scripts.sh 1

```
#!/bin/bash
set -e

# > Normal build
g++ -O2 -Wall -Wextra -std=c++11 -o $1.e $1.cpp

# > Debug build
g++ -fsanitize=address -fsanitize=undefined -D_GLIBCXX_DEBUG \
    -O2 -Wall -Wextra -std=c++11 -o $1.e $1.cpp

# Mac:         -fvisibility=hidden
# Stack limit: -Wl,-stack_size -Wl,16000000 -Wl,-no_pie

# > Compare
./build.sh $1; ./build.sh $2; ./build.sh $3

while :; do
     $3.e > cmp.in; echo -n 0
     $1.e < cmp.in > prog1.out; echo -n 1
     $2.e < cmp.in > prog2.out; echo -n 2
     diff prog1.out prog2.out
     echo -n Y
done
```

template.h 2

```
#include <bits/stdc++.h>
using namespace std;

using ll    = int64_t;
using ull   = uint64_t;
using ld    = long double;
using cmpl  = complex<double>;

#define IT iterator

#define rep(i, b, e)  for (int i = int(b); i < int(e); i++)
#define repd(i, b, e) for (int i = int(b); i >= int(e); i--)
#define each(a, x)    for (auto& a : x)
#define all(x)        (x).begin(), (x).end()
#define sz(x)         int((x).size())

#define gcd           __gcd
#define popcount      __builtin_popcount

// unique_ptr without deallocation
template<typename T>
struct single_ptr {
    T* elem{0};

    single_ptr()                                {}
    single_ptr(nullptr_t)                       {}
    single_ptr(T* v)           : elem(v)        {}
    single_ptr(single_ptr&& r) : elem(r.elem) { r.elem = 0; }

    single_ptr& operator=(nullptr_t)      { elem = 0; return *this; }
    single_ptr& operator=(single_ptr&& r) { elem = r.elem; r.elem = 0; return *this; }

    T* operator->() { return elem; }
    T& operator*()  { return *elem; }
    operator bool() { return elem; }
};
```

math/bellman_inequalities.h 3

```
struct Constraint {
    ll a, b, limit; // a - b >= limit
};

bool solveIneq(vector<Constraint>& edges, vector<ll>& vars) {
    rep(i, 0, sz(vars)) each(e, edges) vars[e.b] = min(vars[e.b], vars[e.a] - e.limit);
    each(e, edges) if (vars[e.a] - e.limit < vars[e.b]) return false;
    return true;
}
```

math/fft.h 4

```
vector<cmpl> bases;

void initFft(int size) {
    bases.resize(size+1);
    rep(i, 0, size+1) bases[i] = exp(cmpl(0, 2*M_PI*i/size));
}

template<bool inv>
void fft(vector<cmpl>::IT in, vector<cmpl>::IT out, int size, int step = 1) {
    if (size == 1) { *out = *in; return; }

    fft<inv>(in,      out,        size*2, step*2);
    fft<inv>(in+step, out+size/2, size*2, step*2);

    rep(i, 0, size/2) {
        auto t = out[i], m = bases[(inv ? i : size-i)*step];
        out[i]        = t + out[i+size/2]*m;
        out[i+size/2] = t - out[i+size/2]*m;
    }
}
```

math/mod_inv.h 5

```
template<class T>
T modInv(T a, T b) {
    T u = 1, v = 0, x = 0, y = 1, m = b;

    while (a > 0) {
        T q = b / a, r = b % a;
        T m = x - u*q, n = y - v*q;
        b = a; a = r; x = u; y = v; u = m; v = n;
    }
    return (b == 1 ? (x < 0 ? x+m : x) : 0);
}
```

math/montgomery.h 6

```
#include "mod_inv.h"

constexpr ll MG_SHIFT = 32;
constexpr ll MG_MULT  = 1LL << MG_SHIFT;
constexpr ll MG_MASK  = MG_MULT - 1;

ll getMgInv(ll mod)      { return MG_MULT - modInv(mod, MG_MULT); }
ll mgShift(ll n, ll mod) { return (n * MG_MULT) % mod; } // Precompute multipliers

ll redc(ll n, ll mod, ll mgInv) {
    ll quot = (n * mgInv) & MG_MASK;
    n = (n + quot*mod) >> MG_SHIFT;
    return (n >= mod ? n-mod : n);
}

// MOD < MG_MULT, gcd(MG_MULT, MOD) must be 1
// mgRedc(mgForm1 * mgForm2)    = Montgomery-form product
// mgRedc(notMgForm1 * mgForm2) = normal number
```

structures/interval_tree.h 7

```
struct IntervalTree {
    using T = int;

    // (+, max)
    static constexpr T T_IDENT = INT_MIN;
    #define opModify(x, y) ((x)+(y))
    #define opQuery(x, y)  max(x, y)
```

```
#define opTimes(x, t)  (x)

// (max, max)
// static constexpr T T_IDENT = INT_MIN;
// #define opModify(x, y) max(x, y)
// #define opQuery(x, y)  max(x, y)
// #define opTimes(x, t)  (x)

// (+, +)
// static constexpr T T_IDENT = 0;
// #define opModify(x, y) ((x)+(y))
// #define opQuery(x, y)  ((x)+(y))
// #define opTimes(x, t)  ((x)*(t))

// (=, max)
// static constexpr T T_IDENT = INT_MIN;
// #define opQuery(x, y)  max(x, y)
// #define opTimes(x, t)  (x)

struct Node {
    T   val{0}, extra{0};
    int nEqual{1};       // EXTENSION: min/max element count
    bool current{false}; // EXTENSION: (set, ?) tree
};

vector<Node> tree;
int len;

IntervalTree(int size) {
    for (len = 1; len < size; len *= 2);
    tree.resize(len*2);

    // EXTENSION: min/max element count
    repd(i, len-1, 0) tree[i].nEqual = tree[i*2].nEqual + tree[i*2+1].nEqual;
}

T query(int vStart, int vFinish, int i = 1, int begin = 0, int end = -1) {
    if (end < 0) end = len;
    // push(i, begin, end); // EXTENSION: (set, ?) tree

    if (vStart >= vFinish || vFinish <= begin || end <= vStart) return T_IDENT;
    if (vStart <= begin && end <= vFinish)                      return tree[i].val;

    int mid = (begin + end) / 2;
    T tmp = opQuery(query(vStart, vFinish, i*2, begin, mid),
                    query(vStart, vFinish, i*2+1, mid, end));

    // return tmp; // EXTENSION: (set, ?) tree -> use this instead of following line
    return opModify(tmp, opTimes(tree[i].extra, min(end, vFinish)-max(begin, vStart)));
}

// EXTENSION: min/max element count -> use this instead of above
T queryCount(int vStart, int vFinish, int& count, int i = 1, int begin = 0, int end = -1) {
    if (end < 0) end = len;
    if (vStart >= vFinish || vFinish <= begin || end <= vStart) return T_IDENT;
    // push(i, begin, end); // EXTENSION: (set, ?) tree

    if (vStart <= begin && end <= vFinish) {
        count = tree[i].nEqual;
        return tree[i].val;
    }

    int eq1 = 0, eq2 = 0, mid = (begin + end) / 2;
    T v1  = queryCount(vStart, vFinish, eq1, i*2, begin, mid);
    T v2  = queryCount(vStart, vFinish, eq2, i*2+1, mid, end);
    T tmp = opQuery(v1, v2);

    count = (v1 == tmp ? eq1 : 0) + (v2 == tmp ? eq2 : 0);
    // return tmp; // EXTENSION: (set, ?) tree -> use this instead of following line
    return opModify(tmp, opTimes(tree[i].extra, min(end, vFinish)-max(begin, vStart)));
}

    void modify(int vStart, int vFinish, T val, int i = 1, int begin = 0, int end = -1) {
        if (end < 0) end = len;
        if (vFinish <= begin || end <= vStart) return;

        if (vStart > begin || end > vFinish) {
            int mid = (begin + end) / 2;
            modify(vStart, vFinish, val, i*2, begin, mid);
            modify(vStart, vFinish, val, i*2+1, mid, end);
        } else {
            tree[i].extra = opModify(tree[i].extra, val);
        }

        if (i < len) tree[i].val = opModify(opQuery(tree[i*2].val, tree[i*2+1].val),
                                            opTimes(tree[i].extra, end-begin));
        else         tree[i].val = tree[i].extra;

        // EXTENSION: min/max element count
        tree[i].nEqual = (i >= len);
        if (i < len) {
            T tmp = opQuery(tree[i*2].val, tree[i*2+1].val);
            if (tmp == tree[i*2].val)   tree[i].nEqual += tree[i*2].nEqual;
            if (tmp == tree[i*2+1].val) tree[i].nEqual += tree[i*2+1].nEqual;
        }
    }

    // EXTENSION: (set, ?) tree -> use this instead of above
    // void modify(int vStart, int vFinish, T val, int i = 1, int begin = 0, int end = -1) {
    //  if (end < 0) end = len;
    //  if (vFinish <= begin || end <= vStart) return;
    //  push(i, begin, end);

    //  if (vStart > begin || end > vFinish) {
    //      int mid = (begin + end) / 2;
    //      modify(vStart, vFinish, val, i*2, begin, mid);
    //      modify(vStart, vFinish, val, i*2+1, mid, end);
    //  } else {
    //      tree[i].extra = val;
    //      tree[i].val = opTimes(val, end-begin);
    //      tree[i].current = true;
    //      tree[i].nEqual = end-begin; // EXTENSION: min/max element count
    //      return;
    //  }

    //  if (i < len) tree[i].val = opQuery(tree[i*2].val, tree[i*2+1].val);

    //  // EXTENSION: min/max element count
    //  tree[i].nEqual = (i >= len);
    //  if (i < len) {
    //      T tmp = opQuery(tree[i*2].val, tree[i*2+1].val);
    //      if (tmp == tree[i*2].val)   tree[i].nEqual += tree[i*2].nEqual;
    //      if (tmp == tree[i*2+1].val) tree[i].nEqual += tree[i*2+1].nEqual;
    //  }
    // }

    // EXTENSION: (set, ?) tree
    void push(int i, int begin, int end) {
        if (!tree[i].current || i >= len) return;
        tree[i*2].val     = tree[i*2+1].val     = opTimes(tree[i].extra, (end-begin)/2);
        tree[i*2].extra   = tree[i*2+1].extra   = tree[i].extra;
        tree[i*2].current = tree[i*2+1].current = true;
        tree[i].current   = false;

        // EXTENSION: min/max element count
        tree[i*2].nEqual = tree[i*2+1].nEqual = (end-begin)/2;
    }
};
```

structures/pairing_heap.h


```
template<class T, class Cmp = less<T>>
```

```
struct PHeap {
    struct Node;
    using NodeP = unique_ptr<Node>; // Or use single_ptr + bump allocator

    struct Node {
        T val;
        NodeP child{nullptr}, next{nullptr};
        Node* prev{nullptr};

        Node(T x = T()) { val = x; }

        NodeP moveChild() {
            if (child) child->prev = nullptr;
            return move(child);
        }

        NodeP moveNext() {
            if (next) next->prev = nullptr;
            return move(next);
        }

        void setChild(NodeP v) {
            if (child) child->prev = nullptr;
            child = move(v);
            if (child) child->prev = this;
        }

        void setNext(NodeP v) {
            if (next) next->prev = nullptr;
            next = move(v);
            if (next) next->prev = this;
        }
    };

    NodeP root{nullptr};

    NodeP merge(NodeP l, NodeP r) {
        if (!l) return move(r);
        if (!r) return move(l);

        if (Cmp()(l->val, r->val)) swap(l, r);

        l->setNext(r->moveChild());
        r->setChild(move(l));
        return move(r);
    }

    NodeP mergePairs(NodeP v) {
        if (!v || !v->next) return v;
        NodeP v2 = v->moveNext(), v3 = v2->moveNext();
        return merge(merge(move(v), move(v2)), mergePairs(move(v3)));
    }

    Node* push(const T& x) {
        NodeP tmp(new Node(x));
        auto ret = &*tmp;
        root = merge(move(root), move(tmp));
        return ret;
    }

    void decrease(Node* v, T val) {
        assert(!Cmp()(v->val, val));
        v->val = val;

        auto prev = v->prev;
        if (!prev) return;
        NodeP uniq;

        if (&*v->prev->child == v) {
            uniq = prev->moveChild();
            prev->setChild(v->moveNext());
        } else {
            uniq = prev->moveNext();
            prev->setNext(v->moveNext());
        }

        root = merge(move(root), move(uniq));
    }

    bool     empty()          { return !root; }
    const T& top()            { return root->val; }
    void     merge(PHeap&& r) { root = merge(move(root), move(r.root)); r.root = nullptr; }
    void     pop()            { root = mergePairs(root->moveChild()); }
};
```

trees/centroid_decomp.h


```
struct Vert {
    vector<Vert*> edges, cEdges;
    vector<int> dists;
    Vert* cParent{nullptr};
    int cDepth{-1}, cSize{0}, cState{0};
};

void dfsSize(Vert* v, int depth) {
    v->cDepth = depth;
    v->cSize = 1;
    v->cState = 0;

    each(e, v->edges) if (e->cState <= 1 && e->cDepth < depth) {
        dfsSize(e, depth);
        v->cSize += e->cSize;
    }
}

void dfsDist(Vert* v, int dist) {
    v->dists.push_back(dist);
    v->cState = 1;
    each(e, v->edges) if (!e->cState) dfsDist(e, dist+1);
}

Vert* centroidDecomp(Vert* v, int depth, Vert* root = 0) {
    dfsSize(v, depth);

    int size = v->cSize;
    Vert *parent = 0, *heavy = 0;

    while (true) {
        int hSize = 0;

        each(e, v->edges) if (e != parent && e->cDepth == depth && hSize < e->cSize) {
            hSize = e->cSize;
            heavy = e;
        }

        if (hSize <= size/2) break;
        parent = v; v = heavy;
    }

    v->cParent = root;
    dfsDist(v, 0);
    v->cSize = size;
    v->cState = 2;

    each(e, v->edges) if (e->cDepth == depth) {
        v->cEdges.push_back(centroidDecomp(e, depth+1, v));
    }
    return v;
}
```

util/alloc.h


```
char memPool[512*1024*1024];
```

```
int memOffset;

void* operator new(size_t n) { memOffset += n; return &memPool[memOffset-n]; }
void  operator delete(void*) {}
```