1. Template

```cpp
#include <bits/stdc++.h>

#define endl '\n'
#define ll long long int
#define dl double
#define ld long double
#define ff __float128
#define fore(i,a,b) for (int i = a; i < b; i++)
#define fi first
#define se second
#define pb push_back
#define all(v) v.begin(), v.end()
#define fast_io ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);

using namespace std;

typedef vector<int> vi;
typedef vector<vi> vvi;
typedef vector<ll> vll;
typedef vector<vll> vvll;
typedef pair<int,int> pii;
typedef pair<ll,ll> pll;
typedef vector<pll> vpll;

const int inf = 1<<30;
const int mod = 1e9+7;

// clear && g++ -std=c++17 -O2 -Wall template.cpp -o template &&
↪  ./template
// ifstream cin("input.txt"); ofstream cout("output.txt");
```

2. Data Structure

2.1. Segment tree with lazy

```cpp
struct Node{
  int l, r;
  ll sum;
  int mark;
  ll lazy;
};//0 +, 1 *, 0 gcd, 1 mcm
const int Neutro = 0;
template<typename TT> struct SegmentTree{
int n, h;
vector<Node> st;

SegmentTree  (int m,  vector<TT> &values) : n(m){
  h = 1 << ((int)( ceil(log2(n)) + 1));
  st.resize( h );
  build(1, 1, n, values);
}
TT merge(TT l, TT r){ return l + r; }//for query
TT getValue(int curr){ return st[curr].sum; }//same^^

int left(int n){ return (n << 1);}
int right(int n){return (n << 1) | 1; }

void initLeaf(int curr, TT value){
  st[curr].mark = 0; st[curr].lazy = Neutro;
  st[curr].sum = value;//
}
void updateFromChildren(int curr){//
  st[curr].sum = st[left(curr)].sum + st[right(curr)].sum;
}
void updateNodeLazy(int curr, TT value){//updates lazy
  int l = st[curr].l, r = st[curr].r;
  st[curr].sum += (r - l + 1) * value;//
  st[curr].mark = 1; st[curr].lazy += value;//
}
void propagateToChildren(int curr){//propagate lazy
  if(st[curr].mark != 0){
    updateNodeLazy(left(curr), st[curr].lazy );
    updateNodeLazy(right(curr), st[curr].lazy);
    st[curr].mark = 0; st[curr].lazy = Neutro;
  }
}
void build(int curr, int l, int r, vector<TT> &values){
  st[curr].l = l; st[curr].r = r;
  if(l == r) {
    initLeaf(curr, values[l]);//
  }else{
    int m = ((r - l) >> 1) + l;
    build(left(curr), l, m, values);
    build(right(curr), m + 1, r, values);
    updateFromChildren(curr);
  }
}
void rangeUpdate(int curr, int l, int r, int ql, int qr, TT value){
  if( r < ql || qr < l ) return;
  else if( ql <= l && r <= qr){
    updateNodeLazy(curr, value);
```

```
    }else{
      propagateToChildren(curr);
      int m = ((r - l) >> 1) +  l;
      rangeUpdate(left(curr), l, m, ql, qr, value);
      rangeUpdate(right(curr), m + 1, r, ql, qr, value);
      updateFromChildren(curr);
    }
  }// not lazy
  // void pointUpdate(int curr, int l, int r, int pos, TT value){
  //   if(l == r){
  //     st[curr].sum += value;
  //   }else{
  //     int m = ((r - l) >> 1) +  l;
  //     if(pos <= m) pointUpdate(left(curr), l, m, pos, value);
  //     else pointUpdate(right(curr), m + 1, r, pos, value);
  //     updateFromChildren(curr);
  //   }
  // }
  TT rangeQuery(int curr, int l, int r, int ql, int qr){
    if( r < ql || qr < l ) return Neutro;
    else if( ql <= l && r <= qr){
      return getValue(curr);
    }else{
      propagateToChildren(curr);
      int m = ((r - l) >> 1) + l;
      return merge( rangeQuery(left(curr), l, m, ql, qr),
↪   rangeQuery(right(curr), m+1, r, ql, qr)) ;
    }
  }
  void update( int ql, int qr, int value){
    rangeUpdate(1, 1, n, ql, qr, value);
  }
  TT query(int ql, int qr){
    return rangeQuery(1, 1, n, ql, qr);
  }
  // void printST(){
  //   cout << endl << "st = ";
  //   fore(i,0,h) cout << st[i].sum << ' '; cout << endl;
  // }
};
// vector<ll> nums(n + 1);
// SegmentTree<ll>* st = new SegmentTree<ll>(n, nums);
// st->update(l,r,x); st->query(l, r);
```

2.2. Segment tree Geometric sum

```
/*  If you have an array [0, 0, 0, 0, 0, 0]
    update(l, r) adds x, x^2, x^3, x^4 starting from l
    Example:      x = 2
    update(2, 5) -> [0, 2, 4, 8, 16, 0]
    query(l, r) -> sum(arr[l], arr[l+1], ..., arr[r]) */
struct Node {
    int l = 0, r = 0;
    lli ls = 0, rs = 0;
    bool flagLazy = false;
    lli lazy_ls = 0, lazy_rs = 0;
    Node() {}
    Node(int l, int r) : l(l), r(r) {}
    // Combine 2 nodes
    Node operator+(const Node &b) {
        Node res(l, b.r);
        res.ls = (ls + b.ls) % MOD;
        res.rs = (rs + b.rs) % MOD;
        return res;
    }
    // Update range
    void updateNode(lli sum_l, lli sum_r) {
        ls = (ls + sum_l) % MOD; rs = (rs + sum_r) % MOD;
        lazy_ls = (lazy_ls + sum_l) % MOD;
        lazy_rs = (lazy_rs + sum_r) % MOD;
        flagLazy = true;
    }

    void resetLazy() {
        flagLazy = false;
        lazy_ls = 0; lazy_rs = 0;
    }
};
struct SegmentTree {
    vector<Node> ST;
    int N;
    lli x, x_inv;
    SegmentTree(int n, lli x) : N(n), x(x) {
        x_inv = powerMod(x, MOD - 2);
        ST.resize(4 * N); build(1, 1, N);
    }
    void build(int curr, int l, int r) {
        ST[curr].l = l, ST[curr].r = r;
        if (l == r) return;
        int mid = l + (r - l) / 2;
        build(2 * curr, l, mid);
        build(2 * curr + 1, mid + 1, r);
```

```
    }
    void pushToChildren(int curr) {
        if (ST[curr].flagLazy) {
            int size_child_left = (ST[2 * curr].r - ST[2 * curr].l + 1);
            int size_child_right =
                (ST[2 * curr + 1].r - ST[2 * curr + 1].l + 1);
            lli sum_r_to_left =
                ST[curr].lazy_rs * powerMod(x_inv, size_child_right) %
↪   MOD;
            lli sum_l_to_right =
                ST[curr].lazy_ls * powerMod(x, size_child_left) % MOD;
            ST[2 * curr].updateNode(ST[curr].lazy_ls, sum_r_to_left);
            ST[2 * curr + 1].updateNode(sum_l_to_right,
↪   ST[curr].lazy_rs);
            ST[curr].resetLazy();
        }
    }
    // UPDATE
    void update(int curr, int l, int r, int ql, int qr, int start) {
        if (l > qr || r < ql) return;
        else if (ql <= l && r <= qr) {
            int offset_l = l - start + 1;
            int offset_r = offset_l + (r - l + 1) - 1;
            ST[curr].updateNode(powerMod(x, offset_l),
                                powerMod(x, offset_r + 1));
            return;
        }
        pushToChildren(curr);
        int mid = l + (r - l) / 2;
        update(2 * curr, l, mid, ql, qr, start);
        update(2 * curr + 1, mid + 1, r, ql, qr, start);
        ST[curr] = ST[2 * curr] + ST[2 * curr + 1];
    }
    void update(int ql, int qr, int start) { update(1, 1, N, ql, qr,
↪   start); }
    // QUERY
    Node query(int curr, int l, int r, int ql, int qr) {
        if (l > qr || r < ql) return Node();
        if (ql <= l && r <= qr) return ST[curr];
        else {
            pushToChildren(curr);
            int mid = l + (r - l) / 2;
            return query(2 * curr, l, mid, ql, qr) +
                   query(2 * curr + 1, mid + 1, r, ql, qr);
        }
    }
    lli query(int ql, int qr) {
        auto ans = query(1, 1, N, ql, qr);
        return (ans.ls - ans.rs + MOD) * powerMod(1 - x + MOD, MOD - 2) %
↪   MOD;
    }
};
```

2.3. Treap

```
/*-----Treap-----*/
struct Node {
    int key, priority, value;
    Node *left, *right;
    Node(int key, int priority)
        : key(key), priority(priority), value(key), left(NULL),
↪   right(NULL) {}
};
// modify this funciton depending on the query that you want
void update(Node *T) {
    if (T) {
        T->value = T->key;
        if (T->left)
            T->value += T->left->value;
        if (T->right)
            T->value += T->right->value;
    }
}
// returns the root of the union of treaps T1 and T2
Node *merge(Node *T1, Node *T2) {
    if (T1 == NULL) return T2;
    if (T2 == NULL) return T1;
    if (T1->priority > T2->priority) {
        T1->right = merge(T1->right, T2);
        update(T1); return T1;
    } else {
        T2->left = merge(T1, T2->left);
        update(T2); return T2;
    }
}

// returns the roots the division of the treap T with parameter x
pair<Node *, Node *> split(Node *T, int x) {
    if (T == NULL) return {NULL, NULL};
    Node *T1, *T2;
    if (T->key < x) {
        tie(T1, T2) = split(T->right, x);
        T->right = T1;
        update(T2); update(T);
        return {T, T2};
```

```cpp
    } else {
        tie(T1, T2) = split(T->left, x);
        T->left = T2;
        update(T1); update(T);
        return {T1, T};
    }
}

// seed to generate random numbers
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
// generates random numbers
uniform_int_distribution<> dis(numeric_limits<int>::min(),
                               numeric_limits<int>::max());

// inserts a new node with key = x in the treap T, returns the new root
Node *insert(Node *T, int x) {
    Node *TN = new Node(x, dis(rng));
    Node *L, *R;
    tie(L, R) = split(T, x);
    TN = merge(L, TN); TN = merge(TN, R);
    return TN;
}
// returns a treap with all the elements in a
Node *create(vi &a) {
    Node *T = NULL;
    forn(i, (int)a.size()) T = insert(T, a[i]);
    return T;
}
// erases the key x from the treap T
Node *erase(Node *T, int x) {
    Node *L, *R, *AUX;
    tie(L, R) = split(T, x);
    tie(AUX, R) = split(R, x + 1);
    return merge(L, R);
}
// returns the sum of all the keys in te range [l,r]
int query(Node **T, int l, int r) {
    Node *L, *X, *R;
    tie(L, X) = split(*T, l);
    tie(X, R) = split(X, r + 1);
    int ans = (X) ? X->value : 0;
    X = merge(L, X); *T = merge(X, R);
    return ans;
}

// returns the k-th node of the sorted nodes in the treap
Node *findKth(Node *T, int k) {
    if (T == NULL) return NULL;
    int aux = T->left ? T->left->value : 0;
    if (aux >= k) return findKth(T->left, k);
    else if (aux + 1 == k) return T;
    else return findKth(T->right, k - aux - 1);
}
```

2.4. Implicit Treap

```cpp
/*-----Implicit Treap  1 indexed-----*/
struct Node {
    int key, priority, size, rev, value;
    Node *left, *right;
    Node(int key, int priority)
        : key(key), priority(priority), size(1), value(key), left(NULL),
          right(NULL) {}
};
// pushes the lazy updates
void push(Node *T) {
    if (T) {
        if (T->rev) {
            swap(T->left, T->right);
            if (T->left) T->left->rev ^= 1;
            if (T->right) T->right->rev ^= 1;
            T->rev = 0;
        }
    }
}
// modify this funciton depending on the query that you want
void update(Node *T) {//updateFromChildren
    //  push(T);
    if (T) {
        T->size = 1; T->value = T->key;
        if (T->left) {
            T->size += T->left->size;
      T->value += T->left->value;
        }
        if (T->right) {
            T->size += T->right->size;
      T->value += T->right->value;
        }
    }
}

// returns the root of the union of treaps T1 and T2
Node *merge(Node *T1, Node *T2) {
    push(T1), push(T2);
    if (T1 == NULL) return T2;
```

```
    if (T2 == NULL) return T1;
    if (T1->priority > T2->priority) {
        T1->right = merge(T1->right, T2);
        update(T1); return T1;
    } else {
        T2->left = merge(T1, T2->left);
        update(T2); return T2;
    }
}
// returns the roots the division of the treap T with parameter x
// T1 contains all nodes from [1,x), T2 contains all nodes from
↪  [x,T->size]
pair<Node *, Node *> split(Node *T, int x) {
    push(T);
    if (T == NULL) return {NULL, NULL};
    int index;
    if (T->left) index = T->left->size + 1;
    else index = 1;
    if (index < x) {
        Node *T1, *T2;
        tie(T1, T2) = split(T->right, x - index);
        T->right = T1;
        update(T); update(T2);
        return {T, T2};
    } else {
        Node *T1, *T2;
        tie(T1, T2) = split(T->left, x);
        T->left = T2;
        update(T1); update(T);
        return {T1, T};
    }
}
// seed to generate random numbers
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
// generates random numbers
uniform_int_distribution<> dis(numeric_limits<int>::min(),
                               numeric_limits<int>::max());
// inserts a new node with key = x in the position ind of the treap T,
↪  returns
// the new root
Node *insert(Node *T, int x, int ind) {
    Node *TN = new Node(x, dis(rng));
    Node *L, *R;
    tie(L, R) = split(T, ind);
    TN = merge(L, TN); TN = merge(TN, R);
    return TN;
}
// returns a treap with all the elements in a
Node *create(vi a) {
    Node *T = NULL;
    forn(i, a.size()) T = insert(T, a[i], i + 1);
    return T;
}
// erases the key x from the treap T
Node *erase(Node *T, int ind) {
    Node *L, *R, *AUX;
    tie(L, R) = split(T, ind + 1);
    tie(L, AUX) = split(L, ind);
    return merge(L, R);
}
//returns the sum of all the keys in te range [l,r]
int query(Node* T, int l, int r){
  Node *L,*X,*R;
  tie(L,X) = split(T,l);
  tie(X,R) = split(X,r-l+2);
  int ans  = (X) ? X->value : 0;
  X = merge(L,X); T = merge(X,R);
  return ans;
}


// returns the k-th node of the treap
Node *findKth(Node *T, int k) {
    if (T == NULL) return NULL;
    int ind = T->left ? T->left->size + 1 : 1;
    if (ind > k) return findKth(T->left, k);
    else if (ind == k) return T;
    else return findKth(T->right, k - ind);
}
// reverser the treap from [l,r]
Node *reverse(Node *T, int l, int r) {
    Node *L, *R, *AUX;
    tie(L, R) = split(T, r + 1);
    tie(L, AUX) = split(L, l);
    if (AUX) AUX->rev ^= 1;
    L = merge(L, AUX); T = merge(L, R);
    return T;
}
// prints the Treap inorder
void inorder(Node *T) {
    if (T) {
        push(T);
        inorder(T->left);
        cout << T->key << " ";
        inorder(T->right);
    }
```

```
}
```

2.5. Policy base

```
// 20
#include <ext/pb_ds/assoc_container.hpp> // Common file
#include <ext/pb_ds/tree_policy.hpp> // Including
↪  tree_order_statistics_node_update
using namespace __gnu_pbds;

template <class T>
using ordered_set =
    tree<T, null_type, less<T>, rb_tree_tag,
↪   tree_order_statistics_node_update>;

ordered_set<int> s;
*s.find_by_order(k); // To get the i-th element, 0-indexed

s.order_of_key(k); // The number of items strictly smaller than n
                   // For multiset, use T = pair
                   // in second parameter a global counter
                   //  s.order_of_key({k, -INF}) will return
                   //  how many elements < k
```

2.6. SQRT and MO's

```
//SQRT decomposition
//if RTE, change limits to min(br, n)
template <typename TT>
struct SQRT{
  int n, s;
  TT neutro = 0;
  vector<TT> A, B;
  vector<TT> lazy, marks;

  SQRT(int m, vector<TT> &arr): n(m){
    s = sqrt(n) + 1;//puede variar
    A.assign(n, neutro);
    B.assign(n / s + 1, neutro);
    lazy.assign(s, neutro); marks.assign(s, neutro);
    fore(i,0,n){ A[i] = arr[i]; B[i/s] += arr[i]; }
  }
  void pushLazy(int block){
    if(marks[block]){
      fore(i,block,(block+1) * s && i < n) A[i] += lazy[block];
      lazy[block] = neutro; marks[block] = 0;
    }
  }
  void rangeUpdate(int l, int r, TT value){
    int bl = l/s, br = r/s;
    if(bl == br){
      pushLazy(bl);
      fore(i,l,r+1) A[i] += value;
      TT res = neutro;
      fore(i, bl*s, (bl+1) * s && i < n) res += A[i];
      B[bl] = res;
    }else{
      pushLazy(bl);pushLazy(br);
      fore(i,l,(bl+1) * s){ A[i] += value; B[bl] += value;}
      fore(i,bl+1, br)    { B[i] += s * value; lazy[i] += value; marks[i]
↪   = 1;}
      fore(i,br * s, r+1) { A[i] += value; B[br] += value;}
    }
  }
  void pointUpdate(int idx, TT value){//not lazy
    int block = idx / s;
    A[idx] = value;
    TT res = neutro;
    fore(i, block * s, (block + 1) * s && i < n) res += A[i];
    B[block] = res;
  }
  TT rangeQuery(int l, int r){
    int bl = l/s, br = r/s;
    TT res = 0;
    if(bl == br){
      pushLazy(bl);
      fore(i,l,r+1) res += value;
    }else{
      pushLazy(bl); pushLazy(br);
      fore(i,l,(bl+1) * s) res += A[i];
      fore(i,bl+1, br)     res += B[i];
      fore(i,br * s, r+1)  res += A[i];
    }
  }
};
//MO's algorithm
//arr 1 or 0 indexed, S = sqrt(n), index = index of the query
ll answer, neutro = 0; vll arr;
struct MOquery{
  int l, r, index, S;
  MOquery(int l, int r, int idx, int S): l(l), r(r), index(idx), S(S){}
  bool operator<(const MOquery & q) const{
    int bl = l / S, bq = q.l / S;
    if(bl == bq && bl & 1) return r > q.r;//a little bit faster
```

```
    if(bl == bq) return r < q.r;
    return bl < bq;
  }
};
void add(int idx){
  answer += arr[idx];
}
void remove(int idx){
  answer -= arr[idx];
}
vector<ll> MO(vector<MOquery> & queries){
  vector<ll> ans(queries.size());
  sort(queries.begin(), queries.end());
  ll current = 0;
  int prevL = 0, prevR = -1;
  int i, j;
  answer = neutro;
  for(const MOquery & q : queries){
    while (prevL > q.l) { prevL--; add(prevL); }
    while (prevR < q.r) { prevR++; add(prevR); }
    while (prevL < q.l) { remove(prevL); prevL++; }
    while (prevR > q.r) { remove(prevR); prevR--;}
    ans[q.index] = answer;
  }
  return ans;
}
```

2.7. Convex Hull Trick

```
const int MX = 200005;
const ll inf = 1e18;

bool Q = 0;
struct Line {
  mutable ll m, b, x;
  // Maximo: m < ot.m
  // Minimo: m > ot.m
  bool operator < (const Line ot) const {
    return Q ? x < ot.x : m < ot.m;
  }
};

ll ceil (ll a, ll b) {
  if (a < 0 != b < 0) return a / b;
  return (abs(a) + abs(b) - 1) / abs(b);
}

ll intersection (const Line &p, const Line &q) {
  return ceil(q.b - p.b, p.m - q.m);
}

struct Hull : multiset<Line> {
  bool valid (auto it) {
    if (it == begin()) {
      auto sig = it; sig++;
      if (sig != end()) sig->x = intersection(*it, *sig);
      return it->x = -inf;
    }
    auto ant = it, sig = it;
    ant--, sig++;
    if (sig == end()) {
      it->x = intersection(*it, *ant);
      return 1;
    }
    ll x = intersection(*it, *ant);
    ll y = intersection(*it, *sig);
    if (x > y) return 0;
    it->x = x, sig->x = y;
    return 1;
  }
  void add (ll m, ll b) {
    auto it = lower_bound({m, b, -inf});
    if (it != end() && it->m == m) {
      //Maximo: it->b > b
      //Minimo: it->b < b
      if (it->b > b) return;
      it->b = b;
    } else  { it = insert({m, b, -inf}); }

    if (!valid(it)) { erase(it); return; }
    auto ant = it;
    while (ant != begin()) {
      if (valid(--ant)) break;
      erase(ant);
      if (it == begin()) { it->x = -inf; break; }
      ant = it;
    }
    auto sig = it; sig++;
    while (sig != end() && !valid(sig)) erase(sig++);
  }

  ll query (ll x) {
    if (empty()) return 0;
    Q = 1;auto it = upper_bound({0, 0, x});
    it--;
```

```
    Q = 0;return x * it->m + it->b;
  }
};
```

2.8. BIT

```
struct Fenwick {
    int n;
    vector<long long> tree;

    Fenwick(int _n) : n(_n), tree(n + 1, 0) {}

    void update(int idx, long long val) {
        for (; idx <= n; idx += idx & -idx) {
            tree[idx] += val;
        }
    }

    long long query(int idx) {
        long long ret = 0;
        for (; idx > 0; idx -= idx & -idx) {
            ret += tree[idx];
        }
        return ret;
    }

    long long query(int x, int y) { return query(y) - query(x - 1); }
};
```

2.9. merge sort tree

```
vi tree[400000];
vi vv;
void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        tree[v] = vi(1, a[tl]);
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        merge(tree[v*2].begin(), tree[v*2].end(), tree[v*2+1].begin(),
↪   tree[v*2+1].end(), back_inserter(tree[v]));
    }
}
void query(int v, int tl, int tr, int l, int r, int x){
    if(l > r) return;
    if(tr <= r){
    for(auto i = tree[v].begin(); i < tree[v].end(); i++){
      if(*i <= x) vv.pb(*i);
      else break;
    }
    return;
  }
  if(tl > r) return;
  int tm = (tl + tr) / 2;
  query(v*2, tl, tm, l, r, x),
  query(v*2+1, tm+1, tr, l, r, x);
  return;
}
```

3. Binary Search

```
int lowerBound(vi &nums, int a) {
    int l = 0, r = nums.size() - 1;
    while(l <= r) {
        int m = ((r - l) >> 1) + l;
  //  if(nums[m] == a) return m;//binary
    nums[m] < a ? l = m + 1 : r = m - 1;  //lower & binary
  //  nums[m] <= a ? l = m + 1 : r = m - 1; //upper
    }
    return l;//return -1; //binary
}

for(int j = 0;j < 300; j++){
  ld mid1 = l + (r - l) / 3, mid2 = r - (r - l) / 3;
  ld f1 = f(p0, Friend1[i + 1], p1, Friend2[i + 1], mid1);
  ld f2 = f(p0, Friend1[i + 1], p1, Friend2[i + 1], mid2);
  if(f1 >= f2) l = mid1;
  else r = mid2;
}
```

4. Flujos

4.0.1. Dinic

```
typedef tuple<int, ll, ll> edge;
class max_flow{
  private:
    int V;
    vector<edge> EL;
```

```
  vvi AL;
  vi d, last;
  vpii p;
  bool BFS(int s, int t){
    d.assign(V, -1);d[s] = 0;
    queue<int> q({s});
    p.assign(V, {-1, -1});
    while( !q.empty()){
      int u = q.front(); q.pop();
      if( u== t) break;
      for(auto &idx: AL[u]){
        auto &[v, cap, flow] = EL[idx];
        if( (cap - flow > 0) && d[v] == -1){
          d[v] = d[u] + 1, q.push(v), p[v] = {u, idx};
        }
      }
    }
    return d[t] != -1;
  }
  ll send_one_flow(int s, int t, ll f = inf){
    if  ( s == t) return f;
    auto &[u,idx] = p[t];
    auto &cap = get<1>(EL[idx]), &flow = get<2>(EL[idx]);
    ll pushed = send_one_flow(s, u, min(f, cap-flow));
    flow += pushed;
    return pushed;
  }
  ll DFS(int u, int t, ll f = inf){
    if( (u == t) || (f == 0)) return f;
    for(int &i = last[u]; i < (int) AL[u].size(); ++i){
      auto &[v, cap, flow] = EL[AL[u][i]];
      if(d[v] != d[u] + 1) continue;
      if(ll pushed = DFS(v, t, min(f, cap - flow))){
        flow += pushed;
        auto &rflow = get<2> (EL[AL[u][i] ^ 1]);
        rflow -= pushed;
        return pushed;
      }
    }
    return 0;
  }
public:
  max_flow(int initialV) : V(initialV){
    EL.clear();
    AL.assign(V, vi());
  }
  void add_edge(int u, int v, ll w, bool directed = true){
    if( u == v) return;
    EL.emplace_back(v, w, 0);
    AL[u].pb(EL.size() - 1);
    EL.emplace_back(u, directed ? 0 : w, 0);
    AL[v].pb(EL.size() - 1);
  }
  ll edmonds_karp(int s, int t){
    ll mf = 0;
    while( BFS(s,t)){
      ll f = send_one_flow(s, t);
      if ( f == 0)break;
      mf += f;
    }
    return mf;
  }
  ll dinic(int s, int t ){
    ll mf = 0;
    while( BFS(s,t)){
      last.assign(V, 0);
      while( ll f = DFS(s,t)){
        mf += f;
      }
    }
    return mf;
  }
};
```

4.0.2. Ford-Fulkerson

```
const int sink = 37;
int C[50][50], F[50][50], visited[50];
int sendFlow(int node, int bottleneck){
  if(node == sink){
    return bottleneck;
  }
  visited[node] = true;
  fore(i,0,sink+1){
    int f = C[node][i] - F[node][i];
    if(f>0 && !visited[i]){
      f = sendFlow(i, min(f, bottleneck));
      if(!f) continue;
      F[node][i] += f;
      F[i][node] -= f;
      return f;
    }
  }
  return 0;
}
```

4.0.3. maxflow mincost

```
template <typename T = int> struct Edge {
    int to;
    T flow, capacity, cost;
    int idx;
    bool rev;
    Edge *res;
    Edge(int to, T capacity, T cost, int idx, bool rev = false)
        : to(to), flow(0), capacity(capacity), cost(cost), idx(idx),
↪   rev(rev) {}
    void addFlow(T flow) {
        this->flow += flow;
        this->res->flow -= flow;
    }
};

// MUCHO OJO CON ESTO
const lli INF_FLOW = 1e15, INF_COST = 1e9;

template <typename T = int> struct MinCostFlow {
    vector<vector<Edge<T> *>> adjList;
    int N;

    MinCostFlow(int N) : N(N) { adjList.resize(N); }

    void addEdge(int u, int v, T capacity, T cost, int idx) {
        Edge<T> *uv = new Edge<T>(v, capacity, cost, idx);
        Edge<T> *vu = new Edge<T>(u, 0, -cost, idx, true);
        uv->res = vu;
        vu->res = uv;
        adjList[u].push_back(uv);
        adjList[v].push_back(vu);
    }

    pair<T, T> getMinCostFlow(int s, int t) {
        vector<T> dist(N), cap(N);
        vector<bool> inQueue(N);
        vector<Edge<T> *> parent(N);

        T minCost = 0, maxFlow = 0;

        while (true) {

            fill(all(dist), INF_COST);
            fill(all(cap), 0);
            fill(all(parent), nullptr);

            queue<int> q;
            q.push(s);
            dist[s] = 0;
            cap[s] = INF_FLOW;

            while (!q.empty()) {
                int u = q.front();
                q.pop();
                inQueue[u] = 0;

                for (auto E : adjList[u]) {
                    int v = E->to;
                    if (E->flow < E->capacity && dist[u] + E->cost <
↪   dist[v]) {

                        dist[v] = dist[u] + E->cost;
                        cap[v] = min(cap[u], E->capacity - E->flow);
                        parent[v] = E;

                        if (!inQueue[v]) {
                            inQueue[v] = 1;
                            q.push(v);
                        }
                    }
                }
            }

            if (!parent[t])
                break;
            // if(dist[t] > 0) break;

            maxFlow += cap[t];
            minCost += cap[t] * dist[t];

            for (int u = t; u != s; u = parent[u]->res->to) {
                parent[u]->addFlow(cap[t]);
            }
        }

        return {minCost, maxFlow};
    }
};
```

5. Strings

5.1. aho-corasik

```
//#define feach(f, g) for(auto &f: g)
const int N=1e5+10, MOD=1e9+7, SIG=26;
int id=1, dp[N];
string s;
vector<int> adj[2*N];

struct node{
  int fail,ch[SIG]={};
  vector<int> lens;
}t[2*N];
void insert(string s){
  int u=1;
  for(auto &c: s){
    c-='a';
    if(!t[u].ch[c]) t[u].ch[c]=++id;
    u=t[u].ch[c];
  }
  t[u].lens.pb(s.size());
}
void dfs(int u){
  t[u].lens.insert(t[u].lens.end(), t[t[u].fail].lens.begin(),
↪  t[t[u].fail].lens.end());
  for(auto &v: adj) dfs(v);
}
void build(){
  queue<int> q;
  int u=1;
  t[1].fail=1;
  fore(i,0,SIG) {
    if(t[u].ch[i]) t[t[u].ch[i]].fail=u, q.push(t[u].ch[i]);
    else t[u].ch[i]=1;
  }
  while(!q.empty()){
    u=q.front(); q.pop();
    fore(i,0,SIG){
      if(t[u].ch[i]) t[t[u].ch[i]].fail=t[t[u].fail].ch[i],
↪  q.push(t[u].ch[i]);
      else t[u].ch[i]=t[t[u].fail].ch[i];
    }
  }
  fore(i,2,id+1) adj[t[i].fail].pb(i);
  dfs(1);
}
```

6. Math

6.1. FFT

```
using cd = complex<double>;
const double PI = acos(-1);


void fft(vector<cd> & a, int inv) {
    int n = a.size();
   for(int i = 1, j = 0; i < n - 1; ++i){
    for(int k = n >> 1; (j ^= k) < k; k >>= 1);
    if(i < j) swap(a[i], a[j]);
  }

  vector<cd> w(n >> 1);

    for (int k = 2; k <= n; k <<= 1) {
        //   cd w1 = polar(1.0,  2 * PI / k * inv) ;
        w[0] = 1;
        for (int j = 1; j < k >> 1; j++) // best precision but slower
            w[j] = polar(1.0, 2 * j * PI / k * inv);
          // w[j] = w[j-1]*w1;
        for (int i = 0; i < n; i += k) {
            for (int j = 0; j < k >> 1; j++) {
                cd u = a[i + j], v = a[i + j + (k >> 1)] * w[j];
                a[i + j] = u + v;
                a[i + j + (k >> 1)] = u - v;
            }
        }
    }
    if (inv == -1) for (cd & x : a) x /= n;
}

vector<ll> multiply(vector<ll> const& a, vector<ll> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size() - 1)  n <<= 1;
  fa.resize(n); fb.resize(n);

  fft(fa, 1); fft(fb, 1);
  fore(i,0,n) fa[i] *= fb[i];
    fft(fa, -1);

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

```
}
/* if it's numbers and not polynomials, we have to normalise */
void normalise(vll &ans) {
  int  carry = 0;
  for (ll i = 0; i < ans.size(); ++i) {
    ans[i] += carry;
    carry = ans[i] / 10;
    ans[i] %= 10;
  }
  if(carry > 0) ans.pb(carry);
  int t = ans.size();
  while(t > 0 && ans[t-1] == 0){
    ans.pop_back();
    t--;
  }
  if(ans.size() == 0) ans.push_back(0);
}
```

6.2. NTT

```
const int  mod = 998244353, g = 3;

lli powMod(lli a, lli b, lli mod){
  lli ans = 1;
  b %= mod-1;
  if(b < 0) b+= mod-1;
  while(b){
    if(b&1) ans = ans * a % mod;
    a = a*a %mod;
    b>>=1;
  }
  return ans;
}
lli inverse(lli a, lli mod){return powMod(a,mod-2, mod);}

template<int mod, int g>
void ntt(vector<int> & X, int inv){
  int n = X.size();
  for(int i = 1, j = 0; i < n - 1; ++i){
    for(int k = n >> 1; (j ^= k) < k; k >>= 1);
    if(i < j) swap(X[i], X[j]);
  }
  vector<lli> wp(n>>1, 1);
  for(int k = 1; k < n; k <<= 1){
    lli wk = powMod(g, inv * (mod - 1) / (k<<1), mod);
    for(int j = 1; j < k; ++j)
      wp[j] = wp[j - 1] * wk % mod;
    for(int i = 0; i < n; i += k << 1){
      for(int j = 0; j < k; ++j){
        int u = X[i + j], v = X[i + j + k] * wp[j] % mod;
        X[i + j] = u + v < mod ? u + v : u + v - mod;
        X[i + j + k] = u - v < 0 ? u - v + mod : u - v;
      }
    }
  }
  if(inv == -1){
    lli nrev = inverse(n, mod);
    for(int i = 0; i < n; ++i)
      X[i] = X[i] * nrev % mod;
  }
}

template<int mod, int g>
vector<int> multiply(vector<int> const& a, vector<int> const& b) {
    vector<int> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size() - 1)  n <<= 1;
  fa.resize(n); fb.resize(n);

  ntt<mod, g>(fa, 1); ntt<mod, g>(fb, 1);
  fore(i,0,n) fa[i] = (1LL * fa[i] * fb[i]) % mod;
    ntt<mod, g>(fa, -1);

    return fa;
}
```

6.3. Field extension

```
int sq = 5;
// const lli sqrt5 = 383008016;//mod1e9+9
const ll mod = 1000000007;//is important to be CONST
struct EX {
  ll re, im;
  EX (ll re = 0, ll im = 0) : re(re), im(im){}

  EX& operator = (EX oth) {
    return re = oth.re, im = oth.im, *this;
  }
  int norm () const {
    return trim((1ll * re * re - 1ll * sq * im % mod * im) % mod);
  }
  EX conj () const {
    return {re, trim(-im)};
  }
```

```
  EX operator * (EX ot) const {
    return {
      int((1ll * re * ot.re + 1ll * sq * im % mod * ot.im) % mod),
      int((1ll * re * ot.im + 1ll * im * ot.re) % mod)
    };
  };
  EX& operator *= (const EX& ot) {
    *this = *this * ot; return *this;
  }
  EX operator * (ll k) const {
    k = ((k % mod) + mod ) % mod;
    return { (re * k) % mod, (im * k) % mod };
  };
  EX operator / (ll n) const {
    return { re * inv(n) % mod, im * inv(n) % mod };
  }
  EX operator / (EX ot) const {
    return *this * ot.conj() / ot.norm();
  }
  EX& operator /= (const EX& ot) {
    *this = *this / ot; return *this;
  }
  EX operator + (EX ot) const {
    return { trim(re + ot.re),  trim(im + ot.im) };
  }
  EX& operator += (const EX& ot) {
    *this = *this + ot; return *this;
  }
  EX operator - (EX ot) const {
    return { trim(re - ot.re),  trim(im - ot.im) };
  }
  EX& operator -= (const EX& ot) {
    *this = *this - ot; return *this;
  }
  EX pow (ll p) const {
    EX res(1), b = *this;
    while (p) {
      if (p & 1) res *= b; b *= b; p /= 2;
    }
    return res;
  }

  bool operator == (EX ot) const {
    return re == ot.re && im == ot.im;
  }
  bool operator != (EX ot) const {
    return !(*this == ot);
  }

  static ll trim(ll a) {
    if (a >= mod) a -= mod;
    if (a < 0) a += mod;
    return a;
  }
  static ll inv (ll b) {
    ll res = 1, p = mod - 2;
    while (p) {
      if (p & 1) res = 1ll * res * b % mod;
      b = 1ll * b * b % mod;
      p /= 2;
    }
    return res;
  };
};
```

6.4. nCr

```
vll fact(200007,0), inv(200007,0), invfact(200007,0);
void factorial(){
  fact[0] = 1; inv[0] = inv[1] = 1; invfact[0] = 1;
  fore(i,1,200005) fact[i] = (fact[i-1] * i) % MOD;
  fore(i,2,200005) inv[i] = inv[MOD % i] * (MOD - MOD / i) % MOD;
  fore(i,1,200005) invfact[i] = invfact[i-1] * inv[i] % MOD;
}
ll ncr(ll n, ll r){
  return fact[n] * invfact[n - r] % MOD * invfact[r] % MOD ;
}
```

6.5. Discrete Root

```
int generator(int p) {
    vector<int> fact;
    int phi = p-1, n = phi;
    for (int i = 2; i * i <= n; ++i)
        if (n % i == 0)
            fact.push_back(i); while (n % i == 0) n /= i;
    if (n > 1) fact.push_back(n);
    for (int res = 2; res <= p; ++res) {
        bool ok = true;
        for (int factor : fact)
            if (powmod(res, phi / factor, p) == 1)
                ok = false; break;
        if (ok) return res;
    }
```

```
    return -1;
}
vi rootK() {// finds all numbers x such that x^k = a (mod n)
    int n, k, a;
    scanf("%d %d %d", &n, &k, &a);
    if (a == 0) return vi(1,1);
    int g = generator(n);
    // Baby-step giant-step discrete logarithm algorithm
    int sq = (int) sqrt (n + .0) + 1;
    vector<pair<int, int>> dec(sq);
    for (int i = 1; i <= sq; ++i)
        dec[i-1] = {powmod(g, i * sq * k % (n - 1), n), i};
    sort(dec.begin(), dec.end());
    int any_ans = -1;
    for (int i = 0; i < sq; ++i) {
        int my = powmod(g, i * k % (n - 1), n) * a % n;
        auto it = lower_bound(dec.begin(), dec.end(), make_pair(my, 0));
        if (it != dec.end() && it->first == my) {
            any_ans = it->second * sq - i;break;
        }
    }
    if (any_ans == -1) return vi(1,-1);
    // Print all possible answers
    int delta = (n-1) / gcd(k, n-1);
    vi ans;
    for (int cur = any_ans % delta; cur < n-1; cur += delta)
        ans.push_back(powmod(g, cur, n));
    sort(ans.begin(), ans.end());
    return ans;
}
```

6.6. Miller Rabin

```
typedef __int128 i128;
//powerModCode
bool singleTest( i128 a, i128 n ){//test a ^( (n - 1)/2^k )= 1 (mod n)
  i128 exp = n - 1;
  while (~exp & 1) exp >>= 1;// while exp is even
  if( powerMod(a, exp, n) == 1 ) return true; // a^exp = a^( (n - 1)/2^k
↪  ) = 1 (mod n)
  while ( exp < n - 1) {
    if( powerMod(a, exp, n) == n - 1 ) return true;
    exp <<= 1;
  }
  return false;
}
```

```
bool MillerRabin(lli n){
    if(n < 2) return false;
    if(n <= 3) return true;
    if( ~n & 1) return false;
    for(lli a: {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}){
        if(n == a) return true;
    if(!singleTest(a, n)) return false;
    }
    return true; //Probability = 1 - (1/4)^size_of(vector_a)
}

bool MillerRabinOther(lli n){
    if(n < 2) return false;
    if(n <= 3) return true;
    if( ~n & 1) return false;
    lli d = n-1, s = 0; //n-1 = 2^s*k
    for(;(~d&1); d >>= 1, s++); //d = k
    for(lli a: {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37}){
        if(n == a) return true;
        i128 residuo = powerMod(a, d, n);
        if(residuo == 1 or residuo == n-1) continue;
        lli x = s;
        while(--x){
            residuo = (residuo * residuo) % n;
            if(residuo == n-1) break;
        }
        if(x==0) return false;
    }
    return true; //Probability = 1 - (1/4)^size_of(vector_a)
}
```

7. Graphs

7.1. bfs-dfs

```
void bfs(vector<vector<int>> &graph, int start){
  int n = graph.size();
  queue<int> q; q.push(start);
  vector<int> visited(n,0);
  while(!q.empty()){
    int current = q.front(); q.pop();
    for(auto next: graph[current])
      if(!visited[next])
        q.push(next);
  }
  return;
```

```
}
void dfs(int current, vector<vector<int>> &graph, vector<int> visited){
  visited[current] = 1;
  for(auto next: graph[current])
    if(!visited[next])  dfs(next, graph, visited );
  return;
}
```

7.2. cicles

```
int hasCycleDirected(int current, vector<vector<int>> &graph, vector<int>
↪  &color){
  if(color[current] == 1) return 1;
  if(color[current] == 2) return 0;
  color[current] = 1;
  int ans = 0;
  for(int child: graph[current]){
    ans = ans | hasCycleDirected(child, graph, color);
  }
  color[current] = 2;
  return ans;
}
int hasCycleUndirected(int current, int father, vector<vector<int>>
↪  &graph, vector<int> &color){
  if(color[current] == 1) return 1;
  color[current] = 1;
  int ans = 0;
  for(int child: graph[current]) if(child != father)
    ans = ans | hasCycleUndirected(child, current, graph, color);
  return ans;
}
int isBipartiteDFS(int current, vector<vector<int>> &graph, vector<int>
↪  &color){
  bool is = 1;
  for(int child : graph[current]){
    if(color[child] == 0){//no esta coloreado
      color[child] = color[current] == 1 ? 2 : 1;// coloreo con el color
↪  opuesto a mi nodo current
      is = is & isBipartiteDFS(child, graph, color);
    }
    else if(color[child] == color[current]) return 0;//si son iguales no
↪  es bipartito
  }
  return is;
}
bool isBipartiteBFS(int start, vector<vector<int>> &graph, vector<int>
↪  &color){
  int n = graph.size();
  queue<pair<int, int>> q; //node, color
  q.push({start, 1});
  color[start] = 1;
  while(!q.empty()){
    auto current = q.front(); q.pop();
    for (int child :graph[current.first]){
      if(color[child] == current.second) return false;
      if(color[child] == 0){
        color[child] = current.second == 1 ? 2 : 1;
        q.push({child, color[child]});
      }
    }
  }
  return true;
}
```

7.3. Dijkstra

```
vi dijkstra(int start, int n, vvpii graph){
  priority_queue<pii, vpii, greater<pii>> pq;
  vi pesos(n, INT_MAX);
  pq.push({0, start});
  pesos[start] = 0;
  while(!pq.empty()){
    int v = pq.top().second;pq.pop();
    for (auto next : graph[v]){
      int u = next.first, w = next.second;
      if(pesos[u] > pesos[v] + w){
        pesos[u] = pesos[v] + w;
        pq.push({pesos[u], u });
      }
    }
  }
  return pesos;
}
```

7.4. DSU

```
struct DSU {
  vector<int> parent;
  vector<int> rank;
  DSU(int n) : parent(n), rank(n) {
    fore(i,0,n) parent[i] = i;
  }
  int find(int u) {
```

```
    return parent[u] = (u == parent[u]? u : find(parent[u]));
  }
  void setUnion(int u, int v) {
    int pv = find(v);
    int pu = find(u);
    if(pv != pu) {
      if(rank[pu] < rank[pv]) swap(pu, pv);
      if(rank[pu] == rank[pv]) rank[pu]++;
      parent[pv] = pu;
    }
  }
  bool isSame(int u, int v) {
    return find(u) == find(v);
  }
};
```

7.5. FloydWarshall

```
void floydWarshall(vvi &mtx){
  int n = mtx.size();
  fore(k,0,n) fore(i,0,n) fore(j,0,n)
    mtx[i][j] = min(mtx[i][j], mtx[i][k] + mtx[k][j]);
}
```

7.6. Kruskal

```
vector<int> parent, rank;
void make_set(int v) {
    parent[v] = v; rank[v] = 0;
}
int find_set(int v) {
    if (v == parent[v]) return v;
    return parent[v] = find_set(parent[v]);
}
void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b]) swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b]) rank[a]++;
    }
}
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};
void Kruskal(){
  vector<Edge> edges, result;
  int n, cost = 0;
  parent.resize(n); rank.resize(n);
  for (int i = 0; i < n; i++) make_set(i);
  sort(edges.begin(), edges.end());
  for (Edge e : edges) {
    if (find_set(e.u) != find_set(e.v)) {
      cost += e.weight;
      result.push_back(e);
      union_sets(e.u, e.v);
    }
  }
}
```

7.7. topoSort

```
vi toposortBFS(vvi &graph){
    int n = graph.size();
    vi incoming_edges(n);
  fore(i,0,n) for(auto v: graph[i]) incoming_edges[v]++;
    queue<int> q;
    fore(i,0,n) if(incoming_edges[i] == 0) q.push(i);
    int cnt = 0;
    vi ans;
    while(!q.empty()){
        int u = q.front(); q.pop();
        ans.push_back(u);
        cnt++;
        for(auto v: graph[u]){
            incoming_edges[v]--;
            if(incoming_edges[v] == 0) q.push(v);
        }
    }
    if(cnt != n) return {-1};
    return ans;
}
void dfs(int curr, vector<bool> &visited, vector<vector<int>> &graph,
↪   vector<int> &ans){
    if(visited[curr]) return;
    visited[curr] = true;
    for(auto nextNode: graph[curr])  dfs(nextNode, visited, graph, ans);
    ans.push_back(curr);
    return;
```

```
}
vector<int> toposort_dfs(vector<vector<int>> &graph){
    int n = graph.size();
    vector<bool> visited(n);
    vector<int> ans;
    fore(i,0,n) if(!visited[i]) dfs(i, visited, graph, ans);
    reverse(ans.begin(), ans.end());//dfs => reverse
    return ans;
}
```

8. Tree

```
struct Tree {
  int n, root, idx;
  vvi ancestros;
  vi level, tin, tout, aplanado, sz;

  Tree(int n, int root, vvi& adj) :root(root),n(n), ancestros(21, vi(n+1,
↪  0)),
            level(n+1), tin(n+1), tout(n+1), aplanado(n+1), sz(n+1, 1){
    idx = 1; dfs(root, root, adj);
    fore(k, 1, 21) fore(x, 1, n + 1) ancestros[k][x] = ancestros[k -
↪  1][ancestros[k - 1][x]];
  }
  void dfs(int curr, int father, vvi &adj) {
    tin[curr] = idx;
    aplanado[idx] = curr;
    for (auto& next : adj[curr]) {
      if (next == father) continue;

      idx++; dfs(next, curr, adj);

      ancestros[0][next] = curr;
      level[next] = level[curr] + 1;
      sz[curr] += sz[next];
    }
    tout[curr] = idx;
  }

  int kAncestorOfx(int k, int x) {
    fore(j,0,21) if (k & (1 << j)) x = ancestros[j][x];
    return x;
  }
  int isAncestor(int u, int v) {
    return tin[u] <= tin[v] && tout[v] <= tout[u];
  }
  int LCA(int u, int v) {
    if(level[u] < level[v]) swap(u,v);
    int k = level[u] - level[v];
    u = kAncestorOfx(k, u);
    if(u == v) return u;
    forr(i, 20, -1){
      if(ancestros[i][u] != ancestros[i][v]){
        u = ancestros[i][u];
        v = ancestros[i][v];
      }
    }
    return ancestros[0][u];
  }
  // int LCA(int u, int v) {
  //   if (isAncestor(u, v)) return u;
  //   if (isAncestor(v, u)) return v;
  //   forr(k, 20, -1) if(!isAncestor(ancestros[k][u], v)) u =
↪  ancestros[k][u];
  //   return ancestros[0][u];
  // }
  int centroid(vvi& adj){
    int vf = 1;
    while(vf){
      vf = 0;
      for(auto next: adj[root]){
        if(sz[next] > n/2){
          int aux = sz[root];
          sz[root] -= sz[next];
          sz[next] = aux;
          root = next;
          vf = 1;
        }
      }
    }
    return root;
  }
  void printAncestros(){
    fore(i,0,4) {fore(j,0,n+1) cout << ancestros[i][j] <<' '; cout <<
↪  endl;}
  }
};
```

8.1. HLD

```
struct HeavyLightDecomp {
  int n, root, idx;
  vi level, size, head, pos, newVals;
  vvi ancestros;

  HeavyLightDecomp(int n, int root, vvi & adj, vi& vals):n(n),
  ↪  root(root), ancestros(21, vi(n+1,0)),level(n+1), size(n+1, 1),
  ↪  head(n+1), pos(n+1), newVals(1){
    dfs(root, root, adj);
    idx = 1; hld(root, root, adj, vals);
  }
  void dfs(int curr, int father, vvi & adj) {
    for (auto& next : adj[curr]) {
      if (next == father) continue;
      ancestros[0][next] = curr;
      level[next] = level[curr] + 1;
      dfs(next, curr, adj);
      size[curr] += size[next];
    }
  }
  void hld(int curr, int nodeHead, vvi &adj, vi &vals){
    head[curr] = nodeHead;
    pos[curr] = idx++;
    newVals.pb(vals[curr]);
    int sz_heavy = 0, heavy = -1;
    for(auto next: adj[curr]){
      if(next == ancestros[0][curr])  continue;
      if(size[next] > sz_heavy)
        sz_heavy = size[next], heavy = next;

    }
    if(heavy != -1) hld(heavy, nodeHead, adj, vals);
    for(auto next: adj[curr])
      if(next != heavy && next != ancestros[0][curr])
        hld(next, next, adj, vals);
  }
  template <class BinaryOperation>
  void traversePath(int u, int v, BinaryOperation op){
    for(; head[u] != head[v]; u = ancestros[0][head[u]]){
      if(level[head[u]] < level[head[v]]) swap(u,v);
      op(pos[head[u]], pos[u]);
    }
    if(pos[u] > pos[v]) swap(u,v);
    op(pos[u], pos[v]);
  }
  template <class DSType> ll query(int u, int v, DSType *st){
    int ans = -1;
    traversePath(u, v, [this, &ans, st](int l, int r){ans = max(ans,
  ↪  st->query(l,r));});
    return ans;
  }
  template <class DSType> void update(int u, ll val, DSType *st){
    traversePath(u, u, [this, &val, st](int l, int r){st->update(l,
  ↪  val);});
  }
};
```

9. Memorización

9.1. Knapsack

```
int mochila(int i, int peso, int &N, int &W, vi &v, vi &w, vvi &dp){
  if(peso > W) return -inf;
  if( i == N) return 0;
  int &ans = dp[i][peso];
  if(ans != -inf) return ans;
  //if choose only one:
  //ans = max( mochila(i+1, peso, N, W, v, w, dp), mochila(i+1,
↪  peso+w[i], N, W, v, w, dp) + v[i]);
  ans = max( mochila(i+1, peso, N, W, v, w, dp), mochila(i, peso+w[i], N,
↪  W, v, w, dp) + v[i]);
  return ans;
}
```