Bamboo Team Notes

Contents

1 Number theory

1.1 Extended Euclide

```cpp
int bezout(int a, int b) {
    // return x such that ax + by == gcd(a, b)
    int xa = 1, xb = 0;
    while (b) {
        int q = a / b;
        int r = a - q * b, xr = xa - q * xb;
        a = b; xa = xb;
        b = r; xb = xr;
    }
    return xa;
}

pair<int, int> solve(int a, int b, int c) {
    // solve ax + by == c
    int d = __gcd(a, b);
    int x = bezout(a, b);
    int y = (d - a * x) / b;
    c /= d;
    return make_pair(x * c, y * c);
}

int main() {
    int a = 100, b = 128;
    int c = __gcd(a, b);
    int x = bezout(a, b);
    int y = (c - a * x) / b;
    cout << x << ' ' << y << endl;
    pair<int, int> xy = solve(100, 128, 40);
    cout << xy.first << ' ' << xy.second << endl;
    return 0;
}
```

1.2 System of linear equations

```cpp
// extended version, uses diophantine equation solver to solve system of congruent equations
pair<int, int> solve(int a, int b, int c) {
    // solve ax + by == c
    int d = __gcd(a, b);
    int x = bezout(a / d, b / d);
    int y = (d - a * x) / b;
    c /= d;
    return make_pair(x * c, y * c);
}

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

int solveSystem(vector<int> a, vector<int> b) {
    // xi mod bi = ai
    int A = a[0], B = b[0];
    // x mod B = A
    for (int i = 1; i < a.size(); ++i) {
        int curB = b[i], curA = a[i];
        // x = Bi + A = curB * j + curA
        pair<int, int> ij = solve(B, -curB, curA - A);
        assert(B * ij.first + A == curB * ij.second + curA);
        int newA = (B * ij.first + A);
        B = lcm(B, curB);
        A = newA % B;
        if (i + 1 == a.size()) return A;
    }
}

int main() {
    vector<int> a = {0, 3, 3};
    vector<int> b = {3, 6, 9};
    cout << solveSystem(a, b) << endl;
    return 0;
}
```

2 String

2.1 Suffix Array

```cpp
#include <bits/stdc++.h>

using namespace std;

struct SuffixArray {
    static const int N = 100010;

    int n;
    char *s;
    int sa[N], tmp[N], pos[N];
    int len, cnt[N], lcp[N];

    SuffixArray(char *t) {
        s = t;
        n = strlen(s + 1);
        buildSA();
    }

    bool cmp(int u, int v) {
        if (pos[u] != pos[v]) {
            return pos[u] < pos[v];
        }
        return (u + len <= n && v + len <= n) ? pos[u + len] < pos[v + len] : u > v;
    }

    void radix(int delta) {
        memset(cnt, 0, sizeof cnt);
        for (int i = 1; i <= n; i++) {
            cnt[i + delta <= n ? pos[i + delta] : 0]++;
        }
        for (int i = 1; i < N; i++) {
            cnt[i] += cnt[i - 1];
        }
        for (int i = n; i > 0; i--) {
            int id = sa[i];
            tmp[cnt[id + delta <= n ? pos[id + delta] : 0]--] = id;
        }
        for (int i = 1; i <= n; i++) {
```

```
            sa[i] = tmp[i];
        }
    }

    void buildSA() {
        for (int i = 1; i <= n; i++) {
            sa[i] = i;
            pos[i] = s[i];
        }
        len = 1;
        while (1) {
            radix(len);
            radix(0);
            tmp[1] = 1;
            for (int i = 2; i <= n; i++) {
                tmp[i] = tmp[i - 1] + cmp(sa[i - 1], sa[i]);
            }
            for (int i = 1; i <= n; i++) {
                pos[sa[i]] = tmp[i];
            }
            if (tmp[n] == n) {
                break;
            }
            len <<= 1;
        }

        len = 0;
        for (int i = 1; i <= n; i++) {
            if (pos[i] == n) {
                continue;
            }
            int j = sa[pos[i] + 1];
            while (s[i + len] == s[j + len]) {
                len++;
            }
            lcp[pos[i]] = len;
            if (len) {
                len--;
            }
        }
    }
};
```

2.2 Aho Corasick

```
struct AhoCorasick {
        const int N = 30030;

        int fail[N];
        int to[N][2];
        int ending[N];
        int sz;

        void add(const string &s) {
                int node = 1;
                for (int i = 0; i < s.size(); ++i) {
                        if (!to[node][s[i] - 'a']) {
                                to[node][s[i] - 'a'] = ++sz;
                        }
                        node = to[node][s[i] - 'a'];
                }
                ending[node] = true;
        }

        void push() {
                queue<int> Q;
                Q.push(1);
                fail[1] = 1;
                while (!Q.empty()) {
                        int u = Q.front(); Q.pop();
                        for (int i = 0; i < 26; ++i) {
                                int &v = to[u][i];
                                if (!v) {
                                        v = u == 1 ? 1 : to[fail[u]][i];
                                } else {
                                        fail[v] = u == 1 ? 1 : to[fail[u]][i];
                                        Q.push(v);
                                }
                        }
                }
        }
};
```

2.3 Z algorithm

```
vector<int> calcZ(const string &s) {
    int L = 0, R = 0;
    int n = s.size();
    vector<int> Z(n);
    Z[0] = n;
    for (int i = 1; i < n; i++) {
        if (i > R)
        {
            L = R = i;
            while (R < n && s[R] == s[R - L]) R++;
            Z[i] = R - L; R--;
        }
        else
        {
            int k = i - L;
            if (Z[k] < R - i + 1) Z[i] = Z[k];
            else
            {
                L = i;
                while (R < n && s[R] == s[R - L]) R++;
                Z[i] = R - L; R--;
            }
        }
    }
    return Z;
}
```

2.4 Suffix Automaton

```
//set last = 0 everytime we add new string
struct SuffixAutomaton {
    static const int N = 100000;
    static const int CHARACTER = 26;
    int suf[N * 2], nxt[N * 2][CHARACTER], cnt, last, len[N * 2];

    SuffixAutomaton() {
        memset(suf, -1, sizeof suf);
        memset(nxt, -1, sizeof nxt);
        memset(len, 0, sizeof len);
        last = cnt = 0;
    }

    int getNode(int last, int u) {
        int q = nxt[last][u];
        if (len[last] + 1 == len[q]) {
            return q;
        }
        int clone = ++cnt;
        len[clone] = len[last] + 1;
        for (int i = 0; i < CHARACTER; i++) {
            nxt[clone][i] = nxt[q][i];
        }
        while (last != -1 && nxt[last][u] == q) {
            nxt[last][u] = clone;
            last = suf[last];
        }
        suf[clone] = suf[q];
        return suf[q] = clone;
    }

    void add(int u) {
        if (nxt[last][u] == -1) {
            int newNode = ++cnt;
            len[newNode] = len[last] + 1;
            while (last != -1 && nxt[last][u] == -1) {
                nxt[last][u] = newNode;
                last = suf[last];
            }
            if (last == -1) {
                suf[newNode] = 0;
                last = newNode;
                return;
            }
            suf[newNode] = getNode(last, u);
            last = newNode;
        } else {
            last = getNode(last, u);
        }
    }
};
```

3 Combinatorial optimization

4 Geometry

4.1 Geometry

```cpp
#define EPS 1e-6
inline int cmp(double a, double b) { return (a < b - EPS) ? -1 : ((a > b + EPS) ? 1 : 0); }
struct Point {
    double x, y;
    Point() { x = y = 0.0; }
    Point(double x, double y) : x(x), y(y) {}

    Point operator + (const Point& a) const { return Point(x+a.x, y+a.y); }
    Point operator - (const Point& a) const { return Point(x-a.x, y-a.y); }
    Point operator * (double k) const { return Point(x*k, y*k); }
    Point operator / (double k) const { return Point(x/k, y/k); }

    double operator * (const Point& a) const { return x*a.x + y*a.y; } // dot product
    double operator % (const Point& a) const { return x*a.y - y*a.x; } // cross product
    double norm() { return x*x + y*y; }
    double len() { return sqrt(norm()); } // hypot(x, y);
    Point rotate(double alpha) {
        double cosa = cos(alpha), sina = sin(alpha);
        return Point(x * cosa - y * sina, x * sina + y * cosa);
    }
};
double angle(Point a, Point o, Point b) { // min of directed angle AOB & BOA
    a = a - o; b = b - o;
    return acos((a * b) / sqrt(a.norm()) / sqrt(b.norm()));
}
double directed_angle(Point a, Point o, Point b) { // angle AOB, in range [0, 2*PI)
    double t = -atan2(a.y - o.y, a.x - o.x)
            + atan2(b.y - o.y, b.x - o.x);
    while (t < 0) t += 2*PI;
    return t;
}
// Distance from p to Line ab (closest Point --> c)
double distToLine(Point p, Point a, Point b, Point &c) {
    Point ap = p - a, ab = b - a;
    double u = (ap * ab) / ab.norm();
    c = a + (ab * u);
    return (p-c).len();
}
// Distance from p to segment ab (closest Point --> c)
double distToLineSegment(Point p, Point a, Point b, Point &c) {
    Point ap = p - a, ab = b - a;
    double u = (ap * ab) / ab.norm();
    if (u < 0.0) {
        c = Point(a.x, a.y);
        return (p - a).len();
    }
    if (u > 1.0) {
        c = Point(b.x, b.y);
        return (p - b).len();
    }
    return distToLine(p, a, b, c);
}
// NOTE: WILL NOT WORK WHEN a = b = 0.
struct Line {
    double a, b, c;
    Point A, B; // Added for polygon intersect line. Do not rely on assumption that these are valid

    Line(double a, double b, double c) : a(a), b(b), c(c) {}

    Line(Point A, Point B) : A(A), B(B) {
        a = B.y - A.y;
        b = A.x - B.x;
        c = - (a * A.x + b * A.y);
    }
    Line(Point P, double m) {
        a = -m; b = 1;
        c = -((a * P.x) + (b * P.y));
    }
    double f(Point A) {
        return a*A.x + b*A.y + c;
    }
};
bool areParallel(Line l1, Line l2) {
    return cmp(l1.a*l2.b, l1.b*l2.a) == 0;
}
bool areSame(Line l1, Line l2) {
    return areParallel(l1 ,l2) && cmp(l1.c*l2.a, l2.c*l1.a) == 0
                && cmp(l1.c*l2.b, l1.b*l2.c) == 0;
}
bool areIntersect(Line l1, Line l2, Point &p) {
    if (areParallel(l1, l2)) return false;
    double dx = l1.b*l2.c - l2.b*l1.c;
    double dy = l1.c*l2.a - l2.c*l1.a;
    double d  = l1.a*l2.b - l2.a*l1.b;
    p = Point(dx/d, dy/d);
    return true;
}
void closestPoint(Line l, Point p, Point &ans) {
    if (fabs(l.b) < EPS) {
        ans.x = -(l.c) / l.a; ans.y = p.y;
        return;
    }
    if (fabs(l.a) < EPS) {
        ans.x = p.x; ans.y = -(l.c) / l.b;
        return;
    }
    Line perp(l.b, -l.a, - (l.b*p.x - l.a*p.y));
    areIntersect(l, perp, ans);
}
void reflectionPoint(Line l, Point p, Point &ans) {
    Point b;
    closestPoint(l, p, b);
    ans = p + (b - p) * 2;
}
struct Circle : Point {
    double r;
    Circle(double x = 0, double y = 0, double r = 0) : Point(x, y), r(r) {}
    Circle(Point p, double r) : Point(p), r(r) {}
    bool contains(Point p) { return (*this - p).len() <= r + EPS; }
};

// Find common tangents to 2 circles
// Tested:
// - http://codeforces.com/gym/100803/ - H
// Helper method
void tangents(Point c, double r1, double r2, vector<Line> & ans) {
    double r = r2 - r1;
    double z = sqr(c.x) + sqr(c.y);
    double d = z - sqr(r);
    if (d < -EPS)  return;
    d = sqrt(fabs(d));
    Line l((c.x * r + c.y * d) / z,
            (c.y * r - c.x * d) / z,
            r1);
    ans.push_back(l);
}
// Actual method: returns vector containing all common tangents
vector<Line> tangents(Circle a, Circle b) {
    vector<Line> ans; ans.clear();
    for (int i=-1; i<=1; i+=2)
        for (int j=-1; j<=1; j+=2)
            tangents(b-a, a.r*i, b.r*j, ans);
    for(int i = 0; i < ans.size(); ++i)
        ans[i].c -= ans[i].a * a.x + ans[i].b * a.y;
    vector<Line> ret;
    for(int i = 0; i < (int) ans.size(); ++i) {
        bool ok = true;
        for(int j = 0; j < i; ++j)
            if (areSame(ret[j], ans[i])) {
                ok = false;
                break;
            }
        if (ok) ret.push_back(ans[i]);
    }
    return ret;
}
// Circle & line intersection
vector<Point> intersection(Line l, Circle cir) {
    double r = cir.r, a = l.a, b = l.b, c = l.c + l.a*cir.x + l.b*cir.y;
    vector<Point> res;
    double x0 = -a*c/(a*a+b*b),  y0 = -b*c/(a*a+b*b);
    if (c*c > r*r*(a*a+b*b)+EPS) return res;
    else if (fabs(c*c - r*r*(a*a+b*b)) < EPS) {
        res.push_back(Point(x0, y0) + Point(cir.x, cir.y));
        return res;
    }
    else {
        double d = r*r - c*c/(a*a+b*b);
        double mult = sqrt (d / (a*a+b*b));
        double ax,ay,bx,by;
        ax = x0 + b * mult;
        bx = x0 - b * mult;
        ay = y0 - a * mult;
        by = y0 + a * mult;
        res.push_back(Point(ax, ay) + Point(cir.x, cir.y));
        res.push_back(Point(bx, by) + Point(cir.x, cir.y));
        return res;
    }
}
```

```cpp
// helper functions for commonCircleArea
double cir_area_solve(double a, double b, double c) {
    return acos((a*a + b*b - c*c) / 2 / a / b);
}
double cir_area_cut(double a, double r) {
    double s1 = a * r * r / 2;
    double s2 = sin(a) * r * r / 2;
    return s1 - s2;
}
double commonCircleArea(Circle c1, Circle c2) { //return the common area of two circle
    if (c1.r < c2.r) swap(c1, c2);
    double d = (c1 - c2).len();
    if (d + c2.r <= c1.r + EPS) return c2.r*c2.r*M_PI;
    if (d >= c1.r + c2.r - EPS) return 0.0;
    double a1 = cir_area_solve(d, c1.r, c2.r);
    double a2 = cir_area_solve(d, c2.r, c1.r);
    return cir_area_cut(a1*2, c1.r) + cir_area_cut(a2*2, c2.r);
}
// Check if 2 circle intersects. Return true if 2 circles touch
bool areIntersect(Circle u, Circle v) {
    if (cmp((u - v).len(), u.r + v.r) > 0) return false;
    if (cmp((u - v).len() + v.r, u.r) < 0) return false;
    if (cmp((u - v).len() + u.r, v.r) < 0) return false;
    return true;
}
// If 2 circle touches, will return 2 (same) points
// If 2 circle are same --> be careful
vector<Point> circleIntersect(Circle u, Circle v) {
    vector<Point> res;
    if (!areIntersect(u, v)) return res;
    double d = (u - v).len();
    double alpha = acos((u.r * u.r + d*d - v.r * v.r) / 2.0 / u.r / d);

    Point p1 = (v - u).rotate(alpha);
    Point p2 = (v - u).rotate(-alpha);
    res.push_back(p1 / p1.len() * u.r + u);
    res.push_back(p2 / p2.len() * u.r + u);
    return res;
}
Point centroid(Polygon p) {
    Point c(0,0);
    double scale = 6.0 * signed_area(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;
}
// Cut a polygon with a line. Returns one half.
// To return the other half, reverse the direction of Line l (by negating l.a, l.b)
// The line must be formed using 2 points
Polygon polygon_cut(const Polygon& P, Line l) {
    Polygon Q;
    for(int i = 0; i < P.size(); ++i) {
        Point A = P[i], B = (i == P.size()-1) ? P[0] : P[i+1];
        if (ccw(l.A, l.B, A) != -1) Q.push_back(A);
        if (ccw(l.A, l.B, A)*ccw(l.A, l.B, B) < 0) {
            Point p; areIntersect(Line(A, B), l, p);
            Q.push_back(p);
        }
    }
    return Q;
}
// Find intersection of 2 convex polygons
// Helper method
bool intersect_1pt(Point a, Point b,
    Point c, Point d, Point &r) {
    double D =  (b - a) % (d - c);
    if (cmp(D, 0) == 0) return false;
    double t =  ((c - a) % (d - c)) / D;
    double s = -((a - c) % (b - a)) / D;
    r = a + (b - a) * t;
    return cmp(t, 0) >= 0 && cmp(t, 1) <= 0 && cmp(s, 0) >= 0 && cmp(s, 1) <= 0;
}
Polygon convex_intersect(Polygon P, Polygon Q) {
    const int n = P.size(), m = Q.size();
    int a = 0, b = 0, aa = 0, ba = 0;
    enum { Pin, Qin, Unknown } in = Unknown;
    Polygon R;
    do {
        int a1 = (a+n-1) % n, b1 = (b+m-1) % m;
        double C = (P[a] - P[a1]) % (Q[b] - Q[b1]);
        double A = (P[a1] - Q[b]) % (P[a] - Q[b]);
        double B = (Q[b1] - P[a]) % (Q[b] - P[a]);
        Point r;
        if (intersect_1pt(P[a1], P[a], Q[b1], Q[b], r)) {
            if (in == Unknown) aa = ba = 0;
            R.push_back( r );
            in = B > 0 ? Pin : A > 0 ? Qin : in;
        }
        if (C == 0 && B == 0 && A == 0) {
            if (in == Pin) { b = (b + 1) % m; ++ba; }
            else           { a = (a + 1) % m; ++aa; }
        } else if (C >= 0) {
            if (A > 0) { if (in == Pin) R.push_back(P[a]); a = (a+1)%n; ++aa; }
            else       { if (in == Qin) R.push_back(Q[b]); b = (b+1)%m; ++ba; }
        } else {
            if (B > 0) { if (in == Qin) R.push_back(Q[b]); b = (b+1)%m; ++ba; }
            else       { if (in == Pin) R.push_back(P[a]); a = (a+1)%n; ++aa; }
        }
    } while ( (aa < n || ba < m) && aa < 2*n && ba < 2*m );
    if (in == Unknown) {
        if (in_convex(Q, P[0])) return P;
        if (in_convex(P, Q[0])) return Q;
    }
    return R;
}
// Find the diameter of polygon.
// Rotating callipers
double convex_diameter(Polygon pt) {
    const int n = pt.size();
    int is = 0, js = 0;
    for (int i = 1; i < n; ++i) {
        if (pt[i].y > pt[is].y) is = i;
        if (pt[i].y < pt[js].y) js = i;
    }
    double maxd = (pt[is]-pt[js]).norm();
    int i, maxi, j, maxj;
    i = maxi = is;
    j = maxj = js;
    do {
        int jj = j+1; if (jj == n) jj = 0;
        if ((pt[i] - pt[jj]).norm() > (pt[i] - pt[j]).norm()) j = (j+1) % n;
        else i = (i+1) % n;
        if ((pt[i]-pt[j]).norm() > maxd) {
            maxd = (pt[i]-pt[j]).norm();
            maxi = i; maxj = j;
        }
    } while (i != is || j != js);
    return maxd; /* farthest pair is (maxi, maxj). */
}
// Check if we can form triangle with edges x, y, z.
bool isSquare(long long x) { /* */ }
bool isIntegerCoordinates(int x, int y, int z) {
    long long s=(long long)(x+y+z)*(x+y-z)*(x+z-y)*(y+z-x);
    return (s%4==0 && isSquare(s/4));
}
// Pick theorem
// Given non-intersecting polygon.
// S = area
// I = number of integer points strictly Inside
// B = number of points on sides of polygon
// S = I + B/2 - 1
// Smallest enclosing circle:
// Given N points. Find the smallest circle enclosing these points.
// Amortized complexity: O(N)
struct SmallestEnclosingCircle {
    Circle getCircle(vector<Point> points) {
        assert(!points.empty());
        random_shuffle(points.begin(), points.end());
        Circle c(points[0], 0);
        int n = points.size();
        for (int i = 1; i < n; i++)
            if ((points[i] - c).len() > c.r + EPS) {
                c = Circle(points[i], 0);
                for (int j = 0; j < i; j++)
                    if ((points[j] - c).len() > c.r + EPS) {
                        c = Circle((points[i] + points[j]) / 2, (points[i] - points[j]).len() / 2);
                        for (int k = 0; k < j; k++)
                            if ((points[k] - c).len() > c.r + EPS)
                                c = getCircumcircle(points[i], points[j], points[k]);
                    }
            }
        return c;
    }
    // NOTE: This code work only when a, b, c are not collinear and no 2 points are same --> DO NOT
    // copy and use in other cases.
    Circle getCircumcircle(Point a, Point b, Point c) {
        assert(a != b && b != c && a != c);
        assert(ccw(a, b, c));
        double d = 2.0 * (a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y));
        assert(fabs(d) > EPS);
        double x = (a.norm() * (b.y - c.y) + b.norm() * (c.y - a.y) + c.norm() * (a.y - b.y)) / d;
        double y = (a.norm() * (c.x - b.x) + b.norm() * (a.x - c.x) + c.norm() * (b.x - a.x)) / d;
        Point p(x, y);
        return Circle(p, (p - a).len());
    }
};
```

5 Numerical algorithms

5.1 Simplex Algorithm

```
/**
 * minimize c^T * x
 * subject to Ax <= b
 * and x >= 0
 * The input matrix a will have the following form
 * 0 c c c c c
 * b A A A A A
 * b A A A A A
 * b A A A A A
 * Result vector will be: val x x x x x
 **/

typedef long double ld;
const ld EPS = 1e-8;
struct LPSolver {
    static vector<ld> simplex(vector<vector<ld>> a) {
        int n = (int) a.size() - 1;
        int m = (int) a[0].size() - 1;
        vector<int> left(n + 1);
        vector<int> up(m + 1);
        iota(left.begin(), left.end(), m);
        iota(up.begin(), up.end(), 0);
        auto pivot = [&](int x, int y) {
            swap(left[x], up[y]);
            ld k = a[x][y];
            a[x][y] = 1;
            vector<int> pos;
            for (int j = 0; j <= m; j++) {
                a[x][j] /= k;
                if (fabs(a[x][j]) > EPS) pos.push_back(j);
            }
            for (int i = 0; i <= n; i++) {
                if (fabs(a[i][y]) < EPS || i == x) continue;
                k = a[i][y];
                a[i][y] = 0;
                for (int j : pos) a[i][j] -= k * a[x][j];
            }
        };
        while (1) {
            int x = -1;
            for (int i = 1; i <= n; i++) {
                if (a[i][0] < -EPS && (x == -1 || a[i][0] < a[x][0])) {
                    x = i;
                }
            }
            if (x == -1) break;
            int y = -1;
                for (int j = 1; j <= m; j++) {
                    if (a[x][j] < -EPS && (y == -1 || a[x][j] < a[x][y])) {
                    y = j;
                }
            }
            if (y == -1) return vector<ld>(); // infeasible
            pivot(x, y);
        }
        while (1) {
            int y = -1;
            for (int j = 1; j <= m; j++) {
                if (a[0][j] > EPS && (y == -1 || a[0][j] > a[0][y])) {
                    y = j;
                }
            }
            if (y == -1) break;
            int x = -1;
            for (int i = 1; i <= n; i++) {
                if (a[i][y] > EPS && (x == -1 || a[i][0] / a[i][y] < a[x][0] / a[x][y])) {
                    x = i;
                }
            }
            if (x == -1) return vector<ld>(); // unbounded
            pivot(x, y);
        }
        vector<ld> ans(m + 1);
        for (int i = 1; i <= n; i++) {
            if (left[i] <= m) ans[left[i]] = a[i][0];
        }
        ans[0] = -a[0][0];
        return ans;
    }
};
```

5.2 NTT

```
//Poly Invert: R(2n) = 2R(n) - R(n) ^ 2 * F where R(z) = invert F(z)
//Poly Sqrt: 2 * S(2n) = S(n) + F * S(n) ^ -1
const int MOD = 998244353;
struct NTT {
    int base = 1;
    int maxBase = 0;
    int root = 2;
    vector<int> w = {0, 1};
    vector<int> rev = {0, 1};
    NTT () {
        int u = MOD - 1;
        while (u % 2 == 0) {
            u >>= 1;
            maxBase++;
        }
        while (1) {
            if (power(root, 1 << maxBase) == 1 && power(root, 1 << (maxBase - 1)) != 1) {
                break;
            }
            root++;
        }
    }
    void ensure(int curBase) {
        assert(curBase <= maxBase);
        if (curBase <= base) return;
        rev.resize(1 << curBase);
        for (int i = 0; i < (1 << curBase); i++) {
            rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (curBase - 1));
        }
        w.resize(1 << curBase);
        for (; base < curBase; base++) {
            int wc = power(root, 1 << (maxBase - base - 1));
            for (int i = 1 << (base - 1); i < (1 << base); i++) {
                w[i << 1] = w[i];
                w[i << 1 | 1] = mul(w[i], wc);
            }
        }
    }
    void fft(vector<int> &a) {
        int n = a.size();
        int curBase = 0;
        while ((1 << curBase) < n) curBase++;
        int shift = base - curBase;
        for (int i = 0; i < n; i++) {
            if (i < (rev[i] >> shift)) swap(a[i], a[rev[i] >> shift]);
        }
        for (int k = 1; k < n; k <<= 1) {
            for (int i = 0; i < k; i++) {
                for (int j = i; j < n; j += k * 2) {
                    int foo = a[j];
                    int bar = mul(a[j + k], w[i + k]);
                    a[j] = add(foo, bar);
                    a[j + k] = sub(foo, bar);
                }
            }
        }
    }
    vector<int> mult(vector<int> a, vector<int> b) {
        int nResult = a.size() + b.size() - 1;
        int curBase = 0;
        while ((1 << curBase) < nResult) curBase++;
        ensure(curBase);
        a.resize(1 << curBase), b.resize(1 << curBase);
        fft(a);
        fft(b);
        for (int i = 0; i < (1 << curBase); i++) {
            a[i] = mul(mul(a[i], b[i]), inv(1 << curBase));
        }
        reverse(a.begin() + 1, a.end());
        fft(a);
        a.resize(nResult);
        return a;
    }
    vector<int> polyInv(vector<int> r, vector<int> f) {
        vector<int> foo = mult(r, f);
        foo.resize(f.size());
        foo[0] = sub(2, foo[0]);
        for (int i = 1; i < foo.size(); i++) {
            foo[i] = sub(0, foo[i]);
        }
        vector<int> res = mult(r, foo);
        res.resize(f.size());
        return res;
    }
    vector<int> polySqrt(vector<int> s, vector<int> invS, vector<int> f) {
        vector<int> res = mult(f, invS);
        res.resize(f.size());
```

```cpp
        for (int i = 0; i < s.size(); i++) {
            res[i] = add(res[i], s[i]);
        }
        for (int i = 0; i < res.size(); i++) {
            res[i] = mul(res[i], INV_2);
        }
        return res;
    }
    vector<int> getSqrt(vector<int> c, int sz) {
        vector<int> sqrtC = {1}, invSqrtC = {1}; //change this if c[0] != 1
        for (int k = 1; k < (1 << sz); k <<= 1) {
            vector<int> foo(c.begin(), c.begin() + (k * 2));
            vector<int> bar = sqrtC;
            bar.resize(bar.size() * 2, 0);
            vector<int> tempInv = polyInv(invSqrtC, bar);
            sqrtC = polySqrt(sqrtC, tempInv, foo);
            invSqrtC = polyInv(invSqrtC, sqrtC);
        }
        return sqrtC;
    }
    vector<int> getInv(vector<int> c, int sz) {
        vector<int> res = {INV_2}; // change this if c[0] != 2
        for (int k = 1; k < (1 << sz); k <<= 1) {
            vector<int> foo(c.begin(), c.begin() + (k * 2));
            res = polyInv(res, foo);
        }
        return res;
    }
} ntt;
```

5.3 Partition Formula

```cpp
/**
 * generating function : PI: (1 / (1 - x ^ k))
 * p(n)=p(n-1)+p(n-2)-p(n-5)-p(n-7)+p(n-12)+p(n-15)-p(n-22)- ...
 * p_k = k * (3k - 1) / 2 with k = 1, -1, 2, -2, 3, -3, ...
 */
```

6 Graph algorithms

6.1 Bipartite Maximum Matching

```cpp
struct BipartiteGraph {
    vector< vector<int> > a;
    vector<int> match;
    vector<bool> was;
    int m, n;

    BipartiteGraph(int m, int n) {
        // zero-indexed
        this->m = m; this->n = n;
        a.resize(m);
        match.assign(n, -1);
        was.assign(n, false);
    }

    void addEdge(int u, int v) {
        a[u].push_back(v);
    }

    bool dfs(int u) {
        for (int v : a[u]) if (!was[v]) {
            was[v] = true;
            if (match[v] == -1 || dfs(match[v])) {
                match[v] = u;
                return true;
            }
        }
        return false;
    }

    int maximumMatching() {
        vector<int> buffer;
        for (int i = 0; i < m; ++i) buffer.push_back(i);
        bool stop = false;
        int ans = 0;
        do {
            stop = true;
            for (int i = 0; i < n; ++i) was[i] = false;
            for (int i = (int)buffer.size() - 1; i >= 0; --i) {
                int u = buffer[i];
                if (dfs(u)) {
                    ++ans;
                    stop = false;
                    buffer[i] = buffer.back();
                    buffer.pop_back();
                }
            }
        } while (!stop);
        return ans;
    }

    vector<int> konig() {
        // returns minimum vertex cover, run this after maximumMatching()
        vector<bool> matched(m);
        for (int i = 0; i < n; ++i) {
            if (match[i] != -1) matched[match[i]] = true;
        }
        queue<int> Q;
        was.assign(m + n, false);
        for (int i = 0; i < m; ++i) {
            if (!matched[i]) {
                was[i] = true;
                Q.push(i);
            }
        }

        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int v : a[u]) if (!was[m + v]) {
                was[m + v] = true;
                if (match[v] != -1 && !was[match[v]]) {
                    was[match[v]] = true;
                    Q.push(match[v]);
                }
            }
        }

        vector<int> res;
        for (int i = 0; i < m; ++i) {
            if (!was[i]) res.push_back(i);
        }
        for (int i = m; i < m + n; ++i) {
            if (was[i]) res.push_back(i);
        }

        return res;
    }
};
```

6.2 Dinic Flow

```cpp
const int V = 1e5;
const int INF = 1e9;
struct Flow {
    vector<int> adj[V];
    int to[V], c[V], f[V];
    int n, s, t, cnt;
    int d[V];
    int cur[V];
    Flow(int n, int s, int t) {
        this->n = n;
        this->s = s;
        this->t = t;
        cnt = 0;
    }
    int addEdge(int u, int v, int _c) {
        to[cnt] = v, c[cnt] = _c, f[cnt] = 0;
        adj[u].push_back(cnt++);
        to[cnt] = u, c[cnt] = 0, f[cnt] = 0;
        adj[v].push_back(cnt++);
    }
    bool bfs() {
        for (int i = 0; i < n; i++) d[i] = -1;
        d[s] = 0;
        queue<int> q;
        q.push(s);
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int id : adj[u]) {
                int v = to[id];
                if (d[v] == -1 && f[id] < c[id]) {
                    d[v] = d[u] + 1;
                    q.push(v);
                }
            }
        }
```

```
        return d[t] != -1;
    }
    int dfs(int u, int res) {
        if (u == t) return res;
        for (int &it = cur[u]; it < adj[u].size(); it++) {
            int id = adj[u][it];
            int v = to[id];
            if (d[v] == d[u] + 1 && f[id] < c[id]) {
                int foo = dfs(v, min(c[id] - f[id], res));
                if (foo) {
                    f[id] += foo;
                    f[id ^ 1] -= foo;
                    return foo;
                }
            }
        }
        return 0;
    }
    int maxFlow() {
        int res = 0;
        while (bfs()) {
            for (int i = 0; i < n; i++) cur[i] = 0;
            while (1) {
                int foo = dfs(s, INF);
                if (!foo) break;
                res += foo;
            }
        }
        return res;
    }
};
```

6.3 Min Cost-Max Flow

```
struct Flow {
    static const int V = 100000;
    int head[V], to[V], c[V], cost[V], f[V], nxt[V], h[V], par[V], inQueue[V];
    int s, t, n, cnt;
    queue <int> q;
    Flow (int n, int s, int t) {
        this->n = n;
        this->s = s;
        this->t = t;
        cnt = 0;
        for (int i=  0; i < n; i++) {
            head[i] = -1;
            inQueue[i] = 0;
        }
    }
    int addEdge(int u, int v, int _c, int _cost) {
        to[cnt] = v, c[cnt] = _c, cost[cnt] = _cost, f[cnt] = 0, nxt[cnt] = head[u], head[u] = cnt++;
        to[cnt] = u, c[cnt] = 0, cost[cnt] = -_cost, f[cnt] = 0, nxt[cnt] = head[v], head[v] = cnt++;
        return cnt - 2;
    }
    pair<int, int> maxFlow () {
        int res = 0, minCost = 0;
        while (1) {
            for (int i = 0; i < n; i++) {
                par[i] = -1;
                h[i] = 2e9;
            }
            h[s] = 0;
            q.push(s);
            inQueue[s] = 1;
            while (!q.empty()) {
                int u = q.front();
                q.pop();
                inQueue[u] = 0;
                for (int id = head[u]; id != -1; id = nxt[id]) {
                    int v = to[id];
                    if (h[v] > h[u] + cost[id] && f[id] < c[id]) {
                        h[v] = h[u] + cost[id];
                        par[v] = id;
                        if (!inQueue[v]) {
                            inQueue[v] = 1;
                            q.push(v);
                        }
                    }
                }
            }
            if (par[t] == -1) {
                break;
            }
            int x = t;
            int now = 2e9;
            while (x != s) {
                int id = par[x];
                now = min(now, c[id] - f[id]);
                x = to[id ^ 1];
            }
            x = t;
            while (x != s) {
                int id = par[x];
                minCost += cost[id] * now;
                f[id] += now;
                f[id ^ 1] -= now;
                x = to[id ^ 1];
            }
            res += now;
        }
        return make_pair(res, minCost);
    }
};
```

6.4 Bounded Feasible Flow

```
struct BoundedFlow {
    int low[N][N], high[N][N];
    int c[N][N];
    int f[N][N];
    int n, s, t;

    void reset() {
        memset(low, 0, sizeof low);
        memset(high, 0, sizeof high);
        memset(c, 0, sizeof c);
        memset(f, 0, sizeof f);
        n = s = t = 0;
    }
    void addEdge(int u, int v, int d, int c) {
        low[u][v] = d; high[u][v] = c;
    }

    int flow;
    int trace[N];

    bool findPath() {
        memset(trace, 0, sizeof trace);
        queue<int> Q;
        Q.push(s);
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int v = 1; v <= n; ++v) if (c[u][v] > f[u][v] && !trace[v]) {
                trace[v] = u;
                if (v == t) return true;
                Q.push(v);
            }
        }
        return false;
    }

    void incFlow() {
        int delta = INF;
        for (int v = t; v != s; v = trace[v])
            delta = min(delta, c[trace[v]][v] - f[trace[v]][v]);
        for (int v = t; v != s; v = trace[v])
            f[trace[v]][v] += delta, f[v][trace[v]] -= delta;
        flow += delta;
    }

    int maxFlow() {
        flow = 0;
        while (findPath()) incFlow();
        return flow;
    }
    bool feasible() {
        c[t][s] = INF;
        s = n + 1; t = n + 2;
        int sum = 0;
        for (int u = 1; u <= n; ++u) for (int v = 1; v <= n; ++v) {
            c[s][v] += low[u][v];
            c[u][t] += low[u][v];
            c[u][v] += high[u][v] - low[u][v];
            sum += low[u][v];
        }
        n += 2;
        return maxFlow() == sum;
    }
};
```

6.5 Hungarian Algorithm

```cpp
struct BipartiteGraph {
    const int INF = 1e9;

    vector<vector<int> > c; // cost matrix
    vector<int> fx, fy; // potentials
    vector<int> matchX, matchY; // corresponding vertex
    vector<int> trace; // last vertex from the left side
    vector<int> d, arg; // distance from the tree && the corresponding node
    queue<int> Q; // queue used for BFS

    int n; // assume that |L| = |R| = n
    int start; // current root of the tree
    int finish; // leaf node of the augmenting path

    BipartiteGraph(int n) {
        this->n = n;
        c = vector<vector<int> >(n + 1, vector<int>(n + 1, INF));
        fx = fy = matchX = matchY = trace = d = arg = vector<int>(n + 1);
    }

    void addEdge(int u, int v, int cost) { c[u][v] = min(c[u][v], cost); }
    int cost(int u, int v) { return c[u][v] - fx[u] - fy[v]; }

    void initBFS(int root) {
        start = root;
        Q = queue<int>(); Q.push(start);
        for (int i = 1; i <= n; ++i) {
            trace[i] = 0;
            d[i] = cost(start, i);
            arg[i] = start;
        }
    }

    int findPath() {
        while (!Q.empty()) {
            int u = Q.front(); Q.pop();
            for (int v = 1; v <= n; ++v) if (trace[v] == 0) {
                int w = cost(u, v);
                if (w == 0) {
                    trace[v] = u;
                    if (matchY[v] == 0) return v;
                    Q.push(matchY[v]);
                }
                if (d[v] > w) d[v] = w, arg[v] = u;
            }
        }
        return 0;
    }

    void enlarge() {
        for (int y = finish, next; y; y = next) {
            int x = trace[y];
            next = matchX[x];
            matchX[x] = y;
            matchY[y] = x;
        }
    }

    void update() {
        int delta = INF;
        for (int i = 1; i <= n; ++i) if (trace[i] == 0) delta = min(delta, d[i]);
        fx[start] += delta;
        for (int i = 1; i <= n; ++i) {
            if (trace[i] != 0) {
                fx[matchY[i]] += delta;
                fy[i] -= delta;
            } else {
                d[i] -= delta;
                if (d[i] == 0) {
                    trace[i] = arg[i];
                    if (matchY[i] == 0)
                        finish = i;
                    else
                        Q.push(matchY[i]);
                }
            }
        }
    }

    void hungarian() {
        for (int i = 1; i <= n; ++i) {
            initBFS(i);
            do {
                finish = findPath();
                if (finish == 0) update();
            } while (finish == 0);
            enlarge();
        }
    }

    void show() {
        int ans = 0;
        for (int i = 1; i <= n; ++i) if (matchX[i]) ans += c[i][matchX[i]];
        cout << ans << endl;
        for (int i = 1; i <= n; ++i) cout << i << ' ' << matchX[i] << endl;
    }
};
```

7 Data structures

8 Miscellaneous