goon

Andwerp, dmot, Duckling

Texas A&M University

February 27, 2024

1 Base Template

```cpp
#include <bits/stdc++.h>
typedef long long ll;
typedef __int128 lll;
typedef long double ld;
typedef __float128 lld;
using namespace std;

signed main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    return 0;
}
```

2 Range Queries

2.1 Segment Tree

Allows for single element modification, and range sum queries, for any binary associative operation.

```cpp
template <typename T>
struct Segtree {
    //note that t[0] is not used
    int n;
    T* t;
    T uneut, qneut;

    //single element modification function
    function<T(T, T)> fmodify;

    //product of two elements for query and updating tree
    function<T(T, T)> fcombine;

    Segtree(int n, T updateNeutral, T queryNeutral,
         function<T(T, T)> fmodify, function<T(T, T)>
         fcombine) {
        this -> n = n;
        t = new T[2 * n];

        this -> fmodify = fmodify;
        this -> fcombine = fcombine;

        uneut = updateNeutral;
        qneut = queryNeutral;

        for(int i = 0; i < n; i++){
            t[i + n] = uneut;
        }
        build();
    }

    void build() { // build the tree after manually assigning
         the values.
        for (int i = n - 1; i > 0; i--) {
            t[i] = fcombine(t[i * 2], t[i * 2 + 1]);
        }
    }

    void modify(int p, T value) { // set value at position p
        p += n;
        t[p] = fmodify(t[p], value);
        for (p /= 2; p > 0; p /= 2) {
            t[p] = fcombine(t[p * 2], t[p * 2 + 1]);
        }
    }

    T query(int l, int r) { // sum on interval [l, r)
        T l_res = qneut, r_res = qneut;
        bool l_none = true, r_none = true;
        for (l += n, r += n; l < r; l /= 2, r /= 2) {
            if (l % 2 == 1) {
                if(l_none) {
                    l_none = false;
                    l_res = t[l];
                }
                else {
                    l_res = fcombine(l_res, t[l]);
                }
                l++;
            }
            if (r % 2 == 1) {
                r--;
                if(r_none) {
                    r_none = false;
                    r_res = t[r];
                }
                else {
                    r_res = fcombine(t[r], r_res);
                }
            }
        }
        if(l_none) {
            return r_res;
        }
        if(r_none) {
            return l_res;
        }
        return fcombine(l_res, r_res);
    }

    T query(int ind) {
        return this->query(ind, ind + 1);
    }
};

//useful examples:
// -- INCREMENT MODIFY, SUM QUERY --
{
    function<int(int, int)> fmodify = [](const int src, const
         int val) -> int{return src + val;};
    function<int(int, int)> fcombine = [](const int a, const
         int b) -> int{return a + b;};
    Segtree<int> segt(n, 0, 0, fmodify, fcombine);
}

// -- ASSIGNMENT MODIFY, MIN QUERY --
{
    function<int(int, int)> fmodify = [](const int src, const
         int val) -> int{return val;};
    function<int(int, int)> fcombine = [](const int a, const
         int b) -> int{return min(a, b);};
    Segtree<int> segt(n, 0, 1e9, fmodify, fcombine);
}

// -- INCREMENT MODIFY, MAX SUBARRAY SUM QUERY --
//subarray has to contain at least 1 element.
{
    struct seg {
        ll max_pfx, max_sfx, max_sum, sum;
        seg() {};
        seg(ll sum) {
            this->sum = sum;
        }
        seg(ll max_pfx, ll max_sfx, ll max_sum, ll sum) {
            this->max_pfx = max_pfx;
            this->max_sfx = max_sfx;
            this->max_sum = max_sum;
            this->sum = sum;
        }
    };
    function<seg(seg, seg)> fmodify = [](const seg src, const
         seg val) -> seg{
        seg next;
        next.max_pfx = src.max_pfx + val.sum;
        next.max_sfx = src.max_sfx + val.sum;
        next.max_sum = src.max_sum + val.sum;
        next.sum = src.sum + val.sum;
        return next;
    };
    function<seg(seg, seg)> fcombine = [](const seg lhs,
         const seg rhs) -> seg{
        seg next;
        next.max_pfx = max(lhs.max_pfx, lhs.sum + rhs.max_pfx);
        next.max_sfx = max(rhs.max_sfx, rhs.sum + lhs.max_sfx);
        next.max_sum = max({lhs.max_sum, rhs.max_sum,
             lhs.max_sfx + rhs.max_pfx});
        next.sum = lhs.sum + rhs.sum;
        return next;
    };
    Segtree<seg> segt(n, {0, 0, 0, 0}, {0, 0, 0, 0}, fmodify,
         fcombine);
}

// -- ASSIGNMENT MODIFY, XOR BASIS QUERY --
//returns the xor basis of a range. Refer to
     https://codeforces.com/blog/entry/68953
{
    struct seg{
        int basis[20];
        int nr_b = 0;
        seg(int val) {
            fill(basis, basis + 20, -1);
            nr_b += basisAdd(val);
        }
        seg() {
            fill(basis, basis + 20, -1);
        }
        bool basisAdd(int val, int start = 0) {
            for(int i = start; i < 20; i++){
                if((val & 1 << i) == 0){
                    continue;
                }
                if(basis[i] == -1){
                    basis[i] = val;
                    return true;
                }
                val ^= basis[i];
            }
            return false;
        }
    };
    function<seg(seg, seg)> fmodify = [](const seg src, const
         seg val) -> seg{
        return val;
    };
    function<seg(seg, seg)> fcombine = [](const seg a, const
         seg b) -> seg{
        if(a.nr_b == 20){
            return a;
        }
        if(b.nr_b == 20){
            return b;
        }
        seg next;
        for(int i = 0; i < 20; i++){
            next.basis[i] = a.basis[i];
        }
        next.nr_b = a.nr_b;
        for(int i = 19; i >= 0; i--){
            if(b.basis[i] != -1){
                next.basisAdd(b.basis[i], i);
            }
        }
        return next;
    };
    Segtree<seg> segt(n, {}, {}, fmodify, fcombine);
}
```

2.2 Lazy Segment Tree

Allows for range modification, and range sum query for any binary associative operation.

```cpp
template <typename T>
struct SegtreeLazy {
    public:
        int n;
        T* t;   //stores product of range
```

```
    T* d;   //lazy tree
    bool* upd; //marks whether or not a lazy change is here
    T uneut, qneut;

    //single element modify
    function<T(T, T)> fmodify;

    //k element modify
    function<T(T, T, int)> fmodifyk;

    //product of two elements for query
    function<T(T, T)> fcombine;

    SegtreeLazy(int maxSize, T updateNeutral, T
        queryNeutral, T initVal, function<T(T, T)>
        fmodify, function<T(T, T, int)> fmodifyk,
        function<T(T, T)> fcombine) {
        n = maxSize;
        uneut = updateNeutral;
        qneut = queryNeutral;

        this -> fmodify = fmodify;
        this -> fmodifyk = fmodifyk;
        this -> fcombine = fcombine;

        //raise n to nearest pow 2
        int x = 1;
        while(x < n) {
            x <<= 1;
        }
        n = x;

        t = new T[n * 2];
        d = new T[n * 2];
        upd = new bool[n * 2];

        //make sure to initialize values
        for(int i = 0; i < n; i++){
            t[i + n] = initVal;
        }
        for(int i = n - 1; i > 0; i--){
            t[i] = fcombine(t[i * 2], t[i * 2 + 1]);
        }
        for(int i = 0; i < n * 2; i++){
            d[i] = uneut;
            upd[i] = false;
        }
    }

    void modify(int l, int r, T val) { //modifies the
        range [l, r)
        _modify(l, r, val, 0, n, 1);
    }

    void modify(int ind, T val) { //modifies the range
        [ind, ind + 1)
        _modify(ind, ind + 1, val, 0, n, 1);
    }

    T query(int l, int r) { //queries the range [l, r)
        return _query(l, r, 0, n, 1);
    }

    T query(int ind) {  //queries the range [ind, ind + 1)
        return _query(ind, ind + 1, 0, n, 1);
    }
private:
    //calculates value of node based off of children
    //k is the amount of values that this node represents.
    void combine(int ind, int k) {
        if(ind >= n){
            return;
        }
        int l = ind * 2;
        int r = ind * 2 + 1;
        //make sure children are correct value before
            calculating
        push(l, k / 2);
        push(r, k / 2);
        t[ind] = fcombine(t[l], t[r]);
    }

    //registers a lazy change llo this node
    void apply(int ind, T val) {
        upd[ind] = true;
        d[ind] = fmodify(d[ind], val);
    }

    //applies lazy change to this node
    //k is the amount of values that this node represents.
    void push(int ind, int k) {
        if(!upd[ind]) {
            return;
        }
        t[ind] = fmodifyk(t[ind], d[ind], k);
        if(ind < n) {
            int l = ind * 2;
            int r = ind * 2 + 1;
            apply(l, d[ind]);
            apply(r, d[ind]);
        }
        upd[ind] = false;
        d[ind] = uneut;
    }

    void _modify(int l, int r, T val, int tl, int tr, int
        ind) {
        if(l == r){
            return;
        }
        if(upd[ind]) {
            push(ind, tr - tl);
        }
        if(l == tl && r == tr) {
            apply(ind, val);
            push(ind, tr - tl);
            return;
        }
        int mid = tl + (tr - tl) / 2;
        if(l < mid) {
            _modify(l, min(r, mid), val, tl, mid, ind * 2);
        }
        if(r > mid) {
            _modify(max(l, mid), r, val, mid, tr, ind * 2 +
                1);
        }
        combine(ind, tr - tl);
    }

    T _query(int l, int r, int tl, int tr, int ind) {
        if(l == r){
            return qneut;
        }
        if(upd[ind]) {
            push(ind, tr - tl);
        }
        if(l == tl && r == tr){
            return t[ind];
        }
        int mid = tl + (tr - tl) / 2;
        T lans = qneut;
        T rans = qneut;
        if(l < mid) {
            lans = _query(l, min(r, mid), tl, mid, ind * 2);
        }
        if(r > mid) {
            rans = _query(max(l, mid), r, mid, tr, ind * 2
                + 1);
        }
        return fcombine(lans, rans);
    }
};

//useful examples
// -- ASSIGNMENT MODIFY, SUM QUERY --
{
    function<int(int, int)> fmodify = [](const int src, const
        int val) -> int{return val;};
    function<int(int, int, int)> fmodifyk = [](const int src,
        const int val, const int k) -> int{return val * k;};
    function<int(int, int)> fcombine = [](const int a, const
        int b) -> int{return a + b;};
    run_segt_tests(n, 0, 0, fmodify, fmodifyk, fcombine);
}

// -- INCREMENT MODIFY, MINIMUM QUERY --
{
    function<int(int, int)> fmodify = [](const int src, const
        int val) -> int{return src + val;};
    function<int(int, int, int)> fmodifyk = [](const int src,
        const int val, const int k) -> int{return src + val;};
    function<int(int, int)> fcombine = [](const int a, const
        int b) -> int{return min(a, b);};
    run_segt_tests(n, 0, 1e9, fmodify, fmodifyk, fcombine);
}

// -- ASSIGNMENT MODIFY, MINIMUM QUERY --
{
    function<int(int, int)> fmodify = [](const int src, const
        int val) -> int{return val;};
    function<int(int, int, int)> fmodifyk = [](const int src,
        const int val, const int k) -> int{return val;};
    function<int(int, int)> fcombine = [](const int a, const
        int b) -> int{return min(a, b);};
    run_segt_tests(n, 0, 1e9, fmodify, fmodifyk, fcombine);
}

// -- 01 ASSIGNMENT MODIFY, FIRST LAST INDEX 01 QUERY --
// when querying, returns a struct that tells you the first
    and last indices of 0 and 1
// in the range that you queried.
{
    struct seg {
        int size;
        bool has_one, has_zero;
        int pfx_one, sfx_one; //distance from beginning and
            end of closest 1
        int pfx_zero, sfx_zero;
        seg(int size, bool which) { //sets the entire segment
            the same value
            this->size = size;
            has_one = which;
            has_zero = !which;
            pfx_one = 0;
            sfx_one = 1;
            pfx_zero = 0;
            sfx_zero = 1;
        }
        seg(int size){
            this->size = size;
            has_one = false;
            has_zero = false;
        }
        seg() {}
    };
    //assignment modify, range 'seg' query.
    function<seg(seg, seg)> fmodify = [](const seg src, const
        seg val) -> seg{
        //set this element to 0 or 1
        seg next(src.size, val.has_one);
        return next;
    };
    function<seg(seg, seg, int)> fmodifyk = [](const seg src,
        const seg val, const int k) -> seg{
        //set the entire range to 0 or 1
        seg next(src.size, val.has_one);
        return next;
    };
    function<seg(seg, seg)> fcombine = [](const seg lhs,
        const seg rhs) -> seg{
        //combines lhs and rhs into one segment
        seg next(lhs.size + rhs.size, false);
        next.has_one = lhs.has_one || rhs.has_one;
        next.has_zero = lhs.has_zero || rhs.has_zero;
        if(next.has_one) {
            next.pfx_one = lhs.has_one? lhs.pfx_one : lhs.size
```

```
        + rhs.pfx_one;
      next.sfx_one = rhs.has_one? rhs.sfx_one : rhs.size
        + lhs.sfx_one;
    }
    if(next.has_zero) {
      next.pfx_zero = lhs.has_zero? lhs.pfx_zero :
        lhs.size + rhs.pfx_zero;
      next.sfx_zero = rhs.has_zero? rhs.sfx_zero :
        rhs.size + lhs.sfx_zero;
    }
    return next;
  };
  SegtreeLazy<seg> segt(n, {0}, {0}, {1, false}, fmodify,
    fmodifyk, fcombine);
  //example use
  segt.modify(0, {0, false}); //set range to 0
  segt.modify(1, 5, {0, true}); //set range to 1
  seg a = segt.query(5, 10); //get attr over range
}
```

2.3 Lazy Segment Tree (Duckling)

```
template<typename T, typename D>
struct Lazy {
  static constexpr T qn = 0; //stores the starting values
    at all nodes,
  static constexpr D ln = 0;
  vector<T> v;    //stores values at each index we are
    querying for
  vector<D> lazy; //base, count of how many polynomials
    start at one at the beginning of this node
  int n, size;
  //if OJ is not up to date, remove all occurrences of ln
  Lazy(int n = 0, T def = qn) {
    this->n = n;
    this->size = 1;
    while(size < n) size *= 2;
    v.assign(size * 2, def);
    lazy.assign(size * 2, ln);
  }
  bool isLeaf(int node) {
    return node >= size;
  }
  T query_comb(T val1, T val2) {//update this depending on
    query type
    return val1 + val2;
  }
  //how we combine lazy updates to lazy
  void lazy_comb(int node, D val) {//update this depending
    on update type. how do we merge the lazy changes?
    lazy[node] += val;
  }
  void main_comb(int node, int size) {//update this
    depending on query type, how does the lazy value
    affect value at v for the query?
    v[node] += lazy[node];
  }
  void push_lazy(int node, int size) {
    main_comb(node, size); //push lazy change to current
      node
    if(!isLeaf(node)) {
      lazy_comb(node * 2, lazy[node]);
      lazy_comb(node * 2 + 1, lazy[node]);
    }
    lazy[node] = ln;
  }
  void update(int l, int r, D val) {
    _update(1,0,size,l,r, val);
  }
  void _update(int node, int currl, int currr, int
    &targetl, int &targetr, D val) {
    if (currl >= targetr || currr <= targetl) return;
    push_lazy(node, currr - currl);
    if(currl >= targetl && currr <= targetr) { //complete
      overlap
      lazy_comb(node, val); //we apply the lazy change
        to this node, then update this node.
    } else { //partial overlap, should never be a leaf,
      otherwise it'd fall under previous categories
      int mid = (currl + currr) / 2;
      _update(node * 2, currl, mid, targetl, targetr,
        val);
      _update(node * 2 + 1, mid, currr, targetl,
        targetr, val);
      push_lazy(node * 2, (currr - currl) / 2);
      push_lazy(node * 2 + 1, (currr - currl) / 2);
      v[node] = query_comb(v[node * 2], v[node * 2 + 1]);
    }
  }
  T query(int l, int r) {
    return _query(1,0,size,l,r);
  }
  T _query(int node, int currl, int currr, int &targetl,
    int &targetr) { //[l,r)
    if(currr <= targetl || currl >= targetr) return qn;
    push_lazy(node, currr-currl); //make pushes necessary
      before getting value, we always check for 2 cases
    if(currl >= targetl && currr <= targetr) { //complete
      overlap
      return v[node];
    } else {
      int mid = (currl + currr) / 2;
      return query_comb(
        _query(node * 2, currl, mid, targetl, targetr),
        _query(node * 2 + 1, mid, currr, targetl,
          targetr)
      );
    }
  }
};
```

3 Graphs

3.1 DSU

```
//basic union find implementation with path compression
struct DSU {
  int N;
  vector<int> dsu;

  DSU(int n) {
    this->N = n;
    this->dsu = vector<int>(n, 0);
    for(int i = 0; i < n; i++){ //initialize roots
      dsu[i] = i;
    }
  }

  void dsu_init() {
    for(int i = 0; i < N; i++){
      dsu[i] = i;
    }
  }

  int find(int a) {
    if(dsu[a] == a) {
      return a;
    }
    return dsu[a] = find(dsu[a]);
  }

  //ret true if updated something
  bool unify(int a, int b) {
    int ra = find(a);
    int rb = find(b);
    if(ra == rb) {
      return false;
    }
    dsu[rb] = ra;
    return true;
  }
};
```

3.2 Bridges and Articulation Points

A bridge is an edge who's deletion will disconnect the graph. Articulation points are the same, but for nodes. The endpoints of a bridge are always articulation points (except in the case where an endpoint has degree 1), but it is possible for a non-bridge to have two articulation points as endpoints.

3.2.1 Bridges

If edge (u, v) is a bridge, then the returned set should contain (u, v) and (v, u). As of 2/22/2024, this is untested.

```
set<pair<int, int>> find_bridges(int n, vector<vector<int>>&
    adj) {
  vector<bool> visited;
  vector<int> tin, low;
  int timer;
  timer = 0;
  visited.assign(n, false);
  tin.assign(n, -1);
  low.assign(n, -1);
  set<pair<int, int>> ans;
  function<void(int, int)> dfs = [&visited, &tin, &low,
      &timer, &dfs, &adj, &ans](int v, int p) -> void {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    for (int to : adj[v]) {
      if (to == p) continue;
      if (visited[to]) {
        low[v] = min(low[v], tin[to]);
      } else {
        dfs(to, v);
        low[v] = min(low[v], low[to]);
        if (low[to] > tin[v]) {
          ans.insert({v, to});
          ans.insert({to, v});
        }
      }
    }
  };
  for (int i = 0; i < n; ++i) {
    if (!visited[i])
      dfs(i, -1);
  }
  return ans;
}
```

3.2.2 Articulation Points

Returns a boolean array, `a[i]` is `true` if node `i` is an articulation point.

```
vector<bool> find_articulation_points(int n,
    vector<vector<int>>& adj) {
  vector<bool> visited(n, false);
  vector<int> tin(n, -1);
  vector<int> low(n, -1);
  vector<bool> is_articulation_point(n, false);
  int timer = 0;
  function<void(int, int)> dfs = [&visited, &tin, &low,
      &is_articulation_point, &timer, &adj, &dfs](int v,
      int p) -> void {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
      if (to == p) continue;
      if (visited[to]) {
        low[v] = min(low[v], tin[to]);
      } else {
        dfs(to, v);
        low[v] = min(low[v], low[to]);
```

```
                if (low[to] >= tin[v] && p!=-1)
                    is_articulation_point[v] = true;
                ++children;
            }
        }
        if(p == -1 && children > 1)
            is_articulation_point[v] = true;
    };
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs (i, -1);
    }
    return is_articulation_point;
}
```

3.3 Tarjan SCC

Returns multiple lists of node ids, each list being a SCC.

```
vector<vector<int>> find_scc(int n, vector<vector<int>>& adj)
     {
    vector<vector<int>> adj_rev(n, vector<int>(0));
    vector<bool> used(n, false);
    vector<int> order(0);
    for(int i = 0; i < n; i++){
        for(int j = 0; j < adj[i].size(); j++){
            adj_rev[adj[i][j]].push_back(i);
        }
    }
    function<void(int)> dfs1 = [&used, &adj, &order,
         &dfs1](int v) -> void {
        used[v] = true;
        for (auto u : adj[v]) {
            if (!used[u]) {
                dfs1(u);
            }
        }
        order.push_back(v);
    };
    for(int i = 0; i < n; i++){
        if(used[i]) {
            continue;
        }
        dfs1(i);
    }
    fill(used.begin(), used.end(), false);
    reverse(order.begin(), order.end());
    function<void(int, vector<int>&)> dfs2 = [&used,
         &adj_rev, &dfs2](int v, vector<int>& component) ->
         void {
        used[v] = true;
        component.push_back(v);
        for (auto u : adj_rev[v]) {
            if (!used[u]) {
                dfs2(u, component);
            }
        }
    };
    vector<vector<int>> ans(0);
    for(int i = 0; i < n; i++){
        if(used[order[i]]){
            continue;
        }
        vector<int> component(0);
        dfs2(order[i], component);
        ans.push_back(component);
    }
    return ans;
}
```

3.4 LCA

Lets you find the Lowest Common Ancestor of two nodes in a rooted tree in $O(log(n))$ time. Note that the segment tree used in this LCA is slightly modified.

```
struct LCA {
    struct Segtree {
        //note that t[0] is not used
        int n;
        int* t;
        int* node_id;
        int uneut, qneut;

        //single element modification function
        function<int(int, int)> fmodify;

        //product of two elements for query and updating tree
        function<int(int, int)> fcombine;

        Segtree() {
            //do nothing
        }

        Segtree(int n, int updateNeutral, int queryNeutral,
             function<int(int, int)> fmodify,
             function<int(int, int)> fcombine) {
            this -> n = n;
            t = new int[2 * n];
            node_id = new int[2 * n];

            this -> fmodify = fmodify;
            this -> fcombine = fcombine;

            uneut = updateNeutral;
            qneut = queryNeutral;

            for(int i = 0; i < 2 * n; i++){
                t[i] = uneut;
                node_id[i] = 0;
            }
        }

        void build() { // build the tree after manually
             assigning the values.
            for (int i = n - 1; i > 0; i--) {
                t[i] = fcombine(t[i * 2], t[i * 2 + 1]);
                node_id[i] = t[i] == t[i * 2]? node_id[i * 2] :
                     node_id[i * 2 + 1];
            }
        }

        int query(int l, int r) { // least deep node on
             interval [l, r)
            int min_depth = 1e9;
            int res = -1;
            for (l += n, r += n; l < r; l /= 2, r /= 2) {
                if (l % 2 == 1) {
                    if(t[l] < min_depth) {
                        res = node_id[l];
                        min_depth = t[l];
                    }
                    l++;
                }
                if (r % 2 == 1) {
                    r--;
                    if(t[r] < min_depth) {
                        res = node_id[r];
                        min_depth = t[r];
                    }
                }
            }
            return res;
        }
    };

    int n;
    int root;
    vector<vector<int>> edges;
    vector<int> depth; //distance of each node from the root

    vector<int> left_occ, right_occ; //leftmost and rightmost
         occurrences for each node in the euler tour

    //single assignment modify, range min query
    //stores the euler tour of the tree to compute lca
    Segtree segt;

    void euler_tour(int cur, int p, vector<int>& ret) {
        left_occ[cur] = ret.size();
        ret.push_back(cur);
        for(int i = 0; i < edges[cur].size(); i++){
            int next = edges[cur][i];
            if(next == p){
                continue;
            }
            euler_tour(next, cur, ret);
            ret.push_back(cur);
        }
        right_occ[cur] = ret.size();
    }

    void find_depth(int cur, int p) {
        for(int i = 0; i < edges[cur].size(); i++){
            int next = edges[cur][i];
            if(next == p){
                continue;
            }
            depth[next] = depth[cur] + 1;
            find_depth(next, cur);
        }
    }

    void init(int n, int root, vector<vector<int>>& edges) {
        this->n = n;
        this->root = root;
        this->edges = edges;

        this->depth = vector<int>(n, 0);
        find_depth(root, -1);

        vector<int> tour(0);
        this->left_occ = vector<int>(n, -1);
        this->right_occ = vector<int>(n, -1);
        euler_tour(root, -1, tour);

        function<int(int, int)> fmodify = [](const int src,
             const int val) -> int{return val;};
        function<int(int, int)> fcombine = [](const int a,
             const int b) -> int{return min(a, b);};
        this->segt = Segtree(tour.size(), 0, 1e9, fmodify,
             fcombine);
        for(int i = 0; i < tour.size(); i++){
            segt.node_id[i + tour.size()] = tour[i];
            segt.t[i + tour.size()] = depth[tour[i]];
        }
        segt.build();
    }

    //adjacency list constructor
    LCA(int n, int root, vector<vector<int>> edges) {
        init(n, root, edges);
    }

    //parent list constructor
    //if node i is the root, then parents[i] must equal -1
    LCA(int n, vector<int> parents) {
        int root = -1;
        vector<vector<int>> edges(n, vector<int>(0));
        for(int i = 0; i < n; i++){
            if(parents[i] == -1){
                root = i;
            }
            edges[parents[i]].push_back(i);
            edges[i].push_back(parents[i]);
        }
        init(n, root, edges);
    }

    int lca(int a, int b) {
        int l = min(left_occ[a], left_occ[b]);
        int r = max(right_occ[a], right_occ[b]);
        int lc = segt.query(l, r);
```

```
    return lc;
  }

  int dist(int a, int b) {
    int lc = lca(a, b);
    return depth[a] + depth[b] - 2 * depth[lc];
  }
};
```

3.5 Centroid Decomposition

A Centroid of a tree is a node such that when the tree is rooted at it, no other nodes have a subtree of size greater than $\frac{N}{2}$. Restructures the tree such that the maximum distance from root to leaf is $O(log(n))$. Useful for solving problems like answering queries that ask what's the closest white node in a white black tree online, with color updates.

```
struct CentroidDecomp {
  CentroidDecomp() {
    //yay
  }

  vector<bool> vis;
  vector<int> centroid_parent;
  vector<int> size; //size of subtree in original tree
  vector<vector<int>> edges;

  int find_size(int cur, int p = -1) {
    if(vis[cur]) {
      return 0;
    }
    size[cur] = 1;
    for(int i = 0; i < edges[cur].size(); i++){
      int next = edges[cur][i];
      if(next != p){
        size[cur] += find_size(next, cur);
      }
    }
    return size[cur];
  }

  int find_centroid(int cur, int p, int sub_size) {
    for(int i = 0; i < edges[cur].size(); i++){
      int next = edges[cur][i];
      if(next == p){
        continue;
      }
      if(!vis[next] && size[next] > sub_size / 2) {
        return find_centroid(next, cur, sub_size);
      }
    }
    return cur;
  }

  void init_centroid(int cur, int p = -1) {
    find_size(cur);
    int centroid = find_centroid(cur, -1, size[cur]);
    vis[centroid] = true;
    centroid_parent[centroid] = p;
    for(int i = 0; i < edges[centroid].size(); i++){
      int next = edges[centroid][i];
      if(!vis[next]){
        init_centroid(next, centroid);
      }

    }
  }

  //returns an array 'a' where the parent of node i is a[i].
  //if i is the root, then a[i] = -1.
  vector<int> calc_centroid_decomp(int n,
     vector<vector<int>>& adj_list) {
    edges = adj_list;
    vis = vector<bool>(n, false);
    centroid_parent = vector<int>(n, -1);
    size = vector<int>(n, -1);
    init_centroid(0);
    return centroid_parent;
  }
};
```

3.5.1 Find Centroids

Takes in adjacency list, returns a list of the centroids of the tree.

```
vector<int> findCentroid(const vector<vector<int>> &g) {
  int n = g.size();
  vector<int> centroid;
  vector<int> sz(n);
  function<void (int, int)> dfs = [&](int u, int prev) {
      sz[u] = 1;
      bool is_centroid = true;
      for (auto v : g[u]) if (v != prev) {
          dfs(v, u);
          sz[u] += sz[v];
          if (sz[v] > n / 2) is_centroid = false;
      }
      if (n - sz[u] > n / 2) is_centroid = false;
      if (is_centroid) centroid.push_back(u);
  };
  dfs(0, -1);
  return centroid;
}
```

3.6 Heavy-Light Decomposition

$O(log(n)^2)$ modify and query over any path in the tree. LCA, and Lazy Segment Tree not included.

```
template <typename T>
struct HLD {
  LCA lca;
  vector<vector<int>> edges;
  vector<bool> toParentHeavy;
  vector<bool> hasOutHeavy;
  vector<int> parent;
  vector<int> subtreeSize;

  SegtreeLazy<T> segt;
  vector<int> segEndInd; //stores the index at which this
      heavy path ends.
  vector<int> segBeginInd; //stores the index at which this
      heavy path begins.
  vector<int> segParent; //what is the parent node of this
      heavy path?
  vector<int> segPos;  //stores the index of each node
      within the segment tree.

  vector<T> maxSegCache; //stores results to segments which
      take up the entire heavy path

  void calcSubtreeSize(int cur, int p = -1) {
    parent[cur] = p;
    subtreeSize[cur] = 1;
    for(int i = 0; i < edges[cur].size(); i++){
      int next = edges[cur][i];
      if(next == p){
        continue;
      }
      calcSubtreeSize(next, cur);
      subtreeSize[cur] += subtreeSize[next];
    }
  }

  void calcHLD(int cur, int p = -1) {
    for(int i = 0; i < edges[cur].size(); i++){
      int next = edges[cur][i];
      if(next == p){
        continue;
      }
      if(subtreeSize[next] > subtreeSize[cur] / 2) {
        hasOutHeavy[cur] = true;
        toParentHeavy[next] = true;
      }
      calcHLD(next, cur);
    }
  }

  HLD(int n, int root, vector<vector<int>> adjList, T
     updateNeutral, T queryNeutral, T initVal,
     function<T(T, T)> fmodify, function<T(T, T, int)>
     fmodifyk, function<T(T, T)> fcombine) {
    this->lca = LCA(n, root, adjList);

    this->edges = adjList;
    this->parent = vector<int>(n, -1);
    this->subtreeSize = vector<int>(n, 0);
    this->toParentHeavy = vector<bool>(n, false);
    this->hasOutHeavy = vector<bool>(n, false);
    this->calcSubtreeSize(root);
    this->calcHLD(root);

    //create the segment tree needed to do the range
        updates.
    this->segt = SegtreeLazy<T>(n, updateNeutral,
        queryNeutral, initVal, fmodify, fmodifyk,
        fcombine);
    this->segBeginInd = vector<int>(n, -1);
    this->segEndInd = vector<int>(n, -1);
    this->segParent = vector<int>(n, -1);
    this->segPos = vector<int>(n, -1);

    //find the positions of the nodes in the segment tree.
    int posPtr = 0;
    for(int i = 0; i < n; i++){
      if(this->hasOutHeavy[i]) {
        //we want to have each heavy path be contiguous
            in the segment tree, so we want to start
            at the beginning.
        continue;
      }
      int begin = posPtr;
      int cur = i;
      vector<int> heavyPath(0);
      heavyPath.push_back(cur);
      this->segPos[cur] = posPtr ++;
      while(toParentHeavy[cur]) {
        cur = parent[cur];
        heavyPath.push_back(cur);
        this->segPos[cur] = posPtr ++;
      }
      cur = parent[cur];
      for(int j = 0; j < heavyPath.size(); j++){
        this->segBeginInd[heavyPath[j]] = begin;
        this->segEndInd[heavyPath[j]] = posPtr;
        this->segParent[heavyPath[j]] = cur;
      }
    }

    //compute max cache values
    this->maxSegCache = vector<T>(n);
    for(int i = 0; i < n; i++){
      if(this->segPos[i] == this->segBeginInd[i]) {
        int begin = this->segBeginInd[i];
        int end = this->segEndInd[i];
        this->maxSegCache[begin] =
            this->segt.query(begin, end);
      }
    }
  }

  void modify(int a, int b, T val) {
    int _lca = this->lca.lca(a, b);
    _modify(a, _lca, val);
    _modify(b, _lca, val);
```

```
        this->modify(_lca, val);
    }

    void modify(int a, T val) {
        this->segt.modify(this->segPos[a], val);

        //update cache
        int begin = this->segBeginInd[a];
        int end = this->segEndInd[a];
        this->maxSegCache[begin] = this->segt.query(begin,
             end);
    }

    T query(int a, int b) {
        int _lca = this->lca.lca(a, b);
        T ret = this->segt.qneut;
        ret = this->segt.fcombine(ret, _query(a, _lca));
        ret = this->segt.fcombine(ret, _query(b, _lca));
        ret = this->segt.fcombine(ret, this->query(_lca));
        return ret;
    }

    T query(int a) {
        return this->segt.query(this->segPos[a]);
    }

    private:
        void _modify(int a, int _lca, int val) {
            //while a and _lca aren't in the same heavy path
            while(this->segEndInd[a] != this->segEndInd[_lca])
                 {
                //modify until the end of the segment a belongs
                     to.
                this->segt.modify(this->segPos[a],
                     this->segEndInd[a], val);
                a = this->segParent[a];

                //update cache
                int begin = this->segBeginInd[a];
                int end = this->segEndInd[a];
                this->maxSegCache[begin] =
                     this->segt.query(begin, end);
            }
            //a and _lca are in the same heavy path. Now, just
                 modify the segment from a to _lca, not
                 including _lca.
            this->segt.modify(this->segPos[a],
                 this->segPos[_lca], val);
        }

        T _query(int a, int _lca) {
            T ret = this->segt.qneut;
            while(this->segEndInd[a] != this->segEndInd[_lca])
                 {
                //see if we can use the cache
                if(this->segBeginInd[a] == this->segPos[a]) {
                    //use the cache
                    ret = this->segt.fcombine(ret,
                         this->maxSegCache[this->segBeginInd[a]]);
                }
                else {
                    ret = this->segt.fcombine(ret,
                         this->segt.query(this->segPos[a],
                         this->segEndInd[a]));
                }
                a = this->segParent[a];
            }
            ret = this->segt.fcombine(ret,
                 this->segt.query(this->segPos[a],
                 this->segPos[_lca]));
            return ret;
        }
};
```

3.7 Min Cost Flow

Given some amount of flow, what's the minimum cost required to achieve that flow? Can also be used to compute max flow if `max_flow = INF`. Slightly modified MCMF template from KACTL.

```
#include <bits/extc++.h>
#define all(x) begin(x), end(x)
#define sz(x) (int) (x).size()
#define rep(i, a, b) for(int i = a; i < (b); i++)
typedef pair<int, int> pii;
typedef vector<ll> VL;
const ll INF = numeric_limits<ll>::max() / 4;

struct MCMF {
    struct edge {
        int from, to, rev;
        ll cap, cost, flow;
    };
    int N;
    vector<vector<edge>> ed;
    vector<int> seen;
    vector<ll> dist, pi;
    vector<edge*> par;

    MCMF(int N) : N(N), ed(N), seen(N), dist(N), pi(N),
         par(N) {}

    void addEdge(int from, int to, ll cap, ll cost) {
        if (from == to) return;
        ed[from].push_back(edge{ from,to,sz(ed[to]),cap,cost,0
             });
        ed[to].push_back(edge{
             to,from,sz(ed[from])-1,0,-cost,0 });
    }

    void path(int s) {
        fill(all(seen), 0);
        fill(all(dist), INF);
        dist[s] = 0; ll di;

        __gnu_pbds::priority_queue<pair<ll, int>> q;
        vector<decltype(q)::point_iterator> its(N);
        q.push({ 0, s });

        while (!q.empty()) {
            s = q.top().second; q.pop();
            seen[s] = 1; di = dist[s] + pi[s];
            for (edge& e : ed[s]) if (!seen[e.to]) {
                ll val = di - pi[e.to] + e.cost;
                if (e.cap - e.flow > 0 && val < dist[e.to]) {
                    dist[e.to] = val;
                    par[e.to] = &e;
                    if (its[e.to] == q.end())
                        its[e.to] = q.push({ -dist[e.to], e.to
                             });
                    else
                        q.modify(its[e.to], { -dist[e.to], e.to
                             });
                }
            }
        }
        rep(i,0,N) pi[i] = min(pi[i] + dist[i], INF);
    }

    pair<ll, ll> maxflow(int s, int t, ll max_flow = INF) {
        ll totflow = 0, totcost = 0;
        while (path(s), seen[t] && totflow < max_flow) {
            ll fl = max_flow - totflow;
            for (edge* x = par[t]; x; x = par[x->from])
                fl = min(fl, x->cap - x->flow);

            totflow += fl;
            for (edge* x = par[t]; x; x = par[x->from]) {
                x->flow += fl;
                ed[x->to][x->rev].flow -= fl;
            }
        }
        rep(i,0,N) for(edge& e : ed[i]) totcost += e.cost *
             e.flow;
        return {totflow, totcost/2};
    }

    // If some costs can be negative, call this before
         maxflow:
    void setpi(int s) { // (otherwise, leave this out)
        fill(all(pi), INF); pi[s] = 0;
        int it = N, ch = 1; ll v;
        while (ch-- && it--)
            rep(i,0,N) if (pi[i] != INF)
                for (edge& e : ed[i]) if (e.cap)
                    if ((v = pi[i] + e.cost) < pi[e.to])
                        pi[e.to] = v, ch = 1;
        assert(it >= 0); // negative cost cycle
    }
};
```

3.8 2SAT

Given a boolean expression like $(a \vee b) \wedge (\neg b \vee c) \wedge (c \wedge \neg a)$, figures out whether there is a valid assignment to the variables, a, b, c. If a valid assignment exists, can produce an example. Depends on `find_scc` to work.

```
struct TSAT {
    int n;
    vector<vector<int>> node_id; //{false, true}
    vector<vector<int>> c;

    TSAT(int n) {
        this->n = n;

        int ptr = 0;
        this->node_id = vector<vector<int>>(n, {0, 0});
        for(int i = 0; i < n; i++){
            node_id[i][0] = ptr ++;
            node_id[i][1] = ptr ++;
        }

        this->c = vector<vector<int>>(n * 2, vector<int>(0));
    }

    //clears all implications
    void clear() {
        this->c = vector<vector<int>>(n * 2, vector<int>(0));
    }

    //a being a_state implies b being b_state
    void imply(int a, bool a_state, int b, bool b_state) {
        int a_id = node_id[a][a_state];
        int b_id = node_id[b][b_state];
        c[a_id].push_back(b_id);          //positive
        c[b_id ^ 1].push_back(a_id ^ 1); //contrapositive
    }

    //forces a to be state
    void set(int a, int state) {
        imply(a, !state, a, state);      //if a is !state,
             then we have a contradiction, so a must be state.
    }

    //at least one of a has to be a_state or b has to be
         b_state
    void addOR(int a, bool a_state, int b, bool b_state) {
        imply(a, !a_state, b, b_state); //if a is not good,
             then b has to be
    }

    //exactly one of a has to be a_state or b has to be
         b_state
    void addXOR(int a, bool a_state, int b, bool b_state) {
        imply(a, !a_state, b, b_state); //normal or
```

```
    imply(a, a_state, b, !b_state); //if a is good, then
         b can't be
}

//either both a and b are good, or both a and b are not
     good
void addXNOR(int a, bool a_state, int b, bool b_state) {
    imply(a, a_state, b, b_state);  //if a is good, then
         b has to be good
    imply(b, b_state, a, a_state);  //if b is good, then
         a has to be good
}

//a and b both have to be good
//equivalent to set(a, a_state); set(b, b_state);
void addAND(int a, bool a_state, int b, bool b_state) {
    imply(a, a_state, b, b_state);  //if a is good, then
         b has to be good
    imply(b, b_state, a, a_state);  //if b is good, then
         a has to be good
    set(a, a_state);                //make sure that a is
         good
}

//if a solution exists, returns a possible configuration
     of the variables.
//otherwise, returns an empty vector
vector<bool> generateSolution() {
    //first, split into sccs.
    vector<vector<int>> scc = find_scc(n * 2, c);
    //check for contradictions, eg if a and !a are in the
         same scc
    vector<int> node_scc(n * 2);
    for(int i = 0; i < scc.size(); i++){
        for(int j = 0; j < scc[i].size(); j++){
            int id = scc[i][j];
            node_scc[id] = i;
        }
    }
    for(int i = 0; i < n; i++){
        if(node_scc[i * 2 + 0] == node_scc[i * 2 + 1]){
            return {};
        }
    }
    //otherwise, a solution always exists
    vector<bool> v(n, false);
    vector<bool> ans(n, false);
    //toposort scc
    vector<vector<int>> scc_c(scc.size(), vector<int>(0));
    for(int i = 0; i < node_scc.size(); i++){
        int cur = i;
        int cur_scc = node_scc[i];
        for(int j = 0; j < this->c[cur].size(); j++){
            int next = this->c[cur][j];
            int next_scc = node_scc[next];
            if(next_scc != cur_scc) {
                scc_c[cur_scc].push_back(next_scc);
            }
        }
    }
    vector<int> scc_indeg(scc.size(), 0);
    for(int i = 0; i < scc_c.size(); i++){
        for(int j = 0; j < scc_c[i].size(); j++){
            int next_scc = scc_c[i][j];
            scc_indeg[next_scc] ++;
        }
    }
    queue<int> q;
    for(int i = 0; i < scc_indeg.size(); i++){
        if(scc_indeg[i] == 0){
            q.push(i);
        }
    }
    vector<int> toporder;
    while(q.size() != 0){
        int cur = q.front();
        q.pop();
        toporder.push_back(cur);
        for(int i = 0; i < scc_c[cur].size(); i++){
            int next = scc_c[cur][i];
            scc_indeg[next] --;
            if(scc_indeg[next] == 0){
                q.push(next);
            }
        }
    }
    //assign the answers in reverse topological order
    for(int i = toporder.size() - 1; i >= 0; i--){
        int cur_scc = toporder[i];
        for(int j = 0; j < scc[cur_scc].size(); j++){
            int cur = scc[cur_scc][j];
            bool state = cur % 2;
            cur /= 2;
            if(v[cur]) {
                break;
            }
            v[cur] = true;
            ans[cur] = state;
        }
    }
    return ans;
}

bool solutionExists() {
    return generateSolution().size() != 0;
}
};
```

4 Strings

4.1 Suffix Tree

Equivalent to a trie that contains all suffixes of a string S, but only uses $O(log(A)|S|)$ memory, where A is the size of the alphabet. Can find the number of occurrences of T in S, compute the suffix array of S, and compute the Longest Common Prefix between two indices in S.

Make sure to modify the terminator character if '\$' is used as a character in the input. Terminator character is to 'flush' the buffered changes so that all suffixes are in the tree.

```
struct SuffixTree {
    public:
        struct SuffixNode {
            //l, r: left and right boundaries [l, r) of the
                 edge that leads to this node.
            //parent: index of parent node
            //link: index of link node
            int index;
            int l, r;
            SuffixNode* parent;
            SuffixNode* link;
            map<char, SuffixNode*> children;
            SuffixNode(int index, int l = 0, int r = 0,
                 SuffixNode* parent = nullptr) : index{index},
                 l{l}, r{r}, parent{parent}, link{nullptr} {}
            int len() {return r - l;}
            SuffixNode* get_child(char c) {
                if(children.find(c) == children.end()) {
                    return nullptr;
                }
                return children[c];
            }
            void set_child(char c, SuffixNode* ptr) {
                if(children.find(c) == children.end()) {
                    children.insert({c, ptr});
                }
                children[c] = ptr;
            }
        };

        int n;
        vector<char> chars;
        vector<SuffixNode*> nodes;

        SuffixTree(string s) {
            //add terminator char
            s.push_back('$');

            //build tree
            this->n = s.size();
            for(int i = 0; i < s.size(); i++){
                this->add_char(s[i]);
            }

            //calculate useful information
            this->calc_leaf_cnt();
            this->calc_suf_arr();
            this->calc_lcp();
        }

        //runs in O(|s|) time.
        bool contains_string(string s) {
            return count_occurrences(s);
        }

        //each leaf node corresponds to a suffix, so it
             suffices to see how many leaf nodes there
        //are in the subtree corresponding to s.
        int count_occurrences(string s) {
            int i = 0;
            SuffixNode* node = this->nodes[0];
            while(i != s.size()) {
                SuffixNode* child = node -> get_child(s[i]);
                if(child == nullptr) {
                    return 0;
                }
                node = child;
                for(int j = child -> l; j < min(child -> r,
                     (int) this->chars.size()) && i < s.size();
                     j++){
                    if(this->chars[j] != s[i]) {
                        return 0;
                    }
                    i ++;
                }
            }
            return this->leaf_cnt[node -> index];
        }

        vector<int> get_suffix_array() {
            return this->suf_arr;
        }

        int get_lcp(int a, int b) {
            if(a == b){
                return this->n - a - 1;
            }
            int a_ind = this->suf_to_suf_ind[a];
            int b_ind = this->suf_to_suf_ind[b];
            if(a_ind > b_ind) {
                swap(a_ind, b_ind);
            }
            return this->lcp_rmq.query(a_ind, b_ind);
        }

    private:
        Segtree<int> lcp_rmq;

        vector<int> suf_arr;  //suffix array
        vector<int> suf_to_suf_ind; //maps suffix indices to
             their locations in the suffix array.
        vector<int> leaf_cnt; //number of leaves in subtree
        vector<int> lcp;  //longest common prefix between
             adjacent suffixes in suffix array.

        //uses kasai's algorithm to compute lcp array in O(n).
        //lcp stands for longest common prefix.
        void calc_lcp() {
```

```
    int k = 0;
    int n = this->n;
    vector<int> lcp(n,0);
    vector<int> rank(n,0);
    for(int i = 0; i < n; i++) {
        rank[this->suf_arr[i]] = i;
    }
    for(int i = 0; i < n; i++, k? k-- : 0) {
        if(rank[i]==n-1) {
            k = 0;
            continue;
        }
        int j = this->suf_arr[rank[i] + 1];
        while(i + k < n && j + k < n && this->chars[i +
             k] == this->chars[j + k]) {
            k++;
        }
        lcp[rank[i]] = k;
    }
    this->lcp = lcp;
    //create lcp_rmq segtree
    function<int(int, int)> fmodify = [](const int
         src, const int val) -> int{return val;};
    function<int(int, int)> fcombine = [](const int a,
         const int b) -> int{return min(a, b);};
    this->lcp_rmq = Segtree<int>(n, 0, 1e9, fmodify,
         fcombine);
    for(int i = 0; i < n; i++){
        this->lcp_rmq.t[n + i] = lcp[i];
    }
    this->lcp_rmq.build();
}

void calc_leaf_cnt() {
    vector<int> ans(this->nodes.size(), 0);
    function<int(SuffixNode*)> dfs = [&ans,
         &dfs](SuffixNode* cur) -> int {
        int cnt = cur -> children.size() == 0;
        for(auto i = cur -> children.begin(); i != cur
             -> children.end(); i++){
            cnt += dfs(i -> second);
        }
        ans[cur -> index] = cnt;
        return cnt;
    };
    dfs(this -> nodes[0]);
    this->leaf_cnt = ans;
}

//do greedy dfs on the tree.
//Ordering can be changed by switching the comparator
     in the ordered map in the node struct.
void calc_suf_arr() {
    vector<int> ans(this->chars.size(), 0);
    int ind = 0;
    int n = this -> n;
    function<void(SuffixNode*, int)> dfs = [&ans,
         &dfs, &ind, &n](SuffixNode* cur, int dist) ->
         void {
        dist += cur -> len();
        if(cur -> children.size() == 0){
            ans[ind++] = n - dist;
            return;
        }
        for(auto i = cur -> children.begin(); i != cur
             -> children.end(); i++){
            dfs(i -> second, dist);
        }
    };
    dfs(this->nodes[0], 0);
    this->suf_arr = ans;
    //compute mapping from indices to suffix array
    this->suf_to_suf_ind = vector<int>(n, 0);
    for(int i = 0; i < n; i++){
        this->suf_to_suf_ind[this->suf_arr[i]] = i;
    }
}
```

```
struct SuffixState {
    SuffixNode* v;
    int pos;
    SuffixState(SuffixNode* v, int pos) : v{v},
         pos{pos} {}
};

SuffixState ptr = SuffixState(nullptr, 0);

//runs in amortized O(1) time
void add_char(char c) {
    this->chars.push_back(c);
    this->tree_extend((int) this->chars.size() - 1);
}

void tree_extend(int pos) {
    if(pos == 0){
        nodes.push_back(new SuffixNode(0));
        ptr = SuffixState(nodes[0], 0);
    }
    while(true) {
        SuffixState nptr = go(ptr, pos, pos + 1);
        if(nptr.v != nullptr) {
            ptr = nptr;
            return;
        }

        SuffixNode* mid = split(ptr);
        SuffixNode* leaf = new SuffixNode(nodes.size(),
             pos, this -> n, mid);
        nodes.push_back(leaf);
        mid -> set_child(chars[pos], leaf);

        ptr.v = get_link(mid);
        ptr.pos = ptr.v -> len();
        if(mid == nodes[0]) {
            break;
        }
    }
}

SuffixState go(SuffixState st, int l, int r) {
    while(l < r) {
        if(st.pos == st.v -> len()) {
            st = SuffixState(st.v ->
                 get_child(chars[l]), 0);
            if(st.v == nullptr) {
                return st;
            }
        }
        else {
            if(chars[st.v -> l + st.pos] != chars[l]) {
                return SuffixState(nullptr, -1);
            }
            if(r - l < st.v -> len() - st.pos) {
                return SuffixState(st.v, st.pos + r -
                     l);
            }
            l += st.v -> len() - st.pos;
            st.pos = st.v -> len();
        }
    }
    return st;
}

SuffixNode* split(SuffixState st) {
    if(st.pos == st.v -> len()) {
        return st.v;
    }
    if(st.pos == 0){
        return st.v -> parent;
    }
    SuffixNode* par = st.v -> parent;
    SuffixNode* new_node = new
         SuffixNode(nodes.size(), st.v -> l, st.v -> l
         + st.pos, par);
    nodes.push_back(new_node);
    par -> set_child(chars[st.v -> l], new_node);
    new_node -> set_child(chars[st.v -> l + st.pos],
             st.v);
        st.v -> parent = new_node;
        st.v -> l += st.pos;
        return new_node;
    }

    SuffixNode* get_link(SuffixNode* v) {
        if(v -> link != nullptr) {
            return v -> link;
        }
        if(v -> parent == nullptr) {
            return nodes[0];
        }
        SuffixNode* to = get_link(v -> parent);
        v -> link = split(go(SuffixState(to, to -> len()),
             v -> l + (v -> parent == nodes[0]), v -> r));
        return v -> link;
    }
};
```

5 Maths

5.1 Modular Arithmetic

REMEMBER: power function is `power(ll, ll)`, not `pow(ll, ll)`.

```
ll mod = 1e9 + 7;
vector<ll> fac;
map<pair<ll, ll>, ll> nckdp;

ll add(ll a, ll b) {
    ll ret = a + b;
    while(ret >= mod) {
        ret -= mod;
    }
    return ret;
}

ll sub(ll a, ll b) {
    ll ans = a - b;
    while(ans < 0){
        ans += mod;
    }
    return ans;
}

ll mul(ll a, ll b) {
    return (a * b) % mod;
}

ll power(ll a, ll b) {
    ll ans = 1;
    ll p = a;
    while(b != 0){
        if(b % 2 == 1){
            ans = mul(ans, p);
        }
        p = mul(p, p);
        b /= 2;
    }
    return ans;
}

ll divide(ll a, ll b){
    return mul(a, power(b, mod - 2));
}

ll gcd(ll a, ll b){
    if(b == 0){
        return a;
    }
    return gcd(b, a % b);
}

void fac_init() {
    fac = vector<ll>(1e6, 1);
```

```
    for(int i = 2; i < fac.size(); i++){
        fac[i] = mul(fac[i - 1], i);
    }
}

ll nck(ll n, ll k) {
    if(nckdp.find({n, k}) != nckdp.end()) {
        return nckdp.find({n, k}) -> second;
    }
    ll ans = divide(fac[n], mul(fac[k], fac[sub(n, k)]));
    nckdp.insert({{n, k}, ans});
    return ans;
}

//true if odd, false if even.
bool nck_parity(ll n, ll k) {
    return (n & (n - k)) == 0;
}

ll catalan(ll n){
    return sub(nck(2 * n, n), nck(2 * n, n + 1));
}

//cantor pairing function, uniquely maps a pair of integers
     back to the set of integers.
ll cantor(ll a, ll b, ll m) {
    return ((a + b) * (a + b + 1) / 2 + b) % m;
}

ll extended_euclidean(ll a, ll b, ll& x, ll& y) {
    x = 1, y = 0;
    ll x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1) {
        ll q = a1 / b1;
        tie(x, x1) = make_tuple(x1, x - q * x1);
        tie(y, y1) = make_tuple(y1, y - q * y1);
        tie(a1, b1) = make_tuple(b1, a1 - q * b1);
    }
    return a1;
}

//modular inverse of a for any mod m.
//if -1 is returned, then there is no solution.
ll mod_inv(ll a, ll m) {
    ll x, y;
    ll g = extended_euclidean(a, m, x, y);
    if (g != 1) {
        return -1;
    }
    else {
        x = (x % m + m) % m;
        return x;
    }
}

//only works when all modulo is coprime.
//if you want to do this with non-coprime modulos, then you
     need to factor all of the modulos,
//and resolve the factors independently; converting them back
     to coprime.
//it is not guaranteed that there is a solution if the
     modulos are not coprime.
ll chinese_remainder_theorem(vector<ll>& modulo, vector<ll>&
     remainder) {
    if(modulo.size() != remainder.size()) {
        return -1;
    }
    ll M = 1;
    for(int i = 0; i < modulo.size(); i++){
        M *= modulo[i];
    }
    ll solution = 0;
    for(int i = 0; i < modulo.size(); i++){
        ll a_i = remainder[i];
        ll M_i = M / modulo[i];
        ll N_i = mod_inv(M_i, modulo[i]);
        solution = (solution + a_i * M_i % M * N_i) % M;
    }
    return solution;
}

//sum of elements in arithmetic sequence from start to start
     + (nr_elem - 1) * inc
ll arith_sum(ll start, ll nr_elem, ll inc) {
    ll ans = start * nr_elem;
    ans += inc * nr_elem * (nr_elem - 1) / 2;
    return ans;
}
```

5.2 Primes, Factors, Divisors

```
vector<int> lp; //lowest prime factor
vector<int> pr; //prime list

void prime_sieve(int n) {
    lp = vector<int>(n + 1);
    pr = vector<int>(0);
    for(int i = 2; i <= n; i++) {
        if(lp[i] == 0) {
            lp[i] = i;
            pr.push_back(i);
        }
        for (int j = 0; i * pr[j] <= n; j++) {
            lp[i * pr[j]] = pr[j];
            if (pr[j] == lp[i]) {
                break;
            }
        }
    }
}

vector<int> find_prime_factors(int val) {
    vector<int> factors(0);
    while(val != 1) {
        factors.push_back(lp[val]);
        val /= lp[val];
    }
    return factors;
}

void find_divisors_helper(vector<int>& p, vector<int>& c, int
     ind, int val, vector<int>& ans) {
    if(ind == p.size()) {
        ans.push_back(val);
        return;
    }
    for(int i = 0; i <= c[ind]; i++){
        find_divisors_helper(p, c, ind + 1, val, ans);
        val *= p[ind];
    }
}

vector<int> find_divisors(int val) {
    vector<int> factors = find_prime_factors(val);
    map<int, int> m;
    vector<int> p(0);
    vector<int> c(0);
    for(int i = 0; i < factors.size(); i++){
        int next = factors[i];
        if(m.find(next) == m.end()) {
            p.push_back(next);
            c.push_back(0);
            m.insert({next, m.size()});
        }
        int ind = m[next];
        c[ind] ++;
    }
    vector<int> div(0);
    find_divisors_helper(p, c, 0, 1, div);
    return div;
}
```

5.3 FFT

For our purposes, allows us to multiply two polynomials of length N in $O(Nlog(N))$ time. Useful when we can convert a problem into polynomial multiplication.

```
const double PI = acos(-1);

void fft(vector<complex<ld>> & a, bool invert) {
    int n = a.size();
    if (n == 1) {
        return;
    }

    vector<complex<ld>> a0(n / 2), a1(n / 2);
    for (int i = 0; 2 * i < n; i++) {
        a0[i] = a[2*i];
        a1[i] = a[2*i+1];
    }
    fft(a0, invert);
    fft(a1, invert);

    double ang = 2 * PI / n * (invert ? -1 : 1);
    complex<ld> w(1), wn(cos(ang), sin(ang));
    for (int i = 0; 2 * i < n; i++) {
        a[i] = a0[i] + w * a1[i];
        a[i + n/2] = a0[i] - w * a1[i];
        if (invert) {
            a[i] /= 2;
            a[i + n/2] /= 2;
        }
        w *= wn;
    }
}

//given two polynomials, returns the product.
//if the two polynomials sizes arent same or powers of 2,
     this handles that.
vector<int> fft_multiply(vector<int> const& a, vector<int>
     const& b) {
    vector<complex<ld>> fa(a.begin(), a.end()), fb(b.begin(),
         b.end());
    int n = 1;
    while (n < a.size() + b.size()) {
        n <<= 1;
    }
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++) {
        fa[i] *= fb[i];
    }
    fft(fa, true);

    vector<int> result(n);
    for (int i = 0; i < n; i++) {
        result[i] = round(fa[i].real());
    }
    return result;
}
```

5.4 Geometry

```
ld pi = acos(-1);
ld epsilon = 1e-9;

struct vec2 {
    ld x, y;
    vec2() {this->x = 0; this->y = 0;}
    vec2(ld x, ld y) {this->x = x; this->y = y;}
};

vec2 add(vec2 a, vec2 b){
```

```
    vec2 ret;
    ret.x = a.x + b.x;
    ret.y = a.y + b.y;
    return ret;
}

vec2 sub(vec2 a, vec2 b) {
    vec2 ret;
    ret.x = a.x - b.x;
    ret.y = a.y - b.y;
    return ret;
}

ld cross(vec2 a, vec2 b) {
    return a.x * b.y - a.y * b.x;
}

ld dot(vec2 a, vec2 b) {
    return a.x * b.x + a.y * b.y;
}

ld length(vec2 a) {
    return sqrt(a.x * a.x + a.y * a.y);
}

ld distance(vec2 a, vec2 b){
    return length(sub(a, b));
}

ld lerp(ld t0, ld t1, ld x0, ld x1, ld t) {
    ld slope = (x1 - x0) / (t1 - t0);
    return x0 + slope * (t - t0);
}

vec2 mul(vec2 a, ld s) {
    a.x *= s;
    a.y *= s;
    return a;
}

vec2 normalize(vec2 a){
    ld len = length(a);
    vec2 ret;
    ret.x = a.x / len;
    ret.y = a.y / len;
    return ret;
}

//angle from the +x axis in range (-pi, pi)
ld polar_angle(vec2 a) {
    return atan2(a.y, a.x);
}

//project a onto b
vec2 project(vec2 a, vec2 b) {
    b = normalize(b);
    ld proj_mag = dot(a, b);
    return mul(b, proj_mag);
}

vec2 rotateCCW(vec2 a, ld theta) {
    vec2 ret(0, 0);
    ret.x = a.x * cos(theta) - a.y * sin(theta);
    ret.y = a.x * sin(theta) + a.y * cos(theta);
    return ret;
}

//returns the coefficients s and t, where p1 + v1 * s = p2 +
     v2 * t
vector<ld> lineLineIntersect(vec2 p1, vec2 v1, vec2 p2, vec2
     v2) {
    if(cross(v1, v2) == 0){
        return {};
    }
    ld s = cross(sub(p2, p1), v2) / cross(v1, v2);
    ld t = cross(sub(p1, p2), v1) / cross(v2, v1);
    return {s, t};
}

ld tri_area(vec2 t1, vec2 t2, vec2 t3) {
    vec2 v1 = sub(t1, t2);
    vec2 v2 = sub(t2, t3);
    return abs(cross(v1, v2) / 2.0);
}

//returns the distance along the ray from ray_a to the
     nearest point on the circle.
ld rayCircleIntersect(vec2 ray_a, vec2 ray_b, vec2 center, ld
     radius) {
    vec2 ray_dir = normalize(sub(ray_b, ray_a));
    vec2 to_center = sub(center, ray_a);
    vec2 center_proj = add(ray_a, mul(ray_dir, dot(ray_dir,
         to_center)));
    ld center_proj_len = length(sub(center, center_proj));
    //radius^2 = center_proj_len^2 + int_depth^2
    //int_depth = sqrt(radius^2 - center_proj_len^2)
    ld int_depth = sqrt(radius * radius - center_proj_len *
         center_proj_len);
    return dot(ray_dir, to_center) - int_depth;
}

//sector area of circle
ld sector_area(ld theta, ld radius) {
    return radius * radius * pi * ((theta) / (2.0 * pi));
}

ld chord_area(ld theta, ld radius) {
    ld sector = sector_area(theta, radius);
    ld tri_area = radius * radius * cos(theta) * sin(theta);
    return sector - tri_area;
}

//dist = distance from center
ld chord_area_dist(ld dist, ld radius) {
    ld theta = acos(dist / radius);
    return chord_area(theta, radius);
}

//length of chord
ld chord_area_length(ld length, ld radius) {
    ld theta = asin((length / 2.0) / radius);
    return chord_area(theta, radius);
}

//given a point inside and outside a circle, find the point
     along the line that intersects the circle.
vec2 find_circle_intersect(vec2 in, vec2 out, vec2 c_center,
     ld c_radius) {
    //just binary search :D
    //i think we can reduce this to some sort of quadratic.
    ld low = 0;
    ld high = 1;
    ld len = length(sub(in, out));
    vec2 norm = normalize(sub(out, in));
    while(abs(high - low) > epsilon) {
        ld mid = (high + low) / 2.0;
        vec2 mid_pt = add(in, mul(norm, len * mid));
        ld mid_dist = length(sub(mid_pt, c_center));
        if(mid_dist < c_radius) {
            low = mid;
        }
        else {
            high = mid;
        }
    }
    return add(in, mul(norm, len * low));
}

//returns the area of the polygon.
//winding direction doesn't matter
//polygon can be self intersecting i think...
ld polygon_area(vector<vec2>& poly) {
    ld area = 0;
    for(int i = 0; i < poly.size(); i++){
        vec2 v0 = poly[i];
        vec2 v1 = poly[(i + 1) % poly.size()];
        area += cross(v0, v1);
    }
    return abs(area / 2.0);
}

//assuming that the density of the polygon is uniform, the
     centroid is the center of mass.
//winding direction matters...
vec2 polygon_centroid(vector<vec2>& poly) {
    vec2 c = vec2();
    for(int i = 0; i < poly.size(); i++){
        vec2 v0 = poly[i];
        vec2 v1 = poly[(i + 1) % poly.size()];
        ld p = cross(v0, v1);
        c.x += (v0.x + v1.x) * p;
        c.y += (v0.y + v1.y) * p;
    }
    ld area = polygon_area(poly);
    c.x /= (6.0 * area);
    c.y /= (6.0 * area);
    return c;
}

//i believe this gives in CCW order, have to verify though.
vector<vec2> convex_hull(vector<vec2> a, bool
     include_collinear = false) {
    function<int(vec2, vec2, vec2)> orientation = [](vec2 a,
         vec2 b, vec2 c) -> int {
        ld v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
        if (v < 0) return -1; // clockwise
        if (v > 0) return +1; // counter-clockwise
        return 0;
    };

    function<bool(vec2, vec2, vec2)> collinear =
         [&orientation](vec2 a, vec2 b, vec2 c) -> bool {
        return orientation(a, b, c) == 0;
    };

    function<bool(vec2, vec2, vec2, bool)> cw =
         [&orientation](vec2 a, vec2 b, vec2 c, bool
         include_collinear) -> bool {
        int o = orientation(a, b, c);
        return o < 0 || (include_collinear && o == 0);
    };

    vec2 p0 = *min_element(a.begin(), a.end(), [](vec2 a,
         vec2 b) {
        return make_pair(a.y, a.x) < make_pair(b.y, b.x);
    });
    sort(a.begin(), a.end(), [&p0, &orientation](const vec2&
         a, const vec2& b) {
        int o = orientation(p0, a, b);
        if (o == 0)
            return (p0.x-a.x)*(p0.x-a.x) +
                   (p0.y-a.y)*(p0.y-a.y)
                 < (p0.x-b.x)*(p0.x-b.x) + (p0.y-b.y)*(p0.y-b.y);
        return o < 0;
    });
    if (include_collinear) {
        int i = (int)a.size()-1;
        while (i >= 0 && collinear(p0, a[i], a.back())) i--;
        reverse(a.begin()+i+1, a.end());
    }

    vector<vec2> st;
    for (int i = 0; i < (int)a.size(); i++) {
        while (st.size() > 1 && !cw(st[st.size()-2],
             st.back(), a[i], include_collinear))
            st.pop_back();
        st.push_back(a[i]);
    }

    //make sure there are no duplicate vertices
    vector<vec2> ans(0);
    for(int i = 0; i < st.size(); i++){
        vec2 v0 = st[i];
        vec2 v1 = st[(i + 1) % st.size()];
        if(v0.x == v1.x && v0.y == v1.y) {
            continue;
        }
```

```
        ans.push_back(st[i]);
    }
    return ans;
}

//checks if the area of the triangle is the same as the three triangle areas formed by drawing lines from pt to the vertices.
//i don't think triangle winding order matters
bool point_inside_triangle(vec2 pt, vec2 t0, vec2 t1, vec2 t2) {
    ld a1 = abs(cross(sub(t1, t0), sub(t2, t0)));
    ld a2 = abs(cross(sub(t0, pt), sub(t1, pt))) +
        abs(cross(sub(t1, pt), sub(t2, pt))) +
        abs(cross(sub(t2, pt), sub(t0, pt)));
    return abs(a1 - a2) < epsilon;
}

//runs in O(n * log(n)) time.
//has to do O(n * log(n)) preprocessing, but after preprocessing can answer queries online in O(log(n))
vector<bool> points_inside_convex_hull(vector<vec2>& pts, vector<vec2>& hull) {
    vector<bool> ans(pts.size(), false);
    //edge case
    if(hull.size() <= 2){
        return ans;
    }
    //find point of hull that has minimum x coordinate
    //if multiple elements have same x, then minimum y.
    int pivot_ind = 0;
    for(int i = 1; i < hull.size(); i++){
        if(hull[i].x < hull[pivot_ind].x || (hull[i].x == hull[pivot_ind].x && hull[i].y < hull[pivot_ind].y)) {
            pivot_ind = i;
        }
    }
    //sort all the remaining elements according to polar angle to the pivot
    vector<vec2> h_pts(0);
    vec2 pivot = hull[pivot_ind];
    for(int i = 0; i < hull.size(); i++){
        if(i != pivot_ind) {
            h_pts.push_back(hull[i]);
        }
    }
    sort(h_pts.begin(), h_pts.end(), [&pivot](vec2& a, vec2& b) -> bool {
        return polar_angle(sub(a, pivot)) < polar_angle(sub(b, pivot));
    });
    //for each point we want to check, compute it's polar angle, then binary search for the sector that should contain it
    for(int i = 0; i < pts.size(); i++){
        vec2 pt = pts[i];
        ld pt_ang = polar_angle(sub(pt, pivot));
        int low = 0;
        int high = h_pts.size() - 2;
        int tri_ind = low;
        while(low <= high) {
            int mid = low + (high - low) / 2;
            if(polar_angle(sub(h_pts[mid], pivot)) <= pt_ang) {
                tri_ind = max(tri_ind, mid);
                low = mid + 1;
            }
            else {
                high = mid - 1;
            }
        }
        ans[i] = point_inside_triangle(pt, pivot, h_pts[tri_ind], h_pts[tri_ind + 1]);
    }
    return ans;
}
```

6 Misc

6.1 Hungarian Algorithm

Given J jobs and W workers ($J <= W$), computes the minimum cost to assign each prefix of jobs to distinct workers. Input: a matrix of dimensions JxW such that `C[j][w]` is the cost to assign J-th job to W-th worker (possibly negative). Returns a vector of length J, with the J-th entry equaling the minimum cost to assign the first $J+1$ jobs to distinct workers

```
template <class T>
bool ckmin(T &a, const T &b) { return b < a ? a = b, 1 : 0; }

template <class T>
vector<T> hungarian(const vector<vector<T>> &C) {
    const int J = (int)size(C), W = (int)size(C[0]);
    assert(J <= W);
    // job[w] = job assigned to w-th worker, or -1 if no job assigned
    // note: a W-th worker was added for convenience
    vector<int> job(W + 1, -1);
    vector<T> ys(J), yt(W + 1); // potentials
    // -yt[W] will equal the sum of all deltas
    vector<T> answers;
    const T inf = numeric_limits<T>::max();
    for (int j_cur = 0; j_cur < J; ++j_cur) { // assign j_cur-th job
        int w_cur = W;
        job[w_cur] = j_cur;
        // min reduced cost over edges from Z to worker w
        vector<T> min_to(W + 1, inf);
        vector<int> prv(W + 1, -1); // previous worker on alternating path
        vector<bool> in_Z(W + 1); // whether worker is in Z
        while (job[w_cur] != -1) { // runs at most j_cur + 1 times
            in_Z[w_cur] = true;
            const int j = job[w_cur];
            T delta = inf;
            int w_next;
            for (int w = 0; w < W; ++w) {
                if (!in_Z[w]) {
                    if (ckmin(min_to[w], C[j][w] - ys[j] - yt[w]))
                        prv[w] = w_cur;
                    if (ckmin(delta, min_to[w])) w_next = w;
                }
            }
            // delta will always be non-negative,
            // except possibly during the first time this loop runs
            // if any entries of C[j_cur] are negative
            for (int w = 0; w <= W; ++w) {
                if (in_Z[w]) ys[job[w]] += delta, yt[w] -= delta;
                else min_to[w] -= delta;
            }
            w_cur = w_next;
        }
        // update assignments along alternating path
        for (int w; w_cur != -1; w_cur = w) job[w_cur] = job[w = prv[w_cur]];
        answers.push_back(-yt[W]);
    }
    return answers;
}
```

6.2 Fast Unordered Set / Map

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
struct chash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
            chrono::steady_clock::now().time_since_epoch().count()
        return splitmix64(x + FIXED_RANDOM);
    }
};
template<typename T> using pb_set = gp_hash_table<T, null_type, chash>; // unordered_set but faster
template<typename T, typename U> using pb_map = gp_hash_table<T, U, chash>; // unordered_map but faster
```

6.3 Ordered Set / Multiset

```
#include <ext/pb_ds/assoc_container.hpp> //0-indexed
#include <ext/pb_ds/tree_policy.hpp>
#include <functional>
using namespace __gnu_pbds;

typedef tree<pair<int, int>, null_type, less<pair<int, int>>, rb_tree_tag, tree_order_statistics_node_update> ordered_multiset;

template <class T>
typedef tree<T, null_type, less<T>, rb_tree_tag, tree_order_statistics_node_update> ordered_set;
```

6.4 Vector Print

```
template<typename T>
std::ostream& operator<<(std::ostream& os, const vector<T> v) {
    for(auto &x : v) os << x << " ";
    return os;
}
```