Official UFRGS ACM/ICPC Team
Programming Contest Reference

Alex Gliesch (alex.gliesch@gmail.com)

September 10, 2015

Contents

1 Ad-Hoc

1.1 Test if argument is numeric

```
bool is_numeric(const string& s) {
    stringstream ss(s);
    int val;
    ss >> val;
    return not ss.fail() and ss.eof();
}
```

1.2 Date and time

```
/* converts Gregorian date to integer (Julian day number) */
int dateToInt(int m, int d, int y) {
    return 1461 * (y + 4800 + (m - 14) / 12) / 4 +
        367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
        3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
        d - 32075;
}

/* converts integer (Julian day number) to Gregorian date: month/day/year */
void intToDate (int jd, int &m, int &d, int &y) {
    int x, n, i, j;
    x = jd + 68569;
    n = 4 * x / 146097;
    x -= (146097 * n + 3) / 4;
    i = (4000 * (x + 1)) / 1461001;
    x -= 1461 * i / 4 - 31;
    j = 80 * x / 2447;
    d = x - 2447 * j / 80;
    x = j / 11;
    m = j + 2 - 12 * x;
    y = 100 * (n - 49) + i + x;
}
```

2 Algorithms and Data Structures

2.1 Merge Sort

```
/*
 * The idea: the number of swaps in bubble sort and merge sort are the
 * same. So we can run merge sort instead (because O(n^2) is too large).
 * */
ll num_swaps = 0;
vector<ll> A(100010), B(100010); /* The vector B is just a temporary */
```

```cpp
void merge(ll l, ll m, ll r) {
    ll h = l, i = l, j = m + 1;
    while (h <= m and j <= r) {
        if (A[h] < A[j]) {
            B[i++] = A[h++];
        } else {
            B[i++] = A[j++];
            num_swaps += j - i;
        }
    }

    if (h > m) {
        for (ll k = j; k <= r; ++k) B[i++] = A[k];
    } else {
        for (ll k = h; k <= m; ++k) B[i++] = A[k];
    }
    copy(B.begin() + l, B.begin() + r + 1, A.begin() + l);
}

void merge_sort(ll l, ll h) {
    if (l < h) {
        ll m = (h + l) / 2;
        merge_sort(l, m);
        merge_sort(m + 1, h);
        merge(l, m, h);
    }
}

int main() {
    // Fill A with N elements
    num_swaps = 0;
    merge_sort(0, N - 1);
    cout << "The number of swaps in merge sort of A is "
        << num_swaps << endl;
}
```

2.2 Segment Tree

```cpp
/* T can be any type of random-access container (vector, string, etc) */
template<typename T>
struct SegmentTree {
public:
    SegmentTree(const T& A) {
        this->A = A;
        n = A.size();
        st.assign(4 * n, 0);
        build(1, 0, n - 1);
    }
```

```
    /* Get index of maximum/minimum element between indices
     * i and j, inclusive. */
    int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }

private:
    vector<int> st;
    T A;
    int n;

    int left(int p) { return p << 1; }
    int right(int p) { return (p << 1) + 1; }

    void build(int p, int L, int R) {
        if (L == R) {
            st[p] = L;
        } else {
            build(left(p), L, (L + R) / 2);
            build(right(p), (L + R) / 2 + 1, R);
            int p1 = st[left(p)], p2 = st[right(p)];
            /* change >= to <= according to necessity */
            st[p] = (A[p1] >= A[p2]) ? p1 : p2;
        }
    }

    int rmq(int p, int L, int R, int i, int j) {
        if (i > R or j < L) return -1;
        if (L >= i and R <= j) return st[p];
        int p1 = rmq(left(p), L, (L + R) / 2, i, j);
        int p2 = rmq(right(p), (L + R) / 2 + 1, R, i, j);
        if (p1 == -1) return p2;
        if (p2 == -1) return p1;
        /* change >= to <= according to necessity */
        return (A[p1] >= A[p2]) ? p1 : p2;
    }
};
```

2.3 Segment Tree 2D

```
#define MAX 1010

typedef long long ll;

/* 2D segment tree node */
struct Point {
    ll x, y, mx;
    Point() {}
    Point(ll x, ll y, ll mx) : x(x), y(y), mx(mx) { }
    bool operator<(const Point& other) const {
```

```cpp
        return mx < other.mx;
    }
};

/* Note: DO NOT allocate the Seg Tree on the stack! */
struct Segtree2d {
private:
    Point T[2 * MAX * MAX];
    ll n, m;

    Point build(ll node, ll a1, ll b1, ll a2, ll b2, ll P[MAX][MAX]) {
        if (a1 > a2 or b1 > b2)
            return def();

        if (a1 == a2 and b1 == b2)
            return T[node] = Point(a1, b1, P[a1][b1]);

        T[node] = def();
        T[node] = maxNode(T[node], build(4 * node - 2, a1, b1, (a1 + a2) / 2, (b1 + b2) /
            2, P));
        T[node] = maxNode(T[node], build(4 * node - 1, (a1 + a2) / 2 + 1, b1, a2, (b1 + b2
            ) / 2, P));
        T[node] = maxNode(T[node], build(4 * node + 0, a1, (b1 + b2) / 2 + 1, (a1 + a2) /
            2, b2, P));
        T[node] = maxNode(T[node], build(4 * node + 1, (a1 + a2) / 2 + 1, (b1 + b2) / 2 +
            1, a2, b2, P));
        return T[node];
    }

    Point query(ll node, ll a1, ll b1, ll a2, ll b2, ll x1, ll y1, ll x2, ll y2) {
        if (x1 > a2 or y1 > b2 or x2 < a1 or y2 < b1 or a1 > a2 or b1 > b2)
            return def();
        if (x1 <= a1 and y1 <= b1 and a2 <= x2 and b2 <= y2)
            return T[node];
        Point mx = def();
        mx = maxNode(mx, query(4 * node - 2, a1, b1, (a1 + a2) / 2, (b1 + b2) / 2, x1, y1,
             x2, y2) );
        mx = maxNode(mx, query(4 * node - 1, (a1 + a2) / 2 + 1, b1, a2, (b1 + b2) / 2, x1,
             y1, x2, y2) );
        mx = maxNode(mx, query(4 * node + 0, a1, (b1 + b2) / 2 + 1, (a1 + a2) / 2, b2, x1,
             y1, x2, y2) );
        mx = maxNode(mx, query(4 * node + 1, (a1 + a2) / 2 + 1, (b1 + b2) / 2 + 1, a2, b2,
             x1, y1, x2, y2));
        return mx;
    }

    Point update(ll node, ll a1, ll b1, ll a2, ll b2, ll x, ll y, ll value) {
        if (a1 > a2 or b1 > b2)
            return def();
```

```
        if (x > a2 or y > b2 or x < a1 or y < b1)
            return T[node];

        if (x == a1 and y == b1 and x == a2 and y == b2)
            return T[node] = Point(x, y, value);

        Point mx = def();
        mx = maxNode(mx, update(4 * node - 2, a1, b1, (a1 + a2) / 2, (b1 + b2) / 2, x, y,
            value) );
        mx = maxNode(mx, update(4 * node - 1, (a1 + a2) / 2 + 1, b1, a2, (b1 + b2) / 2, x,
             y, value));
        mx = maxNode(mx, update(4 * node + 0, a1, (b1 + b2) / 2 + 1, (a1 + a2) / 2, b2, x,
             y, value));
        mx = maxNode(mx, update(4 * node + 1, (a1 + a2) / 2 + 1, (b1 + b2) / 2 + 1, a2, b2
            , x, y, value) );
        return T[node] = mx;
    }

public:

    /*
    * initialize and construct segment tree from grid of values P with size
    * n x m
    * */
    SegTree2D(ll n, ll m, ll P[MAX][MAX]) {
        this->n = n;
        this->m = m;
        build(1, 1, 1, n, m, P);
    }

    /* query from range [ (x1, y1), (x2, y2) ]
    * Time: O(logn) */
    Point query(ll x1, ll y1, ll x2, ll y2) {
        return query(1, 1, 1, n, m, x1, y1, x2, y2);
    }

    /* update the value of (x, y) index to 'value'
    * Time: O(logn) */
    Point update(ll x, ll y, ll value) {
        return update(1, 1, 1, n, m, x, y, value);
    }

    /* change this according to application: get the maximum, the minimum,
     * the sum, the product, etc. Don't foret to change def() */
    Point maxNode(Point a, Point b) {
        return Point(0, 0, a.mx + b.mx);
        //                  max(a.mx, b.mx)
        //                  min(a.mx, b.mx)
        //                  etc...
    }
```

```
    /* default node: change according to maxNode, for instance, if it’s sum,
     * return 0, or if max, return -INF, or if min, +INF.*/
    Point def() {
        return Point(0, 0, 0);
    }
};
```

2.4 Closest Pair Algorithm

```
/* Finds the closest pair of points among a set of n 2D points, in
 * O(n log n). */

struct Point {
    double x, y;
};

double dist2(const Point& a, const Point& b) {
    return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
}

double closest_pair(const vector<Point>& P, int s, int e) {
    if (e - s <= 1) return numeric_limits<double>::max();
    if (e - s == 2) return dist2(P[s], P[s + 1]);
    if (e - s == 3) {
        return min(min(dist2(P[s], P[s + 1]), dist2(P[s + 1], P[s + 2])),
            dist2(P[s], P[s + 2]));
    }
    int mid = (s + (e - s) / 2);
    double d = min(closest_pair(P, s, mid), closest_pair(P, mid, e));

    auto l = upper_bound(begin(P) + s, begin(P) + mid,
        Point{ abs(P[mid].x - d), 0 }, [&](const Point& p1, const Point& p2) {
        return abs(p1.x - P[mid].x) > abs(p2.x - P[mid].x);
    });

    auto u = lower_bound(begin(P) + mid, begin(P) + e,
        Point{ abs(P[mid].x - d), 0 }, [&](const Point& p1, const Point& p2) {
        return abs(p1.x - P[mid].x) < abs(p2.x - P[mid].x);
    });

    vector<Point> Q(l, u);
    sort(begin(Q), end(Q), [](const Point& p1, const Point& p2) {
        return p1.y < p2.y;
    });

    auto best = d;
    for (int i = 0; i < Q.size(); ++i) {
        for (int j = i + 1; j < Q.size() and Q[j].y - Q[i].y < d; ++j)
```

```
            best = min(dist2(Q[i], Q[j]), best);
    }
    return best;
}

int main() {
    vector<Point> P;
    // ... Fill P somehow

    // Important: P must be sorted
    sort(begin(P), end(P), [](const Point& p1, const Point& p2) {
        return p1.x < p2.x;
    });

    double d = sqrt(double(closest_pair(P, 0, P.size())));
    cout << d << " is the distance of the closest pair of points in P."
        << endl;
}
```

3 Math

3.1 Euclidean Division

```
/* Given two integers a and b, with b != 0, there exist unique integers
    q and * r such that a = bq + r
*
* Examples:
* If a = 7 and b = 3, then q = 2 and r = 1, since 7 = 3 * 2 + 1.
* If a = 7 and b = -3, then q = -2 and r = 1, since 7 = -3 * (-2) + 1.
* If a = -7 and b = 3, then q = -3 and r = 2, since -7 = 3 * (-3) + 2.
* If a = -7 and b = -3, then q = 3 and r = 2, since -7 = -3 * 3 + 2.
*/

ll euclidean_mod(ll a, ll b) {
    int r = a % b;
    return r >= 0 ? r : r + std::abs(b);
}

ll euclidean_div(ll a, ll b) {
    return (a - euclidean_mod(a, b)) / b;
}
```

3.2 Binomials using DP

```
ll dp[2510][2510];

ll binom(ll n, ll k) {
```

```
    if (k == 0 or k == n) return 1;
    if (dp[n][k] == -1) {
        dp[n][k] = binom(n - 1, k - 1) + binom(n - 1, k);

        // If a MOD is necessary (because binomials can grow quite large):
        // dp[n][k] = dp[n][k] % MOD;
    }
    return dp[n][k];
}
```

3.3 Catalan Numbers

```
ll binom(ll n, ll k) {
    ll ans = 1;
    if (k > n - 1) k = n - 1;
    for (ll i = 0; i < k; ++i) {
        ans *= (n-i);
        ans /= (i+1);
    }
    return ans;
}

ll catalan(ll n) {
    ll c = binom(2*n, n);
    return c / (n+1);
}
```

3.4 Catalan Numbers - Java

```
import java.util.Scanner;
import java.math.BigInteger;
import java.util.ArrayList;

class Main {
    public static BigInteger Binom(int n, int k) {
        BigInteger ans = BigInteger.ONE;
        if (k > n-k) k = n-k;
        for (int i = 0; i < k; ++i) {
            ans = ans.multiply(BigInteger.valueOf(n-i));
            ans = ans.divide(BigInteger.valueOf(i+1));
        }
        return ans;
    }

    public static BigInteger Catalan(int x) {
        if (x <= 1) return BigInteger.ONE;
        BigInteger c = Binom(2*x, x);
```

```java
        return c.divide(BigInteger.valueOf(x+1));
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int x = sc.nextInt();
        System.out.println(Catalan(x+1).toString());
    }
}
```

3.5 Nth Permutation of a String

```cpp
ll fact[MAX_STRING_SIZE];

string nth_permutation(string s, ll n) {
    if (s.empty()) return "";
    if (n == 0) { return s; }

    ll f = fact[s.size() - 1];
    ll i = n / f;
    n -= f * i;

    char c = s[i];
    s.erase(i, 1);
    return c + nth_permutation(s, n);
}

int main() {
    fact[0] = 1;
    for (ll i = 1; i < MAX_STRING_SIZE; ++i) fact[i] = i * fact[i - 1];

    string s;
    int n;
    // fill s  and n
    sort(begin(s), end(s)); /* IMPORTANT: s must be sorted */
    cout << "the " << n << "th permutation of " << s << " is " <<
        nth_permutation(s, n) << endl;
}
```

3.6 Fibonacci in $O(logn)$

```cpp
void mmult(ll a[2][2], ll b[2][2], ll res[2][2], ll mod = (1LL << 60)) {
    res[0][0] = ((a[0][0] * b[0][0]) + (a[0][1] * b[1][0])) % mod;
    res[0][1] = ((a[0][0] * b[0][1]) + (a[0][1] * b[1][1])) % mod;
    res[1][0] = ((a[1][0] * b[0][0]) + (a[1][1] * b[1][0])) % mod;
    res[1][1] = ((a[1][0] * b[0][1]) + (a[1][1] * b[1][1])) % mod;
```

```
}

ll fib(ll n, ll mod = (1LL << 60)) {
    ll ans[2][2] = {{1, 0}, {0, 1}};
    ll pow[2][2] = {{1, 1}, {1, 0}};
    ll tmp[2][2];
    for (ll i = 0; i < 32; ++i) {
        if (n & (1<<i)) {
            memcpy(tmp, ans, sizeof ans);
            mmult(pow, tmp, ans, mod);
        }
        memcpy(tmp, pow, sizeof pow);
        mmult(tmp, tmp, pow, mod);
    }
    return ans[1][0];
}
```

3.7 Sieve of Eratosthenes

```
bitset<1000100> prime;
vector<ll> primes;

void sieve(ll N) {
    prime.set();
    prime[0] = prime[1] = 0;
    for (ll i = 2; i <= N + 1; ++i) {
        if (prime[i]) {
            for (ll j = i * i; j <= N + 1; j += i)
                prime[j] = false;
            primes.push_back(i);
        }
    }
}
```

3.8 Displaced/Segmented Sieve

```
/* first, run sieve and find primes[] up to sqrt(U), where U is the largest
* number you are looking for. */
bitset<1000010> is_prime;

void segmented_sieve(ll L /* sieve lower bound */,
                     ll U /* sieve upper bound*/) {
    is_prime.set();
    for (ll i = 0; primes[i] <= (ll)sqrt(U); ++i) {
        ll p = primes[i];
        for (ll j = p * ceil(L / (double)p); j <= U + 1; j += p)
            is_prime[j - L] = false;
```

```
    }
}

int main() {
    // run sieve to get primes[] vector
    segmented_sieve(L, U);
    for (int i = L; i <= U; ++i) {
        if (is_prime[i - L]) {
            cout << i << " is prime" << endl;
        }
    }
}
```

3.9 Counting Prime Factors

```
vector<int> num_factors(1000010, 0);

for (int i = 2; i <= 1000000; ++i) {
    if (num_factors[i] == 0) {
        for (int j = i; j <= 1000000; j += i)
            ++num_factors[j];
    }
}

/* ps: if num_factors[i] == 0, then i is prime. */
```

3.10 Base Conversion

```
/* Convert val in base 10 to any base: */
void convert(vector<int>& d, int digits, int base, int val) {
    d.assign(digits, 0);
    int i = digits - 1;

    while (i >= 0) {
        d[i] = val % base;
        val /= base;
        --i;
    }
}
```

4 Dynamic Programming

4.1 Task Selection Problem

```
/* given a set of tasks with a start time, an end time, and the number of
 * "points" each task yields (usually, it's start_time - end_time), selects the
 * set of tasks that yield the maximum number of points, such that no task
 * intersects another in its start_time and end_time */

#define MAX_TIME 3600

struct Task {
    int start, end, points;
    Task() {}
    bool operator<(const Task& t) const { return start < t.start; }
};

int dp[MAX_TIME + 10];

int max_points(int time, int i, vector<Task>& tasks) {
    if (time >= MAX_TIME) return 0;

    while (tasks[i].start < time and i < tasks.size()) ++i;
    if (i == tasks.size()) return 0;

    if (dp[time] == -1) {
        dp[time] = 0;
        int min_end = (1 << 28);
        for (int j = i; j < tasks.size(); ++j) {
            min_end = min(min_end, tasks[j].end);
        }

        for (int j = i; j < tasks.size() and tasks[j].start < min_end; ++j) {
            dp[time] = max(dp[time],
                tasks[j].points + max_points(tasks[j].end, j + 1, tasks));
        }
    }
    return dp[time];
}

int get_max_points(vector<Task>& tasks) {
    memset(dp, -1, sizeof dp);
    sort(tasks.begin(), tasks.end());
    return max_points(tasks[0].start, 0, tasks);
}
```

4.2 Knapsack

```
int p[NUM_ITEMS]; /* prices */
int w[NUM_ITEMS]; /* weights */
int N; /* number of items */
int dp[NUM_ITEMS][MAX_WEIGHT];
```

```cpp
int knapsack(int i, int remW) {
    if (remW <= 0) return 0;
    if (i >= N) return 0;
    if (dp[i][remW] == -1) {
        if (w[i] > remW) {
            dp[i][remW] = knapsack(i + 1, remW);
        }
        else {
            dp[i][remW] = max(knapsack(i + 1, remW),
                              p[i] + knapsack(i + 1, remW - w[i]));
        }
    }
    return dp[i][remW];
}

int main() {
    // fill p
    // fill w
    memset(dp, -1, sizeof dp)
    kanpsack(0, InitialWeight)
}
```

4.3 Longest Common Subsequence (LCS)

```cpp
int dp[1010][1010];

int lcs(int Ia, int Ib, const string& Sa, const string& Sb) {
    if (Ia < 0 or Ib < 0) return 0;
    if (dp[Ia][Ib] == -1) {
        if (Sa[Ia] == Sb[Ib]) dp[Ia][Ib] = 1 + lcs(Ia - 1, Ib - 1, Sa, Sb);
        else dp[Ia][Ib] = max(lcs(Ia - 1, Ib, Sa, Sb),
                              lcs(Ia, Ib - 1, Sa, Sb));
    }
    return dp[Ia][Ib];
}

int main() {
    string Sa, Sb;
    // fill Sa and Sb
    memset(dp, -1, sizeof dp);
    cout << "The size of the LCS between " << Sa << " and " << Sb
        << " is " << lcs(Sa.size() - 1, Sb.size() - 1, Sa, Sb);
}
```

4.4 Longest Increasing Subsequence (LIS)

```cpp
/* O(n^2) version, simpler */
int lisOn2(const vector<int>& v) {
    vector<int> lis(v.size(), 1);
    for (int i = 1; i < v.size(); ++i) {
        for (int j = 0; j < i; ++j) {
            if (v[j] >= v[i]) {
                lis[i] = max(lis[i], 1 + lis[j]);
            }
        }
    }
    int lis_size = *max_element(lis.begin(), lis.end());
    return lis_size;
}

/* O(n log k) version, more complicated */
void lis_nlogk(const vector<int>& A) {
    int N = A.size();
    vector<int> lis(N, 0), parent(N, -1), index(N, 0);
    int i_longest = -1;
    int longest = 0;

    for (int i = 0; i < A.size(); ++i) {
        /* upper_bound: normal lis ;  lower_bound: strictly increasing */
        int pos = lower_bound(lis.begin(), lis.begin() + longest, A[i])
            - lis.begin();
        lis[pos] = A[i];
        index[pos] = i;

        if (pos > 0) parent[i] = index[pos - 1];
        if (pos == longest)
            ++longest, i_longest = i;
    }

    lis.clear();
    while (i_longest != -1) {
        lis.push_back(A[i_longest]);
        i_longest = parent[i_longest];
    }
    return lis.size();
}
```

4.5 Coin Change

```cpp
#define MAX_AMOUNT 6100
#define NUM_COINS 11

ll coins[] = { 1, 2, 4, 10, 20, 40, 100, 200, 400, 1000, 2000 };
ll dp[MAX_AMOUNT][NUM_COINS];
```

```
ll coin_change(ll amt, ll c) {
    if (amt == 0) return 1;
    if (amt < 0 || c >= NUM_COINS) return 0;
    if (dp[amt][c] != -1) return dp[amt][c];
    dp[amt][c] = coin_change(amt, c + 1) + coin_change(amt - coins[c], c);
    return dp[amt][c];
}

int main() {
    memset(dp, -1, sizeof dp);
    cout << "There are " << coin_change(100, 0) << " ways of making 100 "
        << "with these coins." << endl;
}
```

4.6 Travelling Salesman Problem

```
#define MAX_NODES 15
int g[MAX_NODES][MAX_NODES]; /* complete graph */
int n; /* number of nodes */;

/* First dimension of DP: the vertex index, second dimension: bitmask
 * having which vertex were already visited. */
int dp[MAX_NODES][1 << MAX_NODES];

int tsp(int v, int visited) {
    if (dp[v][visited] != -1) return dp[v][visited];
    if (visited == (1 << n) - 1) return dp[v][visited] = g[v][0];
    int best = (1 << 28);
    for (int i = 0; i < n; ++i) {
        if (i != v and not(visited & (1 << i))) {
            best = min(best, g[v][i] + tsp(i, (visited | (1 << i))));
        }
    }
    return dp[v][visited] = best;
}

int main() {
    // fill g and n
    memset(dp, -1, sizeof dp);
    int start;
    // fill starting vertex 'start'
    cout << "The shortest TSP path, starting from " << start <<
        ", has length " << tsp(start, 1<<start) << endl;
}
```

5 Graphs

5.1 BFS

```
int bfs(int s, int t, const vector<vector<int> >& g) {
    queue<int> q;
    vector<int> dist(-1, n);
    dist[s] = 0; q.push(s);
    while (q.size()) {
        int v = q.front(); q.pop();
        if (v == t) return dist[t];
        for (int i = 0; i < g[v].size(); ++i) {
            int u = g[v][i];
            if (dist[u] == -1) {
                dist[u] = 1 + dist[v];
                q.push(u);
            }
        }
    }
    return -1;
}
```

5.2 Dijkstra

```
int dijkstra(int s, int t, const vector<vector<pair<int, int> > >& g) {
    priority_queue<pair<int, int>> pq;
    pq.push(make_pair(0, s)); dist[s] = 0;
    while (pq.size()) {
        int v = pq.top().second, w = -pq.top().first;
        pq.pop();
        if (dist[v] != w) continue;
        if (v == t) return w;
        for (int i = 0; i < g[v].size(); ++i) {
            int u = g[v][i].first, d = g[v][i].second + w;
            if (dist[u] > d) {
                pq.emplace(-(dist[u] = d), u);
            }
        }
    }
    return -1;
}
```

5.3 Union-Find

```
vector<int> pset, size_set;

void init_set(int n) {
```

```
        pset.assign(n, 0); size_set.assign(n, 1);
        for (int i = 0; i < n; ++i) pset[i] = i;
    }

    int find_set(int i) {
        return pset[i] == i ? i : (pset[i] = find_set(pset[i]));
    }

    bool is_same_set(int i, int j) {
        return find_set(i) == find_set(j);
    }

    void union_set(int i, int j) {
        if (is_same_set(i, j)) return;
        size_set[find_set(j)] += size_set[find_set(i)];
        pset[find_set(i)] = find_set(j);
    }
```

5.4 Kruskal

```
struct Edge {
    Edge(double cost, int u, int v) : cost(cost), u(u), v(v) { }
    bool operator<(const Edge& e) const { return cost < e.cost; }
    double cost, u, v;
};

/*
 * Returns the cost of the minimum spanning tree
 * */
double kruskal(vector<Edge>& edges, int num_nodes) {
    double cost = 0;
    int num_sets = num_nodes;

    sort(edges.begin(), edges.end());
    init_set(num_nodes);

    for (int i = 0; i < edges.size(); ++i) {
        Edge& e = edges[i];
        if (not is_same_set(e.u, e.v)){
            cost += e.cost;
            /* i.e., add the edge (e.u, e.v) to the spanning tree */
            union_set(e.u, e.v);
            --num_sets;
        }
    }
    return cost;
}
```

5.5 Prim

```
/* For partial MST (where some edges are required to be used for the MST,
 * regardless of their weights, we should add all those edges
 * to Prim's pq with weight 0, so that the algorithm will surely select
 * them first. */

double prim(const vector<vector<pair<int, double> > >& g) {
    /* g[v][i].first = node, .second = cost
     * note: g must be an undirected graph (if g[i] points to j then g[j] must
     * point to i */
    vector<bool> visited(g.size(), false);
    priority_queue<pair<double, int>, vector<pair<double, int> >,
        greater<pair<double, int> > > pq;
    double cost = 0;

    /* start from first node, no problem */
    for (int i = 0; i < g[0].size(); ++i) {
        pq.push(make_pair(g[0][i].second, g[0][i].first));
    }
    visited[0] = true;

    while (pq.size()) {
        double w = pq.top().first;
        int u = pq.top().second;
        pq.pop();
        if (visited[u]) continue;
        visited[u] = true;
        cost += w;
        for (int i = 0; i < g[u].size(); ++i) {
            int v = g[u][i].first;
            if (not visited[v])
                pq.push(make_pair(g[u][i].second, v));
        }
    }

    /* if we want to retrieve the actual edges on the MST, we can store the
     * previous node in the pq state */
     return cost;
}
```

5.6 Bellman-Ford

```
/* returns the distance from s to t. if there is a negative edge weight
 * cycle, it sets 'has_cycle' to true.
int BellmanFord(int s, int t, const vector<vector<pair<int, int> > >& g,
                bool& has_cycle) {
    int N = g.size();
    vector<int> dist(N, numeric_limits<int>::max());
```

```
    dist[s] = 0;
    has_cycle = false;
    for (int i = 0; i < N; ++i) {
        has_cycle = false;
        for (int u = 0; u < N; ++u) {
            for (int j = 0; j < g[u].size(); ++j) {
                const ii& e = g[u][j];
                if (dist[e.first] > dist[u] + e.second) {
                    dist[e.first] = dist[u] + e.second;
                    has_cycle = true;
                }
            }
        }
    }
    return dist[t];
}
```

5.7 Floyd-Warshall

```
vector<vector<int>> g(110, vector<int>(110, (1 << 20)));

// fill g

for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
```

5.8 MinMax

```
int n, m, s, t, dist[210];
vector<vector<ii>> g;

/* Return the minimum edge cost on the paths with maximum cost starting
 * from a given source node. */
void bfs(int s, int t, const vector<vector<pair<int, int> > >& g) {
    priority_queue<ii> pq;
    vector<int> dist(g.size(), -1);
    dist[s] = INF;
    pq.push(make_pair(INF, s));

    while (pq.size()) {
        int cost = pq.top().first, u = pq.top().second; pq.pop();
        if (u == t) break;
        if (dist[u] != cost) continue;

        for (int i = 0; i < g[u].size(); ++i) {
```

```
                int v = g[u][i].second, w = min(g[u][i].first, cost);
                if (dist[v] < w) {
                    dist[v] = w;
                    pq.push(make_pair(w, v));
                }
            }
        }
        return dist[t];
    }
```

5.9 Toposort

```
    /* Here, in[v] has the number of incoming edges on the vertex v; */

    void toposort_bfs(const vector<vector<int> >& g, const vector<int>& in) {
        /* could be a queue also, but this problem required that we preserve
        * ordering.  */
        priority_queue<int, vector<int>, greater<int> > pq;
        for (int v = 0; v < n; ++v) if (in[v] == 0) pq.push(v);
        while (pq.size()) {
            int v = pq.top(); pq.pop();
            --in[v];
            cout << ' ' << index_to_name[v]; /* i.e., ADD TO TOPOSORT LIST */
            for (auto u : g[v]) {
                if (--in[u] == 0)
                    pq.push(u);
            }
        }
    }
```

5.10 Bipartite Checking

```
    int color[MAX_NODES];
    vector<vector<int> > g;

    bool is_bipartite(int v) {
        for (int i = 0; i < g[v].size(); ++i) {
            int u = g[v][i];
            if (color[u] == color[v])
                return false;
            else if (color[u] == -1) {
                color[u] = 1 - color[v];
                if (!is_bipartite(u)) return false;
            }
        }
        return true;
    }
```

5.11 Articulation Points/Bridges

```
void articulation_point(int u, int parent, int& ix,
    vector<int>& num, vector<int>& low, vector<int>& bridges) {
    low[u] = num[u] = ix++;
    for (int i = 0; i < g[u].size(); ++i) {
        if (num[g[u][i]] == -1) {
            articulation_point(g[u][i], u, ix, num, low, bridges);
            if (low[g[u][i]] >= num[u])
                ++bridges[u];
            low[u] = min(low[u], low[g[u][i]]);
        } else if (g[u][i] != parent) {
            low[u] = min(low[u], num[g[u][i]]);
        }
    }
}

int num_bridges(const vector<vector<int> >& g) {
    int ix = 0;
    int n = g.size();
    vector<int> low(n, 0), num(n, -1), bridges(n, 0);
    articulation_point(0, -1, ix, num, low, bridges);
    if (bridges[0] > 0) --bridges[0];

    int res = 0;
    for (int i = 0; i < N; ++i)
        if (bridges[i] > 0) ++res;
    return res;
}
```

5.12 Strongly Connected Components

```
void tarjan_dfs(int v, int& num_scc, int& ix,
        vector<int>& visited, vector<int>& index,
        vector<int>& low, vector<int>& s) {
    if (index[v] != -1) return;
    visited[v] = true;
    index[v] = low[v] = ix++;
    s.push_back(v);

    for (int i = 0; i < g[v].size(); ++i) {
        if (index[g[v][i]] == -1) {
            tarjan_dfs(g[v][i], num_scc, ix, visited, index, low, s);
            low[v] = min(low[v], low[g[v][i]]);
        } else if (visited[g[v][i]]){
            low[v] = min(low[v], index[g[v][i]]);
        }
    }
```

```
        if (low[v] == index[v]) {
            ++num_scc;
            while (true) {
                int u = s.back(); s.pop_back(); visited[u] = 0;
                if (u == v) break;
            }
        }
    }

    /* returns the number of strongly connected components in g */
    int tarjan_scc(const vector<vector<int> >& g) {
        int n = g.size();
        vector<int> visited(n, false), index(V, -1), low(V, 0), s;
        int ix = 0, num_scc = 0;
        for (int v = 0; v < n; ++v)
            tarjan_dfs(v, num_scc, ix, visited, index, low, s);
        return num_scc;
    }
```

5.13 Hopcroft-Karp Bipartite Matching

```
    struct HopcroftKarp {
        int mate[MAX_V], dist[MAX_V];

        bool bfs() {
            queue<int> q;
            for (int v = 1; v <= 2 * n; ++v) {
                if (mate[v] == 0) {
                    dist[v] = 0;
                    q.push(v);
                } else dist[v] = INF;
            }
            dist[0] = INF;
            while (q.size()) {
                int u = q.front(); q.pop();
                if (u == 0) continue;
                for (int i = 0; i < g[u].size(); ++i) {
                    int v = g[u][i];
                    if (dist[mate[v]] == INF) {
                        dist[mate[v]] = dist[u] + 1;
                        q.push(mate[v]);
                    }
                }
            }
            return dist[0] != INF;
        }

        bool dfs(int u) {
            if (u == 0) return true;
```

```
        for (int i = 0; i < g[u].size(); ++i) {
            int v = g[u][i];
            if (dist[mate[v]] == dist[u] + 1 and dfs(mate[v])) {
                mate[u] = v; mate[v] = u;
                return true;
            }
        }
        dist[u] = INF;
        return false;
    }

    /* IMPORTANT: indices in g must start from 1, not from 0! */
    int bipartite_matching(const vector<vector<int> >& g) {
        int max_matching = 0;
        memset(mate, 0, sizeof mate);
        while (bfs()) {
            for (int v = 1; v <= 2 * n; ++v) {
                if (mate[v] == 0 and dfs(v)) ++max_matching;
            }
        }
        return max_matching;
    }
};
```

5.14 Push Relabel Max Flow

```
struct PushRelabelMaxFlow {
    struct Edge {
        Edge(int from, int to, int cap, int flow)
            : from(from), to(to), cap(cap), flow(flow) { }
        int from, to, cap, flow;
    };
    vector<Edge> edge;
    vector<vector<int> > g;
    int n, s, t;
    vector<int> excess, dist, active, ct;
    queue<int> q;

    PushRelabelMaxFlow(int n, int s, int t) :n(n), s(s), t(t) {
        g.resize(n);
    }

    void add_edge(int u, int v, int c) {
        g[u].push_back(edge.size());
        edge.push_back(Edge(u, v, c, 0));
        g[v].push_back(edge.size());
        edge.push_back(Edge(v, u, 0, 0));
    }
```

```
void enqueue(int v) {
    if (not active[v] and excess[v] > 0) {
        active[v] = true;
        q.push(v);
    }
}

void push(int e) {
    int amt = min(excess[edge[e].from], edge[e].cap - edge[e].flow);
    if (dist[edge[e].from] <= dist[edge[e].to] or amt == 0) return;
    edge[e].flow += amt;
    edge[e ^ 1].flow -= amt;
    excess[edge[e].to] += amt;
    excess[edge[e].from] -= amt;
    enqueue(edge[e].to);
}

void gap(int k) {
    for (int v = 0; v < n; ++v) {
        if (dist[v] < k) continue;
        --ct[dist[v]];
        dist[v] = max(dist[v], n + 1);
        ++ct[dist[v]];
        enqueue(v);
    }
}

void relabel(int v) {
    --ct[dist[v]];
    dist[v] = 2 * n;
    for (int i = 0; i < g[v].size(); ++i) {
        int e = g[v][i];
        if (edge[e].cap - edge[e].flow > 0)
            dist[v] = min(dist[v], dist[edge[e].to] + 1);
    }
    ++ct[dist[v]];
    enqueue(v);
}

void discharge(int v) {
    for (int i = 0; excess[v] > 0 and i < g[v].size(); ++i)
        push(g[v][i]);
    if (excess[v] > 0) {
        if (ct[dist[v]] == 1) gap(dist[v]);
        else relabel(v);
    }
}

int push_relabel_max_flow() {
    excess.assign(n, 0);
```

```
        dist.assign(n, 0);
        active.assign(n, 0);
        ct.assign(2 * n, 0);

        ct[0] = n - 1;
        ct[n] = 1;
        dist[s] = n;
        active[s] = active[t] = true;

        for (int i = 0; i < g[s].size(); ++i) {
            int e = g[s][i];
            excess[s] += edge[e].cap;
            push(e);
        }

        while (q.size()) {
            int v = q.front(); q.pop();
            active[v] = false;
            discharge(v);
        }

        int flow = 0;
        for (int i = 0; i < g[s].size(); ++i) {
            int e = g[s][i];
            flow += edge[e].flow;
        }
        return flow;
    }
};
```

5.15 Min Cost Max Flow

```
struct MCMF {
    struct Edge {
        ll from, to;
        double cap, flow, wt;
        Edge(ll from, ll to, double cap, double flow, double wt)
            : from(from), to(to), cap(cap), flow(flow), wt(wt) {
        }
    };

    vector<Edge> edge;
    vector<ll> pred, in_queue;
    vector<double> dist;
    vector<vector<ll> > g;
    ll n, s, t;

    MCMF(ll n, ll s, ll t) : n(n), s(s), t(t) { g.resize(n + 10); }
```

```
void add_edge(ll u /* from */, ll v /* to */, double w /* cost */,
    double c /* capacity */) {
    g[u].push_back(edge.size());
    edge.push_back(Edge(u, v, c, 0, w));
    g[v].push_back(edge.size());
    edge.push_back(Edge(v, u, 0, 0, -w));
}

bool spfa() {
    dist.assign(n, numeric_limits<double>::max());
    pred.assign(n, -1);
    in_queue.assign(n, false);
    queue<ll> q; q.push(s);
    dist[s] = 0, in_queue[s] = true;
    while (q.size()) {
        ll v = q.front(); q.pop();
        in_queue[v] = false;
        for (ll i = 0; i < g[v].size(); ++i) {
            ll e = g[v][i];
            if (edge[e].cap - edge[e].flow <= 0) continue;
            ll u = edge[e].to;
            double d = dist[v] + edge[e].wt;
            if (dist[u] > d) {
                dist[u] = d;
                pred[u] = e;
                if (not in_queue[u]) {
                    in_queue[u] = true;
                    q.push(u);
                }
            }
        }
    }
    return dist[t] != numeric_limits<double>::max();
}

double min_cost_max_flow() {
    double total_flow = 0, total_cost = 0;
    while (spfa()) {
        double f = numeric_limits<double>::max();
        for (ll e = pred[t]; e != -1; e = pred[edge[e].from]) {
            f = min(f, edge[e].cap - edge[e].flow);
        }
        if (f == 0) continue;
        for (ll e = pred[t]; e != -1; e = pred[edge[e].from]) {
            edge[e].flow += f;
            edge[e ^ 1].flow -= f;
        }
        total_flow += f;
        total_cost += (f * dist[t]);
    }
```

```
        return total_cost;
    }
};
```

5.16 Edmonds-Karp Max Flow

```
struct EdmondsKarpMaxFlow {
    struct Edge {
        Edge(int from, int to, int cap, int flow)
            : from(from), to(to), cap(cap), flow(flow) {
        }
        int from, to, cap, flow;
    };

    vector<Edge> edge;
    vector<int> pred;
    int n, s, t;
    vector<vector<int> > g;

    EdmondsKarpMaxFlow(int n, int s, int t) :n(n), s(s), t(t) {
        g.resize(n);
    }

    void add_edge(int u, int v, int c) {
        g[u].push_back(edge.size());
        edge.push_back(Edge(u, v, c, 0));
        g[v].push_back(edge.size());
        edge.push_back(Edge(v, u, 0, 0));
    }

    bool bfs() {
        pred.assign(n, -1);
        queue<int> q; q.push(s);
        pred[s] = -2; /* some unique value */
        while (q.size()) {
            int v = q.front(); q.pop();
            for (int i = 0; i < g[v].size(); ++i) {
                int e = g[v][i];
                if (edge[e].cap - edge[e].flow <= 0) continue;
                int u = edge[e].to;
                if (pred[u] == -1) {
                    pred[u] = e;
                    q.push(u);
                }
            }
        }
        return pred[t] != -1;
    }
```

```
    int edmonds_karp_max_flow() {
        int total_flow = 0;
        while (bfs()) {
            int f = numeric_limits<int>::max();
            for (int e = pred[t]; e != pred[s]; e = pred[edge[e].from])
                f = min(f, edge[e].cap - edge[e].flow);
            if (f == 0) continue;
            for (int e = pred[t]; e != pred[s]; e = pred[edge[e].from]) {
                edge[e].flow += f;
                edge[e ^ 1].flow -= f;
            }
            total_flow += f;
        }
        return total_flow;
    }
};
```

6 Strings

6.1 Split a String

```
/* the Delim() is a functor that returns true if a given character is a
* delimiter, and false otherwise. Change as needed. */
struct Delim {
    bool operator()(char c){return c==' ';}
};

void split_string(vector<string>& output, const std::string& input,
    bool trim_empty = false) {
    string::const_iterator it = input.begin();
    string::const_iterator it_last = it;
    while (it = find_if(it, input.end(), Delim()), it != input.end()) {
        if (not(trim_empty and it == it_last)) {
            output.push_back(string(it_last, it));
        }
        ++it;
        it_last = it;
    }
    if (it_last != input.end()) output.push_back(string(it_last, it));
}
```

6.2 KMP

```
int b[MAXN]; /* back table for kmp. */

void kmp_preprocess(const string& p /* pattern */) {
    int i = 0, j = -1, m = p.size();
```

```
    b[0] = -1;
    while (i < m) {
        while (j >= 0 and p[i] != p[j]) j = b[j];
        ++i, ++j;
        b[i] = j;
    }
}

/*
 * this kmp_search will simply count the number of occurences of p in s.
 * */
int kmp_search(const string& s, const string& p) {
    int i = 0, j = 0, n = s.size(), m = p.size(), ans = 0;
    while (i < n) {
        while (j >= 0 and s[i] != p[j]) j = b[j];
        ++i, ++j;
        if (j == m) {
            j = b[j];
            ++ans;
        }
    }
    return ans;
}

/*
 * this kmp finds the size of the longest prefix of s that matches p
 * */
int kmp_longest_prefix(const string& s, const string& p) {
    kmp_preprocess(p);
    int i = 0, j = 0, n = s.size(), m = p.size(), ans = 0;
    while (i < n) {
        while (j >= 0 and s[i] != p[j]) j = b[j];
        ++i, ++j;
    }
    return i - j;
}

int main() {
    string s, p;
    // Fill s and p
    kmp_preprocess(p);
    cout << p << " occurs " << kmp_search(s, p) << " times in "
        << s << endl;
}
```

6.3 Suffix Array

```
#define MAXN 100010
```

```cpp
struct SuffixArray {
public:
    /*
     * build a suffix array
     * */
    void build(const string& T) {
        construct_sa(T);
        compute_lcp();
    }

    /*
    * returns the longest repeated substring. if there are more than one,
    * returns the one which compares lexicografically less.
    * */
    string lrs() {
        int ix = 0, max_lcp = -1;
        string ans;
        for (int i = 1; i < n; ++i) {
            if (lcp[i] > max_lcp or
                (lcp[i] == max_lcp and string(t, sa[i], lcp[i]) < ans)) {
                max_lcp = lcp[i], ix = i;
                ans = string(t, sa[i], lcp[i]);
            }
        }

        return string(t, sa[ix], max_lcp);
    }

private:

    int n, sa[MAXN], temp_sa[MAXN], ra[MAXN], temp_ra[MAXN];
    int lcp[MAXN], c[MAXN], plcp[MAXN], phi[MAXN];
    string t;

    void counting_sort(int k) {
        int maxi = max(300, n), sum = 0;
        memset(c, 0, sizeof(c));
        for (int i = 0; i < n; ++i)
            c[i + k < n ? ra[i + k] : 0]++;
        for (int i = 0; i < maxi; ++i) {
            int t = c[i];
            c[i] = sum;
            sum += t;
        }
        for (int i = 0; i < n; ++i)
            temp_sa[c[sa[i] + k < n ? ra[sa[i] + k] : 0]++] = sa[i];
        memcpy(sa, temp_sa, n * sizeof(int));
    }

    void construct_sa(const string& T) {
```

```
        t = T, n = T.size();
        for (int i = 0; i < n; ++i) ra[i] = t[i];
        for (int i = 0; i < n; ++i) sa[i] = i;
        for (int k = 1; k < n; k <<= 1) {
            counting_sort(k);
            counting_sort(0);
            int r = temp_ra[sa[0]] = 0;
            for (int i = 1; i < n; ++i) {
                temp_ra[sa[i]] = ((ra[sa[i]] == ra[sa[i - 1]] and
                    ra[sa[i] + k] == ra[sa[i - 1] + k]) ? r : ++r);
            }
            memcpy(ra, temp_ra, n * sizeof(int));
            if (ra[sa[n - 1]] == n - 1) break;
        }
    }

    void compute_lcp() {
        int L = 0;
        phi[sa[0]] = -1;
        for (int i = 0; i < n; ++i) phi[sa[i]] = sa[i - 1];
        for (int i = 0; i < n; ++i) {
            if (phi[i] == -1) {
                plcp[i] = 0;
                continue;
            }
            while (t[i + L] == t[phi[i] + L]) L++;
            plcp[i] = L;
            L = max(L - 1, 0);
        }
        for (int i = 0; i < n; ++i) lcp[i] = plcp[sa[i]];
    }
} sa;

int main() {
    string s; cin >> s;
    sa.build(s);
    cout << "The longest repeated substring in " << s << " is "
        << sa.lrs() << endl;
}
```

7 Geometry

7.1 Points and Lines

```
#define EPS 1e-10

struct Point {
    Point(double x = 0, double y = 0) : x(x), y(y) { }
```

```
    bool operator<(const Point& p) const {
        return y < p.y or(y == p.y and x < p.x);
    }
    bool operator==(const Point& p) const {
        return abs(x - p.x) < EPS and abs(y - p.y) < EPS;
    }
    double norm() const {
        return sqrt(x*x + y*y);
    }
    double x, y;
};

/*
 * Distance of two points in 2D.
 * */
double dist(const Point& a, const Point& b) {
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

/*
 * Distance of point p to line defined by AB. Closest point on AB is returned on
 * 'ret'.
 * */
double dist_to_line(const Point& p, const Point& A, const Point& B,
                    Point* ret = NULL) {
    double scale = double((p.x - A.x) * (B.x - A.x) + (p.y - A.y) * (B.y - A.y))
        / ((B.x - A.x) * (B.x - A.x) + (B.y - A.y) * (B.y - A.y));
    Point r(A.x + scale * (B.x - A.x), A.y + scale * (B.y - A.y));
    if (ret) *ret = r;
    return dist(p, r);
}

/*
 * Distance of point p to line segment defined by AB. The closest point on AB is
 * returned on 'ret'.
 * */
double dist_to_line_segment(const Point& p, const Point& A, const Point& B,
                            Point* ret = NULL) {
    if ((B.x - A.x) * (p.x - A.x) + (B.y - A.y) * (p.y - A.y) < EPS) {
        if (ret) *ret = A;
        return dist(p, A);
    } else if ((A.x - B.x) * (p.x - B.x) + (A.y - B.y) * (p.y - B.y) < EPS) {
        if (ret) *ret = B;
        return dist(p, B);
    } else {
        return dist_to_line(p, A, B, ret);
    }
}

bool in_range(const Point& p, const Point& q, const Point& r) {
```

```
    return r.x <= max(p.x, q.x) and r.x >= min(p.x, q.x)
        and r.y <= max(p.y, q.y) and r.y >= min(p.y, q.y);
}

/*
 * Returns true if line segments intersect. The intersection is returned to
 * ret, if there is exactly one intersection.
 * */
bool line_segments_intersect(const Point& p1, const Point& q1,
                             const Point& p2, const Point& q2,
                             Point* ret = NULL) {
    double a1 = q1.y - p1.y, b1 = p1.x - q1.x, c1 = a1 * p1.x + b1 * p1.y;
    double a2 = q2.y - p2.y, b2 = p2.x - q2.x, c2 = a2 * p2.x + b2 * p2.y;
    double det = a1 * b2 - a2 * b1;
    if (abs(det) < EPS) { /* Lines are parallel */
        return in_range(p1, q1, p2) or in_range(p1, q1, q2)
            or in_range(p2, q2, p1) or in_range(p2, q2, q1);
    }
    Point r((b2 * c1 - b1 * c2) / det, (a1 * c2 - a2 * c1) / det);
    bool ans = in_range(p1, q1, r) and in_range(p2, q2, r);
    if (ans and ret) *ret = r;
    return ans;
}

/*
 * Minimum distance between two line segments AB and BC. If AB and BC
 * intersect, the minimum distance is 0. Otherwise, it's the min between
 * distance of A to CD, of B to CD, of C to AB, or of D to AB.
 * */
double dist_line_seg_line_seg(const Point& A, const Point& B,
                              const Point& C, const Point& D) {
    if (line_segments_intersect(A, B, C, D)) return 0;
    double dACD = dist_to_line_segment(A, C, D);
    double dBCD = dist_to_line_segment(B, C, D);
    double dCAB = dist_to_line_segment(C, A, B);
    double dDAB = dist_to_line_segment(D, A, B);
    return min(min(dACD, dBCD), min(dCAB, dDAB));
}

/*  Collinear test:
 * > 0 ccw,  < 0 cw,  = 0 (test against EPS!) collinear */
double cross(const Point& p, const Point& q, const Point& r) {
    return (q.x - p.x) * (r.y - p.y) - (q.y - p.y) * (r.x - p.x);
}
```

7.2 Convex Hull

```
/*  > 0 ccw
*  < 0 cw
```

```
    *  = 0 collinear*/
    int cross(const Point& p, const Point& q, const Point& r) {
        return (q.x - p.x) * (r.y - p.y) - (q.y - p.y) * (r.x - p.x);
    }

    vector<Point> convex_hull(vector<Point> P) {
        int n = P.size(), k = 0;
        vector<Point> H(2 * n);

        sort(P.begin(), P.end());

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

        H.resize(k);
        return H;
    }
```

7.3 Polygon

```
    double area_of_polygon(const vector<Point>& P) {
        /* Assumes first vertex is equal to last vertex */
        double sum = 0.0;
        for (int i = 0; i < P.size() - 1; ++i) {
            sum += (P[i].x * P[i + 1].y - P[i + 1].x * P[i].y);
        }
        return abs(sum) / 2.0;
    }

    double angle(const Point& p, const Point& q, const Point& r) {
        double ux = q.x - p.x, uy = q.y - p.y;
        double vx = r.x - p.x, vy = r.y - p.y;
        return acos((ux * vx + uy * vy)
            / sqrt((ux * ux + uy * uy) * (vx * vx + vy * vy)));
    }

    bool point_in_polygon(const Point& p, const vector<Point>& P) {
        /* Assumes first vertex is equal to last vertex */
        if (P.size() == 0) return false;
        double sum = 0;
        for (int i = 0; i < P.size() - 1; ++i) {
            if (cross(p, P[i], P[i + 1]) < 0) {
```

```
            sum -= angle(p, P[i], P[i + 1]);
        } else {
            sum += angle(p, P[i], P[i + 1]);
        }
    }
    return (abs(sum - 2 * M_PI) < EPS or abs(sum + 2 * M_PI) < EPS);
}
```

7.4 Angle Between 3 Points

```
double angle(const Point& p, const Point& q, const Point& r) {
    int ux = q.x - p.x, uy = q.y - p.y;
    int vx = r.x - p.x, vy = r.y - p.y;
    return acos((ux * vx + uy * vy)
        / sqrt((ux * ux + uy * uy) * (vx * vx + vy * vy)));
}
```

7.5 Circles

```
struct Circle {
    double r, x, y;

    Circle(double r = 0, double x = 0, double y = 0) : r(r), x(x), y(y) {}

    /* returns true if p is inside circle */
    bool inside(const Point& p) {
        return dist(p, Point(x, y)) < r;
    }

    /* intersection of this circle and another circle c2. returns the number
     * of solutions (0, 1 or 2), and the intersected points in i1 and i2. */
    int circle_circle_intersection(const Circle& c2, Point& i1, Point& i2) const {
        double d = Point(x - c2.x, y - c2.y).norm();
        // find number of solutions
        if (d - (r + c2.r) > -EPS) {
            // circles are too far apart, no solution(s)
            return 0;
        } else if (std::abs(d) < EPS and std::abs(double(r - c2.r)) < EPS) {
            // circles coincide, infinite solutions
            return 2;
        } else if (d + min(r, c2.r) - max(r, c2.r) < EPS) {
            // one circle contains the other
            return 2;
        } else {
            double a = (r*r - c2.r*c2.r + d*d) / (2.0*d);
            double h = sqrt(r*r - a*a);
```

```
        Point p2(x + (a*(c2.x - x)) / d,
            y + (a*(c2.y - y)) / d);
        i1.x = p2.x + h * (c2.y - y) / d;
        i1.y = p2.y - h * (c2.x - x) / d;

        i2.x = p2.x - h * (c2.y - y) / d;
        i2.y = p2.y + h * (c2.x - x) / d;
        if (abs(d - double(r + c2.r)) < EPS) {
            return 1;
        } else {
            return 2;
        }
    }
}

/* Returns the number of intersections, and the intersected points in
 * out1 and out2 */
int circle_line_intersection(const Point& p1, const Point& p2,
                             Point& out1, Point& out2) {
    if (inside(p1) and inside(p2)) return 0;

    Point lp1(p1.x - x, p1.y - y);
    Point lp2(p2.x - x, p2.y - y);

    /* not 100% sure if normalization is correct here! */
    Point p2minusp1(lp2.x - lp1.x, lp2.y - lp1.y);
    p2minusp1.x /= p2minusp1.norm();
    p2minusp1.y /= p2minusp1.norm();

    double a = pow(p2minusp1.x, 2) + pow(p2minusp1.y, 2);
    double b = 2 * ((p2minusp1.x * lp1.x) + (p2minusp1.y * lp1.y));
    double c = pow(lp1.x, 2) + pow(lp1.y, 2) - pow(r, 2);

    double delta = b * b - 4 * a * c;
    if (delta < 0) {
        /* No intersection */
        return 0;
    } else if (delta == 0) {
        /* One intersection */
        double u = -b / (2.0 * a);
        out1 = Point(p1.x + (u * p2minusp1.x), p1.y + (u * p2minusp1.y));
        return 1;
    } else /*if (delta > 0)*/ {
        /* Two intersections */
        double sd = sqrt(delta);
        double u1 = (-b + sd) / (2.0 * a);
        double u2 = (-b - sd) / (2.0 * a);

        out1 = Point(p1.x + u1 * p2minusp1.x, p1.y + u1 * p2minusp1.y);
        out2 = Point(p1.x + u2 * p2minusp1.x, p1.y + u2 * p2minusp1.y);
```

```
                return 2;
            }
        }
    };
```