(X = x) + \mathbb{P}(X < x)$.
- $\mathbb{P}(X \in A) = \mathbb{P}(X \in A_1) + \mathbb{P}(X \in A_2)$, when $A = A_1 \cup A_2$ and $A_1 \cap A_2 = \varnothing$.
- $\mathbb{P}(X \mid Y) = \mathbb{P}(X \cap Y)/\mathbb{P}(Y)$.

Bayes' Theorem

$$\mathbb{P}(X \mid Y) = \frac{\mathbb{P}(Y \mid X)P(X)}{P(Y)}$$

Independence

Two random variables X and Y are independent if $\mathbb{P}(X \cap Y) = \mathbb{P}(X)\mathbb{P}(Y)$.

When random variables $X_1, \ldots, X_n$ are independent and identically distributed, we have

$$\mathbb{P}(X_1 = \cdots = X_n = x) = \mathbb{P}(X_1 = x)^n.$$

Hölder's inequality

For $p, q \in [1, \infty)$ with $\frac{1}{p} + \frac{1}{q} = 1$ and f, g μ-measurable ("normal" functions) we have

$$\int |fg| \, d\mu \leq \left(\int |f|^p \, d\mu\right)^{\frac{1}{p}} \left(\int |g|^q \, d\mu\right)^{\frac{1}{q}}.$$

Markov Chains

A Markov chain is given by a transition matrix, which gives to probability to transition from one state to another:

$$P_{ij} = \mathbb{P}(X_n = j \mid X_{n-1} = i).$$

In particular Markov chains have the Markov property:

$$\mathbb{P}(X_{n+1} = x \mid X_1 = x_1, \ldots, X_n = x_n)$$
$$= \mathbb{P}(X_{n+1} = x \mid X_n = x_n).$$

Suppose that we have a Markov chain with matrix

$$P = \begin{pmatrix} Q & R \\ 0 & I_r \end{pmatrix}.$$

This Markov chain is absorbing, with r absorbing states (and t transient states). The fundamental matrix is defined as

$$N = (I_t - Q)^{-1}.$$

Then the expected number of steps before being absorbed, starting in state i, is the i-th entry in the vector $N\mathbb{1}$. The probability of being absorbed into absorbing state j, starting from state i, is the i, j-entry in NR.

The probability of visiting transient state j when starting in transient state i is the i, j-entry in $(N - I_t)(N_{\text{dg}})^{-1}$. Here N_{dg} is the diagonal matrix with the same diagonal as N.

Common distributions

Distributions below are over some discrete set (support).

Name	$\mathbb{P}(X=k)$	Supp.	μ	σ^2
Bern.	$p^{1-k}(1-p)^k$	$\{0,1\}$	p	$p(1-p)$
Bin.	$\binom{n}{k}p^k(1-p)^{n-k}$	$\{0,\dots,n\}$	np	$np(1-p)$
Geo.	$(1-p)^{k-1}p$	$\mathbb{N}_0$	$\frac{1}{p}$	$\frac{1-p}{p^2}$
Pois.	$\frac{\lambda^k e^{-\lambda}}{k!}$	$\mathbb{N}_0$	λ	λ
Unif.	$\frac{1}{b-a+1}$	$\{a,\dots,b\}$	$\frac{a+b}{2}$	$\frac{(b-a+1)^2-1}{12}$

Distributions below are over some continuous set (support), given are density functions f.

Name	$f(x)$	Supp.	μ	σ^2
Beta	$\frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$	$[0,1]$	$\frac{\alpha}{\alpha+\beta}$	$\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$
Unif.	$\frac{1}{b-a}$	$[a,b]$	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$
Exp.	$\lambda e^{-\lambda x}$	$[0,\infty)$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$
Gam.	$\frac{x^{k-1}e^{-x/\theta}}{\Gamma(k)\theta^k}$	$(0,\infty)$	$k\theta$	$k\theta^2$
Nor.	$\frac{\exp\left(-\frac{(x-\mu)^2}{(\sigma^2}\right)}{\sigma\sqrt{2\pi}}$	$\mathbb{R}$	μ	σ^2
t, $\nu=1$	$\frac{1}{\pi(1+t^2)}$	$\mathbb{R}$		

The beta and gamma functions are defined as

$$B(\alpha,\beta)=\int_0^1 t^{\alpha-1}(1-t)^{\beta-1}\,dt,$$

$$\Gamma(k)=(k-1)!=\int_0^\infty t^{k-1}e^{-t}\,dt.$$

Common situations

- Suppose an event happens event time unit with probability p, then the probability of having it occur *exactly* n times in the first m time units ($m\geq n$) is equal to
$$\mathbb{P}(S=n)=\binom{m}{n}p^n(1-p)^{m-n}.$$
The probability of it happening *at least* n times is then
$$\mathbb{P}(S\geq n)=\sum_{k=n}^{m}\mathbb{P}(S=k).$$
- Given a game with n rounds, where the prizes making it through to round i are r_i and the probability of making it through round i is p_i. The player can stop at any round (including before the first round). Then the expected winnings W playing optimally, given that the player is currently at round I are
$$\mathbb{E}[W\mid I=i]=\max\left\{r_i,\ p_i\mathbb{E}[W\mid I=i+1]\right\},$$
$$\mathbb{E}[W]=\mathbb{E}[W\mid I=0].$$
- Assume we grab items from a set without putting them back, until there are no more items left. If each item is chosen with the same probability, then the probability of a specific sequence is
$$\mathbb{P}(X_1=x_1,\dots,X_n=x_n)=\frac{1}{n!}.$$
- Usually DP is very useful for probability problems.

Entropy

Entropy indicates the uncertainty factor of a random variable. The definition of entropy is

$$H(X)=-\sum_{x\in\mathcal{X}}p(x)\log_2 p(x)=\mathbb{E}[-\log_2 p(X)].$$

Some properties are:

- For any function f, $H(f(X))\leq H(X)$.
- For X and Y independent, $H(X\mid Y)=H(X)$.
- For any X and Y, $H(X\mid Y)\leq H(X)$.
- For any X and Y, $H(Y,X)=H(X,Y)=H(X\mid Y)+H(Y)$.

1.10. Combinatorics

Burnside's lemma

Let G be a finite group that acts on a set X, let X^g be the set of elements in X that are fixed by g, then the number of orbits is

$$|X/G|=\frac{1}{|G|}\sum_{g\in G}|X^g|.$$

Pólya enumeration theorem (unweighted)

Let G be a group of permutations of a finite set X. Denote Y^X for all functions $X\to Y$. Then the group G acts on Y^X, and the number of orbits is

$$|Y^X/G|=\frac{1}{|G|}\sum_{g\in G}|Y|^{c(g)},$$

where $c(g)$ is the number of cycles of g when viewed as a permutation of X.

k-th combination $\mathcal{O}(n\log n)$

Given a number n, the following returns the k-th combination in the list of bit sequences of length n that contain m ones. Note that the total number of combinations is $\binom{n}{m}$.

```
ll kthComb(ll n, ll m, ll k) {
    if (m >= n)
        return ~(~0LL << n);
    if (m <= 0)
        return 0;
    // Zeros first, then ones
    ll s = binom(n - 1, m);
    if (k < s)
        return nthComb(n - 1, m, k);
    return nthComb(n - 1, m - 1, k - s) |
    (1LL << (n - 1));
}
```

List of combinations $\mathcal{O}(n\binom{n}{m})$

Given numbers n and m, the following produces a list of all bit sequences of length n that contain m ones, without enumerating over all 2^n possible bit sequences of length n.

```
vi allCombs(ll n, ll m) {
    if (m >= n)
        return {~(~0LL << n)};
    if (m <= 0)
        return {0};
    // Prepend both 0 and 1
    vi z = allCombs(n - 1, m);
    vi o = allCombs(n - 1, m - 1);
    for (ll i : o)
        z.pb(i | (1LL << (n - 1)));
    return z;
}
```

1.11. Analysis and calculus

L'Hôpital's rule

If $\lim_{x\to c}f(x)=\lim_{x\to c}g(x)=0$ and the right limit exists, then

$$\lim_{x\to c}\frac{f(x)}{g(x)}=\lim_{x\to c}\frac{f'(x)}{g'(x)}.$$

Integration by parts

$$\int_a^b f'(x)g(x)\ dx = [f(x)g(x)]_a^b - \int_a^b f(x)g'(x)\ dx.$$

Common integrals

$$\int \tan x\ dx = -\log|\cos x| + C$$
$$\int \cot x\ dx = \log|\sin x| + C$$
$$\int \csc x\ dx = \log|\csc x + \cot x| + C$$
$$\int \sec x\ dx = \log|\sec x + \tan x| + C$$
$$\int_0^\infty e^{-ax^2}\ dx = \frac{1}{2}\sqrt{\frac{\pi}{a}}$$
$$\int_0^\infty \frac{x}{e^x - 1}\ dx = \frac{\pi^2}{6}$$

Common derivatives

$$\frac{d}{dx}\sin^{-1} x = \frac{1}{\sqrt{1-x^2}}$$
$$\frac{d}{dx}\cos^{-1} x = -\frac{1}{\sqrt{1-x^2}}$$
$$\frac{d}{dx}\tan^{-1} x = \frac{1}{1+x^2}$$

Common limits

$$\lim_{x\to 0}\frac{a^x - 1}{x} = \log a$$
$$\lim_{x\to 0}\frac{\sin(ax)}{bx} = \frac{a}{b}, \qquad \text{if } b \neq 0$$
$$\lim_{n\to\infty}\frac{n}{\sqrt[n]{n!}} = e$$
$$\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{n!} = 1, \qquad \text{(Stirling)}$$

Partial fractions

A fraction with a product of linear expressions at the bottom can be split:

$$\frac{ax+b}{(x-a_1)\dots(x-a_m)} = \frac{A_1}{x-a_1} + \dots + \frac{A_2}{x-a_m}.$$

Find the A_i by solving the system of equations that you get when transforming the sum into one fraction and comparing the factors before x and the constants.

Newton's method

For most differentiable functions f and sequences $(x_n)_{n=0}^\infty$ with

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

the sequence $(f(x_n))_{n=0}^\infty$ will converge to zero, i.e. the limit of $(x_n)_{n=0}^\infty$ is a root of f. Newton's method uses this property to find roots of functions, by starting with x_0 close to a root.

```
const ld EPS = 1e-12;

// Finds some root of a function f, given f, f' and an
// initial guess x. It would perform well if x is close
// to a root. A max iteration count could be added
ld newton(ld (*f)(ld), ld (*fp)(ld), ld x = 0.0) {
    while (abs(f(x)) > EPS) {
        if (abs(fp(x)) < EPS )
            x = x - EPS;
        else
            x = x - f(x) / fp(x);
    }
    return x;
}
```

Extremum of a parabola

The minimum or maximum of a parabola with the formula $y = ax^2 + bx + c$ is given by the point

$$(x, y) = \left(-\frac{b}{2a}, c - \frac{b^2}{4a}\right).$$

1.12. Fast Fourier transform $\mathcal{O}(n \log n)$

The discrete Fourier transform of the polynomial $A(x) = a_0x^0 + \dots + a_{n-1}x^{n-1}$ with $n = 2^m$ is defined as

$$\mathcal{F}(a_0, \dots, a_{n-1}) = (A(w_n^0), \dots, A(w_n^{n-1})),$$

where $w_n = \exp(\frac{2\pi i}{n})$. When multiplying polynomials A and B we have $\mathcal{F}(A \cdot B) = \mathcal{F}(A) \cdot \mathcal{F}(B)$ where multiplication is done coordinate-wise. This gives

$$A \cdot B = \mathcal{F}^{-1}(\mathcal{F}(A) \cdot \mathcal{F}(B)).$$

FFT can also be used to multiply two very large numbers quickly, by using the multiplication of polynomials and filling in $x = 10$.

```
typedef complex<ld> cd;
const ld PI = acos(-1.0);

// This function calculates the fft and inverse fft of
// a polynomial function coefficients a. The result is
// again put in a. The size of a needs to be 2^n
void fft(vector<cd> &a, bool invert) {
    if (sz(a) == 1)
        return;
    // Determines left and right sides and applies
    // transform on them separately
    vector<cd> l(sz(a) / 2), r(sz(a) / 2);
    REP(i, sz(a) / 2)
        l[i] = a[2 * i], r[i] = a[2 * i + 1];
    // Apply FFT to left and right sides
    fft(l, invert), fft(r, invert);
    // Calculate w1
    ld ang = 2.0 * PI / ld(sz(a)) * (invert ? -1.0 :
    1.0);
    cd w(1.0), wn(cos(ang), sin(ang));
    REP(i, sz(a) / 2) {
        a[i] = l[i] + w * r[i];
        a[i + sz(a) / 2] = l[i] - w * r[i];
        if (invert)
            a[i] /= 2.0, a[i + sz(a) / 2] /= 2.0;
        w *= wn;
    }
}

// Calculates the coefficients of two polynomials
// multiplied
vi mul(const vi &a, const vi &b) {
    vector<cd> fa(all(a)), fb(all(b));
    // Calculate the size rounded up to 2^n
    ll n = 1;
    while (n < sz(a) + sz(b))
        n <<= 1;
    fa.resize(n), fb.resize(n);
    // Apply FFT to two polynomials separately
    fft(fa, false), fft(fb, false);
    REP(i, n)
        fa[i] *= fb[i];
    // Apply inverse FFT to product
    fft(fa, true);
    vi res(n);
```

```
    REP(i, n)
        res[i] = round(fa[i].real());
    return res;
}
```

2. Data structures

```
vector<T> // Contiguous memory, push_back in O(1) time
list<T> // Non-contiguous memory
deque<T> // Contraction/expansion on both sides
array<T, n> // Alternative to C arrays

queue<T> // Push back, pop front
priority_queue<T> // Ordered push, pop front
stack<T> // Push back, pop back

set<T> // Ordered set
multiset<T> // Ordered multiset
map<T, S> // Ordered map
multimap<T, S> // Ordered multimap

unordered_set<T> // Hash table set
multiset<T> // Hash table multiset
unordered_map<T, S> // Hash table map
unordered_multimap<T, S> // Hash table multimap
```

2.1. Disjoint-set union

The disjoint-set structure is used to represent a partition of dataset. It supports two basic operations:

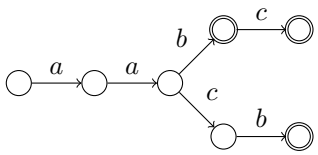
- FIND: Returns the representative of the subset that the given item is a part of. The representative is the same for any item in the same subset. $\mathcal{O}(\alpha(n))$
- COMBINE: (union) Combine two subsets, making the representatives of the items in the set the same. $\mathcal{O}(\alpha(n))$

The implementation uses a list of numbers $\{0, \ldots, n-1\}$. These can then be used as indices of more complex data. The unions are represented as rooted trees, where every node points to its representative. The α in the complexities above is a very slow growing function. The implementation below is an expanded version of the basic DSU data structure.

```
// A disjoint union data structure, initialized as
// singletons. The structure also keeps track of set
// sizes and set sums. The values of all elements are
// initialized at 0
struct DisjointUnion {
    // Parent node, sum of nodes, number of nodes
    vi prev, sums, sizes;
    DisjointUnion(ll n) : prev(n, -1), sums(n, 0),
    sizes(n, 1) { }
    // Find a representative of the
    ll find(ll x) {
        if (prev[x] < 0)
            return x;
        return prev[x] = find(prev[x]);
    }
    // Combine two sets, the root of the largest set
    // becomes the new root
    void combine(ll x, ll y) {
        if ((x = find(x)) == (y = find(y)))
            return;
        if (sizes[x] < sizes[y])
            swap(x, y);
        prev[y] = x;
        sizes[x] += sizes[y];
        sums[x] += sums[y];
    }
    // Get the size of the set that x is contained in
    ll size(ll x) {
        return sizes[find(x)];
    }
    // Update the value of the set that x is contained
    // in by adding v
    void update(ll x, ll v) {
        sums[find(x)] += v;
    }
    // Get the sum of the values of elements of the set
    // that x is contained in
    ll sum(ll x) {
        return sums[find(x)];
    }
};
```

2.2. Trie

A trie is used to represent a collection of strings. It stores the prefixes of strings in a tree structure. For example the strings `aab`, `aabc` and `aacb` are represented as:



The trie supports three basic operations:

- FIND: Check if a string S is in the trie. $\mathcal{O}(|S|)$
- INSERT: Add a string S to the trie. $\mathcal{O}(|S|)$
- ERASE: Remove a string S from the trie. Note that the structure of the tree is not removed, which can hurt overall performance after the erase. $\mathcal{O}(|S|)$

The implementation uses a list of integers to keep track of the children of nodes.

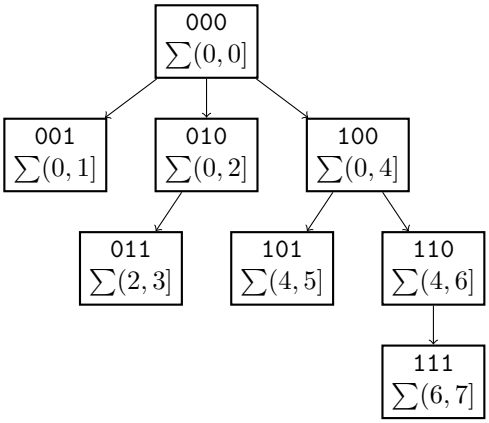
```
// Trie data structure, can store strings efficiently.
// Is empty as initialization. Note that strings can be
// deleted, but this is not recommended. Change the
// base to the lowest ASCII char that should be
// accepted and amt to the amount of accepted chars.
struct Trie {
    const char base = 'a';
    const ll amt = 26;
    vvi children = {vi(amt, -1)};
    vi leaves = {0};
    bool find(const string &s) {
        ll node = 0;
        for (char c : s) {
            node = children[node][c - base];
            if (node < 0)
                return false;
        }
        return leaves[node];
    }
    void insert(const string &s) {
        ll node = 0;
        for (char c : s) {
            if (children[node][c - base] < 0) {
                children[node][c - base] =
                sz(children);
                leaves.pb(0);
                children.pb(vi(amt, -1));
            }
            node = children[node][c - base];
        }
        leaves[node] = 1;
    }
    void erase(const string &s) {
        ll node = 0;
        for (char c : s) {
            if (children[node][c - base] < 0)
                return;
```

```
            node = children[node][c - base];
        }
        leaves[node] = 0;
    }
};
```

2.3. Fenwick tree

Fenwick trees are most commonly used to calculate prefix sums (i.e. sums up to a given point), but can also be used for other binary associative operators on groups. It supports two basic operations:

- SUM: Get the prefix sum before an index i. $\mathcal{O}(\log n)$
- UPDATE: update an element of the tree. $\mathcal{O}(\log n)$



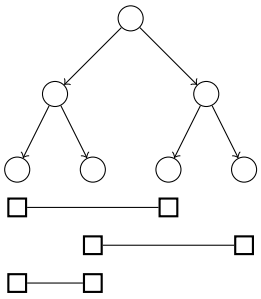
Note that the number of elements in the tree is fixed. The tree is implemented such that every child of the root corresponds to a power of two (each sub-tree is then a Fenwick tree itself). This node then contains a prefix sum.

```
// Implements a fenwick tree, use prefixSum to get the
// sum of the first i elements, use update to change a
// value (adds v to the entry)
struct Fenwick {
    vi data;
    Fenwick(ll n) : data(n + 1, 0) { }
    ll prefixSum(ll i) {
        ll sm = 0;
        i++;
        while (i > 0)
            sm += data[i], i = i - (i & (-i));
        return sm;
    }
    void update(ll i, ll v) {
        i++;
        while (i < sz(data))
            data[i] += v, i = i + (i & (-i));
    }
};
```

2.4. Segment tree

A segment tree is more efficient in calculating sums of intervals than Fenwick trees, and also has more flexibility in the operation that needs to be determined over an interval. Segment trees have two basic operations:

- SUM: Get the sum of the elements on an interval $[a, b]$. $\mathcal{O}(\log n)$
- UPDATE: Update an entry in list that the tree represents. $\mathcal{O}(\log n)$



The implementation uses nodes which store the sum of all of the leaves below it. The start and end points of all of the intervals are considered in the tree.

```
// Segment tree, which supports updating a value and
// finding the sum of an interval [start, end]
struct SegmentTree {
    vi nodes;
    ll n;
    SegmentTree(vi &src) : nodes(vi(2 * sz(src), 0)),
    n(sz(src)) {
        REP(i, n)
            nodes[n + i] = src[i];
        for (ll i = n - 1; i > 0; i--)
            nodes[i] = nodes[i << 1] +
            nodes[i << 1 | 1]; // change op
    }
    void update(ll i, ll val) {
        nodes[i + n] = val, i += n;
        for (ll j = i; j > 1; j >>= 1)
            // change op
            nodes[j >> 1] = nodes[j] + nodes[j ^ 1];
    }
    ll getSum(ll start, ll end) {
        ll r = 0; end++;
        for (start += n, end += n; start < end;
        start >>= 1, end >>= 1) {
            if (start & 1)
                r += nodes[start++]; // change op
            if (end & 1)
                r += nodes[--end]; // change op
        }
        return r;
    }
};
```

2.5. Sparse table

A sparse table is similar to a segment tree, but the values of the entries cannot be changed. At creation a table is generated that can then be used to query the minimum of a certain range. The sparse table has two basic operations:

- BUILD: Initialization creates the lookup table for a source list. $\mathcal{O}(n \log n)$
- QUERY: Get the minimum value in the list in a range of indices $[l, r)$. $\mathcal{O}(1)$

$[0,1)$	$[1,2)$	$[2,3)$	$\cdots$	$[n,n+1)$
$[0,2)$	$[1,3)$	$[2,4)$	$\cdots$	$[n,n+2)$
$[0,4)$	$[1,5)$	$[2,6)$	$\cdots$	$[n,n+4)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$[0,2^k)$	$[1,1+2^k)$	$[2,2+2^k)$	$\cdots$	$[n,n+2^k)$

The implementation calculates the minimum of all intervals

$[i, i + 2^k)$ and puts them in a table. Then the minimum of any interval $[\ell, r)$ can be determined by taking the minimum of the largest interval $[\ell, \ell+2^k)$ that is contained in $[\ell, r)$, and likewise the minimum of the largest interval $[a, r)$. These intervals may overlap, which does not matter in the case of a minimum/maximum.

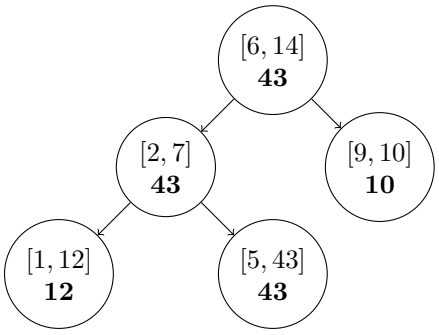
```
// Implements the sparse table data structure, the
// input list has to be given at construction.
struct SparseTable {
    // Lookup table
    vvi data;
    SparseTable(const vi &inp) {
        // Precompute the data table
        ll K = 0, k = sz(inp);
        while (k > 0)
            k /= 2, K++;
        data = vvi(K + 1, vi(sz(inp)));
        data[0] = inp;
        rep(i, 1, K + 1)
            REP(j, sz(inp) - (1LL << i) + 1)
                // min() can be replaced with a
                // different function
                data[i][j] = min(data[i - 1][j],
                data[i - 1][j + (1LL << (i - 1))]);
    }
    // Searches for interval minimum in [l, r)
    ll query(ll l, ll r) {
        if (r <= l)
            return LLONG_MAX;
        ll p = -1, d = r - l;
        while (d > 0)
            d /= 2, p++;
        return min(data[p][l],
        data[p][r - (1LL << p)]);
    }
};
```

2.6. Interval tree

An interval tree is used to find all (closed and bounded) intervals containing a point in the tree. The tree supports two basic operations:

- INSERT: Insert an interval into the tree. $\mathcal{O}(\log n)$
- SEARCH: Find all m intervals that contain a given point. $\mathcal{O}(\log n + m)$

The tree is implemented as a binary search tree with the left bound of the intervals being the key, and containing annotations about the maximum value in a subtree.



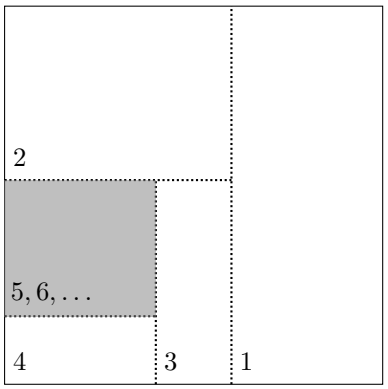

```
// Implementation of an unbalanced interval tree that
// supports inserting intervals and finding point
// matches
struct IntervalTree {
    // List of intervals, maximum values of nodes,
    // children, and root node
    vii vals; vi mx; vii ch; ll root = -1;
    // Insert interval into tree
    void insert(ii val) {
        // Keep track of parent node and which side the
        // child node is
        ll parent = -1, node = root;
        bool side = true;
        // Traverse the tree
        while (node != -1) {
            parent = node;
            // Update maximum
            if (val.y > mx[node])
                mx[node] = val.y;
            // Insert left or right
            if (val.x < vals[node].x)
                node = ch[node].x, side = true;
            else
                node = ch[node].y, side = false;
        }
        if (parent == -1)
            root = 0;
        else
            (side ? ch[parent].x : ch[parent].y) =
            sz(vals);
        vals.pb(val), mx.pb(val.y), ch.pb({-1, -1});
    }
    // Retrieve all overlapping intervals with given
    // point
    vii search(ll val) {
        if (root == -1)
            return {};
        vii res;
        queue<ll> q; q.push(root);
        while (!q.empty()) {
            ll node = q.front();
            q.pop();
            // Current node interval contains point
            if (vals[node].x <= val && val <=
            vals[node].y)
                res.pb(vals[node]);
            // Search left node
            if (ch[node].x != -1 && val <=
            mx[ch[node].x])
                q.push(ch[node].x);
            // Search right node
            if (ch[node].y != -1 && val >=
            vals[node].x)
                q.push(ch[node].y);
        }
        return res;
    }
};
```

2.7. k-d tree

A k-d tree can quickly search and insert points in a multi-dimensional data structure. It supports two basic operations:

- INSERT: Insert a point into the tree. $\mathcal{O}(k \log n)$
- SEARCH: Check if the given point is contained in the tree. $\mathcal{O}(k \log n)$

The tree is implemented by having each layer compare a different dimension in a cycle.



```
// Implements a k-d tree, change dimension based on
// needed. All points inserted must be of size k.
// Supports search and insert operations. Deletion can
// be added by marking points as unused
struct KDTree {
    // Change dimension here
    const ll k = 2;
    // Stores point data and children
    vvi points; vii ch; ll root = -1;
    // Helper insert function, given are the root node
    // of the current subtree and the current dimension
    ll insert(const vi &p, ll i, ll d) {
        // If there is no root, a new node needs to be
        // created
        if (i == -1) {
            points.pb(p);
            ch.pb({-1, -1});
            return sz(points) - 1;
        }
        // Check left or right in current dimension
        if (p[d] < points[i][d]) {
            ll c = insert(p, ch[i].x, (d + 1) % k);
            ch[i].x = c;
        } else {
            ll c = insert(p, ch[i].y, (d + 1) % k);
            ch[i].y = c;
        }
        return i;
    }
    // Insert a point
    void insert(const vi &p) {
        root = insert(p, root, 0);
    }
    // Search function, given are the root node of the
    // current subtree and the current dimension
    bool search(const vi &p, ll i = -2, ll d = 0) {
        if (i == -2)
            i = root;
        if (i == -1)
            return false;
        if (p == points[i])
            return true;
        // Search is done in the same way as insert
        if (p[d] < points[i][d])
            return search(p, ch[i].x, (d + 1) % k);
        return search(p, ch[i].y, (d + 1) % k);
    }
};
```

3. Graphs

3.1. Theorems

Euler's formula

For any connected planar graph (V, E, F), which is a graph that can be drawn in 2-D without intersecting edges, we have

$$V - E + F = 2.$$

Here V is the number of vertices, E the number of edges and F the number of faces (including the outside). For a non-connected planar graph with C connected components, we have

$$V - E + F = C + 1.$$

Eulerian cycle

An Eulerian path is a path that uses each edge exactly once. An Eulerian cycle starts and ends in the same vertex. For an undirected connected graph, an Eulerian cycle exists if and only if every vertex has even degree.

A strongly connected directed graph has an Eulerian cycle if and only if all vertices have the same in degree and out degree.

Eulerian path

For an undirected connected graph, an Eulerian path exists if and only if there are exactly zero or two vertices with odd degree.

A strongly connected directed graph has an Eulerian path if and only if at most one vertex has in degree one more than out degree, at most one vertex has out degree one more than in degree, and every other vertex has equal in degree and out degree.

Dirac's/Ore's theorem

Let (V, E) be a connected graph such that for all $x, y \in V$ with $x \neq y$ we have $d(x) + d(y) \geq |V|$, then G contains a Hamiltonian circuit. The graph G is still Hamiltonian if this only holds for adjacent x and y.

Cayley's theorem

For $n \geq 1$ there are n^{n-2} tree graphs on n labelled vertices.

Kőnig's theorem

In a bipartite graph, the minimum vertex cover has the same size as the maximum matching set.

A vertex cover is a set of vertices such that every edge has at least one vertex that is part of the vertex cover. A matching set is a set of edges such that no vertex is included in more than one edge.

Max-flow min-cut theorem

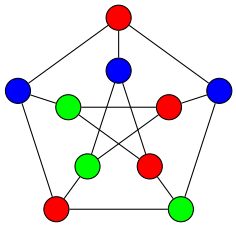
The minimum cut in a flow network is the same as the maximum flow. Min-cut is the minimum total weight of edges to remove such that the source and sink are no longer connected. The max-flow is the maximum flow of some liquid from the source to the sink.

Maximum independent set

The maximum independent set of a graph is the maximum size set of vertices that do not share an edge. The complement of the maximum independent set is a minimum vertex cover.

Petersen graph

The Petersen graph is often used for counterexamples in graph theory:



3.2. Shortest path

Dijkstra $\mathcal{O}(E \log V)$

The shortest path is found by iteratively picking the shortest edge from a node in the currently reached set to any other node.

```
// Calculates the shortest path from the starting
// points to any other point. Returns the distance to
// any point, and the previous nodes in the shortest
```

```
// path. The graph input G represents the list of
// points, each containing a list of connections to
// other points, these are pairs: the other point and
// the distance. Returns LLONG_MAX if a point is not
// reachable
pair<vi, vi> dijkstra(const vector<vii> &G, ll s) {
    vi dist(sz(G), LLONG_MAX), pr(sz(G), -1);
    dist[s] = 0;
    // Priority queue: Lowest distance first
    set<ii> q; q.insert({dist[s], s});
    // While there are points not visited (or other
    // points are not reachable)
    while (!q.empty()) {
        // Vertex with shortest distance
        ll v = q.begin()->y;
        q.erase(q.begin());
        // Go over all connections from this point
        for (const ii &e : G[v])
            // Distance calculation can be altered
            if (dist[v] + e.y < dist[e.x]) {
                q.erase({dist[e.x], e.x});
                dist[e.x] = dist[v] + e.y;
                q.insert({dist[e.x], e.x});
                pr[e.x] = v;
            }
    }
    return {dist, pr};
}
```

Floyd-Warshall $\mathcal{O}(V^3)$

Calculates shortest distances between all nodes by considering a route "via k".

```
// Has as input the graph distances represented using
// an adjacency matrix. This matrix is then modified to
// reflect minimal path distances between nodes. When
// there is no connection between nodes, this should be
// represented with LLONG_MAX
void floydWarshall(vvi &d) {
    REP(k, sz(d)) REP(i, sz(d)) REP(j, sz(d))
        if (d[i][j] > d[i][k] + d[k][j]
        && LLONG_MAX - d[i][k] > d[k][j])
            d[i][j] = d[i][k] + d[k][j];
}
```

Flood fill reachable nodes $\mathcal{O}(EM)$

Only works for graphs with distances in $\{0, 1\}$. Note that the complexity is dependent on M, the maximum distance to any node. Only use this if M is low, otherwise use Dijkstra and select nodes based on distance.

This algorithm can be modified to calculate reachable nodes for any graph, but then the maximum distance requirement has to be dropped.

```
// Determines reachable nodes with a maximum distance
// Input are the graph, the starting node and the
// maximum distance. Returns the set of reachable
// nodes. Set the maximum distance very high to
// determine connected subgraph. Assumes that all
// distances in the graph are either 0 or 1!
vi reachable(const vector<vii> &G, ll s, ll M) {
    // Keeps track of nodes that are reachable
    vi reach(sz(G), 0);
    reach[s] = 1;
    // Create queues for current and next distance
    // steps
    queue<ll> cur, next;
    cur.push(s);
    // Loop over all possible distance steps
    REP(m, M + 1) {
        if (cur.empty())
            break;
        while (!cur.empty()) {
            ll v = cur.front();
            cur.pop();
            reach[v] = 1;
            for (const ii &e : G[v]) {
                if (reach[e.x])
                    continue;
                if (e.y)
                    next.push(e.x);
                else
                    cur.push(e.x);
            }
        }
        cur = next;
    }
    return reach;
}
```

k shortest paths

A modified version of Dijkstra's algorithm can be used to find the k shortest paths in an undirected graph. Note that this implementation is very slow!

```
// Calculates the k shortest paths from source s to
// sink t in an undirected weighted graph G, given as
// adjacency lists
vvi kShortest(const vector<vii> &G, ll s, ll t, ll k) {
    // P contains all found paths
    vvi P;
    // Number of paths found to a certain node
    vi count(sz(G), 0);
    // Heap to store apths to nodes with given lengths.
    // Unlike in normal Dijkstra the entire path needs
    // to be stored
    set<pair<ll, vi>> Q;
    Q.insert({0, {s}});
    while (!Q.empty() && count[t] < k) {
        pair<ll, vi> p = *Q.begin();
        ll u = p.y.back();
        Q.erase(Q.begin()), count[u]++;
        // Save paths to sink node
        if (u == t)
            P.pb(p.y);
        // Go over neighbours and add lengths and paths
        // to the heap
        if (count[u] < k) for (ii e : G[u]) {
            pair<ll, vi> pe = p;
            pe.x += e.y, pe.y.pb(e.x);
            Q.insert(pe);
        }
    }
    return P;
}
```

IDA*

This algorithm is used for finding the shortest path in many AI applications. Suppose we want to find the shortest path from s to t. The algorithm uses a heuristic $h : V \to \mathbb{R}_{\geq 0}$ that has the property

$$h(v) \leq d(v, t), \quad v \in V.$$

It uses a kind of DFS with updating thresholds for further searching.

```
// Fill the heuristic function in here:
ll h(ll i);

// Search function, returns minimum or -1 if
// destination was found, LLONG_MAX if no path was
// found
ll search(const vector<vii> &G, ll s, ll t, vi &path,
ll g, ll bound) {
    ll cur = path.back();
    ll f = g + h(cur);
    if (f > bound)
```

```
        return f;
    if (cur == t)
        return -1;
    ll mn = LLONG_MAX;
    // Go over neighbours
    for (ii e : G[cur]) {
        bool inPath = false;
        for (ll p : path) if (p == e.x) {
            inPath = true;
            break;
        }
        if (inPath)
            continue;
        path.pb(e.x);
        ll c = search(G, s, t, path, g + e.y, bound);
        if (c == -1)
            return -1;
        if (c < mn)
            mn = c;
        path.pop_back();
    }
    return mn;
}

// The input graph G can be directed, and is given as
// adjacency lists, s is the start node, t the
// destination. If no shortest path exists, returns
// empty vector
vi idaStar(const vector<vii> &G, ll s, ll t)
{
    ll bound = h(s);
    vi path = {s};
    // Search until no extra nodes can be reached
    while (bound != LLONG_MAX) {
        ll c = search(G, s, t, path, 0, bound);
        if (c == -1)
            // Here, "bound" will be the distance from
            // s to t
            return path;
        bound = c;
    }
    return {};
}
```

3.3. Minimum spanning tree

When working with a general graph, Prim is likely best. When finding the MST of points (which are all connected by distance) on a 2D surface, Kruskal is likely best.

Prim $\mathcal{O}(E \log V)$

Prim starts at some node and expands the tree by adding the node at the shortest distance from the tree that has not been added yet.

```
// Executes Prim's algorithm on a graph G, represented
// by adjacency lists. Returns the total length of the
// MST and the edges that make it up.
// WARNING: The graph needs to be undirected and
// connected and thus contain (v1, v2) if it contains
// the edge (v2, v1)!
pair<ll, vii> prim(const vector<vii> &G) {
    vi done(sz(G), 0);
    done[0] = 1;
    vii edges;
    ll len = 0;
    set<pair<ll, ii>> q;
    // Find connections of first node
    for (ii c : G[0])
        q.insert({c.y, {0, c.x}});
    while (!q.empty()) {
        // Retrieve edge with lowest length, and check
        // if node has been marked as done
        pair<ll, ii> cur = *q.begin();
        q.erase(q.begin());
        if (done[cur.y.y])
            continue;
        // Mark this node as done
        edges.pb(cur.y);
        done[cur.y.y] = 1;
        len += cur.x;
        // Add all connections of the new node
        for (ii c : G[cur.y.y])
            if (!done[c.x])
                q.insert({c.y, {cur.y.y, c.x}});
    }
    return {len, edges};
}
```

Kruskal $\mathcal{O}(E \log V)$

Kruskal sorts the edges from short to long. And walks through the edges, keeping the vertices in groups using a disjoint-set. If two vertices are already in the same group, edges between them will not be included.

```
// Executes Kruskal's algorithm on a graph G,
// represented by adjacency lists. Returns the total
// length of the MST and the edges that make it up.
// WARNING: The graph needs to be undirected and
// connected and thus contain (v1, v2) if it contains
// the edge (v2, v1)!
pair<ll, vii> kruskal(const vector<vii> &G) {
    // Each node is in a different group initially
    DisjointUnion dsu(sz(G));
    // Gather edges: length, (v1, v2)
    vector<pair<ll, ii>> edges;
    REP(i, sz(G)) REP(j, sz(G[i]))
        if (i < G[i][j].x)
            edges.pb({G[i][j].y, {i, G[i][j].x}});
    sort(all(edges));
    vii mst;
    ll len = 0;
    for (auto &e : edges) {
        if (dsu.find(e.y.x) != dsu.find(e.y.y)) {
            mst.pb(e.y);
            dsu.combine(e.y.x, e.y.y);
            len += e.x;
        }
    }
    return {len, mst};
}
```

3.4. Topological sort $\mathcal{O}(V + E)$

A topological ordering on a directed graph $G = (V, E)$ is an ordering $(v_1, \ldots, v_n)$ of the vertices of the graph such that for any $(v_i, v_j) \in E$ we have $i < j$. A topological ordering exists if and only if G does not contain cycles. A topological ordering can be found using DFS.

```
// Perform depth-first search on a directed graph,
// marking nodes as visited on the way
void dfs(ll i, vi &out, const vvi &Gr,
vector<bool> &done) {
    done[i] = true;
    // Visit ancestor nodes first
    for (ll e : Gr[i]) if (!done[e])
        dfs(e, out, Gr, done);
    out.pb(i);
}

// Finds a topological ordering on a graph G. This
// function assumes that such an ordering exists!
vi toposort(const vvi &G) {
    // Construct reverse graph
    vvi Gr(sz(G));
    REP(i, sz(G)) for (ll e : G[i])
        Gr[e].pb(i);
    // Run DFS for each node without descendents
```

```
    vi out; vector<bool> done(sz(G), false);
    REP(i, sz(G)) if (G[i].empty() && !done[i])
        dfs(i, out, Gr, done);
    return out;
}
```

3.5. Cycle finding $\mathcal{O}(V+E)$

To check if an undirected graph has a cycle: Use a DSU to find number of connected components C and check if $E + C > V$. To find the cycle, use DFS as below. For directed graphs, remove the marked if-statement below.

```
// Returns {} if no cycle is found
vi findCycle(const vvi &G) {
    vector<bool> done(sz(G), false);
    vector<bool> in_path(sz(G), false); vi path;
    // DFS is used to detect path looping back on
    // itself
    function<bool(ll, ll)> dfs = [&](ll i, ll prev) {
        if (done[i])
            return false;
        path.pb(i); in_path[i] = done[i] = true;
        // NOTE: Remove if-statement for directed graph
        for (ll e : G[i]) if (e != prev) {
            if (!done[e] && dfs(e, i))
                return true;
            if (in_path[e]) {
                // Add duplicate node for removal
                path.pb(e);
                return true;
            }
        }
        path.pop_back(); in_path[i] = false;
        return false;
    };
    // For every point not yet visited start a DFS
    REP(i, sz(G)) {
        if (done[i] || !dfs(i, -1))
            continue;
        // Reconstruct path from duplicate node
        ll start = 1;
        while (path[start - 1] != path.back())
            start++;
        return vi(path.begin() + start, path.end());
    }
    return {};
}
```

3.6. Strongly connected components $\mathcal{O}(V+E)$

A directed graph is strongly connected if there exists a path from every node to every other node (so not just in one direction). In an undirected graph, the connected components can be found with a DSU or BFS/DFS. In directed graphs, Kosaraju's algorithm can be used.

```
// The graph G is represented as adjacency lists
vvi components(const vvi &G) {
    // Construct reverse graph (for in-neighbours)
    vvi Gr(sz(G));
    REP(i, sz(G)) for (ll e : G[i])
        Gr[e].pb(i);
    vector<bool> done(sz(G), false);
    vi L;
    function<void(ll)> visit = [&](ll i) {
        if (done[i])
            return;
        done[i] = true;
        for (ll e : G[i])
            visit(e);
        L.pb(i);
    };
    // Component represented by root vertex
    vi comp(sz(G), -1);
    function<void(ll, ll)> assign = [&](ll i, ll root)
    {
        if (comp[i] != -1)
            return;
        comp[i] = root;
        for (ll e : Gr[i])
            assign(e, root);
    };
    REP(i, sz(G))
        visit(i);
    reverse(all(L));
    REP(i, sz(L))
        assign(L[i], L[i]);
    // Construct component lists
    vvi compList(sz(G));
    REP(i, sz(G))
        compList[comp[i]].pb(i);
    // Remove empty components
    vvi out;
    for (const vi &l : compList)
        if (!l.empty())
            out.pb(l);
    return out;
}
```

3.7. Eulerian cycles

Existence in an undirected graph $\mathcal{O}(V+E)$

An undirected graph contains an Eulerian cycle if and only if it is connected and all vertices have even degrees.

Existence in a directed graph $\mathcal{O}(V+E)$

An undirected graph contains an Eulerian cycle if and only if it is strongly connected and all vertices have an equal in and out-degree.

Finding $\mathcal{O}(E \log E)$

An Eulerian cycle (if it exists) can be found by using Hierholzer's algorithm. This algorithm first generates some cycle from the starting vertex and then keeps adding cycles at the vertices in this cycle that have unused edges. If an Eulerian cycle exists, it is impossible to get stuck.

```
// Determines a Eulerian cycle in a graph G, where G is
// given by adjacency lists. Input is the starting
// vertex of the cycle. This function assumes that a
// Eulerian cycle exists
vi eulerCycle(const vvi &G, ll start = 0) {
    // Store edges in a set for quick removal
    set<ii> edges;
    REP(i, sz(G)) for (ll e : G[i])
        edges.insert({i, e});
    // Number of edges from each vertex
    vi deg;
    REP(i, sz(G))
        deg.pb(sz(G[i]));
    // Current path and final result
    vi path, res;
    path.pb(0);
    ll cur = start;
    while (!path.empty()) {
        // Continue current circuit of backtrack
        if (deg[cur] > 0) {
            path.pb(cur);
            ll nxt = edges.lower_bound({cur, 0})->y;
            edges.erase({cur, nxt}), deg[cur]--;
            // For directed graphs, remove this line:
            edges.erase({nxt, cur}), deg[nxt]--;
            cur = nxt;
        } else {
            res.pb(cur);
            cur = path.back();
            path.pop_back();
```

```
        }
    }
    reverse(all(res));
    return res;
}
```

3.8. Max-flow $\mathcal{O}(E^2V)$ / $\mathcal{O}(EV^2)$

The max-flow problem asks for the maximum flow through a directed network (graph) given the capacities of its edges. It can be thought of as a network of water pipes for example.

The problem can be solved using the Edmonds-Karp variant of the Ford-Fulkerson algorithm or Dinic's algorithm.

Edmonds-Karp algorithm $\mathcal{O}(E^2V)$

```
// Checks if there is a path from s to t in the
// residual graph given by its edges. Uses prev to
// store path
bool bfs(const vvi &Gr, const vvi &edges, ll s, ll t,
vi &prev) {
    // Keep track of done vertices
    vi done(sz(prev), 0);
    // Add source vertex to queue
    queue<ll> q;
    q.push(s), done[s] = 1, prev[s] = -1;
    // BFS, consider edges with weight >0
    while (!q.empty()) {
        ll cur = q.front();
        q.pop();
        for (ll v : edges[cur]) {
            if (!done[v] && Gr[cur][v] > 0) {
                prev[v] = cur;
                if (v == t)
                    return true;
                q.push(v), done[v] = 1;
            }
        }
    }
    return false;
}

// Returns maximum flow from source s to sink t in
// graph G and the residual graph as an adjacency
// matrix. The graph should also be given as an
// adjacency matrix
pair<ll, vvi> maxFlow(const vvi &G, ll s, ll t) {
    vvi Gr = G;
    vi prev(sz(G), 0);
    ll mx = 0;
    // Also store original graph as adjacency list
    vvi edges(sz(G));
    REP(i, sz(G)) REP(j, sz(G))
        if (G[i][j] != 0)
            edges[i].pb(j), edges[j].pb(i);
    // Augment flow while there is a path from source
    // to sink
    while (bfs(Gr, edges, s, t, prev)) {
        // Find minimum residual capacity of edges
        // along the path found by BFS
        ll flow = LLONG_MAX, v;
        for (v = t; v != s; v = prev[v])
            flow = min(flow, Gr[prev[v]][v]);
        // Update residual capacities of the edges and
        // reverse edges along the path
        for (v = t; v != s; v = prev[v]) {
            Gr[prev[v]][v] -= flow;
            Gr[v][prev[v]] += flow;
        }
        mx += flow;
    }
    return {mx, Gr};
}
```

Dinic's algorithm $\mathcal{O}(EV^2)$

If the network is a unit network, meaning every node except s and t has either a unique incoming or a unique outgoing edge, and this unique edge has capacity 1, then the complexity is reduced to $\mathcal{O}(E\sqrt{V})$.

```
// Special pair hashing
struct DinicHash {
    hash<ll> h;
    size_t operator()(const ii &p) const {
        return h(p.x + (p.y << 31));
    }
};
// Object to run Dinic's algorithm
struct Dinic {
    // Size of the graph
    ll n;
    // Adjacency lists, with both forward and backward
    // edges included
    vvi adj;
    // Total and residual capacity
    unordered_map<ii, ll, DinicHash> cap, res;
    // Current level of each node
    vi level;
    // Source and sink nodes
    ll s, t;
    // Visited nodes by DFS in `findPath` function, and
    // pruned nodes is DFS that lead to dead ends
    vector<bool> visited, pruned;
    // Input the graph as an adjacency list
    Dinic(const vector<vii> &G, ll s, ll t) : n(sz(G)),
    adj(n), s(s), t(t), visited(n), pruned(n) {
        // For performance: determine approx. number of
        // entries in hash tables
        ll tot = 0;
        REP(i, n) tot += sz(G[i]);
        cap.reserve(tot * 2); res.reserve(tot * 2);
        // Construct adjacency lists, set capacities
        REP(i, n)
            for (const ii &e : G[i]) {
                if (cap.find({i, e.x}) == cap.end()) {
                    cap[{i, e.x}] = cap[{e.x, i}] =
                    res[{i, e.x}] = res[{e.x, i}] = 0;
                    adj[i].pb(e.x), adj[e.x].pb(i);
                }
                cap[{i, e.x}] += e.y;
            }
    }
    // Find the levels of all nodes, given residual
    // graph
    void findLevels() {
        level = vi(n, -1);
        queue<ll> q; q.push(s); level[s] = 0;
        while (!q.empty()) {
            ll i = q.front(); q.pop();
            for (ll e : adj[i])
                if (cap[{i, e}] > res[{i, e}] &&
                level[e] == -1)
                    level[e] = level[i] + 1, q.push(e);
        }
    }
    // Checks if an edge is in the current level graph
    bool inLevel(const ii &e) {
        return cap[e] > res[e] && level[e.x] <
        level[e.y];
    }
    // Find a blocking path using DFS, returns max-flow
    // of path if it exists, otherwise 0
    ll findPath(ll i, vi &path, ll flow) {
        visited[i] = true;
        path.pb(i);
        if (i == t)
            return flow;
        for (ll e : adj[i]) {
            if (inLevel({i, e}) && !visited[e] &&
```

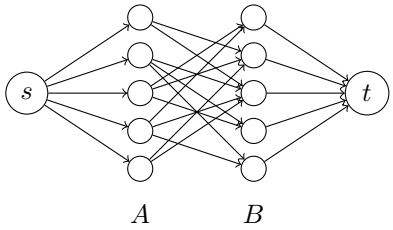
```
            !pruned[e]) {
                ll r = findPath(e, path, min(flow,
                cap[{i, e}] - res[{i, e}]));
                if (r > 0)
                    return r;
            }
        }
        pruned[i] = true;
        path.pop_back();
        return 0;
    }
    // Find the maximum flow of the graph. NOTE: This
    // function can only be called once!
    ll maxFlow() {
        ll flow = 0;
        while (true) {
            findLevels();
            if (level[t] == -1)
                break;
            vi path; ll r;
            fill(all(pruned), false);
            fill(all(visited), false);
            while ((r = findPath(s, path, LLONG_MAX)))
            {
                REP(i, sz(path) - 1) {
                    res[{path[i], path[i + 1]}] += r;
                    res[{path[i + 1], path[i]}] -= r;
                }
                flow += r;
                path.clear();
                fill(all(visited), false);
            }
        }
        return flow;
    }
};
```

Max-flow min-cut theorem

Given a network, the capacity of the minimum cut (which has one partition with s and one with t) is the same as the maximum flow in the network.

3.9. Maximum bipartite matching $\mathcal{O}(E\sqrt{V})$

The maximum matching (maximum number of vertex pairs that have an edge between them) in a bipartite graph with parts A and B can be found using Dinic's algorithm (see above), by setting all capacities to 1.



3.10. Assignment problem $\mathcal{O}(V^3)$

The Hungarian algorithm solves the assignment problem: Given a complete bipartite graph with parts S and T, and a cost c_{ij} for each $i \in S$ and $j \in T$, find a perfect matching (all vertices are in a match) with minimal summed cost.

```
const ll INF = 1LL << 32;

// Find the minimum cost matching given a cost matrix,
// returns the total cost and matching pairs
pair<ll, vii> hungarian(const vvi &A) {
    ll n = sz(A), m = sz(A[0]);
    vi u(n + 1), v(m + 1), p(m + 1), way(m + 1);
    rep(i, 1, n + 1) {
        p[0] = i;
        ll j0 = 0;
        vi minv(m + 1, INF), used(m + 1, 0);
        while (p[j0] != 0) {
            used[j0] = 1;
            ll i0 = p[j0], delta = INF, j1;
            rep(j, 1, m + 1) {
                if (!used[j]) {
                    ll cur = A[i0 - 1][j - 1] - u[i0]
                    - v[j];
                    if (cur < minv[j])
                        minv[j] = cur, way[j] = j0;
                    if (minv[j] < delta)
                        delta = minv[j], j1 = j;
                }
            }
            REP(j, m + 1) {
                if (used[j])
                    u[p[j]] += delta, v[j] -= delta;
                else
                    minv[j] -= delta;
            }
            j0 = j1;
        }
        while (j0) {
            ll j1 = way[j0];
            p[j0] = p[j1], j0 = j1;
        }
    }
    vii res;
    rep(i, 1, m + 1)
        res.pb({p[i] - 1, i - 1});
    return {-v[0], res};
}
```

3.11. Maximum independent set of a tree $\mathcal{O}(E)$

A maximum independent set in a graph is a set of vertices of maximum size such that no two vertices share an edge. In a tree, the maximum independent set can be determined using DFS, by determining the maximum independent set with both the current node included and not included.

```
// The tree should be given as a DIRECTED graph, with
// node 0 the root, the graph is given as an adjacency
// list
vi maxIndep(const vvi &G) {
    // Create list in the right order (children first)
    vi nodes;
    queue<ll> q; q.push(0);
    while (!q.empty()) {
        ll c = q.front();
        q.pop();
        nodes.pb(c);
        for (ll e : G[c])
            q.push(e);
    }
    reverse(all(nodes));
    // Max indep set size given that current node is/is
    // not included
    vi A(sz(G), 0), B(sz(G), 0);
    for (ll c : nodes) {
        A[c] = 1;
        // A[c] = sum(B[e] for e children)
        for (ll e : G[c])
            A[c] += B[e];
        // B[c] = sum(max(A[e], B[e]) for e children)
        for (ll e : G[c]) {
            if (A[e] > B[e])
                B[c] += A[e];
            else
                B[c] += B[e];
        }
    }
    // Construct max indep set
    // queue: (node, can be included)
```

```
    queue<ii> v; v.push({0, 1});
    vi res;
    while (!v.empty()) {
        ii c = v.front();
        v.pop();
        // Include or do not include
        if (c.y && A[c.x] > B[c.x]) {
            res.pb(c.x);
            for (ll e : G[c.x])
                v.push({e, 0});
        } else {
            for (ll e : G[c.x])
                v.push({e, 1});
        }
    }
    return res;
}
```

3.12. Graph coloring

A coloring of a graph is an assignment of colors to the vertices such that two vertices connected by an edge have a different color. The problem is generally NP-hard.

Chromatic polynomial

The chromatic polynomial represents the number of ways a graph can be colored given a maximum number of colors x. Below are some common graph types:

Graph	$\mathcal{P}(x)$
Complete K_n	$x(x-1)(x-2)\dots(x-(n-1))$
Edgeless $\overline{K}_n$	x^n
Tree with n vertices	$x(x-1)^{n-1}$
Cycle C_n	$(x-1)^n+(-1)^n(x-1)$
Path P_n	$x(x-1)^{n-1}$
Star S_n	$x(x-1)^{n-1}$
Ladder $P_2 \times P_n$	$x(x-1)(x^2-3x+3)^{n-1}$

Chromatic number

The chromatic number $\chi(G)$ is the minimum number of colors needed to color the graph G, it has the following properties:

$$\chi(G) := \min\{k \in \mathbb{N} \mid \mathcal{P}(k) > 0\},$$
$$\chi(G)(\xi(G)-1) \leq 2|E|,$$
$$\chi(G) \geq \omega(G),$$
$$\chi(G) \leq 1 + \max_{v \in V} d(v),$$

where $\omega(G)$ is the size of the size of the largest clique (complete subgraph).

2-coloring $\mathcal{O}(V+E)$

A graph can be colored with 2 colors if it is bipartite. This can be found out using BFS.

```
// Check if a graph is bipartite. The graph is given as
// adjacency lists and must be undirected
bool isBipartite(const vvi &G) {
    // Keep track of group of vertices
    vi group(sz(G), -1);
    // Iterate over connected components
    REP(i, sz(G)) {
        if (group[i] != -1)
            continue;
        // Go over vertices in connected component
        // using BFS
        queue<ll> q; q.push(i); group[i] = 0;
        while (!q.empty()) {
            ll c = q.front();
            q.pop();
            for (ll e : G[c]) {
                if (group[c] == group[e])
                    return false;
                if (group[e] != -1)
                    continue;
                group[e] = 1 - group[c];
                q.push(e);
            }
        }
    }
    return true;
}
```

4. Search algorithms

The following are in the C++ STL:

- `find(begin, end, val)`: Returns iterator to found element (linear search).
- `find_if(begin, end, pred)`: Returns iterator for element that returns `true` for given predicate. For example `find_if(all(vec), [] (ll v) { return v % 2 == 0 })` returns the first even element.

4.1. Binary search $\mathcal{O}(\log(max-min))$

Find the lowest number x in an interval $[s,t]$ such that $f(x)$ holds. Assumes that for any $x \in [s,t]$ with $f(x)$ true, $f(y)$ holds for all $y \geq x$.

Binary search on a sorted vector in C++ can be done using `binary_search` (will check if element exists).

```
// Perform binary search on interval [start, end]
// f should evaluate to false for lower values and
// to true for higher values. Returns the lowest value
// for which f is true
ll binarySearch(ll start, ll end) {
    while (start < end) {
        ll mid = (start + end) / 2;
        if (f(mid))
            end = mid;
        else
            start = mid + 1;
    }
    return start;
}
```

4.2. Ternary search $\mathcal{O}(\log(max-min))$

Finds the maximum M of a function that is monotonically increasing for values lower than M and monotonically decreasing for values higher than M. This can be used as a replacement for determining the point where the derivative is zero, for example in the case of a parabola.

```
// Perform ternary search on interval [start, end]
// This will find the maximum M of a function f
ll ternarySearch(ll start, ll end) {
    while (start < end) {
        ll l = start + (end - start) / 3;
        ll r = end - (end - start) / 3;
        if (f(l) < f(r))
            start = l;
        else
            end = r;
    }
    return start;
}
```

4.3. Interpolation search $\mathcal{O}(\log(\log n))$

This algorithm is faster than binary search if the data in a list is uniformly distributed (otherwise it is $\mathcal{O}(\log n)$). The algo-

rithm guesses a position of the element based on the current lower and upper bounds.

```
// Returns the index of the found element, or -1 if it
// wasn't found
ll interpolationSearch(const vi &a, ll x) {
    ll start = 0, end = sz(a) - 1;
    while (start <= end && a[start] <= x &&
    x <= a[end]) {
        if (start == end)
            return a[start] == x ? start : -1;
        ll pos = start + (end - start) * (x - a[start])
            / (a[end] - a[start]);
        if (a[pos] == x)
            return pos;
        if (a[pos] < x)
            start = pos + 1;
        else
            end = pos - 1;
    }
    return -1;
}
```

5. Sort algorithms

A version of Quicksort (combined with linear sort) is implemented in C++, use `sort(all(list), lt)`. Or use `stable_sort(all(list), lt)` for stable sorting.

5.1. Quicksort $\mathcal{O}(n \log n)$

In case more complicated mechanics need to be implemented, the following template can be used:

```
// Partition part a list using the last element as a
// pivot. All lower elements will be left to the pivot
// and higher elements will be to the right. The
// (final) index of the pivot if returned.
ll partition(vi &v, ll start, ll end) {
    ll pivot = v[end];
    ll i = start - 1;
    rep(j, start, end)
        if (v[j] < pivot)
            i++, swap(v[i], v[j]);
    swap(v[i + 1], v[end]);
    return i + 1;
}

// Sort a list by recursively partitioning
void quickSort(vi &v, ll start = 0, ll end = -1) {
    if (end == -1)
        end = sz(v) - 1;
    if (start >= end)
        return;
    ll pi = partition(v, start, end);
    quickSort(v, start, pi - 1);
    quickSort(v, pi + 1, end);
}
```

5.2. Counting sort $\mathcal{O}(max - min + n)$

Can be used when the range of values in the list is very small, searches roughly in linear time. Most of the time, Quicksort (STL) is better.

```
// Sort by the number of occurences of each possible
// number in the list. Can also be used with a set if
// bounds are unknown, but likely quicksort is faster.
// Parameters are the list, the min and the max.
void countSort(vi &list, ll m, ll M) {
    vi occ(M - m + 1, 0);
    // Counts number of occurences
    for (ll &i : list)
        occ[i - m]++;
    // Remake list in correct order
    ll c = 0, i = 0;
    vi res;
    while (i < sz(occ)) {
        res.pb(i + m);
        c++;
        if (c >= occ[i])
            c = 0, i++;
    }
    list = res;
}
```

5.3. Random sort $\mathcal{O}(n \log n)$

```
// Randomize the order of a list
// Note that this implementation is not perfect and
// should not be used for security purposes
template<class T>
void randomSort(vector<T> &list) {
    static auto rng = default_random_engine(
    time(nullptr));
    shuffle(all(list), rng);
}
```

5.4. Count inversions $\mathcal{O}(n \log n)$

This algorithm counts the number of inversions in a list using merge-sort. An inversion is a pair (x_i, x_j) in a list $(x_1, x_2, \ldots, x_n)$ where $i < j$ and $x_i > x_j$.

```
ll invMerge(vi &A, ll lo, ll mid, ll hi) {
    ll i = lo, j = mid + 1, k = 0, r = 0;
    vi B(hi - lo + 1, 0);
    while (i <= mid && j <= hi) {
        if (A[i] <= A[j])
            B[k++] = A[i++];
        else
            B[k++] = A[j++], r += (mid + 1) - i;
    }
    while (i <= mid) B[k++] = A[i++];
    while (j <= hi) B[k++] = A[j++];
    rep(x,lo,hi + 1) A[x] = B[x - lo];
    return r;
}

// Also sorts the array (using mergesort)
ll inversions(vi &A, ll lo = 0, ll hi = -1) {
    if (hi == -1) hi = sz(A) - 1;
    ll r = 0;
    if (lo < hi) {
        ll mid = (lo + hi) / 2;
        r += inversions(A, lo, mid);
        r += inversions(A, mid + 1, hi);
        r += invMerge(A, lo, mid, hi);
    }
    return r;
}
```

6. String algorithms

In C++ some built-in functions on strings are:

- `s.find(it, pos = 0)`, does not use KMP. $\mathcal{O}(mn)$
- `s.substr(start, end = SIZE_MAX)`. $\mathcal{O}(end - start)$

6.1. Knuth-Morris-Pratt $\mathcal{O}(S + P)$

This algorithm determines the position of a pattern $P = p_1 p_2 \ldots p_m$ in a string $S = s_1 s_2 \ldots s_n$, where $n > m$. It does this by first preprocessing the pattern and then matching the string in constant time. In case there are multiple strings where the same pattern needs to be matched, the preprocessing step only has to be done once.

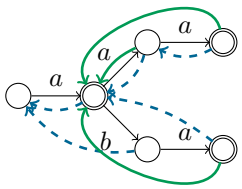
```
// Calculate the preprocessing pattern table for KMP
// This table works for finding a pattern in any string
vi kmpTable(const string &pat) {
    vi T(sz(pat) + 1, -1);
```

```
    ll pos = 1, cnd = 0;
    while (pos < sz(pat)) {
        if (pat[pos] == pat[cnd]) {
            T[pos] = T[cnd];
        } else {
            T[pos] = cnd;
            while (cnd >= 0 && pat[pos] != pat[cnd])
                cnd = T[cnd];
        }
        pos++, cnd++;
    }
    T[pos] = cnd;
    return T;
}

// Returns the first position at which the pattern
// occurs, or -1 if the pattern does not occur
// Function can be modified to return a list of indices
ll kmp(const string &str, const string &pat) {
    ll j = 0, k = 0;
    vi T = kmpTable(pat);
    while (j < sz(str)) {
        if (pat[k] == str[j]) {
            j++, k++;
            if (k == sz(pat)) {
                return j - k;
                // In case we need multiple occurences:
                k = T[k];
            }
        } else {
            k = T[k];
            if (k < 0)
                j++, k++;
        }
    }
    return -1;
}
```

6.2. Aho-Corasick $\mathcal{O}(\sum_i S_i + \sum_i P_i + M)$

Like KMP, it is used for finding patterns in strings. However, all patterns are processed in a batch, which is more cost-efficient than matching using KMP for every pattern. It constructs an automaton on 3 steps: Construct a trie, add blue edges, and add green edges. The blue edges point to the longest strict suffix of the current word in the trie. The green edges point to the longest *accepted* suffix in the tree (may not exist). The complexity is the sum of the total string length, total pattern length, and the number of matches. The number of matches can be removed from the complexity if only the longest suffix is returned for each index.




```
// NOTE: Duplicate patterns will be ignored!
// NOTE: Doesn't allow for adding patterns after
// construction!
struct AhoCorasick {
    // Base character offset and character range
    const char base; const ll range;
    // Standard trie links, and (parent node, char
    // index)
    vvi trie; vii parent;
    // Blue and green edges, accepted word index or -1
    vi blue, green; vi accepted;
    AhoCorasick(const vector<string> &pat, char base,
    char end) : base(base), range(ll(end) - base + 1) {
        addNode(); // Add root node
        createTrie(pat); createBlue(); createGreen();
    }
    // Add empty node to the automaton and return index
    ll addNode() {
        accepted.pb(-1); trie.pb(vi(range, -1));
        blue.pb(-1); green.pb(-1); parent.pb({-1, -1});
        return sz(blue) - 1;
    }
    // Construct trie and mark accepted patterns
    void createTrie(const vector<string> &pat) {
        REP(i, sz(pat)) {
            ll j = 0;
            for (char c : pat[i]) {
                if (trie[j][c - base] == -1) {
                    ll v = addNode();
                    parent[v] = {j, c - base};
                    trie[j][c - base] = v;
                }
                j = trie[j][c - base];
            }
            accepted[j] = i;
        }
    }
    // Create blue edges to longest strict suffix
    // Uses BFS and blue edges of parent
    void createBlue() {
        queue<ll> q; q.push(0);
        while (!q.empty()) {
            ll cur = q.front(); q.pop();
            REP(i, range)
                if (trie[cur][i] != -1)
                    q.push(trie[cur][i]);
            if (parent[cur].x <= 0) {
                blue[cur] = 0;
                continue;
            }
            // Char that points at current node
            ll c = parent[cur].y;
            // Follow blue edge of parent from current
            // node
            ll p = blue[parent[cur].x];
            while (p > 0 && trie[p][c] == -1)
                p = blue[p];
            blue[cur] = trie[p][c] == -1 ? 0 :
            trie[p][c];
        }
    }
    // Create green edges to longest accepted suffix
    // Uses BFS and blue edges
    void createGreen() {
        queue<ll> q; q.push(0);
        while (!q.empty()) {
            ll cur = q.front(); q.pop();
            REP(i, range)
                if (trie[cur][i] != -1)
                    q.push(trie[cur][i]);
            if (blue[cur] == -1)
                continue;
            if (accepted[blue[cur]] != -1)
                green[cur] = blue[cur];
            else
                green[cur] = green[blue[cur]];
        }
    }
    // Returns a list of (end index, pat. index)
    vii match(const string &s) {
        ll cur = 0; vii out;
        REP(i, sz(s)) {
            ll c = ll(s[i]) - base;
            // Move to next node
            while (cur > 0 && trie[cur][c] == -1)
                cur = blue[cur];
            if (trie[cur][c] == -1)
                cur = 0;
```

```
            else
                cur = trie[cur][c];
            // NOTE: Option to return only longest
            // match per index
            #if ONLY_LONGEST_SUFFIX
                if (accepted[cur] != -1)
                    out.pb({i, accepted[cur]});
                else if (green[cur] != -1)
                    out.pb({i, accepted[green[cur]]});
            #else
                if (accepted[cur] != -1)
                    out.pb({i, accepted[cur]});
                ll j = cur;
                while (green[j] != -1) {
                    // green[j] points to next accepted
                    // suffix
                    j = green[j];
                    out.pb({i, accepted[j]});
                }
            #endif
        }
        return out;
    }
};
```

6.3. Edit distance $\mathcal{O}(S_1S_2)$

Given two strings, the edit distance is the minimum amount of operations needed to transform one string into another, with the operations insert character, remove character, replace character. The implementation uses DP on first m characters of S_1 and n characters of S_2.

```
ll editDistance(const string &src, const string &dst) {
    vvi dp(sz(src) + 1, vi(sz(dst) + 1, 0));
    REP(i, sz(src) + 1) REP(j, sz(dst) + 1) {
        if (i == 0)
            dp[i][j] = j;
        else if (j == 0)
            dp[i][j] = i;
        else if (src[i - 1] == dst[j - 1])
            dp[i][j] = dp[i - 1][j - 1];
        else
            dp[i][j] = 1 + min(dp[i][j - 1], min(
            dp[i - 1][j], dp[i - 1][j - 1]));
    }
    return dp[sz(src)][sz(dst)];
}
```

6.4. Longest common subsequence $\mathcal{O}(S_1S_2)$

```
// Return the longest common subsequence in two strings
// Returns just one if there are multiple
string longestCommon(const string &A, const string &B)
{
    // [first i of A][first j of B] = (max, last chars
    // in A and B)
    vector<vector<pair<ll, ii>>> dp(sz(A) + 1,
    vector<pair<ll, ii>>(sz(B) + 1));
    REP(i, sz(A) + 1) REP(j, sz(B) + 1) {
        dp[i][j] = {0, {-1, -1}};
        if (i == 0 || j == 0)
            continue;
        if (dp[i - 1][j].x > dp[i][j].x)
            dp[i][j] = dp[i - 1][j];
        if (dp[i][j - 1].x > dp[i][j].x)
            dp[i][j] = dp[i][j - 1];
        if (dp[i - 1][j - 1].x + 1 > dp[i][j].x &&
        A[i - 1] == B[j - 1])
            dp[i][j] = {dp[i - 1][j - 1].x + 1,
            {i - 1, j - 1}};
    }
    // Reconstruct string from "last char" entries
    string rec = "";
    ll i = sz(A), j = sz(B);
    while (i >= 0 && j >= 0) {
        pair<ll, ii> entry = dp[i][j];
        rec.pb(A[entry.y.x]);
        i = entry.y.x, j = entry.y.y;
    }
    reverse(all(rec));
    return rec;
}
```

7. Miscellaneous

7.1. Knapsack problem $\mathcal{O}(Wn)$

```
// Returns the maximum value (v) that can be put in a
// knapsack of capacity W with weights (wt)
ll knapsack(int W, vi &wt, vi &v) {
    ll n = sz(wt);
    vvi K(n + 1, vi(W + 1));
    REP(i, n + 1) REP(w, W + 1) {
        if (i == 0 || w == 0)
            K[i][w] = 0;
        else if (wt[i - 1] <= w)
            K[i][w] = max(v[i - 1]
            + K[i - 1][w - wt[i - 1]],
            K[i - 1][w]);
        else
            K[i][w] = K[i - 1][w];
    }
    return K[n][W];
}
```

7.2. Hashing

Default C++ structures often support hashing by the default, such as `string`, `long long`, etc. These are automatically used in `unordered_map` and `unordered_set` to give $\mathcal{O}(1)$ lookup times.

Rolling hash function

You can make it so that the hash of an object can be determined based on its individual elements (such as in a vector). For example a string $s_1s_2\ldots s_n$ can be hashed with

$$H(s_1s_2\ldots s_n) = \sum_{k=1}^{n} s_k p^{k-1} \mod M,$$

for some small prime p and large prime M (see "Useful numbers"). This then gives the relations

$$H(s_1s_2\ldots s_ns_{n+1}) \equiv H(s_1s_2\ldots s_n) + s_{n+1}p^n,$$
$$H(s_0s_1\ldots s_n) \equiv s_0 + pH(s_1s_2\ldots s_n),$$
$$H(s_2s_3\ldots s_n) \equiv p^{-1}\left(H(s_1s_2\ldots s_n) - s_1\right),$$
$$H(s_1s_2\ldots s_{n-1}) \equiv H(s_1s_2\ldots s_n) - s_np^{n-1}.$$

```
// Cached prime powers
static vi ppow = {1};
// This can be used to keep track of the rolling hash
// of a string. Access the hash using H. If you want to
// keep track of the contained string, use a deque
struct RollingHash {
    // ip is the inverse of p mod M. Different primes
    // can be found earlier in the reference!
    static const ll p = 8125343,
    ip = 34789068517540942, M = 36028797018963971;
    // Keep track of hash and string size
    ll H, size;
    RollingHash() : H(0), size(0) { }
    // Update cached prime powers
    void pup() {
        while (size >= sz(ppow))
            ppow.pb(modMul(ppow.back(), p, M));
    }
    void norm(ll &c) { c %= M; if (c < 0) c += M; }
    // Append to end of string
    void push_back(ll c) {
```

```
        norm(c);
        H += modMul(c, ppow[size], M); norm(H);
        size++; pup();
    }
    // Remove front end of string
    void pop_back(ll c) {
        norm(c);
        H -= modMul(c, ppow[size - 1], M); norm(H);
        size--;
    }
    // Add to front of string
    void push_front(ll c) {
        norm(c);
        H = c + modMul(p, H, M); norm(H);
        size++; pup();
    }
    // Remove from front of string
    void pop_front(ll c) {
        H = modMul(ip, H - c, M);
        size--;
    }
};
```

Custom hash functions

C++ doesn't have hash functions for vectors and pairs. Reference implementations are added below. These are very slow though! For pairs with low integer values, use `h(p.x + p.y << 31)` as hash instead.

```
namespace std {
    template<class T>
    struct hash<vector<T>> {
        hash<T> ht; hash<size_t> h;
        size_t operator()(const vector<T> &v) const {
            size_t out = 0;
            for (const T &i : v)
                out = h(out ^ ht(i));
            return out;
        }
    };
    template<class T, class S>
    struct hash<pair<T, S>> {
        hash<T> ht; hash<S> hs; hash<size_t> h;
        size_t operator()(const pair<T, S> &p) const {
            return h(ht(p.x) ^ h(hs(p.y)));
        }
    };
}
```

7.3. Gray code

Gray code is a way of ordering binary numbers, such that two consecutive numbers differ in exactly one bit. Iterating over binary numbers in this way can be used for optimizations when using bitmasks.

```
ll gray(ll n) {
    return n ^ (n >> 1);
}

ll grayInverse(ll g) {
    ll n = 0;
    while (g != 0)
        n ^= g, g >>= 1;
    return n;
}
```

7.4. Interval cover $\mathcal{O}(n \log n)$

```
// Check if a collection of intervals covers a specific
// interval. WARNING: Modifies the input
bool intervalCover(vii &vals, ii val) {
    sort(all(vals));
    ll mx = val.x;
    for (ii v : vals) {
        if (v.y <= val.x)
            continue;
        if (v.x > mx)
            return false;
        mx = max(v.y, mx);
    }
    return mx >= val.y;
}
```

7.5. Stable matching problem $\mathcal{O}(n^2)$

The stable matching problem is to find a stable matching given two lists of objects of size n. Each object in both lists has a priority list for the objects in the other list. A matching is *un*stable if both of the following occur:

1. There is an element A of the first list, and an element B of the second list, such that A prefers B over the current assignment, and
2. B prefers A over its assigned element.

A stable matching can always be found, and can be found using the Gale-Shapley algorithm. The algorithm finds the stable matching that is best for the first group and worst for the second group.

```
// Find a stable matching that is best for group A and
// worst for group B. The preference lists are given
// as a list of indices, from most to least preferred.
// It returns a list of indices, the item from group
// A that each item from group B is matched to
vi stableMatching(const vvi &prefA, const vvi &prefB) {
    ll n = sz(prefA);
    // All items are marked as free
    vi freeA(n, 1), freeB(n, 1); ll matched = 0;
    vi match(n);
    // Assign each item from A to an item from B
    while (matched < n) {
        // Pick first free item from list A
        ll a = 0;
        while (!freeA[a]) a++;
        // Go over items from B
        for (ll i = 0; i < n && freeA[a]; i++) {
            ll b = prefA[a][i];
            // If free: assign, otherwise check if it
            // is a better alternative to current
            // assignment
            if (freeB[b]) {
                freeA[a] = freeB[b] = 0;
                match[b] = a;
                matched++;
            } else {
                ll old = match[b];
                // Check if a is preferred over old
                bool pref = false;
                REP(i, n) {
                    if (prefB[b][i] == a) {
                        pref = true; break;
                    } else if (prefB[b][i] == old) {
                        break;
                    }
                }
                if (pref) {
                    match[b] = a;
                    freeA[a] = 0, freeA[old] = 1;
                }
            }
        }
    }
    return match;
}
```

7.6. Subset sum problem $\mathcal{O}(n(max - min))$

The subset sum problem is to determine if there is a subset $\tilde{X}$ of a multiset of integers X, such that $\sum \tilde{X} = T$. The problem is NP-hard when the possible sums are unbounded. It can be solved by keeping track of all possible subset sums of the first k elements. To find the elements that make up the sum, a directed graph can be used with vertices (k, t), which indicates that the first k elements contain a subset sum of t. Then there are edges from (k, t) to $(k + 1, t)$ and $(k + 1, t + x_{k+1})$. A path from $(0, 0)$ to (n, T) can be found with DFS/BFS.

```
// Determines if there is a subset of x that sums up to
// T
bool subsetSum(vi x, ll T) {
    if (T == 0)
        return true;
    // Subset sums of first n items
    vi opt = {0};
    unordered_set<ll> found = {0};
    for (ll v : x) {
        ll s = sz(opt);
        REP(i, s) {
            ll c = opt[i] + v;
            if (c == T)
                return true;
            if (found.find(c) == found.end())
                opt.pb(c), found.insert(c);
        }
    }
    return false;
}
```

7.7. Longest increasing subsequence

```
// Longest increasing subsequence of vector v
ll longestIncreasing(const vi &v) {
    // d[i] = lowest number that subsequence of length
    // i can end in
    vi d(sz(v) + 1, LLONG_MAX);
    d[0] = LLONG_MIN;
    ll r = 0;
    for (ll val : v) {
        ll i = upper_bound(all(d), val) - d.begin();
        // Change to <= for longest non-decreasing
        if (d[i - 1] <= val && val <= d[i]) {
            if (d[i] == LLONG_MAX)
                r++;
            d[i] = val;
        }
    }
    return r;
}
```

7.8. Approximate bounds

Below is a table of approximate upper bounds for n given different algorithm complexities. Keep in mind that constants may also be high, but generally this is not important.

Complexity	Bound
$\mathcal{O}(n!)$	$n \leq 10$
$\mathcal{O}(2^n)$	$n \leq 20$
$\mathcal{O}(n^3)$	$n \leq 500$
$\mathcal{O}(n^2 \log n)$	$n \leq 10^3$
$\mathcal{O}(n^2)$	$n \leq 5 \cdot 10^3$
$\mathcal{O}(n\sqrt{n})$	$n \leq 10^5$
$\mathcal{O}(n \log^2 n)$	$n \leq 10^5$
$\mathcal{O}(n \log n)$	$n \leq 10^6$
$\mathcal{O}(n)$	$n \leq 10^8$
$\mathcal{O}(\sqrt{n})$	$n \leq 10^{15}$
$\mathcal{O}(\log n)$	$n \leq 10^{18}$

8. Pre-competition checklist

Before competition

- What time do we need to wake up?
- Make sure to be on time for test session and competition!
- Can we take snacks with use? Pepsels, Oreo's, etc. ;)
- Remember to bring a bottle of water!
- Do we know the amount of problems there will be?
- Discuss reading order/strategy.
- Hand in reference document, keyboard and mouse at registration.
- How many copies of this document can we use?
- Which IDE is used? Check useful shortcuts. VSCode?
- Which languages are supported? C++? Python?

Test session

- Which feedback is given on incorrect result? Check for no output, wrong answer, timelimit, runtime error, output limit, memory limit, compiler error.
- Check judging features:
  - Is `cerr` considered as output?
  - Difference in time with more/less correct results?
  - Still correct with compiler warnings?
  - Still correct without `return 0;`?
  - Still correct with `exit(0)`?
  - Still correct with memory leaks?
- Which `C++` version is used? Optimization level?

Pink elephant