Dysfunctional Programmer's ICPC Notebook

Yaseen Mowzer   Broson Rudner   David Broodryk

March 24, 2019

Contents

1 Tricks

1.1 Runtime Error Checklist

- Did you check for **formatting errors**?
- Are you using **long longs**?
- Did you check for **stack overflow**?
- Does the recursive function have a **Base case**?
- Are you **dividing by zero**?
- Is there **enough memory**?

1.2 Makefile

```
CXXFLAGS:=-std=c++14 -g -Wall -Wno-parentheses -Wextra -Winline -fsanitize=address,undefined
OFILES:=$(subst .cpp,,$(wildcard *.cpp))
.PHONY: all
all: $(OFILES)
```

1.3 Printing Floats

```
void print_fixed(double x, int pr)  {
    cout.precision(pr);
    cout << fixed << x;
}
```

1.4 Python tester

```python
#!/usr/bin/env python3

from random import randint
from subprocess import Popen, PIPE

while True:
    N = 10
    mountains = [randint(0, 10) for i in range(N)]
    instring = ' '.join(str(x) for x in [N] + mountains).encode('utf-8')

    p1 = Popen('./mountains', stdin=PIPE, stdout=PIPE)
    output1 = p1.communicate(instring)
    p2 = Popen('./mountains.3', stdin=PIPE, stdout=PIPE)
    output2 = p2.communicate(instring)
    if output1 != output2:
        print(instring)
        break
    else:
        print('YES')
```

2 Segment Trees

2.1 Eager

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

using namespace std;

struct node {
    int i;

    node() {
        i = 0;
    }

    node(int i) {
        this->i = i;
    }
    x
    node operator+(node other) {
        return node(i + other.i);
    }
};

struct sgt {

    int n;
    vector<node> a;

    sgt(int n) {
        this->n = n;
        a.resize(n<<1);
    }

    void assign(int i, node value) {
        a[n+i] = value;
    }

    void build() {
        for (int i = n - 1; i > 0; --i) a[i] = a[i<<1] + a[i<<1|1];
    }
    void update(int i, node value) {
        for (a[i += n] = value; i > 1; i >>=1 ) a[i>>1] = a[i] + a[i^1];
    }

    node query(int l, int r) {
        node ret;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (l&1) ret = ret + a[l++];
            if (r&1) ret = ret + a[--r];
        }
        return ret;
    }
};



int main() {
    sgt s(5);
    s.assign(0, node(3));
    s.assign(1, node(4));
    s.assign(2, node(2));
    s.assign(3, node(5));
    s.assign(4, node(11));
    s.build();
    cout << s.query(0,3).i << endl;
    cout << s.query(2,4).i << endl;
    s.update(2,node(-1));
    cout << s.query(0,3).i << endl;
    return 0;
}
```

2.2 Lazy

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

using namespace std;
using ll = long long;

int pow2(int x) {
    int l = 1;
    while (l < x) l <<= 1;
    return l;
}

template<typename T, typename U>
struct lazy_sgt {
    using op = T (*)(T, T);
    struct node {
        T val;
        bool ovr = false;
        U upd;
    };
    using ov = void (*)(node &, U, int);

    const int n;
    vector<node> tree;
    const T id;
    const op merge;
    const ov overwrite;

    lazy_sgt(int n, T id, op m, ov o)
        : n{pow2(n)}, tree{2 * this->n + 1}, id{id}, merge{m}, overwrite{o} {}

    // updates [a, b) with val
    void update(int a, int b, U upd) {
        update(1, a, b, upd, 0, n);
    }

    // queries [a, b)
    T query(int a, int b) {
        return query(1, a, b, 0, n);
    }

    void update(int i, int a, int b, U upd, int left, int right) {
        if (b <= left || a >= right) return; // doesn't intersect
        if (right - left == 1) overwrite(tree[i], upd, right - left);
        else if (a <= left && right <= b) { // wholly contained
            overwrite(tree[i], upd, right - left);
        } else { // intersects
            int mid = left + (right - left) / 2;
            if (tree[i].ovr == true) {
                overwrite(tree[2 * i], tree[i].upd, mid - left);
                overwrite(tree[2 * i + 1], tree[i].upd, right - mid);
                tree[i].ovr = false;
            }
            update(2 * i, a, b, upd, left, mid);
            update(2 * i + 1, a, b, upd, mid, right);
            tree[i].val = merge(tree[2 * i].val, tree[2 * i + 1].val);
        }
    }

    T query(int i, int a, int b, int left, int right) {

        if (b <= left || a >= right) {return id;} // doesn't intersect
        if (right - left == 1) { return tree[i].val;}
        else if (a <= left && right <= b) { // wholly contained
            return tree[i].val;
        } else {
            int mid = left + (right - left) / 2;
            if (tree[i].ovr == true) {
                overwrite(tree[2 * i], tree[i].upd, mid - left);
                overwrite(tree[2 * i + 1], tree[i].upd, right - mid);
                tree[i].ovr = false;
            }
            T l = query(2 * i, a, b, left, mid);
            T r = query(2 * i + 1, a, b, mid, right);
            return merge(l, r);
        }
    }
};

int main() {
    ios::sync_with_stdio(false);
    // Solution to https://www.spoj.com/problems/HORRIBLE/
    using lsgt = lazy_sgt<ll, ll>;
    lsgt::op plus =
        [] (ll a, ll b) {
            return a + b;
        };
    lsgt::ov upd =
        [] (lsgt::node &a, ll u, int width) {
            a.val += u * width;
            if (a.ovr == false) {
                a.upd = u;
            } else {
                a.upd += u;
            }
            a.ovr = true;
        };

    int T;
    cin >> T;
    for (int testcase = 0; testcase < T; ++testcase) {
        int N, C;
        cin >> N >> C;
        lsgt tree(N, 0ll, plus, upd);
        for (int i = 0; i < C; ++i) {
            int c, a, b, v;
            cin >> c >> a >> b;
            if (c == 0) {
                cin >> v;
                tree.update(a - 1, b, (ll)v);
            } else {
                cout << tree.query(a - 1, b) << "\n";
            }
        }
    }
    return 0;
}
```

3 Numerical Integration

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

using namespace std;

double f(double x) {
        return pow(M_E, -x * x); // dummy
}

// error = - (b - a)^3/(12N^2) * f''(c) for some c in [a, b].
double integrate(double a, double b, int n) {
        double res = 0.5 * f(a) + 0.5 * f(b);
        for (int i = 1; i < n; ++i) {
                res += f(a + (b - a) * i / n);
        }
        return res * (b - a) / n;
}

// Useful for binary searches
bool accurate_to(int decimal_places, double high, double low) {
    double p = pow(10, decimal_places);
    return round(high * p) == round(low * p);
}

int main() {
        cout << fixed << setprecision(10) << integrate(-10,10,10000) << '\n';
        cout << fixed << setprecision(10) << integrate(-10,10,1000) << '\n';
        cout << fixed << setprecision(10) << integrate(-10,10,100) << '\n';
        cout << fixed << setprecision(10) << integrate(-10,10,10) << '\n';
}
```

3.1 Simsons

$$\int_a^b P(x)\,dx = \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] - \frac{1}{90}\left(\frac{b-a}{2}\right)^5 f^{(4)}(\xi)$$

Composite error: $-(h^4/180)(b-a)f^{(4)}(\xi)$

4 Geometry

```cpp
// Mostly copied from standford ICPC notebook

#include <bits/stdc++.h>

using ll = long long;

using namespace std;

// If the algorithm is exact ll is used. If it is inexact double is used.
// If it is both then T is used.

// Tips for geometry
// 1 Remember special cases
// 1.1 coincident (e.g. point on line)
// 1.2 colinear
// 1.3 Paralell lines (no intersection or gradient, division by zero)
// 2. Remember when multiplying inequalities about possibilites of negatives
// 3. Use <= for EPS to allow EPS to be 0.
// 4. Avoid gradient use cross product instead


// BEGIN TESTED GEOMETRY

// If dealing with integers make EPS 0
#define EPS 0
#define EPSD 1e-12
#define INF LLONG_MAX

// b in [a, c).
template<typename T> bool between(T a, T b, T c) { return a <= b && b < c; }
// b in [a, c].
template<typename T> bool betweenincl(T a, T b, T c) { return a <= b && b <= c; }
// b in [a, c] or [c, a]
template<typename T> bool betweeneither(T a, T b, T c) { return between(a, b, c) || between(c, b, a); }


// On complex number we already have
// abs, norm, arg, conj and polar
template<typename T> using pt = complex<T>;

const pt<double> Idb = pt<double>(0.0, 1.0);
const pt<ll> Ill = pt<ll>(0ll, 1ll);

// standards compliant magic!
template<typename T> T& x(pt<T> &p) {
    return reinterpret_cast<T*>(&p)[0];
}
template<typename T> T& y(pt<T> &p) {
    return reinterpret_cast<T*>(&p)[1];
}

template<typename T> T dot(const pt<T> a, const pt<T> b) {
    return real(conj(a) * b);
}
template<typename T> T cross(pt<T> a, pt<T> b) {
    return imag(conj(a) * b);
}

// returns true if p is strictly inside the rectangle with corners a and b
// returns false if p is strictly outside
// returns true or false otherwise
template<typename T> bool pt_in_rect(pt<T> p, pt<T> a, pt<T> b) {
    return betweeneither(x(a), x(p), x(b)) && betweeneither(y(a), y(p), y(b));
}

// determine if lines from a to b and c to d are parallel or collinear
// n^2 overflow
template <typename T> bool lines_parallel(pt<T> a, pt<T> b, pt<T> c, pt<T> d) {
    return abs(cross(b-a, c-d)) <= EPS;
}

// determine if point is in a possibly non-convex polygon (by William Randolph
// Franklin modified by Yaseen Mowzer); returns true for strictly interior
// points, false for strictly exterior points, and false or true for the
// remaining points.
// n^2 overflow
template<typename T>
bool pt_in_pgon(pt<T> q, vector<pt<T>> &p) {
    bool c = false;
    for (int i = 0; i < (int) p.size(); i++) {
        pt<T> a = p[i], b = p[(i + 1) % p.size()];
        if (y(a) > y(b)) swap(a, b);
        pt<T> d = b - a;
        if (between(y(a), y(q), y(b))
            && x(q) * y(d) < x(a) * y(d) + x(d) * (y(q) - y(a))) {
            c = !c;
        }
    }
    return c;
}

// determine if point is on the boundary of a polygon
// n^2 overflow
bool pt_on_pgon(pt<ll> q, vector<pt<ll>> &p) {
    for (int i = 0; i < (int) p.size(); i++) {
        pt<ll> a = p[i], b = p[(i + 1) % p.size()];
        if (norm(a - q) <= EPS) return true; // Coincident with a
        if (x(a) < x(b)) swap(a, b);
        if (pt_in_rect(q, a, b)) {
            if (lines_parallel(a, b, a, q)) {
                return true;
            }
        }

    }
    return false;
}

int other_main()
{
    ios::sync_with_stdio(false);
    vector<pt<double>> points {{-1, -1}, {1, -2}, {2, 0}, {2, 2}, {-1, 1}};
    assert(pt_in_pgon({0, 0}, points) == true);
    assert(pt_in_pgon({-2, 0}, points) == false);
    assert(pt_in_pgon({0, 2}, points) == false);
    assert(pt_in_pgon({0, -2}, points) == false);

    // https://codeforces.com/contest/1030/problem/B
    int N, D, M;
    cin >> N >> D;
    vector<pt<ll>> rect {{0, D}, {D, 0}, {N, N - D}, {N - D, N}};
    cin >> M;
    for (int i = 0; i < M; ++i) {
        pt<ll> p;
        cin >> x(p) >> y(p);
        cout << ((pt_on_pgon(p, rect) || pt_in_pgon(p, rect)) ? "YES\n" : "NO\n");
    }
    return 0;
}

// END TESTED GEOMETRY

// project point z onto line through a and b
// assuming a != b (if a == b division by zero will occur).
pt<double> proj_pt_on_line(pt<double> a, pt<double> b, pt<double> z) {
    return a + (b-a)*dot(z-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
pt<double> proj_pt_on_seg(pt<double> a, pt<double> b, pt<double> 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 between point (x,y,z) and plane ax+by+cz=d
double dist_pt_plane(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);
}


template<typename T> bool lines_collinear(pt<T> a, pt<T> b, pt<T> c, pt<T> d) {
    return lines_parallel(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
template<typename T>
bool seg_intersect(pt<T> a, pt<T> b, pt<T> c, pt<T> d) {
    if (lines_collinear(a, b, c, d)) {
        if (norm(a - c) <= EPS || norm(a - d) <= EPS ||
            norm(b - c) <= EPS || norm(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<double> line_intersection(pt<double> a, pt<double> b, pt<double> c, pt<double> 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<double> circle_center(pt<double> a, pt<double> b, pt<double> c) {
    b=(a+b)/2.0;
    c=(a+c)/2.0;
    return line_intersection(b, b+(a-b)*(-Idb), c, c+(a-c)*(-Idb));
}

// determine if point is on the boundary of a polygon
bool pt_on_pgon(vector<pt<double>> &p, pt<double> q) {
    for (int i = 0; i < p.size(); i++)
        if (norm(proj_pt_on_seg(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
// WARNING: FAILS TESTS!
vector<pt<double>> circle_line_intersection(pt<double> a, pt<double> b, pt<double> c, double r) {
    vector<pt<double>> 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 < -EPSD) return ret;
    ret.push_back(c+a+b*(-B+sqrt(D+EPSD))/A);
    if (D > EPSD)
        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<double>> circle_circle_intersection(pt<double> a, pt<double> b, double r, double R) {
    vector<pt<double>> ret;
    double d = sqrt(norm(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<double> v = (b-a)/d;
    ret.push_back(a+v*x + v*Idb*y);
    if (y > 0)
        ret.push_back(a+v*x - v*Idb*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 signed_area(vector<pt<double>> &p) {
    double area = 0;
    for(int i = 0; i < p.size(); i++) {
        int j = (i + 1) % p.size();
        area += cross(p[i], p[j]);
    }
    return area / 2.0;
}

double area(vector<pt<double>> &p) {
    return fabs(signed_area(p));
}

pt<double> centroid(vector<pt<double>> &p) {
```

```cpp
    pt<double> c(0.0 , 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])*cross(p[i], p[j]);
    }
    return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
template<typename T>
bool is_simple(const vector<pt<T>> &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 (seg_intersect(p[i], p[j], p[k], p[l]))
                return false;
        }
    }
    return true;
}

int secondmain() {
    using PT = pt<double>;
    // expected: (-5,2)
    cerr << pt<ll>(2,5)*Ill << endl;

    // expected: (5,-2)
    cerr << pt<ll>(2,5)*(-Ill) << endl;

    // expected: (-5,2)
    cerr << pt<double>(2,5) * polar<double>(1.0, M_PI/2) << endl;

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

    // expected: (5,2) (7.5,3) (2.5,1)
    cerr << proj_pt_on_seg(PT(-5,-2), PT(10,4), PT(3,7)) << " "
         << proj_pt_on_seg(PT(7.5,3), PT(10,4), PT(3,7)) << " "
         << proj_pt_on_seg(PT(-5,-2), PT(2.5,1), PT(3,7)) << endl;

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

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

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

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

    // expected: (1,2)
    cerr << line_intersection(PT(0,0), PT(2,4), PT(3,1), PT(-1,3)) << endl;
    // expected: (1,1)
    cerr << circle_center(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 << pt_in_pgon(PT(2,2), v) << " "
         << pt_in_pgon(PT(2,0), v) << " "
         << pt_in_pgon(PT(0,2), v) << " "
         << pt_in_pgon(PT(5,2), v) << " "
         << pt_in_pgon(PT(2,5), v) << endl;

    // expected: 0 1 1 1 1
    cerr << pt_on_pgon(v, PT(2,2)) << " "
         << pt_on_pgon(v, PT(2,0)) << " "
         << pt_on_pgon(v, PT(0,2)) << " "
         << pt_on_pgon(v, PT(5,2)) << " "
         << pt_on_pgon(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 = circle_line_intersection(PT(0,6), PT(2,6), PT(1,1), 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = circle_line_intersection(PT(0,9), PT(9,0), PT(1,1), 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = circle_line_intersection(PT(1,1), PT(10,10), 5, 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = circle_line_intersection(PT(1,1), PT(8,8), 5, 5);
    for (int i = 0; i < u.size(); i++) cerr << u[i] << " "; cerr << endl;
    u = circle_line_intersection(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 = circle_line_intersection(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, 1.166666)
    PT pa[] = { PT(0,0), PT(5,0), PT(1,1), PT(0,5) };
    vector<PT> p(pa, pa+4);
    PT c = centroid(p);
    cerr << "Area: " << area(p) << endl;
    cerr << "Centroid: " << c << endl;

    return 0;
}
```

4.1 Convex Hull

```
// Mostly copied from stanford ICPC notebook

// Compute the 2D convex hull of a set of points using the monotone chain
// algorithm.  Eliminate redundant points from the hull if REMOVE_REDUNDANT is
// #defined.
//
// Running time: O(n log n)
//
```

```cpp
//   INPUT:   a vector of input points, unordered.
//   OUTPUT:  a vector of points in the convex hull, counterclockwise, starting
//            with bottommost/leftmost point

#include <bits/stdc++.h>
#include "geom.cpp"

using namespace std;

using PT = pt<double>;

//#define REMOVE_REDUNDANT

double area2(PT a, PT b, PT c) {
    vector<PT> s = {a, b, c};
    return signed_area(s);
}

bool operator<(PT a, PT b) {
    return make_pair(y(a), x(a)) < make_pair(y(b), x(b));
}
bool operator<=(PT a, PT b) {
    return make_pair(y(a), x(a)) <= make_pair(y(b), x(b));
}

#ifdef REMOVE_REDUNDANT
template <typename T> bool between(PT a, PT b, PT c) {
    return (fabs(area2(a,b,c)) <= EPS && (x(a)-x(b))*(x(c)-x(b)) <= 0 && (y(a)-y(b))*(y(c)-y(b)) <= 0);
}
#endif

void ConvexHull(vector<PT> &pts) {
    sort(pts.begin(), pts.end(), [] (PT a, PT b) {return a < b;});
    pts.erase(unique(pts.begin(), pts.end()), pts.end());
    vector<PT> up, dn;
    for (int i = 0; i < pts.size(); i++) {
        while (up.size() > 1 && area2(up[up.size()-2], up.back(), pts[i]) >= 0) up.pop_back();
        while (dn.size() > 1 && area2(dn[dn.size()-2], dn.back(), pts[i]) <= 0) dn.pop_back();
        up.push_back(pts[i]);
        dn.push_back(pts[i]);
    }
    pts = dn;
    for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);

#ifdef REMOVE_REDUNDANT
    if (pts.size() <= 2) return;
    dn.clear();
    dn.push_back(pts[0]);
    dn.push_back(pts[1]);
    for (int i = 2; i < pts.size(); i++) {
        if (between(dn[dn.size()-2], dn[dn.size()-1], pts[i])) dn.pop_back();
        dn.push_back(pts[i]);
    }
    if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
        dn[0] = dn.back();
        dn.pop_back();
    }
    pts = dn;
#endif
}

// BEGIN CUT
// The following code solves SPOJ problem #26: Build the Fence (BSHEEP)
```

```
struct cmp {
    bool operator()(PT a, PT b) {
        return a < b;
    }
};

int main() {
    int t;
    scanf("%d", &t);
    for (int caseno = 0; caseno < t; caseno++) {
        int n;
        scanf("%d", &n);
        vector<PT> v(n);
        for (int i = 0; i < n; i++) scanf("%lf%lf", &x(v[i]), &y(v[i]));
        vector<PT> h(v);
        map<PT, int, cmp> index;
        for (int i = n-1; i >= 0; i--) index[v[i]] = i+1;
        ConvexHull(h);

        double len = 0;
        for (int i = 0; i < h.size(); i++) {
            double dx = x(h[i]) - x(h[(i+1)%h.size()]);
            double dy = y(h[i]) - y(h[(i+1)%h.size()]);
            len += sqrt(dx*dx+dy*dy);
        }

        if (caseno > 0) printf("\n");
        printf("%.2f\n", len);
        for (int i = 0; i < h.size(); i++) {
            if (i > 0) printf(" ");
            printf("%d", index[h[i]]);
        }
        printf("\n");
    }
}

// END CUT
```

5 Max-Flow

```
O(VE^2)

algorithm EdmondsKarp
    input:
        graph   (graph[v] should be the list of edges coming out of vertex v in the
                 original graph and their corresponding constructed reverse edges
                 which are used for push-back flow.
                 Each edge should have a capacity, flow, source and sink as parameters,
                 as well as a pointer to the reverse edge.)
        s       (Source vertex)
        t       (Sink vertex)
    output:
        flow    (Value of maximum flow)

    flow := 0   (Initialize flow to zero)
    repeat
        (Run a bfs to find the shortest s-t path.
         We use 'pred' to store the edge taken to get to each vertex,
         so we can recover the path afterwards)
        q := queue()
        q.push(s)
        pred := array(graph.length)
        while not empty(q)
            cur := q.pull()
            for Edge e in graph[cur]
                 if pred[e.t] = null and e.t != s and e.cap > e.flow
                    pred[e.t] := e
                    q.push(e.t)

        if not (pred[t] = null)
            (We found an augmenting path.
             See how much flow we can send)
            df := infinity
            for (e := pred[t]; e != null; e := pred[e.s])
                df := min(df, e.cap - e.flow)
            (And update edges by that amount)
            for (e := pred[t]; e != null; e := pred[e.s])
                e.flow  := e.flow + df
                e.rev.flow := e.rev.flow - df
            flow := flow + df

    until pred[t] = null  (i.e., until no augmenting path was found)
    return flow
```

```cpp
// Max-Flow copied from seanwentzel/icpc-notebook

#include <bits/stdc++.h>

const long long INF = 1<<20;

typedef std::pair<int, int> pii;
typedef std::pair<int, long long> pil;

class PairHasher {
public:
    std::size_t operator() (const pii &p) const {
        return std::hash<int>()(p.first) ^ std::hash<int>()(p.second);
    }
};

void print(std::unordered_map<pii, long long, PairHasher> flow) {
    for (auto it = flow.begin(); it != flow.end(); it++) {
        std::cout<<it->first.first<<" "<<it->first.second<<": "<<it->second<<std::endl;
    }
}

void print(std::vector<int> path) {
    std::cout<<"path ";
    for (int i=0; i<path.size(); i++) {
        std::cout<<path[i]<<" ";
    }
    std::cout<<std::endl;
}

// BFS from source to sink and find a path that has positive capacity
std::pair<std::vector<int>, long long> findpath(
    std::vector<std::vector<pil> > &graph,
    std::unordered_map<pii, long long, PairHasher> &flow,
    int source,
    int sink) {

    // so that we can reverse the path later
    int previous[graph.size()];
    long long previousCapacity[graph.size()];
    bool visited[graph.size()];
    for(int i=0; i<graph.size(); i++) {
        previous[i] = -1;
        visited[i] = false;
    }
    pii coord;
    std::queue<int> q;
    q.push(source);
    bool found = false;
    while(!q.empty()) {
        int cur = q.front();
        q.pop();
        coord.first = cur;

        for (int i=0; i<graph[cur].size(); i++) {
            int next = graph[cur][i].first;
            long long capacity = graph[cur][i].second;
            coord.second = next;
            // if we can't flow through here skip, or
            // if we have already visited this node, skip
            if (visited[next] || capacity - flow[coord] <= 0)
                continue;
            //std::cout<<"("<<cur<<" "<<next<<"): "<<capacity<<" "<<flow[coord]<<"      "<<capacity - flow[c
            visited[next] = true;
            previous[next] = cur;
            previousCapacity[next] = capacity;

            // traverse this next
            q.push(next);

            // if we have reached our destination construct path and return
            if (next == sink) {
                found =true;
                // to break out of the root loop
                while(!q.empty()) q.pop();
                break;
            }
        }
    }
    if (!found) {
        return std::pair<std::vector<int>, long long>(std::vector<int>(), 0l);
    }

    // let us reconstruct the path from the "previous" array
    int nextPath = sink;
    long long capacity = INF;
    std::vector<int> path;
    path.push_back(nextPath);
    do {
        coord.first = previous[nextPath];
        coord.second = nextPath;
        capacity = min(capacity, previousCapacity[nextPath] - flow[coord]);
        nextPath = previous[nextPath];
        path.push_back(nextPath);
    } while(nextPath != source);

    // the path is backward so let us reverse that again
    for (int i=0; i<path.size()/2; i++) {
        int temp = path[i];
        path[i] = path[path.size()-i-1];
        path[path.size()-i-1] = temp;
    }

    return make_pair(path, capacity);
}


// find max flow form source to sink. O(VE^2)
// graph: [u][num] = pair<v, capacity>
long long maxflow(
    std::vector<std::vector<pil> > &graph,
    int source,
    int sink) {
    std::unordered_map<pii, long long, PairHasher> flow;

    std::pair<std::vector<int>, int> result;
    std::vector<int> path;
    pii coord;
    do {
        result = findpath(graph, flow, source, sink);
        path = result.first;
        //print(path);
        int pathcapacity = result.second;
        //std::cout<<"capacity "<<pathcapacity<<std::endl;
        for(int i=1; i<path.size(); i++) {
            // forward link
            coord.first = path[i-1];
            coord.second = path[i];
            flow[coord] += pathcapacity;
            // reverse link
            coord.second = path[i-1];
            coord.first = path[i];
            flow[coord] -= pathcapacity;
        }
        //print(flow);
    } while(path.size() != 0);

    long long maxflow = 0;
    coord.first = source;
    for (int i=0; i<graph[source].size(); i++) {
        coord.second = graph[source][i].first;
        maxflow += flow[coord];
    }
    return maxflow;
}

void ae(std::vector<std::vector<pil> > &graph, int from, int to, int weight) {
    graph[from].push_back(pil(to, weight));
    graph[to].push_back(pil(from, 0));
}

int main() {
    std::vector<std::vector<pil> > graph(4);
    ae(graph, 0, 1, 1000);
    ae(graph, 0, 2, 1000);
    ae(graph, 0, 3, 1);
    ae(graph, 1, 2, 1000);
    ae(graph, 2, 3, 1000);
    ae(graph, 1, 3, 1000);

    assert(maxflow(graph, 0, 3) == 2001);
    graph.clear();
    graph.resize(6);
    std::cout<<std::endl;

    // 0 1 2 3 4 5
    // s o p q r t
    ae(graph, 0, 1, 3);
    ae(graph, 0, 2, 3);
    ae(graph, 1, 2, 2);
    ae(graph, 1, 3, 3);
    ae(graph, 2, 4, 2);
    ae(graph, 3, 4, 4);
    ae(graph, 3, 5, 2);
    ae(graph, 4, 5, 3);
    int flow = maxflow(graph, 0, 5);
    assert(flow == 5);

}
```