

Contents	1.1 GNU PBDS 1	2.1 Geometry Library . 1	3.1 Max Flow Dinic . . . 3	4 String 4
1 STL Useful Tips 1	2 Geometry 1	3 Graph 3	3.2 Bipartite Matching Hopcroft Karp . . . 4	4.1 Suffix Array 4

ACM ICPC Cheat Sheet

Fairuzi10

1 STL Useful Tips

1.1 GNU PBDS

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

// change null_type to int to make it a map<int, int>
typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update> ordered_set;
```

2 Geometry

2.1 Geometry Library

```
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 circumcenter 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));
}

// compute incenter of circle given three points
PT ComputeInCenter(PT a, PT b, PT c) {
    double x = hypot(b.x-c.x, b.y-c.y);
    double y = hypot(a.x-c.x, a.y-c.y);

```

```

    double z = hypot(a.x-b.x, a.y-b.y);

    double rx = x*a.x+y*b.x+z*c.x;
    double ry = x*a.y+y*b.y+z*c.y;
    return PT(rx, ry)/(x+y+z);
}

// check ccw & count angle
bool ccw(PT p, PT q, PT r) { return cross(PT(p, q), PT(p, r)) > 0; }
double angle(PT a, PT o, PT b) { // return AOB in rad
    PT oa = PT(o, a), ob = PT(o, b);
    return acos(dot(oa, ob)/sqrt(dist2(oa, PT(0, 0))*dist2(ob, PT(0,
        → 0))));
}

// test point in polygon from sum of angle
bool PointInPolygon(const vector<PT> &p, PT q) {
    double sum = 0;
    for (int i = 0; i < (int) p.size(); i++) {
        if (ccw(q, p[i], p[i+1]))
            sum += angle(p[i], q, p[(i+1)%p.size()]);
        else
            sum -= angle(p[i], q, p[(i+1)%p.size()]);
    }

    return fabs(fabs(sum) - 2*PI) < EPS;
}

// 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;
}

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;
}

```

3 Graph

3.1 Max Flow Dinic

```

// run in O(V^2*E)
bool bfs() {
    fill_n(lvl, MAXN, INF);
    lvl[SOURCE] = 0;
    q.push(SOURCE);
    while (!q.empty()) {
        int now = q.front();
        q.pop();
        if (lvl[now]+1 > lvl[SINK]) continue;
        for (auto i: edge[now]) {
            if (lvl[now]+1 < lvl[i] && rem[now][i]) {
                lvl[i] = lvl[now]+1;
                q.push(i);
            }
        }
    }
    return lvl[SINK] != INF;
}

int dfs(int now, int cur_flow) {
    if (now == SINK) return cur_flow;
    int used_flow = 0;
    for (auto i: edge[now]) {
        if (lvl[i] == lvl[now]+1 && rem[now][i]) {
            int next_flow = dfs(i, min(rem[now][i], cur_flow-used_flow));
            used_flow += next_flow;
            rem[now][i] -= next_flow;
            rem[i][now] += next_flow;
            if (used_flow == cur_flow) return used_flow;
        }
    }
    return used_flow;
}

// in main()
while (bfs()) {
    ans += dfs(SOURCE, INF);
}

```

```
}
```

3.2 Bipartite Matching Hopcroft Karp

```
// run in O(E*sqrt(V))
bool bfs() {
    memset(lvl, 63, sizeof(lvl));
    for (int i = 0; i < N; i++) {
        if (pairL[i] == NIL) {
            lvl[i] = 0;
            q.push(i); }
    while (!q.empty()) {
        int now = q.front(); q.pop();
        for (auto i: edge[now]) {
            if (lvl[pairR[i]] > lvl[now]+1) {
                lvl[pairR[i]] = lvl[now]+1;
                q.push(pairR[i]);
            }
        }
    }
    return lvl[NIL] < INF;
}

bool dfs(int now) {
    if (now == NIL) return 1;
    for (auto i: edge[now]) {
        if (lvl[pairR[i]] == lvl[now]+1) {
            if (dfs(pairR[i])) {
                pairL[now] = i;
                pairR[i] = now;
                return 1;
            } else lvl[pairR[i]] = INF;
        }
    }
    return 0;
}

int bipartite_matching() {
    for (int i = 0; i < N; i++) pairL[i] = NIL;
    for (int i = 0; i < M; i++) pairR[i] = NIL;
    int ret = 0;
    while (bfs()) {
        for (int i = 0; i < N; i++) {
            if (lvl[i] == 0) {
```

```
                if (dfs(i)) ret++;
                else lvl[i] = INF;
            }
        }
    }
    return ret;
}
```

4 String

4.1 Suffix Array

```
// change sa[i]+k to (sa[i]+k)%N if cyclic
// change sa[i]+k to min(sa[i]+k, sa[i]+limit-k) and k < N to k < limit
→ if the substring length is limited.

void radix(int k) {
    int maxi = max(300, N);
    memset(cnt, 0, sizeof(cnt));
    for (int i = 0; i < N; i++) cnt[sa[i]+k < N? ra[sa[i]+k]: 0]++;
    for (int i = 0; i < maxi; i++) cnt[i] += cnt[i-1];
    for (int i = N-1; i >= 0; i--) tsa[--cnt[sa[i]+k < N? ra[sa[i]+k]: 0]]
        = sa[i];
    for (int i = 0; i < N; i++) sa[i] = tsa[i];
}

void compute_sa() {
    for (int i = 0; i < N; i++) ra[i] = str[i];
    for (int i = 0; i < N; i++) sa[i] = i;
    for (int k = 1; k < N; k <= 1) {
        radix(k);
        radix(0);
        tra[sa[0]] = 0;
        for (int i = 1; i < N; i++)
            tra[sa[i]] = ra[sa[i]] == ra[sa[i-1]] &&
                ra[sa[i]+k] == ra[sa[i-1]+k]? tra[sa[i-1]]: tra[sa[i-1]]+1;
        for (int i = 0; i < N; i++) ra[i] = tra[i];
        if (ra[sa[N-1]] == N-1) break;
    }
}

void compute_lcp() {
```

```

int cur_lcp = 0;
for (int i = 0; i < N; i++) {
    if (ra[i] == 0) {
        lcp[i] = 0;
        continue;
    }
    cur_lcp = max(0, cur_lcp-1);
    while (str[i+cur_lcp] == str[sa[ra[i]-1]+cur_lcp]) cur_lcp++;
    lcp[i] = cur_lcp;
}

// in main()
str += '$' // if not cyclic
N = str.length()

```