

# Cheatsheet for competitive programming

bubblesolve (Elias Riedel Gårding)

March 21, 2020

## Contents

<b>1 Setup</b>	
1.1 Caps Lock as Escape . . . . .	
1.2 .vimrc . . . . .	
1.3 Compilation . . . . .	
<b>2 Graph algorithms</b>	
2.1 DFS . . . . .	
2.2 Dijkstra . . . . .	
2.3 Bellman-Ford . . . . .	
2.4 Floyd-Warshall (all-pairs shortest path) . . . . .	
2.5 Maximum flow (Ford-Fulkerson) . . . . .	
2.6 Minimum spanning tree . . . . .	
<b>3 Data structures</b>	
3.1 Union-Find . . . . .	
3.2 Fenwick Tree . . . . .	
<b>4 String algorithms</b>	
4.1 KMP . . . . .	
<b>5 Number theory</b>	
5.1 Combinatorics . . . . .	
5.2 Extended Euclidean algorithm . . . . .	
5.3 Chinese remainder theorem . . . . .	
<b>6 Standard problems</b>	
6.1 Knapsack . . . . .	
6.2 Longest increasing subsequence . . . . .	
6.3 Interval cover . . . . .	
<b>7 Miscellaneous algorithms</b>	
7.1 Binary search . . . . .	
7.2 Ternary search (unimodal optimization) . . . . .	
7.3 Gaussian elimination . . . . .	
7.4 Simplex algorithm (linear programming) . . . . .	
7.5 Basis conversion . . . . .	
<b>8 Mathematical objects</b>	
8.1 Fraction . . . . .	
8.2 Bignum . . . . .	
8.3 Matrix . . . . .	
8.4 Polynomial . . . . .	

## 1 Setup

### 1.1 Caps Lock as Escape

```
1 xmodmap -e "clear Lock"
1 # If that doesn't work, xmodmap -e "remove Lock = Caps_Lock"
1 xmodmap -e "keycode 66 = Escape NoSymbol Escape"
```

### 1.2 .vimrc

```
1
1
3
3 set nocompatible
4 filetype plugin indent on
4 set autoindent
6 set tabstop=4 shiftwidth=4 expandtab
set foldmethod=marker
6 set number relativenumber
6 syntax on
7
```

### 1.3 Compilation

```
7
7 compile() {
8     g++ -O2 -Wall -Wextra -Wfloat-equal -Wunreachable-code -Wfatal-errors \
8         -Wformat=2 -std=gnu++17 -D_GLIBCXX_DEBUG "$1" -o "${basename "$1"}.cpp"
8 }
9
9 run() {
9     compile "$1"
9     "${realpath "${basename "$1"}.cpp"}"
10 }
10
```

## 2 Graph algorithms

### 2.1 DFS

```
11
11
12 #pragma once
14
16 #include <algorithm>
16 #include <functional>
16 #include <stack>
16 #include <vector>
18
21 using NodeFunc = std::function<void (int)>;
22 using EdgeFunc = std::function<void (int, int)>;
```

```

void node_nop(int) {}
void edge_nop(int, int) {}

/* Depth-first search in a directed graph.
 * Conceptually, a node has one of three colours:
 * · White: untouched
 * · Gray: in the queue
 * · Black: has been popped from the queue
 * This class offers three callbacks:
 * · explore(from, to): called when exploring the edge `from -> to`
 * · discover(node): called when `node` becomes gray
 * · finish(node): called when `node` becomes black
 */
class DFS {
public:
    int const N;
    std::vector<std::vector<int>> const &adj;
    std::vector<bool> visited;
    EdgeFunc explore;
    NodeFunc discover, finish;

    DFS(std::vector<std::vector<int>> const &adj)
        : N(adj.size()), adj(adj), visited(N, false),
          explore(edge_nop), discover(node_nop), finish(node_nop)
    {}

    void dfs_from(int node) {
        if (visited[node]) return;
        visited[node] = true;

        discover(node);

        for (int child : adj[node]) {
            explore(node, child);
            dfs_from(child);
        }

        finish(node);
    }

    void run() {
        for (int node = 0; node < N; ++node)
            dfs_from(node);
    }

    DFS &on_explore(EdgeFunc f) {
        explore = f;
        return *this;
    }

    DFS &on_discover(NodeFunc f) {
        discover = f;
        return *this;
    }
}

```

```

DFS &on_finish(NodeFunc f) {
    finish = f;
    return *this;
}

std::vector<int> topological_sort(std::vector<std::vector<int>> const &adj)
/* Given a directed graph in the form of an adjacency list, return a topological
 * ordering of the graph's DFS forest: an ordering of the nodes such that a
 * parent always appears before all of its descendants. If the graph is acyclic,
 * this is a topological ordering of the graph itself.
 */
{
    std::vector<int> result;
    result.reserve(adj.size());

    DFS(adj).on_finish([&] (int node) { result.push_back(node); }).run();

    std::reverse(result.begin(), result.end());
    return result;
}

std::vector<std::vector<int>> scc_kosaraju(
    std::vector<std::vector<int>> const &adj)
/* Given a directed graph in the form of an adjacency list, return a vector of
 * its strongly connected components (where each component is a list of nodes).
 */
{
    int const N = adj.size();

    std::vector<int> order = topological_sort(adj);

    std::vector<std::vector<int>> adj_T(N);
    for (int i = 0; i < N; ++i)
        for (int j : adj[i])
            adj_T[j].push_back(i);

    std::vector<std::vector<int>> comps(1);
    DFS dfs(adj_T);
    dfs.on_finish([&] (int node) { (comps.end() - 1)->push_back(node); });

    for (int node : order) {
        if (!(comps.end() - 1)->empty())
            comps.emplace_back();

        dfs.dfs_from(node);
    }

    if ((comps.end() - 1)->empty()) comps.erase(comps.end() - 1);

    return comps;
}

```

```

bool is_cyclic(std::vector<std::vector<int>> const &adj)
/* Given a directed graph in the form of an adjacency list, determine whether it
 * contains a cycle or not.
 */
{
    // Whether a node is currently in the recursive call stack
    std::vector<bool> in_stack(adj.size(), false);

    // Use an exception to stop the DFS early
    class CycleFound {};

    try {
        DFS(adj)
            .on_discover([&] (int node) { in_stack[node] = true; })
            .on_finish([&] (int node) { in_stack[node] = false; })
            .on_explore([&] (int, int child) {
                if (in_stack[child]) throw CycleFound();
            })
            .run();
    } catch (CycleFound) {
        return true;
    }

    return false;
}

```

## 2.2 Dijkstra

```
#pragma once
```

```

#include <limits>
#include <queue>
#include <tuple>
#include <vector>

```

```

template<typename D>
std::vector<D> dijkstra(
    std::vector<std::vector<std::pair<int, D>> const &edges,
    int source, int target,
    std::vector<int> &prev,
    D *inf_ptr = nullptr)
/* Returns a vector of the shortest distances to all nodes from the source,
 * but stops whenever target is encountered (set target = -1 to compute
 * all distances).
 * Populates prev with the predecessors to each node (-1 when no predecessor).
 * Nodes numbered from 0 to N - 1 where N = edges.size().
 * Distances are INF if there is no path.
 * If inf_ptr is not null, puts INF into *inf_ptr.
 */
{
    constexpr D INF = std::numeric_limits<D>::max() / 2;
    if (inf_ptr != nullptr) *inf_ptr = INF;

```

```

    int const N = edges.size();

    std::vector<bool> visited(N, false);

    prev.assign(N, -1);
    std::vector<D> dist(N, INF);
    dist[source] = 0;

    using Entry = std::pair<D, int>;
    std::priority_queue<Entry, std::vector<Entry>, std::greater<Entry>> Q;
    Q.emplace(0, source);

    while (!Q.empty()) {
        D node_dist; int node;
        std::tie(node_dist, node) = Q.top();
        Q.pop();

        if (visited[node]) continue;
        visited[node] = true;

        if (node == target) break;

        for (std::pair<int, D> const &edge : edges[node]) {
            int child; D edge_len;
            std::tie(child, edge_len) = edge;

            if (!visited[child] && node_dist + edge_len < dist[child]) {
                dist[child] = node_dist + edge_len;
                prev[child] = node;
                Q.emplace(dist[child], child);
            }
        }
    }

    return dist;
}

```

## 2.3 Bellman-Ford

```
#pragma once
```

```

#include <limits>
#include <tuple>
#include <vector>

```

```

template<typename D>
std::vector<D> bellman_ford(
    std::vector<std::vector<std::pair<int, D>> const &edges,
    int source,
    std::vector<int> &prev,
    bool &negative_cycle,
    D *inf_ptr = nullptr)
/* Returns a vector of the shortest distances to all nodes from the source.
 * Populates prev with the predecessors to each node (-1 when no predecessor).
 */

```

```

* negative_cycle is set if there are negative cycles in the graph.
* Nodes numbered from 0 to N - 1 where N = edges.size().
* Distances are INF if there is no path, -INF if arbitrarily short paths exist.
* If inf_ptr is not null, puts INF into *inf_ptr.
*/
{
    constexpr D INF = std::numeric_limits<D>::max() / 2;
    if (inf_ptr != nullptr) *inf_ptr = INF;
    int const N = edges.size();

    prev.assign(N, -1);
    std::vector<D> dist(N, INF);
    dist[source] = 0;

    // N - 1 phases for finding shortest paths,
    // N phases for finding negative cycles.
    negative_cycle = false;
    for (int ph = 0; ph < 2 * N - 1; ++ph)
        // Iterate over all edges
        for (int u = 0; u < N; ++u)
            if (dist[u] < INF) // prevent catching negative INF -> INF edges
                for (std::pair<int, D> const &edge : edges[u]) {
                    int v; D w;
                    std::tie(v, w) = edge;
                    if (dist[v] > dist[u] + w) {
                        if (ph < N - 1) {
                            dist[v] = dist[u] + w;
                            prev[v] = u;
                        }
                        else {
                            negative_cycle = true;
                            dist[v] = -INF;
                        }
                    }
                }
    }

    return dist;
}

```

## 2.4 Floyd-Warshall (all-pairs shortest path)

```
#pragma once
```

```
#include <limits>
#include <tuple>
#include <vector>
```

```

template<typename D>
std::vector<std::vector<D>> floyd_warshall(
    std::vector<std::vector<std::pair<int, D>> const &edges,
    bool &negative_cycle,
    D *inf_ptr = nullptr)
/* Returns a matrix of the shortest distances between all nodes.
* negative_cycle is set if there are negative cycles in the graph.

```

```

* Nodes numbered from 0 to N - 1 where N = edges.size().
* Distances are INF if there is no path, -INF if arbitrarily short paths exist.
* If inf_ptr is not null, puts INF into *inf_ptr.
*/
{
    constexpr D INF = std::numeric_limits<D>::max() / 2;
    if (inf_ptr != nullptr) *inf_ptr = INF;
    int const N = edges.size();

    // Initialize distance matrix
    std::vector<std::vector<D>> dist(N, std::vector<D>(N, INF));
    for (int u = 0; u < N; ++u) {
        dist[u][u] = 0;
        for (std::pair<int, D> const &edge : edges[u]) {
            int v; D w;
            std::tie(v, w) = edge;
            dist[u][v] = std::min(dist[u][v], w);
        }
    }

    // Main loop
    for (int k = 0; k < N; ++k)
        for (int i = 0; i < N; ++i)
            for (int j = 0; j < N; ++j)
                if (dist[i][k] < INF && dist[k][j] < INF)
                    dist[i][j] = std::min(dist[i][j], dist[i][k] + dist[k][j]);

    // Propagate negative cycles
    negative_cycle = false;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            for (int k = 0; k < N; ++k)
                if (dist[i][k] < INF && dist[k][j] < INF && dist[k][k] < 0) {
                    negative_cycle = true;
                    dist[i][j] = -INF;
                }
    }

    return dist;
}

```

## 2.5 Maximum flow (Ford-Fulkerson)

```
#pragma once
```

```
#include <limits>
#include <queue>
#include <vector>
```

```

template<typename D>
struct Edge {
    int from, to;
    D cap;
    D flow;

```

```

Edge(int from, int to, D cap)
    : from(from), to(to), cap(cap), flow(0) {}
Edge() = default;

int other(int origin) const {
    return origin == from ? to : from;
}

D max_increase(int origin) const {
    return origin == from ? cap - flow : flow;
}

void increase(int origin, D inc) {
    flow += (origin == from ? inc : -inc);
}

bool saturated(int origin) const {
    return max_increase(origin) == 0;
}
};

template<typename D>
std::vector<std::vector<int>> adj_list(int N, std::vector<Edge<D>> const &edges)
{
    std::vector<std::vector<int>> adj(N);
    for (int i = 0; i < (int) edges.size(); ++i) {
        Edge<D> const &e = edges[i];
        adj[e.from].push_back(i);
        adj[e.to].push_back(i);
    }

    return adj;
}

template<typename D>
D max_flow_body(int N, std::vector<Edge<D>> &edges,
               std::vector<std::vector<int>> const &adj, int source, int sink)
{
    for (Edge<D> &e : edges) e.flow = 0;

    auto augment =
        [&] () -> D {
            std::vector<int> pred(N, -1);

            std::queue<int> Q;
            Q.push(source);

            bool found = false;
            while (!found && !Q.empty()) {
                int node = Q.front();
                Q.pop();

                for (int i : adj[node]) {
                    Edge<D> const &e = edges[i];
                    int other = e.other(node);

```

```

                    if (pred[other] == -1 && !e.saturated(node)) {
                        Q.push(other);
                        pred[other] = i;
                        if (other == sink) {
                            found = true;
                            break;
                        }
                    }
                }
            }

            if (found) {
                D max_inc = std::numeric_limits<D>::max();
                int node = sink;
                while (node != source) {
                    Edge<D> const &e = edges[pred[node]];
                    max_inc = std::min(max_inc,
                                        e.max_increase(e.other(node)));
                    node = e.other(node);
                }

                node = sink;
                while (node != source) {
                    Edge<D> &e = edges[pred[node]];
                    e.increase(e.other(node), max_inc);
                    node = e.other(node);
                }

                return max_inc;
            }

            return 0;
        };

    D max_flow = 0;
    D inc;
    do {
        inc = augment();
        max_flow += inc;
    } while (inc > 0);

    return max_flow;
}

template<typename D>
D max_flow(int N, std::vector<Edge<D>> &edges, int source, int sink)
/* Given a directed graph with edge capacities, computes the maximum flow
 * from source to sink.
 * Returns the maximum flow. Mutates input: assigns flows to the edges.
 */
{
    std::vector<std::vector<int>> adj = adj_list(N, edges);
    return max_flow_body(N, edges, adj, source, sink);
}

```

```

template<typename D>
D min_cut(int N, std::vector<Edge<D>> &edges,
         int source, int sink, std::vector<int> &out)
/* Given a directed weighted graph, constructs a minimum cut such that one
 * side A contains source and the other side B contains sink.
 * Returns the capacity of the cut (sum of weights of edges from A to B, but not
 * from B to A), which is equal to the max flow from source to sink.
 * Populates out with the nodes in A.
 * Mutates input: assigns flows to the edges.
 */
{
    std::vector<std::vector<int>> adj = adj_list(N, edges);
    D flow = max_flow_body(N, edges, adj, source, sink);

    out.clear();
    std::vector<bool> visited(N, false);
    std::queue<int> Q;

    Q.push(source);
    while (!Q.empty()) {
        int node = Q.front();
        Q.pop();

        if (visited[node]) continue;
        visited[node] = true;
        out.push_back(node);

        for (int i : adj[node]) {
            Edge<D> const &e = edges[i];
            int other = e.other(node);
            if (!visited[other] && !e.saturated(node))
                Q.push(other);
        }
    }

    return flow;
}

```

## 2.6 Minimum spanning tree

*#pragma once*

*#include "union\_find.hpp"*

*#include <algorithm>*  
*#include <limits>*  
*#include <numeric>*  
*#include <vector>*

```

template<typename D>
struct Edge {
    int a, b;
    D d;
};

```

```

template<typename D>
D minimum_spanning_tree(int N, std::vector<Edge<D>> const &edges,
                       std::vector<int> &ans)
/* Given a weighted undirected graph, constructs a minimum-weight spanning
 * tree and returns its cost. Populates ans with the indices in edges
 * of the edges in the minimum spanning tree.
 * If there is no minimum spanning tree (the graph is disconnected),
 * returns std::numeric_limits<D>::max().
 */
{
    std::vector<int> idx(edges.size());
    std::iota(idx.begin(), idx.end(), 0);
    std::sort(idx.begin(), idx.end(),
             [&](int i, int j) {
                 return edges[i].d < edges[j].d;
             });

    D cost = 0;
    DisjointSets uf(N);
    for (int i : idx) {
        Edge<D> const &e = edges[i];
        if (!uf.same_set(e.a, e.b)) {
            uf.unite(e.a, e.b);
            cost += e.d;
            ans.push_back(i);
        }
    }

    if (uf.n_disjoint == 1)
        return cost;
    else {
        ans.clear();
        return std::numeric_limits<D>::max();
    }
}

```

## 3 Data structures

### 3.1 Union-Find

*#pragma once*

*#include <numeric>*  
*#include <vector>*

```

class DisjointSets {
public:
    int const size;
    std::vector<int> setsize;
    int n_disjoint;
private:
    std::vector<int> parent;
    std::vector<int> rank;
};

```

```

void set_parent(int child_root, int parent_root) {
    parent[child_root] = parent_root;
    setsize[parent_root] += setsize[child_root];
}
public:
DisjointSets(int size)
    : size(size), setsize(size, 1), n_disjoint(size), rank(size, 0)
{
    parent.assign(size, 0);
    std::iota(parent.begin(), parent.end(), 0);
}

bool same_set(int i, int j) {
    return find(i) == find(j);
}

int find(int i) {
    if (parent[i] != i)
        parent[i] = find(parent[i]);
    return parent[i];
}

void unite(int i, int j) {
    i = find(i);
    j = find(j);
    if (i != j) {
        --n_disjoint;
        if (rank[i] > rank[j])
            set_parent(j, i);
        else if (rank[j] > rank[i])
            set_parent(i, j);
        else {
            set_parent(j, i);
            ++rank[i];
        }
    }
}
};

```

## 3.2 Fenwick Tree

*#pragma once*

*#include <vector>*

```

template<typename T, typename F>
struct FenwickTree

```

```

/* Given an associative binary operation op (e.g. +) with identity element id
 * (e.g. 0), the Fenwick tree represents an array x[0], ..., x[N - 1] and
 * allows "adding" a value to any x[i], as well as computing the prefix "sum"
 * x[0] + ... + x[i - 1], both in time O(n log n).
 * While the implementation uses 1-indexing for bit-twiddling purposes,
 * the API is 0-indexed.

```

```

*/
{
    std::vector<T> data;
    F op; // F is e.g. T (*)(T, T) or std::function<T (T, T)>
    T id;

    FenwickTree(int N, F op, T id)
        : data(N, 0), op(op), id(id) {}

    // Internal one-indexing:
    // Parent of i: i - LSB(i)
    // Successor of parent of i: i + LSB(i)
    // where LSB(i) = i & -i
    T &a(int i) { return data[i - 1]; }

    // Sum of the first i elements, from 0 to i - 1
    T sum(int i) {
        // --i; ++i; (1-indexing cancels)
        T acc = id;
        while (i) {
            acc = op(acc, a(i));
            i -= i & -i;
        }

        return acc;
    }

    void add(int i, T val) {
        ++i; // 1-indexing
        while (i <= (int) data.size()) {
            a(i) = op(a(i), val);
            i += i & -i;
        }
    }
};

// Specialization for prefix sums: op = +, id = 0
template<typename T>
struct StandardFenwickTree : FenwickTree<T, T (*)(T, T)> {
    StandardFenwickTree(int N) :
        FenwickTree<T, T (*)(T, T)>(
            N, [] (T a, T b) { return a + b; }, T(0)) {}
};

```

## 4 String algorithms

### 4.1 KMP

*#include <string>*

*#include <vector>*

```

class KMP {
public:
    int const m; // Pattern length

```

```

std::string const P; // Pattern

// Prefix function
// pf[q] = max {k | k < q, P[0..=k] suffix of P[0..=q]}
// pf[q] ∈ {-1, 0, 1, 2, ...}
std::vector<int> pf;

KMP(std::string const &pattern)
: m(pattern.length()), P(pattern), pf(m)
{
    // Compute prefix function
    pf[0] = -1;
    for (int q = 0; q < m - 1; ++q) {
        int k = pf[q];

        while (k != -1 && P[k + 1] != P[q + 1])
            k = pf[k];

        pf[q + 1] = P[k + 1] == P[q + 1] ? k + 1 : -1;
    }
}

std::vector<int> match(std::string const &T)
// Returns the index in T of all matches of P.
{
    int const n = T.length();
    std::vector<int> matches;

    int q = -1; // index in P of last matched letter
    for (int i = 0; i < n; ++i) {
        while (q != -1 && P[q + 1] != T[i])
            q = pf[q];

        if (P[q + 1] == T[i]) ++q;

        if (q == m - 1) {
            matches.push_back(i - m + 1);
            q = pf[q];
        }
    }

    return matches;
}
};

```

## 5 Number theory

### 5.1 Combinatorics

*#pragma once*

```

template<typename T = long long int>
T int_pow(T x, int n) {
    T ans = 1;

```

```

    while (n > 0)
        if (n % 2 == 0) x *= x, n /= 2;
        else ans *= x, --n;

    return ans;
}

template<typename T = long long int>
T fact(int n) {
    T ans = 1;
    for (int k = 1; k <= n; ++k) ans *= k;

    return ans;
}

template<typename T = long long int>
T nPr(int n, int k) {
    T ans = 1;
    for (int i = 0; i < k; ++i) ans *= n - i;

    return ans;
}

template<typename T = long long int>
T nCr(int n, int k) {
    if (k > n / 2) k = n - k;

    T ans = 1;
    for (int i = 0; i < k; ++i)
        ans = (ans * (n - i)) / (i + 1);

    return ans;
}

```

### 5.2 Extended Euclidean algorithm

*#pragma once*

*#include <tuple>*  
*#include <utility>*

```

template<typename T>
T sign(T x) {
    return (x > T(0)) - (x < T(0));
}

```

```

template<typename T>
T mod(T x, T m) {
    return (x % m + m) % m;
}

```

```

template<typename T>
std::pair<T, T> extended_euclidean(T a, T b, T *gcd = nullptr)
/* Solve a x + b y = gcd(a, b).

```



```

* If gcd is not a null pointer, put gcd(a, b) into it.
* All solutions are given by (x + kb, y - ka) for all integers k.
* a x + b y = c has a solution if and only if gcd(a, b) | c.
*/
{
    if (b == 0) {
        if (gcd != nullptr) *gcd = a >= 0 ? a : -a;
        return {sign(a), 0};
    }

    // Solve b x' + (a % b) y' = gcd(a, b) = gcd(b, a % b)
    T xp, yp;
    std::tie(xp, yp) = extended_euclidean(b, mod(a, b), gcd);

    return {yp, xp - yp * (a / b)};
}

```

```

template<typename T>
T modular_inverse(T x, T n)
/* Find x^{-1} such that x x^{-1} \equiv 1 mod n.
* Return 0 if no inverse exists.
*/
{
    // Solve a x + k n = 1
    T a, k, gcd;
    std::tie(a, k) = extended_euclidean(x, n, &gcd);
    if (a < 0) a += n;
    return gcd == 1 ? a : 0;
}

```

### 5.3 Chinese remainder theorem

```

#pragma once

#include "extended_euclidean.hpp"

#include <cassert>
#include <tuple>
#include <vector>

template<typename T>
T chinese_remainder(std::vector<T> const& a, std::vector<T> const& m)
/* Given a set of relatively prime integers {m_i}, solve the system
* {x \equiv a_i (mod m_i)}.
* Solution is unique modulo * m_0 m_1 ... m_{i-1}.
*/
{
    T x = 0;
    T M = 1;
    for (int i = 0; i < (int) a.size(); ++i) {
        // Solve {x' \equiv x (mod M := m_0 m_1 ... m_{i-1}); x' \equiv a_i (mod m_i)}:
        // Know gcd(M, m_i) = 1
        // x' = x - k M; x' = a_i + h m_i
        // x - a_i = k M + h m_i

```

```

// Can take 0 \le k < m_i and 0 \le h < M.
T gcd;
T k = (x - a[i]) * extended_euclidean(M, m[i], &gcd).first;
assert(gcd == 1);
k = mod(k, m[i]);

x -= k * M;

M *= m[i];
x = mod(x, M);

return x;
}

```

## 6 Standard problems

### 6.1 Knapsack

```

#pragma once

#include <cassert>
#include <vector>

template<typename V>
V knapsack_unbounded(std::vector<V> const& v, std::vector<int> const& w,
                    int capacity)
{
    int const N = v.size();
    assert((int) w.size() == N);

    // A[c] = max value using total weight <= c
    std::vector<V> A(capacity + 1, 0);
    for (int c = 0; c <= capacity; ++c)
        /* TODO */;
}

template<typename V>
V knapsack_0_1(std::vector<V> const& v, std::vector<int> const& w, int capacity,
              std::vector<int> &ans)
/* Given a list of weights and values for a set of items, computes a subset
* that maximizes the total value while keeping the total weight below the
* capacity. Returns the total value and populates ans with the set of indices
* (in decreasing order).
*/
{
    int const N = v.size();
    assert((int) w.size() == N);

    // A[k][c] = max value using total weight <= c and only
    // the first k elements
    // Include/exclude: A[k][c] = max(A[k-1][c], A[k-1][c - w[k]] + v[k])
    std::vector<std::vector<V>> A(N + 1, std::vector<V>(capacity + 1, 0));
    for (int k = 0; k < N; ++k) {

```

```

    A[k + 1] = A[k];
    for (int c = w[k]; c <= capacity; ++c)
        A[k + 1][c] = std::max(A[k][c], A[k][c - w[k]] + v[k]);
}

// Find best capacity
int best_c = 0;
for (int c = 0; c <= capacity; ++c)
    if (A[N][c] > A[N][best_c])
        best_c = c;

// Backtrack
int c = best_c;
for (int k = N - 1; k >= 0; --k)
    // Either keep or remove element k
    if (w[k] <= c && A[k + 1][c] == A[k][c - w[k]] + v[k]) {
        ans.push_back(k);
        c -= w[k];
    }

return A[N][best_c];
}

```

## 6.2 Longest increasing subsequence

```

#pragma once

#include "binary_search.hpp"

#include <algorithm>
#include <vector>

template<typename T>
int longest_increasing_subsequence(std::vector<T> const &x,
                                  std::vector<int> &ans)
/* Given a sequence x, constructs the longest strictly increasing subsequence;
 * i.e. an index sequence ans such that
 * x[ans[0]] < x[ans[1]] < ...
 * Returns the length of this sequence.
 */
{
    int const N = x.size();
    std::vector<int> pred(N, -1);

    // At each time i: A[j] = index k (0 <= k < i) of smallest endpoint x[k] of
    // an increasing subsequence of length j + 1.
    std::vector<int> A;

    for (int i = 0; i < N; ++i) {
        int const J = A.size(); // Length of longest subsequence so far

        // Binary search for the location j to insert x:
        // x[A[j]] < x[i] <= x[A[j + 1]]
        auto p = [&] (int j) { return x[A[j]] < x[i]; };

```

```

        int j = int_binsearch_last(p, 0, J);

        if (j == J - 1)
            // x[i] is the endpoint of a new sequence of length J + 1
            A.push_back(i);
        else
            // x[i] < x[A[j + 1]]
            // x[i] is a smaller endpoint of a sequence of length j
            A[j + 1] = i;

        if (j != -1) pred[i] = A[j];
    }

    // Backtrack
    ans.assign(A.size(), -1);
    int i = A.empty() ? -1 : A[A.size() - 1];
    for (auto it = ans.rbegin(); it != ans.rend(); ++it) {
        *it = i;
        i = pred[i];
    }

    return A.size();
}

```

## 6.3 Interval cover

```

#pragma once

#include "binary_search.hpp"

#include <algorithm>
#include <limits>
#include <numeric>
#include <vector>

template<typename T>
int interval_cover(std::vector<std::pair<T, T>> const &intervals, T lo, T hi,
                  std::vector<int> &ans)
/*
 * Given a vector of closed intervals [a_i, b_i], select a smallest subset
 * that covers the target interval [lo, hi].
 * Returns the minimal number of such intervals, and populates ans with their
 * indices in intervals.
 */
{
    T const NEG_INF = std::numeric_limits<T>::lowest();
    int const N = intervals.size();

    // Sort intervals by starting point
    // retaining information about original order
    std::vector<int> idx(N);
    std::iota(idx.begin(), idx.end(), 0);
    std::sort(idx.begin(), idx.end(),
              [&] (int i, int j) {

```

```

        return intervals[i].first < intervals[j].first; });

// To save typing: I(i) = intervals[idx[i]]
auto I = [&] (int i) -> std::pair<T, T> const & {
    return intervals[idx[i]];
};

// Invariant: (lo, hi] is the interval that is not yet covered,
// except for before the first interval is added - then [lo, hi] is not yet
// covered.
T last_lo = NEG_INF;
do {
    // Find the intervals with starting points in [last_lo, lo] with a
    // binary search, then find the one with the largest endpoint
    // by linear search
    T best_endpoint = lo;
    int best_i = -1;

    int i = int_binsearch_first(
        [&] (int i) { return I(i).first >= last_lo; },
        0, N);
    for (; i < N && I(i).first <= lo; ++i)
        if ((ans.empty() && I(i).second >= best_endpoint)
            || I(i).second > best_endpoint)
        {
            best_endpoint = I(i).second;
            best_i = i;
        }

    if (best_i == -1) {
        ans.clear();
        return -1; // Impossible to cover
    }

    ans.push_back(idx[best_i]);
    last_lo = lo;
    lo = best_endpoint;
} while (lo < hi);

return ans.size();
}

```

## 7 Miscellaneous algorithms

### 7.1 Binary search

```
#pragma once
```

```
#include <limits>
```

```

template<typename P>
int int_binsearch_last(P p, int lo, int hi)
/* Takes a predicate p: int -> bool.
 * Assumes that there is some I (lo <= I < hi) such that p(i) = (i <= I),

```

```

 * i.e. p starts out true and then switches to false after I.
 * Finds and returns I (the last number i such that p(i) is true),
 * or lo - 1 if no such I exists (including if lo = hi).
 */
{
    while (hi - lo > 1) {
        int mid = lo + (hi - lo) / 2;
        if (p(mid)) lo = mid; // mid <= I
        else hi = mid; // I < mid
    }

    return (lo != hi && p(lo)) ? lo : lo - 1;
}

template<typename P>
int int_binsearch_first(P p, int lo, int hi)
/* Takes a predicate p: int -> bool.
 * Assumes that there is some I (lo <= I < hi) such that p(i) = (i >= I).
 * i.e. p starts out false and then switches to true at I.
 * Finds and returns I (the first number i such that p(i) is true),
 * or hi if no such I exists (including if lo = hi).
 */
{
    while (hi - lo > 1) {
        int mid = lo + (hi - lo - 1) / 2;
        if (p(mid)) hi = mid + 1; // mid >= I, search [lo, mid + 1]
        else lo = mid + 1; // I > mid, search [mid + 1, hi]
    }

    return (lo != hi && p(lo)) ? lo : hi;
}

template<typename P>
double double_binsearch_last(P p, double lo, double hi, int n_it)
/* Takes a predicate P: double -> bool.
 * Assumes that there is some X (lo <= X <= hi) such that p(x) = (x <= X),
 * i.e. p starts out true and then switches to false after X.
 * Finds and returns X (the last number x such that p(x) is true),
 * or -infinity if no such X exists.
 * This version runs a fixed number of iterations; to get results with a given
 * precision, use `while (hi - lo > eps)` instead (but this is probably a bad
 * idea because of rounding errors).
 */
{
    while (n_it-- > 0) {
        double mid = (lo + hi) / 2;
        if (p(mid)) lo = mid; // mid <= X
        else hi = mid; // X < mid
    }

    return p(lo) ? lo : -std::numeric_limits<double>::infinity();
}

template<typename P>
double double_binsearch_first(P p, double lo, double hi, int n_it)

```

```

/* Takes a predicate P: double -> bool.
 * Assumes that there is some X (lo <= X <= hi) such that p(x) = (x >= X),
 * i.e. p starts out false and then switches to true at X.
 * Finds and returns X (the first number x such that p(x) is true),
 * or +infinity if no such X exists.
 * This version runs a fixed number of iterations; to get results with a given
 * precision, use `while (hi - lo > eps)` instead (but this is probably a bad
 * idea because of rounding errors).
 */
{
    while (n_it--) {
        double mid = (lo + hi) / 2;
        if (p(mid)) hi = mid; // mid >= X
        else lo = mid; // X > mid
    }

    return p(hi) ? hi : +std::numeric_limits<double>::infinity();
}

```

## 7.2 Ternary search (unimodal optimization)

```
#pragma once
```

```
#include <limits>
#include <utility>
```

```

template<typename F>
double double_ternsearch_max(F f, double lo, double hi, int n_it)
/* Given a unimodal (increasing, then decreasing) function f,
 * returns a value x close to, but never greater than, the one maximizing f.
 * The interval shrinks with a factor 2/3 each iteration.
 */
{
    while (n_it--) {
        double a = (2 * lo + hi) / 3, b = (lo + 2 * hi) / 3;
        if (f(a) < f(b)) lo = a;
        else hi = b;
    }

    return lo;
}

```

```

template<typename T, typename F>
int int_ternsearch_max_first(F f, int lo, int hi, T *opt = nullptr)
/* Given a unimodal function f: [lo, hi] -> T that first attains its
 * maximum at I (f(lo) < f(a + 1) < ... < f(I) >= f(I + 1) >= ... >= f(hi)),
 * find I (lo <= I <= hi).
 * If opt is not nullptr, put the f(I) into *opt.
 * Call as int_ternsearch_max_first<T>(...) (F is deduced automatically).
 */
{
    // Search in [lo, hi]
    while (hi - lo >= 3) {
        // Choosing a = mid, b = mid + 1 gives fast convergence.

```

```

        int mid = lo + (hi - lo - 1) / 2;
        if (f(mid) < f(mid + 1))
            lo = mid + 1;
        else
            hi = mid;
    }

    // Linear search through last remaining elements
    int best_i = -1;
    T best = std::numeric_limits<T>::lowest();
    for (int i = lo; i <= hi; ++i) {
        T x = f(i);
        if (x > best) {
            best_i = i;
            best = x;
        }
    }

    if (opt != nullptr) *opt = best;
    return best_i;
}

```

```

template<typename T, typename F>
int int_ternsearch_max_last(F f, int lo, int hi, T *opt = nullptr)
/* Given a unimodal function f: [lo, hi] -> T that last attains its
 * maximum at I (f(lo) <= f(a + 1) <= ... <= f(I) > f(I + 1) > ... > f(hi)),
 * find I (lo <= I <= hi).
 * If opt is not nullptr, put the f(I) into *opt.
 * Call as int_ternsearch_max_last<T>(...) (F is deduced automatically).
 */
{
    // g(i) = f(j) where j = hi - (i - lo)
    int best_j = int_ternsearch_max_first(
        [&](int i) { return f(hi - (i - lo)); },
        lo, hi, opt);

    return hi - (best_j - lo);
}

```

## 7.3 Gaussian elimination

```
#pragma once
```

```
#include <cmath>
#include <limits>
#include <vector>
```

```

// Usage example:
// vector<vector<double>> A = {
//     {2, 1, -1},
//     {-3, -1, 2},
//     {-2, 1, 2}
// };
// vector<double> b = {8, -11, -3};

```

```

// vector<double> x;
// GaussianSolver<double> solver(A, b, x);
// solver.solve()
// Solution to Ax = b is now in x.
// A and b are modified into the reduced row echelon form of the system.
// Other variables: solver.consistent, solver.rank, solver.determinant

template<typename T>
class GaussianSolver {
private:
    // Add row i to row j
    void add(int i, int j, T factor) {
        b[j] += factor * b[i];
        for (int k = 0; k < M; ++k)
            A[j][k] += factor * A[i][k];
    }

    void swap_rows(int i, int j) {
        std::swap(b[i], b[j]);
        for (int k = 0; k < M; ++k)
            std::swap(A[i][k], A[j][k]);

        determinant = -determinant;
    }

    void scale(int i, T factor) {
        b[i] *= factor;
        for (int k = 0; k < M; ++k)
            A[i][k] *= factor;

        determinant /= factor;
    }

public:
    std::vector<std::vector<T>> &A;
    std::vector<T> &b;
    std::vector<T> &x;
    int const N, M;
    // sqrt(std::numeric_limits<T>::epsilon()) by default, epsilon is too small!
    T eps;
    int rank;
    T determinant;
    bool consistent;
    std::vector<bool> uniquely_determined;

    GaussianSolver(std::vector<std::vector<T>> &A,
                  std::vector<T> &b,
                  std::vector<T> &x,
                  T eps = sqrt(std::numeric_limits<T>::epsilon()))
        : A(A), b(b), x(x), N(A.size()), M(A[0].size()), eps(eps),
          rank(0), determinant(1), consistent(true)
    {
        uniquely_determined.assign(M, false);
    }
}

```

```

void solve() {
    // pr, pc: pivot row and column
    for (int pr = 0, pc = 0; pr < N && pc < M; ++pr, ++pc) {
        // Find pivot with largest absolute value
        int best_r = -1;
        T best = 0;
        for (int r = pr; r < N; ++r)
            if (std::abs(A[r][pc]) > best) {
                best_r = r;
                best = std::abs(A[r][pc]);
            }

        if (std::abs(best) <= eps) { // No pivot found
            --pr; // only increase pc in the next iteration
            continue;
        }

        // Rank = number of pivots
        ++rank;

        // Move pivot to top and scale to 1
        swap_rows(pr, best_r);
        scale(pr, (T) 1 / A[pr][pc]);
        A[pr][pc] = 1; // for numerical stability

        // Eliminate entries below pivot
        for (int r = pr + 1; r < N; ++r) {
            add(pr, r, -A[r][pc]);
            A[r][pc] = 0; // for numerical stability
        }

        // Eliminate entries above pivots
        for (int pr = rank - 1; pr >= 0; --pr) {
            // Find pivot
            int pc = 0;
            while (std::abs(A[pr][pc]) <= eps) ++pc;

            for (int r = pr - 1; r >= 0; --r) {
                add(pr, r, -A[r][pc]);
                A[r][pc] = 0; // for numerical stability
            }

            // Check for inconsistency: an equation of the form 0 = 1
            for (int r = N - 1; r >= rank; --r)
                if (std::abs(b[r]) > eps) {
                    consistent = false;
                    return;
                }

            // Calculate a solution for x
            // One solution is setting all non-pivot variables to 0
            x.assign(M, 0);
            for (int pr = 0; pr < rank; ++pr)

```

```

    for (int pc = 0; pc < M; ++pc)
        if (std::abs(A[pr][pc]) > eps) { // Pivot; A[pr][pc] == 1
            x[pc] = b[pr];
            break;
        }

    // Mark variables as uniquely determined or not
    for (int pr = 0; pr < rank; ++pr) {
        int nonzero_count = 0;
        int pc = -1;
        for (int c = 0; c < M; ++c)
            if (std::abs(A[pr][c]) > eps) {
                if (nonzero_count == 0) pc = c; // Pivot
                ++nonzero_count;
            }
        if (nonzero_count == 1)
            uniquely_determined[pc] = true;
    }
}
};

```

## 7.4 Simplex algorithm (linear programming)

```

// Simplex algorithm for linear programming.
// Written using the theory from
// Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms

```

```
#pragma once
```

```

#include <algorithm>
#include <cassert>
#include <cmath>
#include <limits>
#include <numeric>
#include <vector>

```

```

// For debug()
#include <cstdio>
#include <iostream>

```

```

template<typename T>
class SimplexSolver {
public:
    // Standard form: Maximize  $c \cdot x$  subject to  $A \cdot x \leq b$ ;  $x \geq 0$ .
    // Slack form: Basic variables  $xb$  and nonbasic variables  $xn$ .  $x = [xb, xn]$ 
    // Maximize  $z = nu + c \cdot xn$  subject to  $xb = b - A \cdot xn$ ;  $x \geq 0$ .

    T const eps;

    int n; // number of nonbasic variables
    int m; // number of constraints
    std::vector<std::vector<T>> A;
    std::vector<T> b;
    std::vector<T> c;

```

```

    T nu;

    // Holds INDICES of all variables (ranging from 0 to  $n + m - 1$ )
    std::vector<int> nonbasic_vars, basic_vars;

    bool feasible;

    SimplexSolver(std::vector<std::vector<T>> A,
                 std::vector<T> b,
                 std::vector<T> c,
                 T eps = sqrt(std::numeric_limits<T>::epsilon()))
        : eps(eps), n(c.size()), m(b.size()), A(A), b(b), c(c), nu(0),
          nonbasic_vars(n), basic_vars(m), feasible(true)
    {
        assert((int) A.size() == m && (int) A[0].size() == n);

        // Transform from standard form to slack form
        // Initially: nonbasic_vars: 0 to  $n - 1$ , basic_vars:  $n$  to  $n + m - 1$ 
        std::iota(nonbasic_vars.begin(), nonbasic_vars.end(), 0);
        std::iota(basic_vars.begin(), basic_vars.end(), n);
    }

    //  $xn_e$ : "entering" variable: nonbasic  $\rightarrow$  basic
    //  $xb_l$ : "leaving" variable: basic  $\rightarrow$  nonbasic
    void pivot(int e, int l) {
        std::swap(nonbasic_vars[e], basic_vars[l]);
        int const e_new = l, l_new = e; // Just to avoid confusion

        std::vector<std::vector<T>> A_new(A);
        std::vector<T> b_new(b);
        std::vector<T> c_new(c);
        T nu_new;

        // New constraint for  $xn_e$ : replace
        //  $xb_l = b_l - A_{lj} \cdot xn_j$ 
        // with
        //  $(*) \cdot xn_e = b_l / A_{le} - xb_l / A_{le} - A_{lj} \cdot xn_j / A_{le}$  ( $j \neq e$ )
        b_new[e_new] = b[l] / A[l][e];
        A_new[e_new][l_new] = (T) 1 / A[l][e];
        for (int j = 0; j < n; ++j)
            if (j != e)
                A_new[e_new][j] = A[l][j] / A[l][e];

        // Substitute  $(*)$  in the other constraint equations:
        // In each  $xb_i = b_i - A_{ij} \cdot xn_j$  ( $i \neq l$ ), replace  $A_{ie} \cdot xn_e$ 
        // with  $A_{ie} \cdot (b_l / A_{le} - xb_l / A_{le} - A_{lj} \cdot xn_j / A_{le})$  ( $j \neq e$ )
        for (int i = 0; i < m; ++i)
            if (i != l) {
                b_new[i] -= A[i][e] / A[l][e] * b[l];
                A_new[i][l_new] = -A[i][e] / A[l][e];
                for (int j = 0; j < n; ++j)
                    if (j != e)
                        A_new[i][j] -= A[i][e] * A[l][j] / A[l][e];
            }
    }

```

```

// Substitute (*) in the objective function:
// In nu + c_j xn_j, replace c_e xn_e with
// c_e (b_l / A_le - xb_l / A_le - A_lj xn_j / A_le (j ≠ e))
nu_new = nu + c[e] * b[l] / A[l][e];
c_new[l_new] = -c[e] / A[l][e];
for (int j = 0; j < n; ++j)
    if (j != e)
        c_new[j] -= c[e] * A[l][j] / A[l][e];

A = A_new;
b = b_new;
c = c_new;
nu = nu_new;
}

T solve(std::vector<T> &sol) {
    initial_solution();
    if (feasible)
        return solve_body(sol);
    else
        return std::numeric_limits<T>::lowest();
}

T solve_body(std::vector<T> &sol) {
    T const INF = std::numeric_limits<T>::max();

    while (true) {
        // Find a nonbasic variable with a positive coefficient in c
        int e = -1;
        for (int j = 0; j < n; ++j)
            if (c[j] > 0) {
                e = j;
                break;
            }

        if (e == -1) break; // c ≤ 0; optimal solution reached

        // Find the basic variable xb_l which most severely limits how much
        // we can increase xn_e
        T max_increase = INF;
        int l = -1;
        for (int i = 0; i < m; ++i) {
            T inc = A[i][e] > 0 ? b[i] / A[i][e] : INF;
            if (inc < max_increase) {
                max_increase = inc;
                l = i;
            }
        }

        if (l == -1)
            return INF;
        else
            pivot(e, l);
    }
}

```

```

// Construct solution in terms of the original variables x[0]..x[n - 1]
sol.assign(n, 0);
for (int i = 0; i < m; ++i)
    if (basic_vars[i] < n)
        sol[basic_vars[i]] = b[i];

// Return optimum of objective function
return nu;
}

void initial_solution() {
    // Find index l of basic variable xb_l with minimum b_l
    int l = -1;
    T b_min = std::numeric_limits<T>::max();
    for (int i = 0; i < m; ++i)
        if (b[i] < b_min) {
            b_min = b[i];
            l = i;
        }

    if (b_min >= 0) return;

    // Add an extra nonbasic variable x0
    ++n;
    nonbasic_vars.push_back(n + m - 1);

    // Add -x0 to the LHS of every constraint
    for (std::vector<T> &row : A)
        row.push_back(-1);

    // Change the objective function to -x0
    std::vector<T> original_c = c;
    T original_nu = nu;
    c.assign(n, 0);
    c[n - 1] = -1;

    // Perform a pivot x0 ↔ xb_l.
    // After this, the basic solution is feasible.
    pivot(n - 1, l);

    // Find an optimal solution to this auxiliary problem
    std::vector<T> sol;
    T ans = solve_body(sol); // ans = optimum of -x0

    if (ans >= -eps) {
        // Is x0 basic?
        auto it = std::find(basic_vars.begin(), basic_vars.end(),
                           n + m - 1);
        if (it != basic_vars.end())
        {
            // Pivot with an arbitrary non-basic variable
            int e = it - basic_vars.begin();
            pivot(e, 0);
        }
    }
}

```

```

// Find the index of x0, now a non-basic variable
int j = std::find(nonbasic_vars.begin(), nonbasic_vars.end(),
                 n + m - 1) - nonbasic_vars.begin();

// Erase x0 from the constraints and the list of variables
for (std::vector<T> &row : A) row.erase(row.begin() + j);
nonbasic_vars.erase(nonbasic_vars.begin() + j);
--n;

// Restore original objective function, substituting basic variables
// with their RHS
nu = original_nu;
c.assign(n, 0);
// Loop over all originally non-basic variables
for (int var = 0; var < n; ++var) {
    int j = std::find(nonbasic_vars.begin(), nonbasic_vars.end(),
                    var) - nonbasic_vars.begin();

    if (j != n)
        c[j] += original_c[var];
    else {
        int i = std::find(basic_vars.begin(), basic_vars.end(),
                        var) - basic_vars.begin();
        // Substitute  $x_{b_i} = b_i - A_{ij} x_{n_j}$ 
        nu += original_c[var] * b[i];
        for (int j = 0; j < n; ++j)
            c[j] -= original_c[var] * A[i][j];
    }
}
}
else {
    --n;
    feasible = false;
}
}

void debug() const {
    printf("Nonbasic vars: ");
    for (int j : nonbasic_vars) printf("x[%d] ", j);
    std::cout << std::endl;
    printf("Basic vars: ");
    for (int i : basic_vars) printf("x[%d] ", i);
    std::cout << std::endl;

    std::cout << "Optimize " << nu;
    for (int j = 0; j < n; ++j) {
        std::cout << " + (" << c[j] << ") * ";
        printf("x[%d]", nonbasic_vars[j]);
    }
    puts("");
    for (int i = 0; i < m; ++i) {
        printf("x[%d] = ", basic_vars[i]);
        std::cout << "(" << b[i] << ")";
        for (int j = 0; j < n; ++j) {
            std::cout << " - (" << A[i][j] << ") * ";
            printf("x[%d]", nonbasic_vars[j]);
        }
    }
}

```

```

    }
    puts("");
}

    puts("");
}
};

```

## 7.5 Basis conversion

```
#pragma once
```

```
#include <algorithm>
#include <string>
#include <vector>
```

```
std::string const DIGITS = "0123456789ABCDEF";
```

```
unsigned long long int basis_string_to_number(std::string &s, int b) {
    unsigned long long int result = 0ULL;
    for (char d : s) {
        result = b * result
            + (std::find(DIGITS.begin(), DIGITS.end(), d) - DIGITS.begin());
    }
}

```

```
return result;
```

```
}
```

```
std::string number_to_basis_string(unsigned long long int n, int b) {
    std::vector<char> ds;
    do {
        ds.push_back(DIGITS[n % b]);
        n = n / b;
    } while (n != 0);
}

```

```
return std::string(ds.rbegin(), ds.rend());
```

```
}
```

## 8 Mathematical objects

### 8.1 Fraction

```
#pragma once
```

```
#include <algorithm>
#include <cassert>
#include <cstdio>
#include <iostream>
#include <limits>
```

```
template<typename T = long long int>
struct Fraction {
    T n, d;
```



```

Fraction(T n, T d) : n(n), d(d) {
    reduce();
};

Fraction() : Fraction(0) {}
Fraction(T n) : Fraction(n, 1) {}
Fraction(Fraction const &) = default;
Fraction &operator=(Fraction const &) = default;

void reduce() {
    T gcd = std::__gcd(n, d);
    n /= gcd;
    d /= gcd;

    if (d < 0) {
        n = -n;
        d = -d;
    }
}

bool operator==(Fraction const &other) const {
    return n * other.d == other.n * d;
}

bool operator<(Fraction const &other) const {
    assert(d > 0 && other.d > 0);
    return n * other.d < other.n * d;
}

bool operator>(Fraction const &other) const {
    assert(d > 0 && other.d > 0);
    return n * other.d > other.n * d;
}

bool operator<=(Fraction const &other) const {
    return !(*this > other);
}

bool operator>=(Fraction const &other) const {
    return !(*this < other);
}

Fraction operator+(Fraction const &other) const {
    return Fraction(n * other.d + other.n * d, d * other.d);
}

Fraction operator-(Fraction const &other) const {
    return Fraction(n * other.d - other.n * d, d * other.d);
}

Fraction operator*(Fraction const &other) const {
    return Fraction(n * other.n, d * other.d);
}

Fraction operator/(Fraction const &other) const {
    return Fraction(n * other.d, d * other.n);
}

Fraction &operator+=(Fraction const &other) {
    return *this = *this + other;
}

Fraction &operator-=(Fraction const &other) {
    return *this = *this - other;
}

Fraction &operator*=(Fraction const &other) {
    return *this = *this * other;
}

Fraction &operator/=(Fraction const &other) {
    return *this = *this / other;
}

Fraction operator+() const {
    return *this;
}

Fraction operator-() const {
    return Fraction(-n, d);
}

void print(FILE *f) const {
    fprintf(f, "%lld / %lld", (long long int) n, (long long int) d);
}

friend std::ostream &operator<<(std::ostream &s, Fraction const &frac) {
    return s << frac.n << " / " << frac.d;
}
};

namespace std {
    template<typename T>
    class numeric_limits<Fraction<T>> {
    public:
        static constexpr Fraction<T> max() {
            return Fraction<T>(numeric_limits<T>::max());
        }
        static constexpr Fraction<T> lowest() {
            return Fraction<T>(numeric_limits<T>::lowest());
        }
        static constexpr Fraction<T> epsilon() {
            return Fraction<T>(0);
        }
    };
};

template<typename T>
Fraction<T> abs(Fraction<T> const &f) {
    return Fraction<T>(abs(f.n), abs(f.d));
}

```

```
}
```

## 8.2 Bignum

```
#pragma once
```

```
#include <algorithm>
#include <cassert>
#include <cmath>
#include <cstdio>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
#include <string_view>
#include <tuple>
#include <vector>

class Bignum {
public:
    static int const DEFAULT_BASIS = 10;
    static constexpr char DIGITS[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    std::vector<int> digits; // In reverse order!
    int basis;
    int sign; // 0 should always have sign +1

    Bignum(long long int value, int basis = DEFAULT_BASIS)
        : basis(basis), sign(value >= 0 ? 1 : -1)
    {
        value = std::abs(value);
        while (value != 0) {
            digits.push_back(value % basis);
            value /= basis;
        }

        trim();
    }

    Bignum(std::vector<int> digits, int basis, int sign)
        : digits(digits), basis(basis), sign(sign)
    {
        trim();
    }

    Bignum(std::string_view s, int basis)
        : basis(basis), sign(1)
    {
        auto it = s.begin();
        if (*it == '-') {
            sign = -1;
            ++it;
        }
    }
};
```

```
do {
    int digit = std::find(std::begin(DIGITS),
                        std::end(DIGITS), *it) - DIGITS;
    assert(digit < basis);
    digits.push_back(digit);
} while (++it != s.end());

std::reverse(digits.begin(), digits.end());

trim();
}

Bignum() : Bignum(0, DEFAULT_BASIS) {}

static Bignum from_decimal(std::string_view s, int basis = DEFAULT_BASIS) {
    return Bignum(s, 10).basis_convert(basis);
}

int n_digits() const {
    return digits.size();
}

int get_digit(int i) const {
    return i < n_digits() ? digits[i] : 0;
}

bool operator==(Bignum const &other) const {
    Bignum const &o = other.basis == basis
        ? other : other.basis_convert(basis);

    return sign == o.sign && digits == o.digits;
}

bool operator<(Bignum const &other) const {
    Bignum const &o = other.basis == basis
        ? other : other.basis_convert(basis);

    if (sign != other.sign) return sign < other.sign;

    for (int i = std::max(n_digits(), o.n_digits()) - 1; i >= 0; --i) {
        if (get_digit(i) < o.get_digit(i)) return sign == 1;
        if (get_digit(i) > o.get_digit(i)) return sign == -1;
    }

    return false;
}

bool operator>(Bignum const &other) const {
    return other < *this;
}

bool operator<=(Bignum const &other) const {
    return !(*this > other);
}
```

```

bool operator>=(Bignum const &other) const {
    return !(*this < other);
}

Bignum operator-() const {
    Bignum ans = *this;
    ans.sign = -ans.sign;

    return ans.trim();
}

Bignum abs() const {
    Bignum ans = *this;
    ans.sign = 1;
    return ans;
}

Bignum &operator+=(Bignum const &other) {
    check_basis(other);

    if (sign != other.sign) return *this -= -other;

    digits.resize(1 + std::max(n_digits(), other.n_digits()));
    int carry = 0;
    for (int i = 0; i < n_digits(); ++i) {
        int next_digit = get_digit(i) + other.get_digit(i) + carry;

        carry = next_digit / basis;
        next_digit %= basis;

        digits[i] = next_digit;
    }
    assert(carry == 0);

    return trim();
}

Bignum &operator--(Bignum const &other) {
    check_basis(other);

    if (sign != other.sign) return *this += -other;

    if (abs() < other.abs())
        return *this = -(other - *this);

    digits.resize(1 + std::max(n_digits(), other.n_digits()));
    bool borrow = false;
    for (int i = 0; i < n_digits() - 1; ++i) {
        int next_digit = get_digit(i) - other.get_digit(i);

        borrow = next_digit < 0;
        if (borrow) {
            --digits[i + 1];
            next_digit += basis;
        }

        digits[i] = next_digit;
    }

    return trim();
}

Bignum &operator+=(int n) {
    return *this += Bignum(n, basis);
}

Bignum &operator--(int n) {
    return *this -= Bignum(n, basis);
}

Bignum &operator++() {
    return *this += Bignum(1, basis);
}

Bignum &operator--() {
    return *this -= Bignum(1, basis);
}

Bignum operator+(int n) {
    return *this + Bignum(n, basis);
}

Bignum operator-(int n) {
    return *this - Bignum(n, basis);
}

friend Bignum operator+(int n, Bignum const &b) {
    return b + n;
}

friend Bignum operator-(int n, Bignum const &b) {
    return Bignum(n, b.basis) - b;
}

Bignum &operator*=(int n) {
    if (n < 0) {
        sign = -sign;
    }
}

```

```

    return *this *= -n;
}
if (n == 0) return *this = Bignum(0, basis);
if (n == 1) return *this;

if (n % 2 == 0) {
    *this += *this;
    return *this *= n / 2;
}
else {
    Bignum tmp = *this;
    return (*this *= n - 1) += tmp;
}
}

Bignum operator*(int n) const {
    Bignum ans = *this;
    return ans *= n;
}

friend Bignum operator*(int n, Bignum const &b) {
    return b * n;
}

Bignum operator*(Bignum const &other) const {
    return (std::max(n_digits(), other.n_digits()) < KARATSUBA_CUTOFF)
        ? naive_mult(other)
        : karatsuba_mult(other);
}

Bignum naive_mult(Bignum const &other) const {
    check_basis(other);

    Bignum ans(0, basis);
    for (int i = 0; i < n_digits(); ++i)
        ans += (other << i) * digits[i];

    return ans;
}

Bignum karatsuba_mult(Bignum const &other) const {
    check_basis(other);

    if (n_digits() == 1 || other.n_digits() == 1)
        return naive_mult(other);

    if (sign == -1) return -abs().karatsuba_mult(other);
    if (other.sign == -1) return -karatsuba_mult(other.abs());

    int k = std::max(n_digits(), other.n_digits()) / 2;
    Bignum a, b, c, d;
    std::tie(a, b) = split(k);
    std::tie(c, d) = other.split(k);

    // *this = a << k + b

```

```

    // other = c << k + d
    // *this * other = ac << 2k + (ad + bc) << k + bd
    // ad + bc = (a + b)(c + d) - ac - bd
    Bignum ac = a * c;
    Bignum bd = b * d;
    Bignum ad_plus_bc = (a + b) * (c + d) - ac - bd;

    return (ac << (2 * k)) + (ad_plus_bc << k) + bd;
}

Bignum &operator*=(Bignum const &other) {
    check_basis(other);

    return *this = *this * other;
}

Bignum &operator/=(int n) {
    assert(std::abs(n) < basis);

    if (n < 0) {
        sign = -sign;
        return *this /= -n;
    }
    if (n == 0) throw std::runtime_error("Division by zero");
    if (n == 1) return *this;

    int carry = 0;
    for (auto it = digits.rbegin(); it != digits.rend(); ++it) {
        long long numerator = *it + ((long long) carry) * basis;
        *it = numerator / n;
        carry = numerator % n;
    }

    return trim();
}

Bignum operator/(int n) const {
    Bignum ans = *this;
    return ans /= n;
}

Bignum operator/(Bignum const &n) const {
    if (n < 0) return -*this / -n;
    if (sign < 0) return -(abs() / n);
    if (n == 0) throw std::runtime_error("Division by zero");
    if (n == 1) return *this;

    // Binary search for last number x such that n * x <= *this
    // Total time: O(log(n) d log(d)) = O(d log^2(d))
    // where d is the number of digits in n
    Bignum lo(0, basis), hi(*this);
    while (hi - lo > 1) {
        Bignum mid = lo + (hi - lo) / 2;
        if (n * mid <= *this) lo = mid;
        else hi = mid;
    }
}

```

```

    }
    return lo;
}
Bignum &operator/=(Bignum const &n) {
    return *this = *this / n;
}

Bignum &operator<<=(int shift)
// Shifts in steps of basis, _not_ 2!
{
    digits.insert(digits.begin(), shift, 0);
    return trim();
}

Bignum &operator>>=(int shift)
// Shifts in steps of basis, _not_ 2!
{
    digits.erase(digits.begin(), digits.begin() + shift);
    return trim();
}

Bignum operator<<(int shift) const {
    Bignum ans = *this;
    return ans <<= shift;
}

Bignum operator>>(int shift) const {
    Bignum ans = *this;
    return ans >>= shift;
}

long long int value() const {
    long long int ans = 0;
    for (auto it = digits.rbegin(); it != digits.rend(); ++it)
        ans = basis * ans + *it;

    return sign * ans;
}

Bignum basis_convert(int new_basis) const {
    if (new_basis == basis) return *this;

    Bignum ans(0, new_basis);

    // Powers of the basis written in the new basis
    Bignum b(1, new_basis);

    for (int digit : digits) {
        ans += digit * b;
        b *= basis;
    }

    ans.sign = sign;
}

```

```

    return ans;
}

friend std::ostream &operator<<(std::ostream &s, Bignum const &n) {
    std::vector<int> const &digits = n.basis_convert(10).digits;
    if (n.sign == -1) s << '-';
    for (auto it = digits.rbegin(); it != digits.rend(); ++it)
        s << *it;

    return s;
}

void print(FILE *f) const {
    std::vector<int> digits = basis_convert(10).digits;
    if (sign == -1) fprintf(f, "%c", '-');
    for (auto it = digits.rbegin(); it != digits.rend(); ++it)
        fprintf(f, "%d", *it);
}

std::string str() const {
    std::ostringstream ss;
    ss << *this;
    return ss.str();
}

std::ostream &repr(std::ostream &s) const {
    assert(basis < (int) sizeof(DIGITS));

    if (sign == -1) s << '-';
    std::for_each(digits.rbegin(), digits.rend(),
        [&] (int d) { s << DIGITS[d]; });

    return s;
}

std::string repr() const {
    std::ostringstream ss;
    repr(ss);
    return ss.str();
}

private:
    static int const KARATSUBA_CUTOFF = 10;

    Bignum &trim() {
        int i = n_digits() - 1;
        while (i >= 1 && digits[i] == 0) --i;
        digits.resize(i + 1);

        // Handle the case of 0
        if (digits.empty()) digits.resize(1);
        if (n_digits() == 1 && digits[0] == 0) sign = 1;

        return *this;
    }
}

```

```

void check_basis(Bignum const &other) const {
    assert(basis == other.basis);
}

std::pair<Bignum, Bignum> split(int shift) const
// Returns (a, b) such that n = a << shift + b and b < 1 << shift.
// That is, b is the first `shift` digits, a is the remaining digits.
{
    if (shift >= n_digits()) return {Bignum(0, basis), *this};

    std::vector<int> a_digits(n_digits() - shift), b_digits(shift);
    std::copy_n(digits.begin(), shift, b_digits.begin());
    std::copy(digits.begin() + shift, digits.end(), a_digits.begin());

    return {Bignum(a_digits, basis, sign), Bignum(b_digits, basis, sign)};
}

};

namespace std {
    Bignum abs(Bignum n) {
        return n.abs();
    }
}

```

## 8.3 Matrix

```
#pragma once
```

```
#include <vector>
```

```

template<typename T>
struct Matrix {
    int n_rows, n_cols;
    std::vector<std::vector<T>> data;

    Matrix(std::vector<std::vector<T>> data)
        : n_rows(data.size()), n_cols(data[0].size()), data(data) {}

    Matrix(int n_rows, int n_cols)
        : n_rows(n_rows), n_cols(n_cols),
          data(std::vector<std::vector<T>>(n_rows, std::vector<T>(n_cols))) {}

    static Matrix eye(int n) {
        Matrix A(n, n);
        for (int i = 0; i < n; ++i)
            A[i][i] = 1;

        return A;
    }

    static Matrix row_vector(std::vector<T> const &v) {
        return Matrix(std::vector<std::vector<T>>({v}));
    }
}

```

```

static Matrix column_vector(std::vector<T> const &v) {
    Matrix A(v.size(), 1);
    for (int i = 0; i < (int) v.size(); ++i)
        A[i][0] = v[i];

    return A;
}

std::vector<T> const &operator[](int i) const { return data[i]; }
std::vector<T> &operator[](int i) { return data[i]; }

bool operator==(Matrix const &other) const {
    return data == other.data;
}

Matrix &operator+=(Matrix const &other) {
    assert(n_rows == other.n_rows && n_cols == other.n_cols);
    for (int i = 0; i < n_rows; ++i)
        for (int j = 0; j < n_cols; ++j)
            (*this)[i][j] += other[i][j];
}

Matrix &operator-=(Matrix const &other) {
    assert(n_rows == other.n_rows && n_cols == other.n_cols);
    for (int i = 0; i < n_rows; ++i)
        for (int j = 0; j < n_cols; ++j)
            (*this)[i][j] -= other[i][j];
}

Matrix &operator*=(T const &factor) {
    for (std::vector<T> &row : data)
        for (T &x : row)
            x *= factor;
}

Matrix &operator/=(T const &factor) {
    for (std::vector<T> &row : data)
        for (T &x : row)
            x /= factor;
}

Matrix operator+(T const &other) const {
    Matrix A(*this);
    A += other;
    return A;
}

Matrix operator-(T const &other) const {
    Matrix A(*this);
    A -= other;
    return A;
}

Matrix operator*(T const &factor) const {

```

```

    Matrix A(*this);
    A *= factor;
    return A;
}

Matrix operator/(T const &factor) const {
    Matrix A(*this);
    A /= factor;
    return A;
}

Matrix operator*(Matrix const &other) const {
    assert(n_cols == other.n_rows);
    Matrix A(n_rows, other.n_cols);
    for (int i = 0; i < n_rows; ++i)
        for (int j = 0; j < other.n_cols; ++j)
            for (int k = 0; k < n_cols; ++k)
                A[i][j] += (*this)[i][k] * other[k][j];

    return A;
}

Matrix &operator*=(Matrix const &other) {
    *this = *this * other;
}
};

```

## 8.4 Polynomial

```
#pragma once
```

```
#include <algorithm>
#include <cassert>
#include <tuple>
#include <vector>
```

```
template<typename T = long long int>
class Polynomial {
private:
    static constexpr int KARATSUBA_CUTOFF = 100;
```

```

    Polynomial &trim() {
        int i = degree();
        while (i >= 1 && coeff[i] == 0) --i;
        coeff.resize(i + 1);

        return *this;
    }

```

```

    std::pair<Polynomial, Polynomial> split(int k) const
    // Split as  $p = a x^k + b$ , return {a, b}
    {
        if (k > (int) coeff.size())
            return {Polynomial(0), *this};
    }

```

```

    std::vector<T> b(k), a(coeff.size() - k);

    std::copy_n(coeff.begin(), k, b.begin());
    std::copy(coeff.begin() + k, coeff.end(), a.begin());

    return {Polynomial(a), Polynomial(b)};
}

Polynomial shift(int n)
// Multiply by  $x^n$ 
{
    assert(n >= 0);

    Polynomial ans(std::vector<T>(coeff.size() + n));
    std::copy(coeff.begin(), coeff.end(), ans.coeff.begin() + n);

    return ans;
}

public:
    std::vector<T> coeff; // Constant term first

    Polynomial(std::vector<T> coeff) : coeff(coeff) {}
    Polynomial(T c) : coeff({c}) {}
    Polynomial() : coeff({0}) {}

    int degree() const {
        return coeff.size() - 1;
    }

    Polynomial &operator+=(Polynomial const &other) {
        if (other.degree() > degree())
            coeff.resize(other.coeff.size());

        for (int i = 0; i <= other.degree(); ++i)
            coeff[i] += other.coeff[i];

        return trim();
    }

    Polynomial &operator-=(Polynomial const &other) {
        if (other.degree() > degree())
            coeff.resize(other.coeff.size());

        for (int i = 0; i <= other.degree(); ++i)
            coeff[i] -= other.coeff[i];

        return trim();
    }

    Polynomial operator+(Polynomial const &other) const {
        Polynomial p(*this);
        p += other;
        return p;
    }
}

```

```

Polynomial operator-(Polynomial const &other) const {
    Polynomial p(*this);
    p -= other;
    return p;
}

Polynomial operator-() const {
    return *this * Polynomial(-1);
}

Polynomial operator*(Polynomial const &other) const {
    if (std::max(degree(), other.degree()) < KARATSUBA_CUTOFF)
        return naive_mult(other);
    else
        return karatsuba_mult(other);
}

Polynomial naive_mult(Polynomial const &other) const {
    int new_degree = degree() + other.degree();
    Polynomial ans(std::vector<T>(new_degree + 1, 0));

    for (int i = 0; i <= degree(); ++i)
        for (int j = 0; j <= other.degree(); ++j)
            ans.coeff[i + j] += coeff[i] * other.coeff[j];

    return ans.trin();
}

Polynomial karatsuba_mult(Polynomial const &other) const {
    if (degree() == 0 || other.degree() == 0)
        return naive_mult(other);

    int k = std::max(coeff.size(), other.coeff.size()) / 2;
    Polynomial a, b, c, d;
    std::tie(a, b) = split(k);
    std::tie(c, d) = other.split(k);

    // (a x^k + b)(c x^k + d) = ac x^2k + (ad + bc) x^k + bd
    // ad + bc = (a + b)(c + d) - ac - bd
    Polynomial ac = a * c;
    Polynomial bd = b * d;
    Polynomial ad_plus_bc = (a + b) * (c + d) - ac - bd;

    return ac.shift(2 * k) + ad_plus_bc.shift(k) + bd;
}
};

```