

Contents

1	src/geom/convexhull.hpp	2
2	src/geom/basic.hpp	3
3	src/numtheory/diophantine.hpp	4
4	src/numtheory/basic.hpp	6
5	src/graph/spanningtree.hpp	7
6	src/graph/stronglyconnected.hpp	8
7	src/graph/maxflow.hpp	10
8	src/string/suffixarray.hpp	12
9	src/string/z.hpp	14
10	src/boilerplate.hpp	15
11	src/game/nim.hpp	16
12	src/datastruct/querysegtree.hpp	17
13	src/datastruct/bindtree.hpp	19
14	src/datastruct/unionfind.hpp	20

1 src/geom/convexhull.hpp

```
#pragma once
#include "boilerplate.hpp"

#include "geom/basic.hpp"

/// Compute the convex hull of given set of points. The points on the edges
/// of the hull are not listed in the result (if floating point numbers are
/// used, inaccuracy may cause them to be listed).
///
/// Runs in linear time, uses Andrew's monotone chain algorithm.
vector<V> computeConvexHull(vector<V> points) {
    auto cmp = [](V a, V b) {
        if(a.real() == b.real()) {
            return a.imag() < b.imag();
        } else {
            return a.real() < b.real();
        }
    };
    sort(points.begin(), points.end(), cmp);
    points.erase(unique(points.begin(), points.end()), points.end());

    int n = points.size();
    if(n <= 2) return points;

    vector<V> hull;
    hull.push_back(points[0]);
    for(int dir = 0; dir != 2; ++dir) {
        if(dir) reverse(points.begin(), points.end());

        int start = hull.size();
        for(int i = 1; i < n; ++i) {
            while(
                hull.size() > start &&
                !ccw(hull[hull.size() - 2], hull.back(), points[i])
            ) {
                hull.pop_back();
            }
            hull.push_back(points[i]);
        }
    }
    hull.pop_back();

    return hull;
}
```

2 src/geom/basic.hpp

```
#pragma once
#include "boilerplate.hpp"

// WARNING: be careful with overflows and accuracy. Check what the code does.

/// Return true iff points a, b, c are CCW oriented.
bool ccw(V a, V b, V c) {
    return ((c - a) * conj(b - a)).imag() > 0;
}

/// Return true iff points a, b, c are collinear.
/// NOTE: doesn't make much sense with non-integer COORD_TYPE.
bool collinear(V a, V b, V c) {
    return ((c - a) * conj(b - a)).imag() == 0;
}

/// Check whether segments [a, b] and [c, d] intersect.
/// The segments must not be collinear. Doesn't handle edge cases (endpoint of
/// a segment on the other segment) consistently.
bool intersects(V a, V b, V c, V d) {
    return ccw(a, d, b) != ccw(a, c, b) && ccw(c, a, d) != ccw(c, b, d);
}

/// Interpolate between points a and b with parameter t.
V interpolate(VC t, V a, V b) {
    return a + t * (b - a);
}

/// Return interpolation parameter between a and b of projection of v to the
/// line defined by a and b.
/// NOTE: no rounding behavior specified for integers.
VC projectionParam(V v, V a, V b) {
    return ((v - a) / (b - a)).real();
}
```

3 src/numtheory/diophantine.hpp

```
#pragma once
#include "boilerplate.hpp"

struct DiophantineSolution {
    int64_t x;
    int64_t y;
    int64_t dx;
    int64_t dy;
};

void assign(int64_t& a, int64_t& b, int64_t aval, int64_t bval) {
    a = aval;
    b = bval;
}

/// Solve linear diophantine equation  $ax + by = c$ . If there is no solution
/// (iff  $\gcd(a, b)$  does not divide  $c$ ), fails. Returns structure containing
/// a solution  $(x, y)$ , all solutions are  $(x + t * dx, y + t * dy)$ , for all
/// integers  $t$ .  $a$  and  $b$  must be nonzero.  $|a|$ ,  $|b|$  and  $|c|$  should have couple
/// of high bits zero so that computations do not overflow.
DiophantineSolution solveLinearDiophantine(int64_t a, int64_t b, int64_t c) {
    if(a == 0 || b == 0) fail();
    if(a < 0 || b < 0 || c < 0) {
        DiophantineSolution ret = solveLinearDiophantine(abs(a), abs(b), abs(c));

        if((a < 0) != (c < 0)) {
            ret.x *= -1;
            ret.dx *= -1;
        }
        if((b < 0) != (c < 0)) {
            ret.y *= -1;
            ret.dy *= -1;
        }

        return ret;
    }

    int64_t d = a;
    int64_t e = b;

    int64_t x = 0, y = 1, lx = 1, ly = 0;
    while(e != 0) {
        int64_t q = d / e;
        assign(d, e, e, d % e);
        assign(x, lx, lx - q * x, x);
        assign(y, ly, ly - q * y, y);
    }

    if(c % d != 0) fail();
    int64_t coef = c / d;
```

```

DiophantineSolution ret;
ret.x = 0;
ret.y = 0;
ret.dx = b / d;
ret.dy = -a / d;

int64_t B = int64_t(1) << 62;
for(int i = 0; i <= 62; ++i) {
    ret.x <= 1;
    ret.y <= 1;

    if(coef & B) {
        ret.x += lx;
        ret.y += ly;
    }

    int64_t t = (ret.y - ret.x) / (ret.dx - ret.dy);
    ret.x += t * ret.dx;
    ret.y += t * ret.dy;

    B >>= 1;
}

return ret;
}

```

4 src/numtheory/basic.hpp

```
#pragma once
#include "boilerplate.hpp"

Z gcd_(Z a, Z b) {
    if(a == 0) return b;
    return gcd_(b % a, a);
}

/// Return the greatest common divisor of two integers. Returns 0 if a and b
/// are zero, otherwise result is always positive.
Z gcd(Z a, Z b) {
    return gcd_(abs(a), abs(b));
}

/// Return lookup table of primes in [0, n[. (1 for prime, 0 for non-prime).
vector<char> genPrimeTable(int n) {
    vector<char> ret(n, 1);
    if(n > 0) ret[0] = 0;
    if(n > 1) ret[1] = 0;
    for(int i = 2; i * i <= n; ++i) {
        int x = i * i;
        while(x < n) {
            ret[x] = 0;
            x += i;
        }
    }
    return ret;
}
```

5 src/graph/spanningtree.hpp

```
#pragma once
#include "boilerplate.hpp"

#include "datastruct/unionfind.hpp"

/// Find the minimum spanning tree of a graph with vertices 0,...,n-1 and
/// edges given by (weight, (vertex1, vertex2)) using Kruskal algorithm.
/// If the input graph is not connected, the result consists of spanning trees
/// for each component. The return value is a pair of the resulting forest and
/// its total weight.
/// Weight type may be replaced with any number type.
pair<vector<vector<int> >, double> kruskal(
    int n,
    vector<pair<double, pair<int, int> > > edges
) {
    sort(edges.begin(), edges.end());

    vector<vector<int> > span(n);
    double weight = 0.0;
    UnionFind c(n);
    for(int i = 0; i < edges.size(); ++i) {
        int v1 = edges[i].second.first;
        int v2 = edges[i].second.second;
        if(c.find(v1) != c.find(v2)) {
            c.merge(v1, v2);
            span[v1].push_back(v2);
            span[v2].push_back(v1);
            weight += edges[i].first;
        }
    }

    return pair<vector<vector<int> >, double>(span, weight);
}
```

6 src/graph/stronglyconnected.hpp

```
#pragma once
#include "boilerplate.hpp"

/// Compute the strongly connected components of a graph given as adjacency
/// lists. The result is a vector of the strongly connected components, each
/// given as a vector of vertex indices.
///
/// Uses Tarjan's algorithm, runs in  $O(V + E)$  time.
vector<vector<int>> computeStronglyConnectedComponents(const vector<vector<int>>& G) {
    int n = G.size();

    vector<int> ind(n, -1);
    vector<int> low(n);
    vector<int> parent(n, -1);
    vector<int> pos(n, 0);
    vector<char> in_tarj(n, 0);

    stack<int> dfs, tarj;

    int cur_ind = 0;

    vector<vector<int>> ret;
    for(int s = 0; s < n; ++s) {
        if(ind[s] != -1) continue;

        dfs.push(s);
        while(!dfs.empty()) {
            int v = dfs.top();
            dfs.pop();

            if(pos[v] == 0) {
                if(ind[v] != -1) fail();
                ind[v] = cur_ind;
                low[v] = cur_ind;
                ++cur_ind;
                tarj.push(v);
                in_tarj[v] = 1;
            }

            if(pos[v] == G[v].size()) {
                if(parent[v] != -1) {
                    low[parent[v]] = min(low[parent[v]], low[v]);
                }
                if(low[v] == ind[v]) {
                    vector<int> comp;
                    int r;
                    do {
                        r = tarj.top();
                        tarj.pop();
                        in_tarj[r] = 0;
                    } while(r != v);
                    ret.push_back(comp);
                }
            }
        }
    }
}
```



```

        comp.push_back(r);
    } while(r != v);
    ret.push_back(move(comp));
} else {
    dfs.push(v);

    int x = G[v][pos[v]];
    if(ind[x] == -1) {
        parent[x] = v;
        dfs.push(x);
    } else if(in_tarj[x]) {
        low[v] = min(low[v], ind[x]);
    }

    ++pos[v];
}
}

return ret;
}

```

7 src/graph/maxflow.hpp

```
#pragma once
#include "boilerplate.hpp"

/// Edmonds-Karp algorithm for computing max flow in a graph. Edges are given
/// with addEdge(source, destination, capacity), and maxFlow computes the
/// maximum flow, populating the flow fields of the edges.
struct EdmondsKarp {
    struct Edge {
        int dest;
        Z cap; // Remaining capacity in the edge.
        Z flow; // Flow through the edge. >= 0 in normal edges, <= 0 in
                // backwards edges.
        int back; // Corresponding backwards edge.
    };

    EdmondsKarp(int n) : G(n) { }

    void addEdge(int src, int dest, Z cap) {
        if(src == dest) return;
        Edge e1 = {dest, cap, 0, (int)G[dest].size()};
        Edge e2 = {src, 0, 0, (int)G[src].size()};
        G[src].push_back(e1);
        G[dest].push_back(e2);
    }

    Z maxFlow(int s, int t) {
        if(s == t) fail();

        Z ret = 0;

        typedef pair<int, int> P;
        vector<P> parent;
        while(true) {
            parent.clear();
            parent.resize(G.size(), P(-1, -1));

            queue<int> Q;
            Q.push(s);
            parent[s] = P(s, -1);

            while(!Q.empty()) {
                int v = Q.front();
                Q.pop();

                if(v == t) break;

                for(int i = 0; i < G[v].size(); ++i) {
                    if(G[v][i].flow == G[v][i].cap) continue;
                    int x = G[v][i].dest;
                    if(parent[x].first != -1) continue;
```

```

        parent[x] = P(v, i);
        Q.push(x);
    }
}

if(parent[t].first == -1) break;

Z a = -1;
int v = t;
while(v != s) {
    P p = parent[v];
    Z rem = G[p.first][p.second].cap - G[p.first][p.second].flow;
    if(a < 0 || rem < a) a = rem;
    v = p.first;
}
ret += a;
v = t;
while(v != s) {
    P p = parent[v];
    G[p.first][p.second].flow += a;
    G[v][G[p.first][p.second].back].flow -= a;
    v = p.first;
}

return ret;
}

vector<vector<Edge> > G;
};

```

8 src/string/suffixarray.hpp

```
#pragma once
#include "boilerplate.hpp"

namespace suffixarray_ {
struct Elem {
    int start;
    int parts[2];
};
}

/// Return the start indices of the suffices of string S sorted in
/// lexicographical order. Characters past the end come before other characters
/// in the order. The empty suffix is included.
vector<int> constructSuffixArray(const vector<int>& S) {
    using suffixarray_::Elem;

    int n = S.size();
    vector<pair<int, int> > C(n + 1);
    vector<int> tmp(n + 1);
    vector<Elem> T(n + 1);
    vector<Elem> T2(n + 1);

    for(int i = 0; i <= n; ++i) {
        C[i].first = (i == n) ? INT_MIN : S[i];
        C[i].second = -i;
    }
    sort(C.begin(), C.end());

    for(int i = 0; i <= n; ++i) {
        int start = -C[i].second;
        T[i].start = start;
        if(start == n) {
            T[i].parts[0] = 0;
        } else {
            T[i].parts[0] = 1;
            T[i].parts[1] = S[start];
        }
    }

    int t = 1;
    while(t <= n) {
        for(int i = 0; i <= n; ++i) {
            if(i != 0 &&
                T[i].parts[0] == T[i - 1].parts[0] &&
                T[i].parts[1] == T[i - 1].parts[1]
            ) {
                tmp[T[i].start] = tmp[T[i - 1].start];
            } else {
                tmp[T[i].start] = i;
            }
        }
    }
}
```

```

    }

    for(int i = 0; i <= n; ++i) {
        T[i].start = i;
        T[i].parts[0] = (i + t > n) ? 0 : tmp[i + t];
        T[i].parts[1] = tmp[i];
    }

    for(int s = 0; s < 2; ++s) {
        fill(tmp.begin(), tmp.end(), 0);
        for(int i = 0; i <= n; ++i) {
            ++tmp[T[i].parts[s]];
        }

        int x = 0;
        for(int i = 0; i <= n; ++i) {
            int y = tmp[i];
            tmp[i] = x;
            x += y;
        }

        for(int i = 0; i <= n; ++i) {
            T2[tmp[T[i].parts[s]]++] = T[i];
        }

        swap(T, T2);
    }

    t *= 2;
}

for(int i = 0; i <= n; ++i) {
    tmp[i] = T[i].start;
}

return tmp;
}

```

9 src/string/z.hpp

```
#pragma once
#include "boilerplate.hpp"

/// Return vector containing for each position i the length of the longest
/// substring that is also a prefix of the string.
vector<int> zAlgorithm(const vector<int>& S) {
    int n = S.size();
    vector<int> ret(n, 0);

    int L = 0;
    int R = 0;
    for(int i = 1; i < n; ++i) {
        if(i > R) {
            L = i;
            R = i;
            while(R < n && S[R - L] == S[R]) ++R;
            ret[i] = R - L;
            --R;
        } else if(ret[i - L] < R - i + 1) {
            ret[i] = ret[i - L];
        } else {
            L = i;
            while(R < n && S[R - L] == S[R]) ++R;
            ret[i] = R - L;
            --R;
        }
    }

    return ret;
}
```

10 src/boilerplate.hpp

```
#pragma once

// Boilerplate required by other modules.

using namespace std;

#include <algorithm> // abs, binary_search, copy, equal_range, lower_bound, max,
                    // merge, min, upper_bound, sort, swap, ...
#include <bitset>    // bitset
#include <complex>   // complex
#include <climits>   // INT_MAX, INT_MIN, ...
#include <cstdlib>   // exit
#include <deque>     // deque
#include <iostream>  // cin, cout, cerr
#include <list>      // list
#include <map>       // map
#include <queue>     // queue, priority_queue
#include <random>    // mt19937, uniform_int_distribution, uniform_real_distribution, ...
#include <set>       // set
#include <stack>     // stack
#include <string>    // string
#include <sstream>   // stringstream
#include <vector>    // vector

// Complex number/vector type for geometry.
#ifdef COORD_TYPE
    typedef COORD_TYPE VC;
#else
    typedef double VC;
#endif
typedef complex<VC> V;

// Signed integer type used for integers apart from indices.
#ifdef INT_TYPE
    typedef INT_TYPE Z;
#else
    typedef int Z;
#endif

const double PI = 4 * atan(1);

void fail(string msg) {
    cerr << "FAIL: " << msg << "\n";
    abort();
}

void fail() {
    cerr << "FAIL\n";
    abort();
}
```

11 src/game/nim.hpp

```
#pragma once
#include "boilerplate.hpp"

/// In the Nim game, there are heaps of coins with  $h[i]$  coins in  $i$ :th heap.
/// The players remove any positive number of coins from one heap in turns.
/// The player who removes the last coin wins. The function returns a pair
/// consisting of the index of the heap and the number of coins to remove
/// to win, or pair (-1, -1) if there is no winning strategy.
pair<int, Z> solveNim(const vector<Z>& h) {
    Z nim = 0;
    for(int i = 0; i < h.size(); ++i) {
        nim ^= h[i];
    }

    if(nim == 0) return pair<int, Z>(-1, -1);

    for(int i = 0; i < h.size(); ++i) {
        if((nim ^ h[i]) < h[i]) return pair<int, Z>(i, h[i] - (nim ^ h[i]));
    }

    fail();
    return pair<int, Z>(-2, -2);
}
```


12 src/datastruct/querysegtree.hpp

```
#pragma once
#include "boilerplate.hpp"

/// Array [0, n[ -> Z with fast queries of A[i] (x) A[i + 1] (x) ... (x) A[j - 1]
/// for given i, j where (x) is the associative operator 'oper' (implement).
struct QuerySegmentTree {
    /// The associative operator to use. To be implemented by the user.
    static Z oper(Z a, Z b); // { return <result>; }

    QuerySegmentTree(vector<Z> src) {
        N = 1;
        while(N < src.size()) N *= 2;
        tree.resize(2 * N);
        copy(src.begin(), src.end(), tree.begin() + N);
        for(int x = N - 1; x != 0; --x) {
            tree[x] = oper(tree[2 * x], tree[2 * x + 1]);
        }
    }

    /// Query A[x] (x) A[x + 1] (x) ... (x) A[y - 1].
    Z query(int x, int y) {
        x += N;
        y += N;

        if(x == y) fail();
        if(x + 1 == y) return tree[x];

        Z a = tree[x];
        Z b = tree[--y];
        while(x / 2 != y / 2) {
            if(x % 2 == 0) a = oper(a, tree[x + 1]);
            if(y % 2 == 1) b = oper(tree[y - 1], b);

            x /= 2;
            y /= 2;
        }
        return oper(a, b);
    }

    /// Set A[x] to val.
    void set(int x, Z val) {
        x += N;
        tree[x] = val;
        x /= 2;
        while(x != 0) {
            tree[x] = oper(tree[2 * x], tree[2 * x + 1]);
            x /= 2;
        }
    }
}
```

```
int N;  
vector<Z> tree;  
};
```

13 src/datastruct/bindtree.hpp

```
#pragma once
#include "boilerplate.hpp"

/// Array [0, n[ -> Z with fast query of sums [0, x[.
/// Initially set to zero.
struct BinIndexedTree {
    BinIndexedTree(int n) : n(n), tree(n + 1, 0) { }

    /// Add x to i:th value, 0 <= i < n.
    void change(int i, Z x) {
        ++i;
        while(i <= n) {
            tree[i] += x;
            i += (i & -i);
        }
    }

    /// Get sum of elements [0, i[, 0 <= i <= n.
    Z sum(int i) {
        Z ret = 0;
        while(i != 0) {
            ret += tree[i];
            i -= (i & -i);
        }
        return ret;
    }

    int n;
    vector<Z> tree;
};
```

14 src/datastruct/unionfind.hpp

```
#pragma once
#include "boilerplate.hpp"

/// Data structure containing a subdivision of set  $0, \dots, n-1$ . Each set has a
/// representative element of the set, and two sets can be merged fast.
/// Initially each element is in its own set.
struct UnionFind {
    UnionFind(int n) : parent(n) {
        for(int i = 0; i < n; ++i) {
            parent[i] = i;
        }
    }

    /// Find the representative element for set containing x.
    int find(int x) {
        if(parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    /// Merge sets containing x and y.
    void merge(int x, int y) {
        x = find(x);
        y = find(y);
        parent[x] = y;
    }

    vector<int> parent;
};
```