

## SuiGeneris ICPC document

## Contents

Kushagra, Siddharth, Harshit

<b>1</b>	<b>Combinatorial optimization</b>	<b>1</b>
1.1	Max bipartite machine . . . . .	1
<b>2</b>	<b>Geometry</b>	<b>2</b>
2.1	Convex hull . . . . .	2
2.2	Miscellaneous geometry . . . . .	3
<b>3</b>	<b>Numerical algorithms</b>	<b>7</b>
3.1	Number theory (modular, Chinese remainder, linear Diophantine) . . . . .	7
3.2	Systems of linear equations, matrix inverse, determinant . . . . .	9
3.3	Reduced row echelon form, matrix rank . . . . .	10
3.4	Fast Fourier transform . . . . .	11
<b>4</b>	<b>Graph algorithms</b>	<b>13</b>
4.1	Bellman-Ford shortest paths with negative edge weights (C++) . . . . .	13
4.2	Floyd-Warshall . . . . .	14
4.3	Fast Dijkstra's algorithm . . . . .	14
4.4	Strongly connected components . . . . .	14
4.5	Eulerian path . . . . .	14
4.6	Kruskal's alternative . . . . .	15
4.7	Prim alternative . . . . .	16
4.8	Bridge Finding . . . . .	17
<b>5</b>	<b>Data structures</b>	<b>18</b>
5.1	Suffix array . . . . .	18
5.2	Binary Indexed Tree . . . . .	18
5.3	Union-find set . . . . .	19
5.4	Lazy segment tree . . . . .	19
5.5	Lowest common ancestor . . . . .	21
5.6	Sparse Table . . . . .	22
5.7	Segmented Sieve . . . . .	23
<b>6</b>	<b>Miscellaneous</b>	<b>24</b>
6.1	Vimrc file . . . . .	24

## 1 Combinatorial optimization

## 1.1 Max bipartite machine

```

// This code performs maximum bipartite matching.
//
// Running time:  $O(|E| |V|)$  -- often much faster in practice
//
// INPUT: w[i][j] = edge between row node i and column node j
// OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//          mc[j] = assignment for column node j, -1 if unassigned
//          function returns number of matches made

#include <vector>
using namespace std;
typedef vector<int> VI;
typedef vector<VI> VVI;

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
    for (int j = 0; j < w[i].size(); j++) {
        if (w[i][j] && !seen[j]) {
            seen[j] = true;
            if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
                mr[i] = j;
                mc[j] = i;
                return true;
            }
        }
    }
    return false;
}

```

```

    }
}
return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
    mr = VI(w.size(), -1);
    mc = VI(w[0].size(), -1);

    int ct = 0;
    for (int i = 0; i < w.size(); i++) {
        VI seen(w[0].size());
        if (FindMatch(i, w, mr, mc, seen)) ct++;
    }
    return ct;
}

```

## 2 Geometry

### 2.1 Convex hull

```

#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define ld long double

struct Point {
    ld x, y;

    bool operator < (const Point &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

ld cross(const Point &O, const Point &A, const Point &B)
{
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

vector<Point> convex_hull(vector<Point> P)
{
    int n = P.size();
    int k = 0;
    vector<Point> H(2*n);
    sort(P.begin(), P.end());
    for(int i=0; i<n; i++)
    {
        while(k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }

    for(int i=n-2, t = k+1; i>=0; i--)
    {
        while(k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
        H[k++] = P[i];
    }

    H.resize(k-1);
    return H;
}

int main()
{
    ios_base::sync_with_stdio(false);
    return 0;
}

```

## 2.2 Miscellaneous geometry

```
// C++ routines for computational geometry.
#include <iostream>
#include <vector>
#include <cmath>
#include <cassert>

using namespace std;

double INF = 1e100;
double EPS = 1e-12;

struct PT {
    double x, y;
    PT() {}
    PT(double x, double y) : x(x), y(y) {}
    PT(const PT &p) : x(p.x), y(p.y) {}
    PT operator + (const PT &p) const { return PT(x+p.x, y+p.y); }
    PT operator - (const PT &p) const { return PT(x-p.x, y-p.y); }
    PT operator * (double c) const { return PT(x*c, y*c); }
    PT operator / (double c) const { return PT(x/c, y/c); }
};

double dot(PT p, PT q) { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q) { return dot(p-q, p-q); }
double cross(PT p, PT q) { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
    os << "(" << p.x << ", " << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p) { return PT(-p.y, p.x); }
PT RotateCW90(PT p) { return PT(p.y, -p.x); }
PT RotateCCW(PT p, double t) {
    return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
    return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
PT ProjectPointSegment(PT a, PT b, PT c) {
    double r = dot(b-a, b-a);
    if (fabs(r) < EPS) return a;
    r = dot(c-a, b-a)/r;
    if (r < 0) return a;
    if (r > 1) return b;
    return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
    return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// compute distance between point (x,y,z) and plane ax+by+cz=d
double DistancePointPlane(double x, double y, double z,
                           double a, double b, double c, double d)
{
    return fabs(a*x+b*y+c*z-d)/sqrt(a*a+b*b+c*c);
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
    return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
    return LinesParallel(a, b, c, d)
        && fabs(cross(a-b, a-c)) < EPS;
}
```

```

    && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
    if (LinesCollinear(a, b, c, d)) {
        if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
            dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
        if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
            return false;
        return true;
    }
    if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
    if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
    return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
    b=b-a; d=c-d; c=c-a;
    assert(dot(b, b) > EPS && dot(d, d) > EPS);
    return a + b*cross(c, d)/cross(b, d);
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
    b=(a+b)/2;
    c=(a+c)/2;
    return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
    bool c = 0;
    for (int i = 0; i < p.size(); i++) {
        int j = (i+1)%p.size();
        if ((p[i].y <= q.y && q.y < p[j].y ||
            p[j].y <= q.y && q.y < p[i].y) &&
            q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
            c = !c;
    }
    return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
    for (int i = 0; i < p.size(); i++)
        if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
            return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
    vector<PT> ret;
    b = b-a;
    a = a-c;
    double A = dot(b, b);
    double B = dot(a, b);
    double C = dot(a, a) - r*r;
    double D = B*B - A*C;
    if (D < -EPS) return ret;
    ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);

```

```

    if (D > EPS)
        ret.push_back(c+a+b*(-B-sqrt(D))/A);
    return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
    vector<PT> ret;
    double d = sqrt(dist2(a, b));
    if (d > r+R || d+min(r, R) < max(r, R)) return ret;
    double x = (d*d-R*R+r*r)/(2*d);
    double y = sqrt(r*r-x*x);
    PT v = (b-a)/d;
    ret.push_back(a+v*x + RotateCCW90(v)*y);
    if (y > 0)
        ret.push_back(a+v*x - RotateCCW90(v)*y);
    return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion. Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
    double area = 0;
    for(int i = 0; i < p.size(); i++) {
        int j = (i+1) % p.size();
        area += p[i].x*p[j].y - p[j].x*p[i].y;
    }
    return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
    return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
    PT c(0,0);
    double scale = 6.0 * ComputeSignedArea(p);
    for (int i = 0; i < p.size(); i++) {
        int j = (i+1) % p.size();
        c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
    }
    return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
    for (int i = 0; i < p.size(); i++) {
        for (int k = i+1; k < p.size(); k++) {
            int j = (i+1) % p.size();
            int l = (k+1) % p.size();
            if (i == l || j == k) continue;
            if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
    return true;
}

int main() {
    // expected: (-5,2)
    cerr << RotateCCW90(PT(2,5)) << endl;

    // expected: (5,-2)
    cerr << RotateCW90(PT(2,5)) << endl;

    // expected: (-5,2)
    cerr << RotateCCW(PT(2,5), M_PI/2) << endl;

    // expected: (5,2)
    cerr << ProjectPointLine(PT(-5,-2), PT(10,4), PT(3,7)) << endl;

    // expected: (5,2) (7.5,3) (2.5,1)

```

```

cerr << ProjectPointSegment(PT(-5,-2), PT(10,4), PT(3,7)) << " "
    << ProjectPointSegment(PT(7.5,3), PT(10,4), PT(3,7)) << " "
    << ProjectPointSegment(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

// expected: 6.78903
cerr << DistancePointPlane(4,-4,3,2,-2,5,-8) << endl;

// expected: 1 0 1
cerr << LinesParallel(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
    << LinesParallel(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
    << LinesParallel(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

// expected: 0 0 1
cerr << LinesCollinear(PT(1,1), PT(3,5), PT(2,1), PT(4,5)) << " "
    << LinesCollinear(PT(1,1), PT(3,5), PT(2,0), PT(4,5)) << " "
    << LinesCollinear(PT(1,1), PT(3,5), PT(5,9), PT(7,13)) << endl;

// expected: 1 1 1 0
cerr << SegmentsIntersect(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << " "
    << SegmentsIntersect(PT(0,0), PT(2,4), PT(4,3), PT(0,5)) << " "
    << SegmentsIntersect(PT(0,0), PT(2,4), PT(2,-1), PT(-2,1)) << " "
    << SegmentsIntersect(PT(0,0), PT(2,4), PT(5,5), PT(1,7)) << endl;

// expected: (1,2)
cerr << ComputeLineIntersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << endl;

// expected: (1,1)
cerr << ComputeCircleCenter(PT(-3,4), PT(6,1), PT(4,5)) << endl;

vector<PT> v;
v.push_back(PT(0,0));
v.push_back(PT(5,0));
v.push_back(PT(5,5));
v.push_back(PT(0,5));

// expected: 1 1 1 0 0
cerr << PointInPolygon(v, PT(2,2)) << " "
    << PointInPolygon(v, PT(2,0)) << " "
    << PointInPolygon(v, PT(0,2)) << " "
    << PointInPolygon(v, PT(5,2)) << " "
    << PointInPolygon(v, PT(2,5)) << endl;

// expected: 0 1 1 1 1
cerr << PointOnPolygon(v, PT(2,2)) << " "
    << PointOnPolygon(v, PT(2,0)) << " "
    << PointOnPolygon(v, PT(0,2)) << " "
    << PointOnPolygon(v, PT(5,2)) << " "
    << PointOnPolygon(v, PT(2,5)) << endl;

// expected: (1,6)
// (5,4) (4,5)
// blank line
// (4,5) (5,4)
// blank line
// (4,5) (5,4)
vector<PT> u = CircleLineIntersection(PT(0,6), PT(2,6), PT(1,1), 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleLineIntersection(PT(0,9), PT(9,0), PT(1,1), 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(10,10), 5, 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(8,8), 5, 5);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 10, sqrt(2.0)/2.0);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
u = CircleCircleIntersection(PT(1,1), PT(4.5,4.5), 5, sqrt(2.0)/2.0);
for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;

// area should be 5.0
// centroid should be (1.1666666, 1.1666666)
PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
vector<PT> p(pa, pa+4);
PT c = ComputeCentroid(p);
cerr << "Area: " << ComputeArea(p) << endl;
cerr << "Centroid: " << c << endl;

```

```

    return 0;
}

```

## 3 Numerical algorithms

### 3.1 Number theory (modular, Chinese remainder, linear Diophantine)

*// This is a collection of useful code for solving problems that involve modular linear equations. Note that all of the algorithms described here work on nonnegative integers.*

```

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
typedef vector<int> VI;
typedef pair<int, int> PII;

// return a % b (positive value)
int mod(int a, int b) {
    return ((a%b) + b) % b;
}

// computes gcd(a,b)
int gcd(int a, int b) {
    while (b) { int t = a%b; a = b; b = t; }
    return a;
}

// computes lcm(a,b)
int lcm(int a, int b) {
    return a / gcd(a, b)*b;
}

// (a^b) mod m via successive squaring
int powermod(int a, int b, int m)
{
    int ret = 1;
    while (b)
    {
        if (b & 1) ret = mod(ret*a, m);
        a = mod(a*a, m);
        b >>= 1;
    }
    return ret;
}

// returns g = gcd(a, b); finds x, y such that d = ax + by
int extended_euclid(int a, int b, int &x, int &y) {
    int xx = y = 0;
    int yy = x = 1;
    while (b) {
        int q = a / b;
        int t = b; b = a%b; a = t;
        t = xx; xx = x - q*xx; x = t;
        t = yy; yy = y - q*yy; y = t;
    }
    return a;
}

// finds all solutions to ax = b (mod n)
VI modular_linear_equation_solver(int a, int b, int n) {
    int x, y;
    VI ret;
    int g = extended_euclid(a, n, x, y);
    if (!(b%g)) {
        x = mod(x*(b / g), n);
        for (int i = 0; i < g; i++)
            ret.push_back(mod(x + i*(n / g), n));
    }
}

```

```

        return ret;
    }

    // computes b such that ab = 1 (mod n), returns -1 on failure
    int mod_inverse(int a, int n) {
        int x, y;
        int g = extended_euclid(a, n, x, y);
        if (g > 1) return -1;
        return mod(x, n);
    }

    // Chinese remainder theorem (special case): find z such that
    // z % m1 = r1, z % m2 = r2. Here, z is unique modulo M = lcm(m1, m2).
    // Return (z, M). On failure, M = -1.
    PII chinese_remainder_theorem(int m1, int r1, int m2, int r2) {
        int s, t;
        int g = extended_euclid(m1, m2, s, t);
        if (r1%g != r2%g) return make_pair(0, -1);
        return make_pair(mod(s*r2*m1 + t*r1*m2, m1*m2) / g, m1*m2 / g);
    }

    // Chinese remainder theorem: find z such that
    // z % m[i] = r[i] for all i. Note that the solution is
    // unique modulo M = lcm_i (m[i]). Return (z, M). On
    // failure, M = -1. Note that we do not require the a[i]'s
    // to be relatively prime.
    PII chinese_remainder_theorem(const VI &m, const VI &r) {
        PII ret = make_pair(r[0], m[0]);
        for (int i = 1; i < m.size(); i++) {
            ret = chinese_remainder_theorem(ret.second, ret.first, m[i], r[i]);
            if (ret.second == -1) break;
        }
        return ret;
    }

    // computes x and y such that ax + by = c
    // returns whether the solution exists
    bool linear_diophantine(int a, int b, int c, int &x, int &y) {
        if (!a && !b)
        {
            if (c) return false;
            x = 0; y = 0;
            return true;
        }
        if (!a)
        {
            if (c % b) return false;
            x = 0; y = c / b;
            return true;
        }
        if (!b)
        {
            if (c % a) return false;
            x = c / a; y = 0;
            return true;
        }
        int g = gcd(a, b);
        if (c % g) return false;
        x = c / g * mod_inverse(a / g, b / g);
        y = (c - a*x) / b;
        return true;
    }

    int main() {
        // expected: 2
        cout << gcd(14, 30) << endl;

        // expected: 2 -2 1
        int x, y;
        int g = extended_euclid(14, 30, x, y);
        cout << g << " " << x << " " << y << endl;

        // expected: 95 451
        VI sols = modular_linear_equation_solver(14, 30, 100);
    }

```



```

    for (int i = 0; i < sols.size(); i++) cout << sols[i] << " ";
    cout << endl;

    // expected: 8
    cout << mod_inverse(8, 9) << endl;

    // expected: 23 105
    //           11 12
    PII ret = chinese_remainder_theorem(VI({ 3, 5, 7 }), VI({ 2, 3, 2 }));
    cout << ret.first << " " << ret.second << endl;
    ret = chinese_remainder_theorem(VI({ 4, 6 }), VI({ 3, 5 }));
    cout << ret.first << " " << ret.second << endl;

    // expected: 5 -15
    if (!linear_diophantine(7, 2, 5, x, y)) cout << "ERROR" << endl;
    cout << x << " " << y << endl;
    return 0;
}

```

## 3.2 Systems of linear equations, matrix inverse, determinant

```

// Gauss-Jordan elimination with full pivoting.
//
// Uses:
// (1) solving systems of linear equations (AX=B)
// (2) inverting matrices (AX=I)
// (3) computing determinants of square matrices
//
// Running time: O(n^3)
//
// INPUT:      a[][] = an nxn matrix
//             b[][] = an nxm matrix
//
// OUTPUT:     X      = an nxm matrix (stored in b[][])
//             A^{-1} = an nxn matrix (stored in a[][])
//             returns determinant of a[][]

#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

const double EPS = 1e-10;

typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {
    const int n = a.size();
    const int m = b[0].size();
    VI irow(n), icol(n), ipiv(n);
    T det = 1;

    for (int i = 0; i < n; i++) {
        int pj = -1, pk = -1;
        for (int j = 0; j < n; j++) if (!ipiv[j])
            for (int k = 0; k < n; k++) if (!ipiv[k])
                if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
        if (fabs(a[pj][pk]) < EPS) { cerr << "Matrix is singular." << endl; exit(0); }
        ipiv[pk]++;
        swap(a[pj], a[pk]);
        swap(b[pj], b[pk]);
        if (pj != pk) det *= -1;
        irow[i] = pj;
        icol[i] = pk;

        T c = 1.0 / a[pk][pk];
        det *= a[pk][pk];
        a[pk][pk] = 1.0;
    }
}

```

```

    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
        c = a[p][pk];
        a[p][pk] = 0;
        for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
        for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
}

for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
}

return det;
}

int main() {
    const int n = 4;
    const int m = 2;
    double A[n][n] = { {1,2,3,4}, {1,0,1,0}, {5,3,2,4}, {6,1,4,6} };
    double B[n][m] = { {1,2}, {4,3}, {5,6}, {8,7} };
    VVT a(n), b(m);
    for (int i = 0; i < n; i++) {
        a[i] = VT(A[i], A[i] + n);
        b[i] = VT(B[i], B[i] + m);
    }

    double det = GaussJordan(a, b);

    // expected: 60
    cout << "Determinant: " << det << endl;

    // expected: -0.233333 0.166667 0.133333 0.0666667
    //              0.166667 0.166667 0.333333 -0.333333
    //              0.233333 0.833333 -0.133333 -0.0666667
    //              0.05 -0.75 -0.1 0.2
    cout << "Inverse: " << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            cout << a[i][j] << ' ';
        cout << endl;
    }

    // expected: 1.63333 1.3
    //              -0.166667 0.5
    //              2.36667 1.7
    //              -1.85 -1.35
    cout << "Solution: " << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++)
            cout << b[i][j] << ' ';
        cout << endl;
    }
}

```

### 3.3 Reduced row echelon form, matrix rank

```

// Reduced row echelon form via Gauss-Jordan elimination
// with partial pivoting. This can be used for computing
// the rank of a matrix.
//
// Running time: O(n^3)
//
// INPUT:      a[][] = an nxm matrix
//
// OUTPUT:     rref[][] = an nxm matrix (stored in a[][])
//             returns rank of a[][]

#include <iostream>
#include <vector>
#include <cmath>

```

```

using namespace std;
const double EPSILON = 1e-10;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;
int rref(VVT &a) {
    int n = a.size();
    int m = a[0].size();
    int r = 0;
    for (int c = 0; c < m && r < n; c++) {
        int j = r;
        for (int i = r + 1; i < n; i++)
            if (fabs(a[i][c]) > fabs(a[j][c])) j = i;
        if (fabs(a[j][c]) < EPSILON) continue;
        swap(a[j], a[r]);

        T s = 1.0 / a[r][c];
        for (int j = 0; j < m; j++) a[r][j] *= s;
        for (int i = 0; i < n; i++) if (i != r) {
            T t = a[i][c];
            for (int j = 0; j < m; j++) a[i][j] -= t * a[r][j];
        }
        r++;
    }
    return r;
}
int main() {
    const int n = 5, m = 4;
    double A[n][m] = {
        {16, 2, 3, 13},
        {5, 11, 10, 8},
        {9, 7, 6, 12},
        {4, 14, 15, 1},
        {13, 21, 21, 13}};
    VVT a(n);
    for (int i = 0; i < n; i++)
        a[i] = VT(A[i], A[i] + m);
    int rank = rref(a);
    // expected: 3
    cout << "Rank: " << rank << endl;
    // expected: 1 0 0 1
    //              0 1 0 3
    //              0 0 1 -3
    //              0 0 0 3.10862e-15
    //              0 0 0 2.22045e-15
    cout << "rref: " << endl;
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 4; j++)
            cout << a[i][j] << ' ';
        cout << endl;
    }
}

```

### 3.4 Fast Fourier transform

```

#include <cassert>
#include <cstdio>
#include <cmath>
struct cpx
{
    cpx() {}
    cpx(double aa):a(aa),b(0) {}
    cpx(double aa, double bb):a(aa),b(bb) {}
    double a;
    double b;
    double modsq(void) const

```

```

    {
        return a * a + b * b;
    }
    cpx bar(void) const
    {
        return cpx(a, -b);
    }
};

cpx operator +(cpx a, cpx b)
{
    return cpx(a.a + b.a, a.b + b.b);
}

cpx operator *(cpx a, cpx b)
{
    return cpx(a.a * b.a - a.b * b.b, a.a * b.b + a.b * b.a);
}

cpx operator /(cpx a, cpx b)
{
    cpx r = a * b.bar();
    return cpx(r.a / b.modsq(), r.b / b.modsq());
}

cpx EXP(double theta)
{
    return cpx(cos(theta), sin(theta));
}

const double two_pi = 4 * acos(0);

// in:      input array
// out:      output array
// step:     {SET TO 1} (used internally)
// size:     length of the input/output {MUST BE A POWER OF 2}
// dir:      either plus or minus one (direction of the FFT)
// RESULT:   out[k] = \sum_{j=0}^{size-1} in[j] * exp(dir * 2pi * i * j * k /
//           size)
void FFT(cpx *in, cpx *out, int step, int size, int dir)
{
    if(size < 1) return;
    if(size == 1)
    {
        out[0] = in[0];
        return;
    }
    FFT(in, out, step * 2, size / 2, dir);
    FFT(in + step, out + size / 2, step * 2, size / 2, dir);
    for(int i = 0 ; i < size / 2 ; i++)
    {
        cpx even = out[i];
        cpx odd = out[i + size / 2];
        out[i] = even + EXP(dir * two_pi * i / size) * odd;
        out[i + size / 2] = even + EXP(dir * two_pi * (i + size / 2) / size) * odd;
    }
}

// Usage:
// f[0...N-1] and g[0..N-1] are numbers
// Want to compute the convolution h, defined by
// h[n] = sum of f[k]g[n-k] (k = 0, ..., N-1).
// Here, the index is cyclic; f[-1] = f[N-1], f[-2] = f[N-2], etc.
// Let F[0...N-1] be FFT(f), and similarly, define G and H.
// The convolution theorem says H[n] = F[n]G[n] (element-wise product).
// To compute h[] in O(N log N) time, do the following:
// 1. Compute F and G (pass dir = 1 as the argument).
// 2. Get H by element-wise multiplying F and G.
// 3. Get h by taking the inverse FFT (use dir = -1 as the argument)
//    and *dividing by N*. DO NOT FORGET THIS SCALING FACTOR.

int main(void)
{
    printf("If rows come in identical pairs, then everything works.\n");
}

```

```

cpx a[8] = {0, 1, cpx(1,3), cpx(0,5), 1, 0, 2, 0};
cpx b[8] = {1, cpx(0,-2), cpx(0,1), 3, -1, -3, 1, -2};
cpx A[8];
cpx B[8];
FFT(a, A, 1, 8, 1);
FFT(b, B, 1, 8, 1);
for(int i = 0 ; i < 8 ; i++)
{
    printf("%7.2lf%7.2lf", A[i].a, A[i].b);
}
printf("\n");
for(int i = 0 ; i < 8 ; i++)
{
    cpx Ai(0,0);
    for(int j = 0 ; j < 8 ; j++)
    {
        Ai = Ai + a[j] * EXP(j * i * two_pi / 8);
    }
    printf("%7.2lf%7.2lf", Ai.a, Ai.b);
}
printf("\n");
cpx AB[8];
for(int i = 0 ; i < 8 ; i++)
    AB[i] = A[i] * B[i];
cpx aconvb[8];
FFT(AB, aconvb, 1, 8, -1);
for(int i = 0 ; i < 8 ; i++)
    aconvb[i] = aconvb[i] / 8;
for(int i = 0 ; i < 8 ; i++)
{
    printf("%7.2lf%7.2lf", aconvb[i].a, aconvb[i].b);
}
printf("\n");
for(int i = 0 ; i < 8 ; i++)
{
    cpx aconvbi(0,0);
    for(int j = 0 ; j < 8 ; j++)
    {
        aconvbi = aconvbi + a[j] * b[(8 + i - j) % 8];
    }
    printf("%7.2lf%7.2lf", aconvbi.a, aconvbi.b);
}
printf("\n");
return 0;
}

```

## 4 Graph algorithms

### 4.1 Bellman-Ford shortest paths with negative edge weights (C++)

```

tor<int> dist(V, INF); dist[s] = 0; // INF = 2B not MAX_INT to avoid overflow
REP (i, 0, V - 1) // relax all E edges V-1 times, O(V)
    REP (u, 0, V - 1) // these two loops = O(E)
        TRvii (AdjList[u], v) // has edge and can be relaxed
            dist[v->first] = min(dist[v->first], dist[u] + v->second);
REP (i, 0, V - 1)
    printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

bool negative_cycle_exist = false;
REP (u, 0, V - 1) // one more pass to check
    TRvii (AdjList[u], v)
        if (dist[v->first] > dist[u] + v->second) // should be false, but if
            possible
                negative_cycle_exist = true; // then negative cycle exists!
printf("Negative Cycle Exist? %s\n", negative_cycle_exist ? "Yes" : "No")
;

```

## 4.2 Floyd-Warshall

```

EP (k, 0, V - 1) // recall that #define REP(i, a, b) for (int i = int(a); i <=
    int(b); i++)
    REP (i, 0, V - 1)
    REP (j, 0, V - 1)
    AdjMatrix[i][j] = min(AdjMatrix[i][j], AdjMatrix[i][k] + AdjMatrix[k][j])
    ;

```

## 4.3 Fast Dijkstra's algorithm

```

le (!pq.empty()) { // main loop
    ii top = pq.top(); pq.pop(); // greedy: pick shortest unvisited vertex
    int d = top.first, u = top.second;
    if (d == dist[u]) // This check is important! We want to process vertex u
        only once but we can
        // actually enqueue u several times in priority_queue...
        // Fortunately, other occurrences of u
        // in priority_queue will have greater distances and can be
        // ignored (the overhead is small :)
    TRvii (AdjList[u], it) { // all outgoing edges from u
        int v = it->first, weight_u_v = it->second;
        if (dist[u] + weight_u_v < dist[v]) { // if can relax
            dist[v] = dist[u] + weight_u_v; // relax
            pq.push(ii(dist[v], v)); // enqueue this neighbor
        } } } // regardless whether it is already in pq or not

```

## 4.4 Strongly connected components

```

tack<int> dfs_scc; // additional information for SCC
set<int> in_stack; // for dfs_low update check
void tarjanSCC(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
    dfs_scc.push(u); in_stack.insert(u); // stores u based on order of
    visitation
    TRvii (AdjList[u], v) {
        if (dfs_num[v->first] == DFS_WHITE) // a tree edge
            tarjanSCC(v->first);
        if (in_stack.find(v->first) != in_stack.end()) // condition for
            update
            dfs_low[u] = min(dfs_low[u], dfs_low[v->first]); //
            update dfs_low[u]
    }
    if (dfs_low[u] == dfs_num[u]) { // if this is a root of SCC
        printf("SCC: ");
        while (!dfs_scc.empty() && dfs_scc.top() != u) {
            printf("%d ", dfs_scc.top()); in_stack.erase(dfs_scc.top());
            dfs_scc.pop();
        }
        printf("%d\n", dfs_scc.top()); in_stack.erase(dfs_scc.top());
        dfs_scc.pop();
    } }

```

## 4.5 Eulerian path

```

struct Edge;
typedef list<Edge>::iterator iter;
struct Edge
{
    int next_vertex;
    iter reverse_edge;

```

```

        Edge(int next_vertex)
            :next_vertex(next_vertex)
            { }
    };

    const int max_vertices = ;
    int num_vertices;
    list<Edge> adj[max_vertices];           // adjacency list
    vector<int> path;
    void find_path(int v)
    {
        while(adj[v].size() > 0)
        {
            int vn = adj[v].front().next_vertex;
            adj[vn].erase(adj[v].front().reverse_edge);
            adj[v].pop_front();
            find_path(vn);
        }
        path.push_back(v);
    }
    void add_edge(int a, int b)
    {
        adj[a].push_front(Edge(b));
        iter ita = adj[a].begin();
        adj[b].push_front(Edge(a));
        iter itb = adj[b].begin();
        ita->reverse_edge = itb;
        itb->reverse_edge = ita;
    }

```

## 4.6 Kruskal's alternative

```

#include<bits/stdc++.h>
using namespace std;
typedef pair<int, int> iPair;
struct Graph{
    int V, E;
    vector< pair<int, iPair> > edges;
    Graph(int V, int E){
        this->V = V;
        this->E = E;
    }
    void addEdge(int u, int v, int w){
        edges.push_back({w, {u, v}});
    }
    int kruskalMST();
};

struct DisjointSets{
    int *parent, *rnk;
    int n;
    DisjointSets(int n){
        this->n = n;
        parent = new int[n+1];
        rnk = new int[n+1];
        for (int i = 0; i <= n; i++){
            rnk[i] = 0;
            parent[i] = i;
        }
    }
    int find(int u){
        if (u != parent[u])
            parent[u] = find(parent[u]);
        return parent[u];
    }
    void merge(int x, int y){
        x = find(x), y = find(y);
    }

```

```

        if (rnk[x] > rnk[y])
            parent[y] = x;
        else
            parent[x] = y;
        if (rnk[x] == rnk[y])
            rnk[y]++;
    }
};

int Graph::kruskalMST() {
    int mst_wt = 0; // Initialize result
    sort(edges.begin(), edges.end());
    DisjointSets ds(V);
    vector< pair<int, iPair> >::iterator it;
    for (it=edges.begin(); it!=edges.end(); it++) {
        int u = it->second.first;
        int v = it->second.second;
        int set_u = ds.find(u);
        int set_v = ds.find(v);
        if (set_u != set_v) {
            cout << u << " - " << v << endl;
            mst_wt += it->first;
            ds.merge(set_u, set_v);
        }
    }
    return mst_wt;
}

int main() {
    int V = 9, E = 14;
    Graph g(V, E);
    g.addEdge(0, 1, 4); g.addEdge(1, 2, 8); g.addEdge(1, 7, 11); g.addEdge(2, 3,
        7); g.addEdge(2, 8, 2); g.addEdge(2, 5, 4); g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14); g.addEdge(4, 5, 10); g.addEdge(5, 6, 2); g.addEdge(6, 7,
        1); g.addEdge(6, 8, 6); g.addEdge(7, 8, 7); g.addEdge(0, 7, 8);
    cout << "Edges of MST are \n";
    int mst_wt = g.kruskalMST();
    cout << "\nWeight of MST is " << mst_wt;
    return 0;
}

```

## 4.7 Prim alternative

```

#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f
typedef pair<int, int> iPair;

class Graph{
    int V; // No. of vertices
    list< pair<int, int> > *adj;
public:
    Graph(int V); // Constructor
    void addEdge(int u, int v, int w);
    void primMST();
};

Graph::Graph(int V) {
    this->V = V;
    adj = new list<iPair> [V];
}

void Graph::addEdge(int u, int v, int w) {
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}

void Graph::primMST() {
    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;
    int src = 0; // Taking vertex 0 as source
    vector<int> key(V, INF);
    vector<int> parent(V, -1);
}

```



```

vector<bool> inMST(V, false);
pq.push(make_pair(0, src));
key[src] = 0;
while (!pq.empty())
{
    int u = pq.top().second;
    pq.pop();
    inMST[u] = true; // Include vertex in MST
    list< pair<int, int> >::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        int v = (*i).first;
        int weight = (*i).second;
        if (inMST[v] == false && key[v] > weight)
        {
            key[v] = weight;
            pq.push(make_pair(key[v], v));
            parent[v] = u;
        }
    }
}
for (int i = 1; i < V; ++i)
    printf("%d - %d\n", parent[i], i);
}

int main() {
    int V = 9;
    Graph g(V); g.addEdge(0, 1, 4); g.addEdge(0, 7, 8); g.addEdge(1, 2, 8); g.addEdge
        (1, 7, 11); g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2); g.addEdge(2, 5, 4); g.addEdge(3, 4, 9); g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2); g.addEdge(6, 7, 1); g.addEdge(6, 8, 6); g.addEdge(7, 8, 7);
    g.primMST();
    return 0;
}

```

## 4.8 Bridge Finding

```

ulationPointAndBridge(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
    TRvii (AdjList[u], v)
        if (dfs_num[v->first] == DFS_WHITE) { // a tree edge
            dfs_parent[v->first] = u; // parent of this children is
            me
            if (u == dfsRoot) // special case
                rootChildren++; // count children of root
            articulationPointAndBridge(v->first);
            if (dfs_low[v->first] >= dfs_num[u]) // for articulation
                point
                articulation_vertex[u] = true; // store this
                information first
            if (dfs_low[v->first] > dfs_num[u]) // for bridge
                printf(" Edge (%d, %d) is a bridge\n", u, v->
                    first);
            dfs_low[u] = min(dfs_low[u], dfs_low[v->first]); //
                update dfs_low[u]
        }
        else if (v->first != dfs_parent[u]) // a back edge and not direct
            cycle
            dfs_low[u] = min(dfs_low[u], dfs_num[v->first]); //
                update dfs_low[u]
    }

// inside int main()
dfsNumberCounter = 0;
memset(dfs_num, DFS_WHITE, sizeof dfs_num);
printf("Bridges:\n");
REP (i, 0, V - 1)
    if (dfs_num[i] == DFS_WHITE) {
        dfsRoot = i; rootChildren = 0;
    }
}

```

```

        articulationPointAndBridge(i);
        articulation_vertex[dfsRoot] = (rootChildren > 1); // special
        case
    }
    printf("Articulation Points:\n");
    REP (i, 0, V - 1)
        if (articulation_vertex[i])
            printf(" Vertex %d\n", i);

```

## 5 Data structures

### 5.1 Suffix array

```

#define MAXN 65536
#define MAXLG 17
char A[MAXN];
struct entry
{
    int nr[2];
    int p;
} L[MAXN];
int P[MAXLG][MAXN];
int N, i;
int stp, cnt;
int cmp(struct entry a, struct entry b)
{
    return a.nr[0]==b.nr[0] ? (a.nr[1]<b.nr[1] ? 1: 0) : (a.nr[0]<b.nr[0]
        ? 1: 0);
}

int main()
{
    gets(A);
    for(N=strlen(A), i = 0; i < N; i++)
        P[0][i] = A[i] - 'a';
    for(stp=1, cnt = 1; cnt < N; stp++, cnt *= 2)
    {
        for(i=0; i < N; i++)
        {
            L[i].nr[0]=P[stp-1][i];
            L[i].nr[1]=i+cnt < N? P[stp-1][i+cnt]:-1;
            L[i].p= i;
        }
        sort(L, L+N, cmp);
        for(i=0; i < N; i++)
            P[stp][L[i].p] =i> 0 && L[i].nr[0]==L[i-1].nr[0] && L[i].nr[1] ==
                L[i-1].nr[1] ? P[stp][L[i-1].p] : i;
        return 0;
    }
    /*
    LCP Pseudocode :-
    FindLCP (x, y)
        answer = 0
        for k = ceil(log N) to 0
            if SortIndex[k][x] = SortIndex[k][y]
                // sort-index is same if the first k characters are
                same
                answer += 2k
                // now we wish to find the characters that are same
                in the remaining strings
                x += 2k
                y += 2k
    */

```

### 5.2 Binary Indexed Tree

```

#include <iostream>
using namespace std;
#define LOGSZ 17
int tree[(1<<LOGSZ)+1];
int N = (1<<LOGSZ);
// add v to value at x
void set(int x, int v) {
    while(x <= N) {
        tree[x] += v;
        x += (x & -x);
    }
}
// get cumulative sum up to and including x
int get(int x) {
    int res = 0;
    while(x) {
        res += tree[x];
        x -= (x & -x);
    }
    return res;
}
// get largest value with cumulative sum less than or equal to x;
// for smallest, pass x-1 and add 1 to result
int getind(int x) {
    int idx = 0, mask = N;
    while(mask && idx < N) {
        int t = idx + mask;
        if(x >= tree[t]) {
            idx = t;
            x -= tree[t];
        }
        mask >>= 1;
    }
    return idx;
}

```

---

### 5.3 Union-find set

```

#include <iostream>
#include <vector>
using namespace std;
int find(vector<int> &C, int x) { return (C[x] == x) ? x : C[x] = find(C, C[x]); }
void merge(vector<int> &C, int x, int y) { C[find(C, x)] = find(C, y); }
int main()
{
    int n = 5;
    vector<int> C(n);
    for (int i = 0; i < n; i++) C[i] = i;
    merge(C, 0, 2);
    merge(C, 1, 0);
    merge(C, 3, 4);
    for (int i = 0; i < n; i++) cout << i << " " << find(C, i) << endl;
    return 0;
}

```

---

### 5.4 Lazy segment tree

```

#include <bits/stdc++.h>
using namespace std;
const int N = 1e5+10;
int A[N];
int tree[4*N];

```

```

void build(int node, int start, int end)
{
    if(start == end){
        tree[node] = A[start];
    }
    else{
        int mid = (start + end) / 2;
        build(2*node, start, mid);
        build(2*node+1, mid+1, end);
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}

void update(int node, int start, int end, int idx, int val){
    if(start == end){
        A[idx] += val;
        tree[node] += val;
    }
    else{
        int mid = (start + end) / 2;
        if(start <= idx and idx <= mid){
            update(2*node, start, mid, idx, val);
        }
        else{
            update(2*node+1, mid+1, end, idx, val);
        }
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}

int query(int node, int start, int end, int l, int r){
    if(r < start or end < l){
        return 0;
    }
    if(l <= start and end <= r){
        return tree[node];
    }
    int mid = (start + end) / 2;
    int p1 = query(2*node, start, mid, l, r);
    int p2 = query(2*node+1, mid+1, end, l, r);
    return (p1 + p2);
}

void updateRange(int node, int start, int end, int l, int r, int val){
    if(lazy[node] != 0){
        tree[node] += (end - start + 1) * lazy[node];    // Update it
        if(start != end){
            lazy[node*2] += lazy[node];                // Mark
            child as lazy
            lazy[node*2+1] += lazy[node];              // Mark
            child as lazy
        }
        lazy[node] = 0;                                // Reset it
    }
    if(start > end or start > r or end < l)              // Current segment
        is not within range [l, r]
        return;
    if(start >= l and end <= r){
        tree[node] += (end - start + 1) * val;
        if(start != end){
            lazy[node*2] += val;
            lazy[node*2+1] += val;
        }
        return;
    }
    int mid = (start + end) / 2;
    updateRange(node*2, start, mid, l, r, val);        // Updating left child
    updateRange(node*2 + 1, mid + 1, end, l, r, val);  // Updating right
    child
    tree[node] = tree[node*2] + tree[node*2+1];        // Updating root with
    max value
}

```

```

int queryRange(int node, int start, int end, int l, int r){
    if(start > end or start > r or end < l)
        return 0; // Out of range
    if(lazy[node] != 0){
        tree[node] += (end - start + 1) * lazy[node]; //
        Update it
        if(start != end)
        {
            lazy[node*2] += lazy[node]; // Mark child as lazy
            lazy[node*2+1] += lazy[node]; // Mark child as lazy
        }
        lazy[node] = 0; // Reset it
    }
    if(start >= l and end <= r) // Current segment is totally
        within range [l, r]
        return tree[node];
    int mid = (start + end) / 2;
    int p1 = queryRange(node*2, start, mid, l, r); // Query left
    child
    int p2 = queryRange(node*2 + 1, mid + 1, end, l, r); // Query right child
    return (p1 + p2);
}

int main(){
    return 0;
}

```

## 5.5 Lowest common ancestor

```

const int max_nodes, log_max_nodes;
int num_nodes, log_num_nodes, root;
vector<int> children[max_nodes]; // children[i] contains the children of
node i
int A[max_nodes][log_max_nodes+1]; // A[i][j] is the 2^j-th ancestor of node
i, or -1 if that ancestor does not exist
int L[max_nodes]; // L[i] is the distance between node i
and the root

// floor of the binary logarithm of n
int lb(unsigned int n)
{
    if(n==0)
        return -1;
    int p = 0;
    if (n >= 1<<16) { n >>= 16; p += 16; }
    if (n >= 1<< 8) { n >>= 8; p += 8; }
    if (n >= 1<< 4) { n >>= 4; p += 4; }
    if (n >= 1<< 2) { n >>= 2; p += 2; }
    if (n >= 1<< 1) { p += 1; }
    return p;
}

void DFS(int i, int l)
{
    L[i] = l;
    for(int j = 0; j < children[i].size(); j++)
        DFS(children[i][j], l+1);
}

int LCA(int p, int q)
{
    // ensure node p is at least as deep as node q
    if(L[p] < L[q])
        swap(p, q);
    // "binary search" for the ancestor of node p situated on the same level as q
    for(int i = log_num_nodes; i >= 0; i--)
        if(L[p] - (1<<i) >= L[q])
            p = A[p][i];
    if(p == q)
        return p;
    // "binary search" for the LCA
}

```

```

    for(int i = log_num_nodes; i >= 0; i--)
        if(A[p][i] != -1 && A[p][i] != A[q][i])
        {
            p = A[p][i];
            q = A[q][i];
        }
    return A[p][0];
}

int main(int argc, char* argv[])
{
    // read num_nodes, the total number of nodes
    log_num_nodes = lb(num_nodes);
    for(int i = 0; i < num_nodes; i++)
    {
        int p;
        // read p, the parent of node i or -1 if node i is the root
        A[i][0] = p;
        if(p != -1)
            children[p].push_back(i);
        else
            root = i;
    }
    // precompute A using dynamic programming
    for(int j = 1; j <= log_num_nodes; j++)
        for(int i = 0; i < num_nodes; i++)
            if(A[i][j-1] != -1)
                A[i][j] = A[A[i][j-1]][j-1];
            else
                A[i][j] = -1;
    // precompute L
    DFS(root, 0);
    return 0;
}

```

## 5.6 Sparse Table

```

#include <bits/stdc++.h>
using namespace std;
#define ll long long
const int MAX_SIZE = 1e5+10;
const int K = 16;
const int ZERO = 0; //define zero according to F
ll arr[MAX_SIZE];
ll Table[MAX_SIZE][K+1];
ll F(ll a, ll b)
{
    return max(a, b);
}

void buildTable(int n)
{
    for(int i=0; i<n; i++)
        Table[i][0] = arr[i];
    for(int j = 1; j<=K; j++)
    {
        for(int i=0; i<= n - (1<<j); i++)
        {
            Table[i][j] = F(Table[i][j-1], Table[i + (1<<(j-1))][j-1]);
        }
    }
}

ll query(int l, int r)
{
    ll answer = ZERO;
    for(int j=K; j>=0; j--)

```

```

        {
            if(1 + (1<<j) - 1 <= r)
                answer = F(answer, Table[1][j]);
            l += (1<<j);
        }
    }

int main()
{
    return 0;
}

```

## 5.7 Segmented Sieve

```

#include <bits/stdc++.h>
using namespace std;
//all primes strictly smaller than limit -> stored in vectpr prime
void simpleSieve(int limit, vector<int> &prime)
{
    bool mark[limit + 1];
    memset(mark, true, sizeof(mark));
    for(int p=2; p*p < limit; p++)
    {
        if(mark[p] == true)
        {
            prime.push_back(p);
            for(int i=p*p; i<limit; i += p)
                mark[i] = false;
        }
    }
}

void segmentedSieve(int n)
{
    int limit = floor(sqrt(n)) + 1;
    vector<int> prime;
    simpleSieve(limit, prime);
    int low = limit;
    int high = 2*limit;
    while(low < n)
    {
        bool mark[limit + 1];
        memset(mark, true, sizeof(mark));
        //mark non-primes in bucket
        for(int i=0; i<prime.size(); i++)
        {
            //get smallest non-prime divisible by prime[i];
            int loLim = floor(low/prime[i]) * prime[i];
            if(loLim < low)
                loLim += prime[i];
            //mark all multiples of prime[i]
            for(int j=loLim; j<high; j += prime[i])
                mark[j - low] = false;
        }
        //print primes
        for(int i=low; i<high; i++)
            if(mark[i - low] == true)
                cout << i << " ";

        // update low and high
        low += limit;
        high += limit;
        if(high >= n) high = n;
    }
}

```

```
}  
  
int main()  
{  
    return 0;  
}
```

---

## 6 Miscellaneous

### 6.1 Vimrc file

```
set autoindent  
set smartindent  
set cindent  
set shiftwidth=4  
set tabstop=4  
set autoread  
set cmdheight=1  
set number  
set splitright  
set splitbelow  
set makeprg=g++\ -std=c++14\ %\ -o\ out  
map <F8> :!./out < input > output <ENTER><ENTER>  
map <F7< :make \| cope 8 \| wincmd J <ENTER><ENTER>  
map <C-Right> <C-W><Right>  
map <C-Left> <C-W><Left>  
map <C-Up> <C-W><Up>  
map <C-Down> <C-W><Down>  
only  
40vsp input  
20vsp output  
wincmd w
```

---