Assiut University

# Ala7lam_team

ICPC World Finals 2023

2024-11-03

1 Data structures 1
1.1 Ordered Set 1
1.2 Sparse Table 1
1.3 Fenwick Tree 1
1.4 Segment Tree 2
1.5 Treap 3
1.6 Line Container 5
1.7 MO 5

2 Graph 5
2.1 Graph Data structures 5
2.2 Flow 8
2.2.1 Bipartite Matching 8
2.2.2 Max Flow 8
2.2.3 Min Cost Max Flow 9
2.3 Shortest Path 9
2.4 Tarjan 10

3 Strings 11
3.1 Rolling Hash 11
3.2 String matching 11
3.3 Suffix structures 12
3.4 Manacher Hash 14

4 Math 14
4.1 Prime 14
4.2 ModInverse 15
4.3 Number Theory 15
4.4 Combinatorics 16
4.5 Matrix 16
4.6 FFT 17

5 Geometry 18
5.1 Basic Geometry 18
5.2 Circles 19
5.3 Polygons 20

6 Misc 21

7 Mathematics Notes 22
7.1 Sums 22
7.2 Congruence 22
7.3 $E(x^2)$ 22

8 KACTL Maths Notes 22
8.1 Equations 22
8.2 Recurrences 22
8.3 Trigonometry 22
8.4 Geometry 23
8.4.1 Triangles 23
8.4.2 Spherical coordinates 23
8.5 Derivatives/Integrals 23
8.6 Sums 23
8.7 Series 23
8.8 Probability theory 23
8.8.1 Discrete distributions 23
8.8.2 Continuous distributions 23

# Data structures (1)

## 1.1 Ordered Set

OrderedSet.h
2b4dee, 10 lines

```
#include<ext/pb_ds/assoc_container.hpp> // keep-include
#include<ext/pb_ds/tree_policy.hpp> // keep-include
using namespace __gnu_pbds;
template<typename key>
using ordered_set = tree<key, null_type, less<key>, rb_tree_tag
    , tree_order_statistics_node_update>;
// find_by_order(k) :
// It returns to an iterator to the k-th element (counting
    from zero) in the set in O(logn) time.
// To find the first element k must be zero.
// order_of_key(k) :
// It returns to the number of items that are strictly smaller
     than our item k in O(logn) time.
```

## 1.2 Sparse Table

SparseTable.h
588e6e, 41 lines

```
template<typename T>
struct sparse_table {
  vector<vector<T>> sparseTable;
  using F = function<T(T,T)>;
  F merge;
  static int LOG2(int x) { //floor(log2(x))
    return 31 - __builtin_clz(x);
  }
  sparse_table(vector<T> &v, F merge) :
      merge(merge) {
    int n = v.size();
    int logN = LOG2(n);
    sparseTable = vector < vector < T >> (logN + 1);
    sparseTable[0] = v;
    for (int k = 1, len = 1; k <= logN; k++, len <<= 1) {
      sparseTable[k].resize(n);
      for (int i = 0; i + len < n; i++)
        sparseTable[k][i] = merge(sparseTable[k - 1][i],
            sparseTable[k - 1][i + len]);
    }
  }
  T query(int l, int r) {
    int k = LOG2(r - l + 1); // max k ==> 2^k <= length of
         range
    //check first 2^k from left and last 2^k from right //
         overlap
    return merge(sparseTable[k][l], sparseTable[k][r - (1 << k)
         + 1]);
  }
  T query_shifting(int l, int r) {
    T res;
    bool first = true;
    for (int i = (int) sparseTable.size() - 1; i >= 0; i--)
      if (l + (1 << i) - 1 <= r) {
        if (first)
          res = sparseTable[i][l];
        else
          res = merge(res, sparseTable[i][l]);
        first = false;
        l += (1 << i);
      }
    return res;
  }
};
```

## 1.3 Fenwick Tree

FenwickTree.h
01ae07, 57 lines

```
template<typename T>
struct fenwick_tree {
    int n;
    vector<T> BIT;
    fenwick_tree(int n) : n(n), BIT(n + 1) {}
    T getAccum(int idx) {
        T sum = 0;
        while (idx) {
            sum += BIT[idx];
            idx -= (idx & -idx);
        }
        return sum;
    }
    void add(int idx, T val) {
        assert(idx != 0);
        while (idx <= n) {
            BIT[idx] += val;
            idx += (idx & -idx);
        }
    }
    T getValue(int idx) {
        return getAccum(idx) - getAccum(idx - 1);
    }
    // ordered statistics tree
    // get index that has value >= accum
    // values must be positive
    int getIdx(T accum) {
        int start = 1, end = n, rt = -1;
        while (start <= end) {
            int mid = start + end >> 1;
            T val = getAccum(mid);
            if (val >= accum)
                rt = mid, end = mid - 1;
            else
                start = mid + 1;
        }
        return rt;
    }
    //need review (from topcoder)
    //last index less than or equal accum  O(logn)
    int find(T accum) {
        int i = 1, idx = 0;
        while ((1 << i) <= n)
            i <<= 1;
        for (; i > 0; i >>= 1) {
            int tidx = idx + i;
            if (tidx > n)
                continue;
            if (accum >= BIT[tidx]) {
                idx = tidx;
                accum -= BIT[tidx];
            }
        }
        if (!accum || idx + 1 > n)return -1;
        return idx + 1;
    }
};
```

FenwickTreeUpdateRange.h
9f5704, 47 lines

```
// x[i] = a[i] - a[i-1] //a is original array
// y[i] = x[i]*(i-1)
// sum(1,3) = a[1] + a[2] + a[3]
// = (x[1]) + (x[2] + x[1]) + (x[3] + x[2] + x[1])
// = 3*(x[1] + x[2] + x[3]) - 0*x[1] - 1*x[2] - 2*x[3] -> same
     equation but more complex
// = sumX(1,3) * 3 - sumY(1,3)
// so sum(1,n) = sumX(1,n)*n - sumY(1,n)
// update:
// x[l] += val,x[r+1] -= val
// y[l] += val *(l-1),y[r+1] -= r*val

template<typename T>
class fenwick_tree {
  int n;
  vector<T> x, y;
  T getAccum(vector<T> &BIT, int idx) {
    T sum = 0;
    while (idx) {
      sum += BIT[idx];
      idx -= (idx & -idx);
    }
    return sum;
  }
  void add(vector<T> &BIT, int idx, T val) {
    assert(idx != 0);
    while (idx <= n) {
      BIT[idx] += val;
      idx += (idx & -idx);
    }
  }
  T prefix_sum(int idx) {
    return getAccum(x, idx) * idx - getAccum(y, idx);
  }
public:
  fenwick_tree(int n) :
      n(n), x(n + 1), y(n + 1) {
  }
  void update_range(int l, int r, T val) {
    add(x, l, val);
    add(x, r + 1, -val);
    add(y, l, val * (l - 1));
    add(y, r + 1, -val * r);
  }
  T range_sum(int l, int r) {
    return prefix_sum(r) - prefix_sum(l - 1);
  }
};
```

## 1.4 Segment Tree

SegmentTree.h
6cbbea, 85 lines

```
// for efficient memory (2*n)
// #define LEFT (idx+1)
// #define MID ((start+end)>>1)
// #define RIGHT (idx+((MID-start+1)<<1))
template<typename T>
class segment_tree {//1-based
#define LEFT (idx<<1)
#define RIGHT (idx<<1|1)
#define MID ((start+end)>>1)
  int n;
  vector<T> tree, lazy;
  T merge(const T& left, const T& right) {}
  inline void pushdown(int idx, int start, int end) {
    if (lazy[idx] == 0) return;
    //update tree[idx] with lazy[idx]
    tree[idx] += lazy[idx];
    if (start != end) {
      lazy[LEFT] += lazy[idx];
      lazy[RIGHT] += lazy[idx];
    }
    //clear lazy
    lazy[idx] = 0;
  }
  inline void pushup(int idx) {
    tree[idx] = merge(tree[LEFT], tree[RIGHT]);
  }
  void build(int idx, int start, int end) {
    if (start == end)return;
    build(LEFT, start, MID);
    build(RIGHT, MID + 1, end);
    pushup(idx);
  }
  void build(int idx, int start, int end, const vector<T>& arr)
       {
    if (start == end) {
      tree[idx] = arr[start];
      return;
    }
    build(LEFT, start, MID, arr);
    build(RIGHT, MID + 1, end, arr);
    pushup(idx);
  }
  T query(int idx, int start, int end, int from, int to) {
    pushdown(idx, start, end);
    if (from <= start && end <= to)
      return tree[idx];
    if (to <= MID)
      return query(LEFT, start, MID, from, to);
    if (MID < from)
      return query(RIGHT, MID + 1, end, from, to);
    return merge(query(LEFT, start, MID, from, to),
      query(RIGHT, MID + 1, end, from, to));
  }
  void update(int idx, int start, int end, int lq, int rq,
    const T & val) {
    pushdown(idx, start, end);
    if (rq < start || end < lq)
      return;
    if (lq <= start && end <= rq) {
      lazy[idx] += val;//update lazy
      pushdown(idx, start, end);
      return;
    }
    update(LEFT, start, MID, lq, rq, val);
    update(RIGHT, MID + 1, end, lq, rq, val);
    pushup(idx);
  }
public:
  segment_tree(int n) :n(n), tree(n << 2), lazy(n << 2) {
  }
  segment_tree(const vector<T>& v) {
    n = v.size() - 1;
    tree = vector<T>(n << 2);
    lazy = vector<T>(n << 2);
    build(1, 1, n, v);
  }
  T query(int l, int r) {
    return query(1, 1, n, l, r);
  }
  void update(int l, int r, const T& val) {
    update(1, 1, n, l, r, val);
  }
#undef LEFT
#undef RIGHT
#undef MID
};
```

ExtendedSegmentTree.h
b83b87, 64 lines

```
struct segtree {
  segtree *left = nullptr, *right = nullptr;
  int mx = 0;
  segtree(int val = 0) :
      mx(val) {
  }
  void extend() {
    if (left == nullptr) {
      left = new segtree();
      right = new segtree();
    }
  }
  void pushup() {
    mx = max(left->mx, right->mx);
  }
  ~segtree() {
    if (left == nullptr)return;
    delete left;
    delete right;
  }
};

class extened_segment_tree {
#define MID ((start+end)>>1)
  void update(segtree *root, int start, int end, int pos, int
       val) {
    if (pos < start || end < pos)
      return;
    if (start == end) {
      root->mx = max(root->mx, val);
      return;
    }
    root->extend();
    update(root->left, start, MID, pos, val);
    update(root->right, MID + 1, end, pos, val);
    root->pushup();
  }
  int query(segtree *root, int start, int end, int l, int r) {
    if (root == nullptr || r < start || end < l)
      return 0;
    if (l <= start && end <= r)
      return root->mx;
    return max(query(root->left, start, MID, l, r),
        query(root->right, MID + 1, end, l, r));
  }
public:
  int start, end;
  segtree *root;
  extened_segment_tree() {
  }
  ~extened_segment_tree() {
    delete root;
  }
  extened_segment_tree(int start, int end) :
      start(start), end(end) {
    root = new segtree();
  }
  void update(int pos, int val) {
    update(root, start, end, pos, val);
  }
  int query(int l, int r) {
    return query(root, start, end, l, r);
  }
#undef MID
```

```
};
```

### PersistentSegmentTree.h

f73066, 86 lines

```
struct segtree {
  static segtree *sentinel;
  segtree *left, *right;
  bool dirty = false;
  ll sum = 0;
  ll lazy = 0;
  segtree(ll val = 0) :
      sum(val) {
    left = right = this;
  }
  segtree(segtree *left, segtree *right) :
      left(left), right(right) {
    sum = left->sum + right->sum;
  }
};

segtree *segtree::sentinel = new segtree();

class persistent_segment_tree {
#define MID ((start+end)>>1)
  segtree* apply(segtree *root, int start, int end, ll val) {
    segtree *rt = new segtree(*root);
    rt->dirty = true;
    rt->sum += (end - start + 1) * val;
    rt->lazy += val;
    return rt;
  }
  void pushdown(segtree *root, int start, int end) {
    if (root->dirty == false || start == end)
      return;
    root->left = apply(root->left, start, MID, root->lazy);
    root->right = apply(root->right, MID + 1, end, root->lazy);
    root->lazy = 0;
    root->dirty = 0;
  }
  segtree* build(int start, int end, const vector<int> &v) {
    if (start == end)
      return new segtree(v[start]);
    return new segtree(build(start, MID, v), build(MID + 1, end
        , v));
  }
  segtree* Set(segtree *root, int start, int end, int pos, ll
       new_val) {
    pushdown(root, start, end);
    if (pos < start || end < pos)
      return root;
    if (pos <= start && end <= pos)
      return new segtree(new_val);
    return new segtree(Set(root->left, start, MID, pos, new_val
        ),
        Set(root->right, MID + 1, end, pos, new_val));
  }
  segtree* update(segtree *root, int start, int end, int l, int
       r, ll val) {
    pushdown(root, start, end);
    if (r < start || end < l)
      return root;
    if (l <= start && end <= r)
      return apply(root, start, end, val);
    return new segtree(update(root->left, start, MID, l, r, val
        ),
        update(root->right, MID + 1, end, l, r, val));
  }
  ll query(segtree *root, int start, int end, int l, int r) {
    pushdown(root, start, end);
    if (r < start || end < l)
      return 0;
    if (l <= start && end <= r)
      return root->sum;
    return query(root->left, start, MID, l, r)
        + query(root->right, MID + 1, end, l, r);
  }
public:
  int start, end;
  vector<segtree*> versions;
  persistent_segment_tree(int start, int end) :
      start(start), end(end) {
    versions.push_back(segtree::sentinel);
  }
  persistent_segment_tree(const vector<int> &v) :
      start(0), end(v.size() - 1) {
    versions.push_back(build(start, end, v));
  }
  void update(int l, int r, ll val) {
    versions.push_back(update(versions.back(), start, end, l, r
        , val));
  }
  ll query(int time, int l, int r) {
    return query(versions[time], start, end, l, r);
  }
#undef MID
};
```

### IterativeSegmentTree.h

2bed1e, 67 lines

```
struct node {
  node() { //set Default value

  }
  node(const node &a, const node &b) {

  }
  void apply(int val) {

  }
};

struct segment_tree {
  int n; //0 to n-1
  vector<node> tree;
  segment_tree(int n) {
    resize(n);
    build();
  }
  template<typename T>
  segment_tree(const vector<T> &arr) {
    resize(arr.size());
    for (int i = 0; i < arr.size(); i++)
      tree[n + i] = arr[i];
    build();
  }
  void resize(int n) {
    int p = 1;
    while (p < n)
      p <<= 1;
    this->n = p;
    tree = vector < node > (p << 1);
  }
  void build() {
    for (int i = n - 1; i > 0; i--)
      tree[i] = node(tree[i << 1], tree[i << 1 | 1]);
  }
  template<typename T>
  void update(int p, const T &value) {
    tree[p += n].apply(value);
    for (int i = p / 2; i > 0; i >>= 1)
      tree[i] = node(tree[i << 1], tree[i << 1 | 1]);
  }
  node query(int l, int r) {  //[l, r]
    node resl, resr;  //set default value in node
    for (l += n, r += n + 1; l < r; l >>= 1, r >>= 1) {
      if (l & 1) {
        resl = node(resl, tree[l]);
        l++;
      }
      if (r & 1) {
        r--;
        resr = node(tree[r], resr);
      }
    }
    return node(resl, resr);
  }
  int kth_one(int k, int i = 1) {
    if (k > tree[i])
      return -1;
    if (i >= n)
      return i - n;
    if (tree[i << 1] >= k)
      return kth_one(k, i << 1);
    return kth_one(k - tree[i << 1], i << 1 | 1);
  }
};
```

## 1.5 Treap

### ImplicitTreap.h

f7c8f9, 167 lines

```
enum DIR {
    L, R
};

template<typename T>
struct cartesian_tree {
    static cartesian_tree<T> *sentinel;
    T key;
    int priority = 0, size = 0;
    bool reverse = false;
    cartesian_tree *child[2];
    cartesian_tree *parent;
    cartesian_tree() {
        key = T();
        priority = 0;
        child[L] = child[R] = parent = this;
    }
    cartesian_tree(const T &x, int y) :
            key(x), priority(y) {
        size = 1;
        child[L] = child[R] = sentinel;
        parent = sentinel;
    }
    void push_down() {
        if (!reverse)
            return;
        reverse = 0;
        child[L]->doReverse();
        child[R]->doReverse();
    }
    void doReverse() {
        reverse ^= 1;
        swap(child[L], child[R]);
    }
    void push_up() {
        if (child[L] != sentinel)child[L]->parent = this;
        if (child[R] != sentinel)child[R]->parent = this;
```

```cpp
        size = child[L]->size + child[R]->size + 1;
    }

};

template<typename T>
cartesian_tree<T> *cartesian_tree<T>::sentinel = new
     cartesian_tree<T>();

template<typename T, template<typename > class cartesian_tree>
class implicit_treap { //1 based
    typedef cartesian_tree<T> node;
    typedef cartesian_tree<T> *nodeptr;
#define emptyNode cartesian_tree<T>::sentinel
    nodeptr root;

    void split(nodeptr root, nodeptr &l, nodeptr &r, int
         firstXElment) {
        if (root == emptyNode) {
            l = r = emptyNode;
            return;
        }
        root->push_down();
        if (firstXElment <= root->child[L]->size) {
            split(root->child[L], l, root->child[L],
                 firstXElment);
            r = root;
        } else {
            split(root->child[R], root->child[R], r,
                  firstXElment - root->child[L]->size - 1);
            l = root;
        }
        root->push_up();
    }

    nodeptr merge(nodeptr l, nodeptr r) {
        l->push_down();
        r->push_down();
        if (l == emptyNode || r == emptyNode)
            return (l == emptyNode ? r : l);
        if (l->priority > r->priority) {
            l->child[R] = merge(l->child[R], r);
            l->push_up();
            return l;
        }
        r->child[L] = merge(l, r->child[L]);
        r->push_up();
        return r;
    }
    vector<nodeptr> split_range(int s, int e) { // [x<s,s<=x<=e
         ,e<x]
        nodeptr l, m, r, tmp;
        split(root, l, tmp, s - 1);
        split(tmp, m, r, e - s + 1);
        return {l, m, r};
    }
    map <T, nodeptr> mp;
public:
    implicit_treap() :
            root(emptyNode) {
    }
    int size() {
        return root->size;
    }
    void insert(int pos, const T &key) {
        nodeptr tmp = new node(key, rand());
        nodeptr l, r;
        split(root, l, r, pos - 1);
        root = merge(merge(l, tmp), r);
    }
    void push_back(const T &value) {
        nodeptr tmp = new node(value, rand());
        mp[value] = tmp;
        root = merge(root, tmp);
    }
    T getByIndex(int pos) {
        vector<nodeptr> tmp = split_range(pos, pos);
        nodeptr l = tmp[0], m = tmp[1], r = tmp[2];
        T rt = m->key;
        root = merge(merge(l, m), r);
        return rt;
    }
    void erase(int pos) {
        vector<nodeptr> tmp = split_range(pos, pos);
        nodeptr l = tmp[0], m = tmp[1], r = tmp[2];
        delete m;
        root = merge(l, r);
    }
    void cyclic_shift(int s, int e) { //to the right
        vector<nodeptr> tmp = split_range(s, e);
        nodeptr l = tmp[0], m = tmp[1], r = tmp[2];
        nodeptr first, second;
        split(m, first, second, e - s);
        root = merge(merge(merge(l, second), first), r);
    }
    void reverse_range(int s, int e) {
        vector<nodeptr> tmp = split_range(s, e);
        nodeptr l = tmp[0], m = tmp[1], r = tmp[2];
        m->doReverse();
        root = merge(merge(l, m), r);
    }
    node range_query(int s, int e) {
        vector<nodeptr> tmp = split_range(s, e);
        nodeptr l = tmp[0], m = tmp[1], r = tmp[2];
        node rt = *m;
        root = merge(merge(l, m), r);
    }
    int getIndexByValue(const T &value) {
        nodeptr cur = mp[value];

        vector<nodeptr> path;
        path.push_back(cur);
        while (cur != root) {
            cur = cur->parent;
            path.push_back(cur);
        }

        for (int i = path.size() - 1; i >= 0; i--)
            path[i]->push_down();
        for (auto &it: path)
            it->push_up();

        cur = mp[value];
        int cnt = cur->child[L]->size + 1;
        while (cur != root) {
            nodeptr par = cur->parent;
            if (par->child[R] == cur)cnt += par->child[L]->size
                 + 1;
            cur = par;
        }
        return cnt;
    }
};

implicit_treap<long long, cartesian_tree> tp;
```

## TreapOrderedMultiset.h

275e8e, 157 lines

```cpp
enum DIR {
  L, R
};

template<typename T>
struct cartesian_tree {
  static cartesian_tree<T> *sentinel;
  T key;
  int priority = 0, frequency = 0, size = 0;
  cartesian_tree *child[2];
  cartesian_tree() {
    key = T();
    priority = 0;
    child[L] = child[R] = this;
  }
  cartesian_tree(const T &x, int y) :
      key(x), priority(y) {
    size = frequency = 1;
    child[L] = child[R] = sentinel;
  }
  void push_down() {

  }
  void push_up() {
    size = child[L]->size + child[R]->size + frequency;
  }
};

template<typename T>
cartesian_tree<T> *cartesian_tree<T>::sentinel = new
     cartesian_tree<T>();

template<typename T>
void split(cartesian_tree<T> *root, T key, cartesian_tree<T> *&
     l,
    cartesian_tree<T> *&r) {
  if (root == cartesian_tree<T>::sentinel) {
    l = r = cartesian_tree<T>::sentinel;
    return;
  }
  root->push_down();
  if (root->key <= key) {
    split(root->child[R], key, root->child[R], r);
    l = root;
  } else {
    split(root->child[L], key, l, root->child[L]);
    r = root;
  }
  root->push_up();
}

template<typename T>
cartesian_tree<T>* merge(cartesian_tree<T> *l, cartesian_tree<T
     > *r) {
  l->push_down();
  r->push_down();
  if (l == cartesian_tree<T>::sentinel || r == cartesian_tree<T
       >::sentinel)
    return (l == cartesian_tree<T>::sentinel ? r : l);
  if (l->priority > r->priority) {
    l->child[R] = merge(l->child[R], r);
    l->push_up();
    return l;
  }
  r->child[L] = merge(l, r->child[L]);
  r->push_up();
  return r;
```

```
}

template<typename T, template<typename > class cartesian_tree>
class treap {
  typedef cartesian_tree<T> node;
  typedef node *nodeptr;
#define emptyNode node::sentinel
  nodeptr root;
  void insert(nodeptr &root, nodeptr it) {
    if (root == emptyNode) {
      root = it;
    } else if (it->priority > root->priority) {
      split(root, it->key, it->child[L], it->child[R]);
      root = it;
    } else
      insert(root->child[root->key < it->key], it);
    root->push_up();
  }
  bool increment(nodeptr root, const T &key) {
    if (root == emptyNode)
      return 0;
    if (root->key == key) {
      root->frequency++;
      root->push_up();
      return root;
    }
    bool rt = increment(root->child[root->key < key], key);
    root->push_up();
    return rt;
  }
  nodeptr find(nodeptr root, const T &key) {
    if (root == emptyNode || root->key == key)
      return root;
    return find(root->child[root->key < key], key);
  }
  void erase(nodeptr &root, const T &key) {
    if (root == emptyNode)
      return;
    if (root->key == key) {
      if (--(root->frequency) == 0)
        root = merge(root->child[L], root->child[R]);
    } else
      erase(root->child[root->key < key], key);
    root->push_up();
  }
  T kth(nodeptr root, int k) {
    if (root->child[L]->size >= k)
      return kth(root->child[L], k);
    k -= root->child[L]->size;
    if (k <= root->frequency)
      return root->key;
    return kth(root->child[R], k - root->frequency);
  }
  int order_of_key(nodeptr root, const T &key) {
    if (root == emptyNode)
      return 0;
    if (key < root->key)
      return order_of_key(root->child[L], key);
    if (key == root->key)
      return root->child[L]->size;
    return root->child[L]->size + root->frequency
        + order_of_key(root->child[R], key);
  }
public:
  treap() :
      root(emptyNode) {
  }
  void insert(const T &x) {
    if (increment(root, x)) //change it to find(x) to make it
        as a set
      return;
    insert(root, new node(x, rand()));
  }
  void erase(const T &x) {
    erase(root, x);
  }
  bool find(const T &x) {
    return (find(root, x) != emptyNode);
  }
  int get_kth_number(int k) {
    assert(1 <= k && k <= size());
    return kth(root, k);
  }
  int order_of_key(const T &x) {
    return order_of_key(root, x);
  }
  int size() {
    return root->size;
  }
};

int main() {
  run();
  treap<int, cartesian_tree> tp;
}
```

## 1.6 Line Container

LineContainer.h d4cef2, 44 lines

```
struct Line {
  ll m, c;
  mutable ll p; //p is intersection between cur and next
  bool operator<(const Line &o) const {
    //change to (m>o.m,c < o.c) to get min
    if (m != o.m)
      return m < o.m;
    return c > o.c;
  }
  bool operator<(ll x) const {
    return p < x;
  }
};

struct LineContainer: multiset<Line, less<>> {
  // (for doubles, use inf = INFINITY, div(a,b) = a/b)
  static const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b);
  }
  bool isect(iterator x, iterator y) {
    if (y == end())
      return x->p = inf, 0;
    if (x->m == y->m)
      x->p = inf;
    else
      x->p = div(y->c - x->c, x->m - y->m);
    return x->p >= y->p;
  }
  void add(ll m, ll c) {
    auto z = insert( { m, c, 0 }), y = z++, x = y;
    while (isect(y, z))
      z = erase(z);
    if (x != begin() && isect(--x, y))
      isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }
  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.m * x + l.c;
  }
};
```

## 1.7 MO

MO.h 3ed675, 40 lines

```
int sqrtN; //use a constent value
struct query {
  int l, r, qIdx, block;
  query(int l, int r, int qIdx) :
      l(l), r(r), qIdx(qIdx), block(l / sqrtN) {
  }
  bool operator <(const query &o) const {
    if (block != o.block)
      return block < o.block;
    return (block % 2 == 0 ? r < o.r : r > o.r);
  }
};

int curL, curR, ans;
vector<query> q;
void add(int index);
void remove(int index);

void solve(int l, int r) {
  while (curL > l)
    add(--curL);
  while (curR < r)
    add(++curR);
  while (curL < l)
    remove(curL++);
  while (curR > r)
    remove(curR--);
}

vector<int> MO() {
  vector<int> rt(q.size());
  ans = curL = curR = 0;
  add(0);//v[0]
  sort(q.begin(), q.end());
  for (auto it : q) {
    solve(it.l, it.r);
    rt[it.qIdx] = ans;
  }
  return rt;
}
```

# Graph (2)

## 2.1 Graph Data structures

DSU.h f926d1, 32 lines

```
struct DSU {
  vector<int> rank, parent, size;
  int forsets;
  DSU(int n) {
    size = rank = parent = vector<int>(n + 1, 1);
    forsets = n;
    for (int i = 0; i <= n; i++) {
      parent[i] = i;
    }
  }
  int find_set(int v) {
```

```
    if (v == parent[v])
      return v;
    return parent[v] = find_set(parent[v]);
  }
  void link(int par, int node) {
    parent[node] = par;
    size[par] += size[node];
    if (rank[par] == rank[node])
      rank[par]++;
    forsets--;
  }
  bool union_sets(int v, int u) {
    v = find_set(v), u = find_set(u);
    if (v != u) {
      if (rank[v] < rank[u])
        swap(v, u);
      link(v, u);
    }
    return v != u;
  }
};
```

## DSURollback.h

f76712, 62 lines

```
struct dsu_save {
    int v, rnkv, u, rnku;

    dsu_save() {}

    dsu_save(int _v, int _rnkv, int _u, int _rnku)
        : v(_v), rnkv(_rnkv), u(_u), rnku(_rnku) {}
};

struct dsu_with_rollbacks {
    vector<int> p, rnk;
    int comps;
    stack<dsu_save> op;

    dsu_with_rollbacks() {}

    dsu_with_rollbacks(int n) {
        p.resize(n + 1);
        rnk.resize(n + 1);
        for (int i = 1; i <= n; i++) {
            p[i] = i;
            rnk[i] = 0;
        }
        comps = n;
    }

    int find_set(int v) {
        return (v == p[v]) ? v : find_set(p[v]);
    }

    bool same_group(int v, int u) {
        v = find_set(v);
        u = find_set(u);
        if (v == u)
            return false;
        comps--;
        if (rnk[v] > rnk[u])
            swap(v, u);
        op.push(dsu_save(v, rnk[v], u, rnk[u]));
        p[v] = u;
        if (rnk[u] == rnk[v])
            rnk[u]++;
        return true;
    }
    void snapshot() {
        // this function save the current trees (merged) and
           don't rollback them any more
        while (!op.empty())
            op.pop();
    }
    void rollback() {
        // you can erase the while loop if you want to rollback
            just the last merge
        while (!op.empty()) {
            dsu_save x = op.top();
            op.pop();
            comps++;
            p[x.v] = x.v;
            rnk[x.v] = x.rnkv;
            p[x.u] = x.u;
            rnk[x.u] = x.rnku;
        }
    }
};
```

## LCA.h

31639d, 61 lines

```
class LCA {
  int n, logN, root = 1;
  vector<int> depth;
  vector<vector<int>> adj, lca;
  void dfs(int node, int parent) {
    lca[node][0] = parent;
    depth[node] = (~parent ? depth[parent] + 1 : 0);
    for (int k = 1; k <= logN; k++) {
      int up_parent = lca[node][k - 1];
      if (~up_parent)
        lca[node][k] = lca[up_parent][k - 1];
    }
    for (int child : adj[node])
      if (child != parent)
        dfs(child, node);
  }
public:
  LCA(const vector<vector<int>> &_adj, int root = 1) :
      root(root), adj(_adj) {
    adj = _adj;
    n = adj.size() - 1;
    logN = log2(n);
    lca = vector<vector<int>>(n + 1, vector<int>(logN + 1, -1))
        ;
    depth = vector<int>(n + 1);
    dfs(root, -1);
  }
  int get_LCA(int x, int y) {
    if (depth[x] < depth[y])
      swap(x, y);
    for (int k = logN; k >= 0; k--)
      if (depth[x] - (1 << k) >= depth[y])
        x = lca[x][k];
    if (x == y)
      return x;
    for (int k = logN; k >= 0; k--) {
      if (lca[x][k] != lca[y][k]) {
        x = lca[x][k];
        y = lca[y][k];
      }
    }
    return lca[x][0];
  }
  int get_distance(int u, int v) {
    return depth[u] + depth[v] - 2 * depth[get_LCA(u, v)];
  }
  int kth_ancestor(int node, int dist) {
    for (int i = logN; i >= 0 && ~node; i--)
      if (dist & (1 << i))
        node = lca[node][i];
    return node;
  }
  edge get_path(int u, int LCA) {
    edge rt;
    for (int k = logN; k >= 0; k--)
      if (depth[u] - (1 << k) >= depth[LCA]) {
        rt = merge(rt, lca[u][k]);
        u = lca[u][k].to;
      }
    return rt;
  }
};
```

## HLD.h

a18637, 129 lines

```
// 1-based,if value in node,just update it after build chains
// don't forget to call build_chains after add edges.

class heavy_light_decomposition {
  int n, is_value_in_edge;
  vector<int> parent, depth, heavy, root, pos_in_array,
      pos_to_node, size;
  const static int merge(int a, int b); //implement it
  struct array_ds { //implement it
    int n;
    array_ds(int n) :
        n(n) {
    }
    void update(int pos, int value);
    int query(int l, int r);
  } seg;
  struct TREE {
    int cnt_edges = 1;
    vector<vector<int>> adj;
    //need for value in edges
    vector<vector<int>> edge_idx;
    //edge_to need for undirected tree //end of edge in
        directed tree
    vector<int> edge_to, edge_cost;
    TREE(int n) :
        adj(n + 1), edge_idx(n + 1), edge_to(n + 1), edge_cost(
            n + 1) {
    }
    void add_edge(int u, int v, int c) {
      adj[u].push_back(v);
      adj[v].push_back(u);
      edge_idx[u].push_back(cnt_edges);
      edge_idx[v].push_back(cnt_edges);
      edge_cost[cnt_edges] = c;
      cnt_edges++;
    }
  } tree;
  int dfs_hld(int node) {
    int size = 1, max_sub_tree = 0;
    for (int i = 0; i < (int) tree.adj[node].size(); i++) {
      int ch = tree.adj[node][i], edge_idx = tree.edge_idx[node
          ][i];
      if (ch != parent[node]) {
        tree.edge_to[edge_idx] = ch;
        parent[ch] = node;
        depth[ch] = depth[node] + 1;
        int child_size = dfs_hld(ch);
        if (child_size > max_sub_tree)
          heavy[node] = ch, max_sub_tree = child_size;
        size += child_size;
      }
```

```
    }
    return size;
  }
  vector<tuple<int, int, bool>> get_path(int u, int v) { //l,r,
      must_reverse?
    vector<pair<int, int>> tmp[2];
    bool idx = 1;
    while (root[u] != root[v]) {
      if (depth[root[u]] > depth[root[v]]) {
        swap(u, v);
        idx = !idx;
      }
      //if value in edges ,you need value of root[v] also (
          connecter edge)
      tmp[idx].push_back( { pos_in_array[root[v]], pos_in_array
          [v] });
      v = parent[root[v]];
    }
    if (depth[u] > depth[v]) {
      swap(u, v);
      idx = !idx;
    }
    if (!is_value_in_edge || u != v)
      tmp[idx].push_back( { pos_in_array[u] + is_value_in_edge,
          pos_in_array[v] });
    reverse(all(tmp[1]));
    vector<tuple<int, int, bool>> rt;
    for (int i = 0; i < 2; i++)
      for (auto &it : tmp[i])
        rt.emplace_back(it.first, it.second, i == 0);
    return rt; //u is LCA
  }
public:
  heavy_light_decomposition(int n, bool is_value_in_edge) :
      n(n), is_value_in_edge(is_value_in_edge), seg(n + 1),
          tree(n + 1) {
    heavy = vector<int>(n + 1, -1);
    parent = depth = root = pos_in_array = pos_to_node = size =
        vector<int>(
      n + 1);
  }
  void add_edge(int u, int v, int c = 0) {
    tree.add_edge(u, v, c);
  }
  void build_chains(int src = 1) {
    parent[src] = -1;
    dfs_hld(src);
    for (int chain_root = 1, pos = 1; chain_root <= n;
        chain_root++) {
      if (parent[chain_root] == -1
          || heavy[parent[chain_root]] != chain_root)
        for (int j = chain_root; j != -1; j = heavy[j]) {
          root[j] = chain_root;
          pos_in_array[j] = pos++;
          pos_to_node[pos_in_array[j]] = j;
        }
    }
    if (is_value_in_edge)
      for (int i = 1; i < n; i++)
        update_edge(i, tree.edge_cost[i]);
  }
  void update_node(int node, int value) {
    seg.update(pos_in_array[node], value);
  }
  void update_edge(int edge_idx, int value) {
    update_node(tree.edge_to[edge_idx], value);
  }
  void update_path(int u, int v, ll c) {
    vector<tuple<int, int, bool>> intervals = get_path(u, v);
    for (auto &it : intervals)
      seg.update(get<0>(it), get<1>(it), c);
  }
  node query_in_path(int u, int v) {
    vector<tuple<int, int, bool>> intervals = get_path(u, v);
    //initial value,check if handling u == v
    node query_res = 0;
    for (auto &it : intervals) {
      int l, r;
      bool rev;
      tie(l, r, rev) = it;
      node cur = seg.query(l, r);
      if (rev)
        cur.reverse();
      query_res = node(query_res, cur);
    }
    return query_res;
  }
};
```

## VirtualTree.h

e35cff, 62 lines

```
int subsize[N], depth[N], dfs_num[N], id[N], have[N], timer, n;
vector<vector<int>> adj;

void calc(int node) {
    dfs_num[node] = ++timer;
    subsize[node] = 1;
    for (auto ch: adj[node]) {
        adj[ch].erase(find(all(adj[ch]), node));
        depth[ch] = depth[node] + 1;
        calc(ch);
        subsize[node] += subsize[ch];
    }
}

bool parent(int node, int par) {
    return dfs_num[par] <= dfs_num[node] && dfs_num[node] <
         dfs_num[par] + subsize[par];
}

struct virtual_tree {
    vector<int> nodes;
    vector<vector<int>> adj;

    virtual_tree(const vector<int> &v) : nodes(v) {
        for (auto &it: nodes) have[it] = true;

        sort(all(nodes), [&](int a, int b) {
            return dfs_num[a] < dfs_num[b];
        });

        int tmp = nodes.size();
        for (int j = 0; j + 1 < tmp; j++) {
            int lca = tree.get_LCA(nodes[j], nodes[j + 1]);
            nodes.push_back(lca);
        }
        nodes.push_back(1);
        sort(all(nodes));
        nodes.erase(unique(all(nodes)), nodes.end());

        sort(all(nodes), [&](int a, int b) {
            return dfs_num[a] < dfs_num[b];
        });

        int cnt = 0;
        for (auto &it: nodes)id[it] = cnt++;

        stack<int> stk;
        adj = vector<vector<int>>(cnt);
        for (auto &it: nodes) {
            while (!stk.empty() && !(parent(it, stk.top())))
                stk.pop();
            if (stk.size()) {
                adj[id[stk.top()]].push_back(it); // it or id[
                    it] ??
            }
            stk.push(it);
        }
        dfs(1);

        for (auto &it: nodes)have[it] = false;
    }

};
```

## CentroidDecomposition.h

4ea09b, 84 lines

```
class centroid_decomposition {
  vector<bool> centroidMarked;
  vector<int> size;
  void dfsSize(int node, int par) {
    size[node] = 1;
    for (int ch : adj[node])
      if (ch != par && !centroidMarked[ch]) {
        dfsSize(ch, node);
        size[node] += size[ch];
      }
  }
  int getCenter(int node, int par, int size_of_tree) {
    for (int ch : adj[node]) {
      if (ch == par || centroidMarked[ch])
        continue;
      if (size[ch] * 2 > size_of_tree)
        return getCenter(ch, node, size_of_tree);
    }
    return node;
  }
  int getCentroid(int src) {
    dfsSize(src, -1);
    int centroid = getCenter(src, -1, size[src]);
    centroidMarked[centroid] = true;
    return centroid;
  }
  int decomposeTree(int root) {
    root = getCentroid(root);
    solve(root);
    for (int ch : adj[root]) {
      if (centroidMarked[ch])
        continue;
      int centroid_of_subtree = decomposeTree(ch);
      //note: root and centroid_of_subtree probably not have a
          direct edge in adj
      centroidTree[root].push_back(centroid_of_subtree);
      centroidParent[centroid_of_subtree] = root;
    }
    return root;
  }
  void calc(int node, int par) {
    //TO-DO
    for (int ch : adj[node])
      if (ch != par && !centroidMarked[ch])
        calc(ch, node);
  }
  void add(int node, int par) {
    //TO-DO
    for (int ch : adj[node])
      if (ch != par && !centroidMarked[ch])
```

```cpp
      add(ch, node);
  }
  void remove(int node, int par) {
    //TO-DO
    for (int ch : adj[node])
      if (ch != par && !centroidMarked[ch])
        remove(ch, node);
  }
  void solve(int root) {
    //add root
    for (int ch : adj[root])
      if (!centroidMarked[ch]) {
        calc(ch, root);
        add(ch, root);
      }
    //TO-DO
    //remove root
    for (int ch : adj[root])
      if (!centroidMarked[ch])
        remove(ch, root);
  }
public:
  int n, root;
  vector<vector<int>> adj, centroidTree;
  vector<int> centroidParent;
  centroid_decomposition(vector<vector<int>> &adj) :
      adj(adj) {
    n = (int) adj.size() - 1;
    size = vector<int>(n + 1);
    centroidTree = vector<vector<int>>(n + 1);
    centroidParent = vector<int>(n + 1, -1);
    centroidMarked = vector<bool>(n + 1);
    root = decomposeTree(1);
  }
};
```

Sack.h — 81546e, 20 lines

```cpp
void add(int node, int par, int big);
void remove(int node, int par, int big);

void dfs(int node, int par, bool keep) {
    int mx = -1, big = -1;
    for (int ch: adj[node]) {
        if (ch != par && subsize[ch] > mx)
            mx = subsize[ch], big = ch;
    }
    for (int ch: adj[node]) {
        if (ch != par && ch != big)
            dfs(ch, node, false);
    }
    if (big != -1) dfs(big, node, true);
    add(node, par, big);
    // SOLVE
    if (!keep) {
        remove(node, par, -1);
    }
}
```

## 2.2 Flow

### 2.2.1 Bipartite Matching

MaximumBipartiteMatching.h — 7e53b6, 42 lines

```cpp
// O(VE) in worst case, but in practice it runs in O(E) or O(
      sqrt(V)E)
vector<vector<int>> adj;
vector<int> rowAssign, colAssign, vis;//make vis array instance
      of vector
int test_id;
bool canMatch(int i) {
  if (vis[i] == test_id)
    return false;
  vis[i] = test_id;
  for (int j : adj[i]) {
    if (colAssign[j] == -1) {
      colAssign[j] = i;
      rowAssign[i] = j;
      return true;
    }
  }
  for (int j : adj[i]) {
    if (canMatch(colAssign[j])) {
      colAssign[j] = i;
      rowAssign[i] = j;
      return true;
    }
  }
  return false;
}

// O(rows * edges) //number of operation could by strictly less
      than order
int maximum_bipartite_matching(int rows, int cols) {
  int maxFlow = 0;
  rowAssign = vector<int>(rows, -1);
  colAssign = vector<int>(cols, -1);
  vis = vector<int>(rows);
  for (int i = 0; i < rows; i++) {
    test_id++;
    if (canMatch(i))
      maxFlow++;
  }
  vector<pair<int, int>> matches;
  for (int j = 0; j < cols; j++)
    if (~colAssign[j])
      matches.push_back( { colAssign[j], j });
  return maxFlow;
}
```

HopcroftKarp.h — f4cb9b, 58 lines

```cpp
//Hopcroft-Karp algorithm for maximum bipartite matching
//O(sqrt(V) * E)
struct Hopcroft_Karp {//1-based
#define NIL 0
#define INF INT_MAX
  int n, m;
  vector<vector<int>> adj;
  vector<int> rowAssign, colAssign, dist;
  bool bfs() {
    queue<int> q;
    dist = vector<int>(adj.size(), INF);
    for (int i = 1; i <= n; i++)
      if (rowAssign[i] == NIL) {
        dist[i] = 0;
        q.push(i);
      }
    while (!q.empty()) {
      int cur = q.front();
      q.pop();
      if (dist[cur] >= dist[NIL])break;
      for (auto& nxt : adj[cur]) {
        if (dist[colAssign[nxt]] == INF) {
          dist[colAssign[nxt]] = dist[cur] + 1;
          q.push(colAssign[nxt]);
        }
      }
    }
    return dist[NIL] != INF;
  }
  bool dfs(int i) {
    if (i == NIL)
      return true;
    for (int j : adj[i]) {
      if (dist[colAssign[j]] == dist[i] + 1 && dfs(colAssign[j
          ])) {
        colAssign[j] = i;
        rowAssign[i] = j;
        return true;
      }
    }
    dist[i] = INF;
    return false;
  }
  Hopcroft_Karp(int n, int m)
    :n(n), m(m), adj(n + 1), rowAssign(n + 1), colAssign(m + 1)
        {
  }
  void addEdge(int u, int v) {
    adj[u].push_back(v);
  }
  int maximum_bipartite_matching() {
    int rt = 0;
    while (bfs()) {
      for (int i = 1; i <= n; i++)
        if (rowAssign[i] == NIL && dfs(i))
          rt++;
    }
    return rt;
  }
};
```

### 2.2.2 Max Flow

Dinic.h — 71a43c, 70 lines

```cpp
struct Dinic {
  struct flowEdge {
    int from, to;
    ll cap, flow = 0;
    flowEdge(int from, int to, ll cap) :
        from(from), to(to), cap(cap) {
    }
  };
  vector<flowEdge> edges;
  int n, m = 0, source, sink;
  vector<vector<int>> adj;
  vector<int> level, ptr;
  Dinic(int n, int source, int sink) :
      n(n), source(source), sink(sink), adj(n), level(n), ptr(n
          ) {
  }
  void addEdge(int u, int v, ll cap) {
    edges.emplace_back(u, v, cap);
    edges.emplace_back(v, u, 0);
    adj[u].push_back(m);
    adj[v].push_back(m + 1);
    m += 2;
  }
  bool bfs() {
    queue<int> q;
    level = vector<int>(n, -1);
    level[source] = 0;
    q.push(source);
    while (!q.empty()) {
      int cur = q.front();
      q.pop();
```

```
      for (auto &id : adj[cur]) {
        if (edges[id].cap - edges[id].flow <= 0)
          continue;
        int nxt = edges[id].to;
        if (level[nxt] != -1)
          continue;
        level[nxt] = level[cur] + 1;
        q.push(nxt);
      }
    }
    return level[sink] != -1;
  }
  ll dfs(int node, ll cur_flow) {
    if (cur_flow == 0 || node == sink)
      return cur_flow;
    for (int &cid = ptr[node]; cid < adj[node].size(); cid++) {
      int id = adj[node][cid];
      int nxt = edges[id].to;
      if (level[node] + 1 != level[nxt]
          || edges[id].cap - edges[id].flow <= 0)
        continue;
      ll tmp = dfs(nxt, min(cur_flow, edges[id].cap - edges[id
          ].flow));
      if (tmp == 0)
        continue;
      edges[id].flow += tmp;
      edges[id ^ 1].flow -= tmp;
      return tmp;
    }
    return 0;
  }
  ll flow() {
    ll max_flow = 0;
    while (bfs()) {
      fill(ptr.begin(), ptr.end(), 0);
      while (ll pushed = dfs(source, INF))
        max_flow += pushed;
    }
    return max_flow;
  }
};
```

EdmondsKarp.h

c79b45, 39 lines

```
//O( V * E * E)

int n;
int capacity[101][101];
int getPath(int src, int dest, vector<int> &parent) {
  parent = vector<int>(n + 1, -1);
  queue<pair<int, int>> q;
  q.push( { src, INF });
  while (q.size()) {
    int cur = q.front().first, flow = q.front().second;
    q.pop();
    if (cur == dest)
      return flow;
    for (int i = 1; i <= n; i++)
      if (parent[i] == -1 && capacity[cur][i]) {
        parent[i] = cur;
        q.push( { i, min(flow, capacity[cur][i]) });
        if (i == dest)
          return q.back().second;
      }
  }
  return 0;
}
int Edmonds_Karp(int source, int sink) {
  int max_flow = 0;
  int new_flow = 0;
  vector<int> parent(n + 1, -1);
  while (new_flow = getPath(source, sink, parent)) {
    max_flow += new_flow;
    int cur = sink;
    while (cur != source) {
      int prev = parent[cur];
      capacity[prev][cur] -= new_flow;
      capacity[cur][prev] += new_flow;
      cur = prev;
    };
  }
  return max_flow;
}
```

### 2.2.3 Min Cost Max Flow

MCMF.h

702ef1, 89 lines

```
struct MCMF { //0-based
  struct edge {
    int from, to, cost, cap, flow, backEdge;
    edge() {
      from = to = cost = cap = flow = backEdge = 0;
    }
    edge(int from, int to, int cost, int cap, int flow, int
         backEdge) :
        from(from), to(to), cost(cost), cap(cap), flow(flow),
            backEdge(
           backEdge) {
    }
    bool operator <(const edge &other) const {
      return cost < other.cost;
    }
  };
  int n, src, dest;
  vector<vector<edge>> adj;
  const int OO = 1e9;
  MCMF(int n, int src, int dest) :
      n(n), src(src), dest(dest), adj(n) {
  }
  void addEdge(int u, int v, int cost, int cap) {
    edge e1 = edge(u, v, cost, cap, 0, adj[v].size());
    edge e2 = edge(v, u, -cost, 0, 0, adj[u].size());
    adj[u].push_back(e1);
    adj[v].push_back(e2);
  }
  pair<int, int> minCostMaxFlow() {
    int maxFlow = 0, cost = 0;
    while (true) {
      vector<pair<int, int>> path = spfa();
      if (path.empty())
        break;
      int new_flow = OO;
      for (auto &it : path) {
        edge &e = adj[it.first][it.second];
        new_flow = min(new_flow, e.cap - e.flow);
      }
      for (auto &it : path) {
        edge &e = adj[it.first][it.second];
        e.flow += new_flow;
        cost += new_flow * e.cost;
        adj[e.to][e.backEdge].flow -= new_flow;
      }
      maxFlow += new_flow;
    }
    return {maxFlow,cost};
  }
  enum visit {
    finished, in_queue, not_visited
  };
  vector<pair<int, int>> spfa() {
    vector<int> dis(n, OO), prev(n, -1), from_edge(n), state(n,
        not_visited);
    deque<int> q;
    dis[src] = 0;
    q.push_back(src);
    while (!q.empty()) {
      int u = q.front();
      q.pop_front();
      state[u] = finished;
      for (int i = 0; i < adj[u].size(); i++) {
        edge e = adj[u][i];
        if (e.flow >= e.cap || dis[e.to] <= dis[u] + e.cost)
          continue;
        dis[e.to] = dis[u] + e.cost;
        prev[e.to] = u;
        from_edge[e.to] = i;
        if (state[e.to] == in_queue)
          continue;
        if (state[e.to] == finished
            || (!q.empty() && dis[q.front()] > dis[e.to]))
          q.push_front(e.to);
        else
          q.push_back(e.to);
        state[e.to] = in_queue;
      }
    }
    if (dis[dest] == OO)
      return {};
    vector<pair<int, int>> path;
    int cur = dest;
    while (cur != src) {
      path.push_back( { prev[cur], from_edge[cur] });
      cur = prev[cur];
    }
    reverse(path.begin(), path.end());
    return path;
  }
};
```

## 2.3 Shortest Path

Bellmanford.h

ee0d03, 70 lines

```
#define oo 0x3f3f3f3fLL
struct edge {
  int from, to, weight;
  edge() {
    from = to = weight = 0;
  }
  edge(int from, int to, int weight) :
      from(from), to(to), weight(weight) {
  }
  bool operator <(const edge &other) const {
    return weight > other.weight;
  }
};

vector<edge> edgeList;
//O(V*E)
void bellmanford(int n, int src, int dest = -1) {
  vector<int> dis(n + 1, oo), prev(n + 1, -1);
  dis[src] = 0;
  bool negativeCycle = false;
  int last = -1, tmp = n;
  while (tmp--) {
    last = -1;
    for (edge e : edgeList)
      if (dis[e.to] > dis[e.from] + e.weight) {
```

```
        dis[e.to] = dis[e.from] + e.weight;
        prev[e.to] = e.from;
        last = e.to;
      }
    if (last == -1)
      break;
    if (tmp == 0)
      negativeCycle = true;
  }
  if (last != -1) {
    for (int i = 0; i < n; i++)
      last = prev[last];
    vector<int> cycle;
    for (int cur = last; cur != last || cycle.size() > 1; cur =
         prev[cur])
      cycle.push_back(cur);
    reverse(cycle.begin(), cycle.end());
  }
  vector<int> path;
  while (dest != -1) {
    path.push_back(dest);
    dest = prev[dest];
  }
  reverse(path.begin(), path.end());
}

void difference_constraints() {
  int m;
  cin >> m;
  int cnt = 1;
  while (m--) {
    string x1, x2;
    int w; // x1 - x2 <= w
    cin >> x1 >> x2 >> w;
    map<string, int> id;
    if (id.find(x1) == id.end())
      id[x1] = cnt++;
    if (id.find(x2) == id.end())
      id[x2] = cnt++;
    edgeList.emplace_back(id[x2], id[x1], w);
  }
  for (int i = 1; i < cnt; i++)
    edgeList.emplace_back(cnt, i, 0);
  bellmanford(cnt, cnt);
}
```

### SPFA.h

7a4ab9, 43 lines

```
struct edge {
  int from, to;
  ll weight;
  edge() {
    from = to = weight = 0;
  }
  edge(int from, int to, ll weight) :
      from(from), to(to), weight(weight) {
  }
  bool operator <(const edge &other) const {
    return weight < other.weight;
  }
};

vector<vector<edge>> adj;

void spfa(int src) {
  enum visit {
    finished, in_queue, not_visited
  };
  int n = adj.size();
  vector<int> dis(n, INF), prev(n, -1), state(n, not_visited);
  dis[src] = 0;
  deque<int> q;
  q.push_back(src);
  while (!q.empty()) {
    int u = q.front();
    q.pop_front();
    state[u] = finished;
    for (auto &e : adj[u]) {
      if (dis[e.to] > dis[e.from] + e.cost) {
        dis[e.to] = dis[e.from] + e.cost;
        prev[e.to] = e.from;
        if (state[e.to] == not_visited) {
          q.push_back(e.to);
        } else if (state[e.to] == finished) {
          q.push_front(e.to);
        }
        state[e.to] = in_queue;
      }
    }
  }
}
```

## 2.4 Tarjan

### EdgeClassification.h

c32777, 20 lines

```
vector<vector<int>> adj;
vector<int> start, finish;
int timer;

void dfsEdgeClassification(int node) {
  start[node] = timer++;
  for (int child : adj[node]) {
    if (start[child] == -1)
      dfsEdgeClassification(child);
    else {
      if (finish[child] == -1)
        ; // Back Edge
      else if (start[node] < start[child])
        ; // Forward Edge
      else
        ; // Cross Edge
    }
  }
  finish[node] = timer++;
}
```

### SCC.h

89d031, 48 lines

```
vector<vector<int>> adj, scc;
vector<set<int>> dag;
vector<int> dfs_num, dfs_low, compId;
vector<bool> inStack;
stack<int> stk;
int timer;

void dfs(int node) {
  dfs_num[node] = dfs_low[node] = ++timer;
  stk.push(node);
  inStack[node] = 1;
  for (int child : adj[node])
    if (!dfs_num[child]) {
      dfs(child);
      dfs_low[node] = min(dfs_low[node], dfs_low[child]);
    } else if (inStack[child])
      dfs_low[node] = min(dfs_low[node], dfs_num[child]);
  //can be dfs_low[node] = min(dfs_low[node], dfs_low[child]);
  if (dfs_low[node] == dfs_num[node]) {
    scc.push_back(vector<int>());
    int v = -1;
    while (v != node) {
      v = stk.top();
      stk.pop();
      inStack[v] = 0;
      scc.back().push_back(v);
      compId[v] = scc.size() - 1;
    }
  }
}

void SCC() {
  timer = 0;
  dfs_num = dfs_low = compId = vector<int>(adj.size());
  inStack = vector<bool>(adj.size());
  scc = vector<vector<int>>();
  for (int i = 1; i < adj.size(); i++)
    if (!dfs_num[i])
      dfs(i);
}

void DAG() {
  dag = vector<set<int>>(scc.size());
  for (int i = 1; i < adj.size(); i++)
    for (int j : adj[i])
      if (compId[i] != compId[j])
        dag[compId[i]].insert(compId[j]);
}
```

### ArticulationPointsAndBridges.h

edcf05, 50 lines

```
vector<vector<int>> adj;
vector<int> dfs_num, dfs_low;
vector<bool> articulation_point;
vector<pair<int, int>> bridge;
stack<pair<int, int>> edges;
vector<vector<pair<int, int>>> BCC; //biconnected components
int timer, cntChild;

// O(n + m)
void dfs(int node, int par) {
  dfs_num[node] = dfs_low[node] = ++timer;
  for (int child : adj[node]) {
    if (par != child && dfs_num[child] < dfs_num[node])
      edges.push( { node, child });

    if (!dfs_num[child]) {
      if (par == -1)
        cntChild++;
      dfs(child, node);
      if (dfs_low[child] >= dfs_num[node]) {
        articulation_point[node] = 1;
        //get biconnected component
        BCC.push_back(vector<pair<int, int>>());
        pair<int, int> edge;
        do {
          edge = edges.top();
          BCC.back().push_back(edge);
          edges.pop();
        } while (edge.first != node || edge.second != child);
      }
      if (dfs_low[child] > dfs_num[node]) //can be (dfs_low[
          child] == dfs_num[child])
        bridge.push_back( { node, child });
      dfs_low[node] = min(dfs_low[node], dfs_low[child]);
    } else if (child != par)
      dfs_low[node] = min(dfs_low[node], dfs_num[child]);
  }
}
```

```
void articulation_points_and_bridges() {
  timer = 0;
  dfs_num = dfs_low = vector<int>(adj.size());
  articulation_point = vector<bool>(adj.size());
  bridge = vector<pair<int, int>>();
  for (int i = 1; i < adj.size(); i++)
    if (!dfs_num[i]) {
      cntChild = 0;
      dfs(i, -1);
      articulation_point[i] = cntChild > 1;
    }
}
```

2SAT.h

"SCC.h" 56cc5b, 30 lines

```
int n;
int Not(int x) {
  return (x > n ? x - n : x + n);
}

void addEdge(int a, int b) {
  adj[Not(a)].push_back(b);
  adj[Not(b)].push_back(a);
}

void add_xor_edge(int a, int b) {
  addEdge(Not(a), Not(b));
  addEdge(a, b);
}

bool _2SAT(vector<int> &value) {
  SCC();
  for (int i = 1; i <= n; i++)
    if (compId[i] == compId[Not(i)])
      return false;
  vector<int> assign(scc.size(), -1);
  for (int i = 0; i < scc.size(); i++)
    if (assign[i] == -1) {
      assign[i] = true;
      assign[compId[Not(scc[i].back())]] = false;
    }
  for (int i = 1; i <= n; i++)
    value[i] = assign[compId[i]];
  return true;
}
```

ShortestCycle.h

de063a, 30 lines

```
//Shortest cycle starting from node O(n^2)
int get_shortest_cycle(int start) {
    vector<int> dist(n + 1, INF), parent(n + 1), group(n + 1);
    vector<bool> vis(n + 1);
    dist[start] = 0;
    group[start] = start;
    while (true) {
        int cur = 0;
        for (int i = 1; i <= n; i++)
            if (!vis[i] && dist[i] < dist[cur])cur = i;
        if (dist[cur] == (int) INF)break;
        vis[cur] = true;
        for (int ch = 1; ch <= n; ch++) {
            if (dist[ch] > dist[cur] + adj[cur][ch]) {
                dist[ch] = dist[cur] + adj[cur][ch];
                parent[ch] = cur;
                group[ch] = (cur == start ? ch : group[cur]);
            }
        }
    }
    int mn = INF;
    for (int i = 1; i <= n; i++)
        for (int j = i + 1; j <= n; j++) {
            if (parent[i] == j || parent[j] == i)continue;
            if (group[i] == group[j]) continue;
            if (dist[i] == (int) INF || dist[j] == (int) INF)
                continue;
            mn = min(mn, dist[i] + dist[j] + adj[i][j]);
        }
    return mn;
}
```

# Strings (3)

## 3.1 Rolling Hash

Hashing.h

3a850c, 23 lines

```
struct hashing {
    int MOD, BASE;
    vector<int> Hash, modInv;
    hashing(string s, int MOD, int BASE, char first_char = 'a')
          :
            MOD(MOD), BASE(BASE), Hash(sz(s) + 1), modInv(sz(s)
                + 1) {
        ll BASE_INV = power(BASE, MOD - 2, MOD);
        modInv[0] = 1;
        ll base = 1;
        for (int i = 1; i <= sz(s); i++) {
            Hash[i] = (Hash[i - 1] + (s[i - 1] - first_char +
                1) * base) % MOD;
            modInv[i] = (modInv[i - 1] * BASE_INV) % MOD;
            base = (base * BASE) % MOD;
        }
    }
    int getHash(int l, int r) { //1-based
        return (1LL * (Hash[r] - Hash[l - 1] + MOD) % MOD *
             modInv[l - 1]) % MOD;
    }
};

//MOD = 1e9 + 9  ,BASE = 31
//MOD = 2000000011  ,BASE = 53 ->careful of overflow
//MOD = 998634293,BASE = 953
//MOD = 986464091,BASE = 1013
```

## 3.2 String matching

KMP.h

505aa1, 92 lines

```
struct KMP {
  string pattern;
  vector<int> longestPrefix;
  KMP(const string& str) :pattern(str) {
    failure_function();
  }
  int fail(int k, char nxt) {
    while (k > 0 && pattern[k] != nxt)
      k = longestPrefix[k - 1];
    if (nxt == pattern[k])
      k++;
    return k;
  }
  void failure_function() {
    int n = pattern.size();
    longestPrefix = vector<int>(n);
    for (int i = 1, k = 0; i < n; i++)
      longestPrefix[i] = k = fail(k, pattern[i]);
  }
  void match(const string& str) {
    int n = str.size();
    int m = pattern.size();
    for (int i = 0, k = 0; i < n; i++) {
      k = fail(k, str[i]);
      if (k == m) {
        cout << i - m + 1 << endl; //0-based
        k = longestPrefix[k - 1]; // if you want next match
      }
    }
  }
};

vector<bool> suffix_pal(string s) { //[i..n-1] pal?
  string r = s;
  reverse(all(r));
  vector<bool> v(s.size());
  v[0] = (s == r);
  string pattern = r + "#" + s;
  int n = pattern.size();
  vector<int> longestPrefix(n);
  int k = 0;
  for (int i = 1; i < n; i++) {
    while (k > 0 && pattern[k] != pattern[i])
      k = longestPrefix[k - 1];
    if (pattern[i] == pattern[k])
      k++;
    longestPrefix[i] = k;
  }
  while (k > 0) {
    v[s.size() - k] = true;
    k = longestPrefix[k - 1];
  }
  return v;
}

vector<bool> prefix_pal(string s) { // [0..i] pal?
  string r = s;
  reverse(all(r));
  vector<bool> v(s.size());
  v.back() = (s == r);
  string pattern = s + "#" + r;
  int n = pattern.size();
  vector<int> longestPrefix(n);
  int k = 0;
  for (int i = 1; i < n; i++) {
    while (k > 0 && pattern[k] != pattern[i])
      k = longestPrefix[k - 1];
    if (pattern[i] == pattern[k])
      k++;
    longestPrefix[i] = k;
  }
  while (k > 0) {
    v[k - 1] = true;
    k = longestPrefix[k - 1];
  }
  return v;
}

//frq[i] = number of occur s[0..i] in s
vector<int> build_fre_prefix(const string& s) {
  KMP kmp(s);
  kmp.failure_function();
  vector<int> f = kmp.longestPrefix;
  int n = sz(s);
  vector<int> frq(n);
  for (int i = n - 1; i >= 0; i--)
    if (f[i])
      frq[f[i] - 1] += frq[i] + 1;
```

```
  for (auto& it : frq)it++;
  return frq;
}
```

## AhoCorasick.h

54393a, 82 lines

```
#define all(v) v.begin(),v.end()

struct aho_corasick {
  struct trie_node {
    vector<int> pIdxs; //probably take memory limit
    map<char, int> next;
    int fail;
    trie_node() :
        fail(0) {
    }
    bool have_next(char ch) {
      return next.find(ch) != next.end();
    }
    int& operator[](char ch) {
      return next[ch];
    }
  };
  vector<trie_node> t;
  vector<string> patterns;
  vector<int> end_of_pattern;
  vector<vector<int>> adj;
  int insert(const string &s, int patternIdx) {
    int root = 0;
    for (const char &ch : s) {
      if (!t[root].have_next(ch)) {
        t.push_back(trie_node());
        t[root][ch] = t.size() - 1;
      }
      root = t[root][ch];
    }
    t[root].pIdxs.push_back(patternIdx);
    return root;
  }
  int next_state(int cur, char ch) {
    while (cur > 0 && !t[cur].have_next(ch))
      cur = t[cur].fail;
    if (t[cur].have_next(ch))
      return t[cur][ch];
    return 0;
  }
  void buildAhoTree() {
    queue<int> q;
    for (auto &child : t[0].next)
      q.push(child.second);
    while (!q.empty()) {
      int cur = q.front();
      q.pop();
      for (auto &child : t[cur].next) {
        int k = next_state(t[cur].fail, child.first);
        t[child.second].fail = k;
        vector<int> &idxs = t[child.second].pIdxs;
        //dp[child.second] = max(dp[child.second],dp[k]);
        idxs.insert(idxs.end(), all(t[k].pIdxs));
        q.push(child.second);
      }
    }
  }
  void buildFailureTree() {
    adj = vector<vector<int>>(t.size());
    for (int i = 1; i < t.size(); i++)
      adj[t[i].fail].push_back(i);
  }
  aho_corasick(const vector<string> &_patterns) {
    t.push_back(trie_node());
    patterns = _patterns;
    end_of_pattern = vector<int>(patterns.size());
    for (int i = 0; i < patterns.size(); i++)
      end_of_pattern[i] = insert(patterns[i], i);
    buildAhoTree();
    buildFailureTree();
  }
  vector<vector<int>> match(const string &str) {
    int k = 0;
    vector<vector<int>> rt(patterns.size());
    for (int i = 0; i < str.size(); i++) {
      k = next_state(k, str[i]);
      for (auto &it : t[k].pIdxs)
        rt[it].push_back(i);
    }
    return rt;
  }
};
```

## ZAlgo.h

36a580, 18 lines

```
// z[i] equal the length of the longest substring starting from
      s[i] which is also a prefix of s
vector<int> z_algo(string s) {
  int n = s.size();
  vector<int> z(n);
  z[0] = n;
  for (int i = 1, L = 1, R = 1; i < n; i++) {
    int k = i - L;
    if (z[k] + i >= R) {
      L = i;
      R = max(R, i);
      while (R < n && s[R - L] == s[R])
        R++;
      z[i] = R - L;
    } else
      z[i] = z[k];
  }
  return z;
}
```

## 3.3 Suffix structures

### SuffixArray.h

76bd5d, 220 lines

```
#define all(v) v.begin(),v.end()
class suffix_array {
    const static int alpha = 128;
    int getOrder(int a) const {
        return (a < (int) order.size() ? order[a] : 0);
    }
    void radix_sort(int k) {
        vector<int> frq(n), tmp(n);
        for (auto &it : suf)
            frq[getOrder(it + k)]++;
        for (int i = 1; i < n; i++)
            frq[i] += frq[i - 1];
        for (int i = n - 1; i >= 0; i--)
            tmp[--frq[getOrder(suf[i] + k)]] = suf[i];
        suf = tmp;
    }
public:
    int n;
    string s;
    vector<int> suf, lcp, order; // order store position of
          suffix i in suf array
    suffix_array(const string &s) :
            n(s.size() + 1), s(s) {
        suf = order = lcp = vector<int>(n);
        vector<int> bucket_idx(n), newOrder(n), newsuff(n);
        vector<int> prev(n), head(alpha, -1);
        for (int i = 0; i < n; i++) {
            prev[i] = head[s[i]];
            head[s[i]] = i;
        }
        int buc = -1, idx = 0;
        for (int i = 0; i < alpha; i++) {
            if (head[i] == -1)
                continue;
            bucket_idx[++buc] = idx;
            for (int j = head[i]; ~j; j = prev[j])
                suf[idx++] = j, order[j] = buc;
        }
        int len = 1;
        do {
            auto cmp = [&](int a, int b) {
                if (order[a] != order[b])
                    return order[a] < order[b];
                return getOrder(a + len) < getOrder(b + len);
            };
            for (int i = 0; i < n; i++) {
                int j = suf[i] - len;
                if (j < 0)
                    continue;
                newsuff[bucket_idx[order[j]]++] = j;
            }
            for (int i = 1; i < n; i++) {
                suf[i] = newsuff[i];
                bool cmpres = cmp(suf[i - 1], suf[i]);
                newOrder[suf[i]] = newOrder[suf[i - 1]] +
                      cmpres;
                if (cmpres)
                    bucket_idx[newOrder[suf[i]]] = i;
            }
            order = newOrder;
            len <<= 1;
        } while (order[suf[n - 1]] != n - 1);
        buildLCP();
    }
    /*
     * longest common prefix
     * O(n)
     * lcp[i] = lcp(suf[i],suf[i-1])
     */
    void buildLCP() {
        lcp = vector<int>(n);
        int k = 0;
        for (int i = 0; i < n - 1; i++) {
            int pos = order[i];
            int j = suf[pos - 1];
            while (s[i + k] == s[j + k])
                k++;
            lcp[pos] = k;
            if (k)
                k--;
        }
    }
    int LCP_by_order(int a, int b) {
        if (a > b)
            swap(a, b);
        int mn = n - suf[a] - 1;
        for (int k = a + 1; k <= b; k++)
            mn = min(mn, lcp[k]);
        return mn;
    }
    //LCP(i,j) : longest common prefix between suffix i and
          suffix j
    int LCP(int i, int j) {
```

```
        //return LCP_by_order(order[i],order[j]);
        if (order[j] < order[i])
            swap(i, j);
        int mn = n - i - 1;
        for (int k = order[i] + 1; k <= order[j]; k++)
            mn = min(mn, lcp[k]);
        return mn;
    }
    //compare s[a.first..a.second] with s[b.first..b.second]
    //-1:a<b ,0:a==b,1:a>b
    int compare_substrings(pair<int, int> a, pair<int, int> b)
     {
        int lcp = min(
                { LCP(a.first, b.first), a.second - a.first +
                     1, b.second

        a.first += lcp;
        b.first += lcp;
        if (a.first <= a.second) {
            if (b.first <= b.second) {
                if (s[a.first] == s[b.first])
                    return 0;
                return (s[a.first] < s[b.first] ? -1 : 1);
            }
            return 1;
        }
        return (b.first <= b.second ? -1 : 0);
    }
    pair<int, int> find_string(const string &x) {
        int st = 0, ed = n;
        for (int i = 0; i < sz(x) && st < ed; i++) {
            auto cmp = [&](int a, int b) {
                if (a == -1)
                    return x[i] < s[b + i];
                return s[a + i] < x[i];
            };
            st = lower_bound(suf.begin() + st, suf.begin() + ed
                 , -1, cmp)
                 - suf.begin();
            ed = upper_bound(suf.begin() + st, suf.begin() + ed
                 , -1, cmp)
                 - suf.begin();
        }
        return {st,ed-1};
    }
};

ll number_of_different_substrings(string s) {
    int n = s.size();
    suffix_array sa(s);
    ll cnt = 0;
    for (int i = 0; i <= n; i++)
        cnt += n - sa.suf[i] - sa.lcp[i];
    return cnt;
}
```

```
string longest_common_substring(const string &s1, const string
     &s2) {
    suffix_array sa(s1 + "#" + s2);
    vector<int> suf = sa.suf, lcp = sa.lcp;
    auto type = [&](int idx) {
        return idx <= s1.size();
    };
    int mx = 0, idx = 0;
    int len = s1.size() + 1 + s2.size();
    for (int i = 1; i <= len; i++)
        if (type(suf[i - 1]) != type(suf[i]) && lcp[i] > mx) {
            mx = lcp[i];
            idx = min(suf[i - 1], suf[i]);
        }
    return s1.substr(idx, mx);
}

int longest_common_substring(const vector<string> &v) {
    int n = v.size();
    int len = n - 1;
    for (auto &it : v)
        len += it.size();
    string s(len, '.');
    vector<int> type(len + 1, n), frq(n + 1);
    for (int i = 0, j = 0; i < v.size(); i++) {
        if (i)
            s[j] = 'z' + i;
        for (char ch : v[i]) {
            s[j] = ch;
            type[j] = i;
            j++;
        }
    }
    suffix_array sa(s);
    vector<int> suf = sa.suf, lcp = sa.lcp;
    monoqueue q;
    int st = 0, ed = 0, cnt = 0, mx = 0;
    while (st <= s.size()) {
        while (ed <= s.size() && cnt < v.size()) {
            q.push(lcp[ed], ed);
            if (++frq[type[suf[ed]]] == 1)
                cnt++;
            ed++;
        }
        q.pop(st);
        if (cnt == v.size())
            mx = max(mx, q.getMin()); //st+1,ed
        if (--frq[type[suf[st]]] == 0)
            cnt--;
        st++;
    }
    return mx;
}

string kth_substring(string s, int k) { //1-based,repated
    int n = s.size();
    suffix_array sa(s);
    vector<int> suf = sa.suf, lcp = sa.lcp;
    for (int i = 1; i <= n; i++) {
        int len = n - suf[i];
        int cnt = 0;
        for (int l = 1; l <= len; l++) {
            cnt++;
            int st = i + 1, ed = n, ans = i;
            while (st <= ed) {
                int md = st + ed >> 1;
                if (sa.LCP_by_order(i, md) >= l)
                    st = md + 1, ans = md;
                else
                    ed = md - 1;
            }
            cnt += ans - i;
            if (cnt >= k)
                return s.substr(suf[i], l);
        }
        k -= len;
    }
    assert(0);
}
```

## SuffixAutomaton.h

97eac6, 116 lines

```
struct suffix_automaton {
  struct state {
    int len, link = 0, cnt = 0;
    bool terminal = false, is_clone = false;
    map<char, int> next;
    state(int len = 0) :
        len(len) {
    }
    bool have_next(char ch) {
      return next.find(ch) != next.end();
    }
    void clone(const state &other, int nlen) {
      len = nlen;
      next = other.next;
      link = other.link;
      is_clone = true;
    }
  };
  vector<state> st;
  int last = 0;
  suffix_automaton() {
    st.push_back(state());
    st[0].link = -1;
  }
  suffix_automaton(const string &s) :
      suffix_automaton() {
    for (char ch : s)
      extend(ch);
    mark_terminals();
  }
  void extend(char c) {
    int cur = st.size();
    st.push_back(state(st[last].len + 1));
    st[cur].cnt = 1;
    int p = last;
    last = cur;
    while (p != -1 && !st[p].have_next(c)) {
      st[p].next[c] = cur;
      p = st[p].link;
    }
    if (p == -1)
      return;
    int q = st[p].next[c];
    if (st[p].len + 1 == st[q].len) {
      st[cur].link = q;
      return;
    }
    int clone = st.size();
    st.push_back(state());
    st[clone].clone(st[q], st[p].len + 1);
    while (p != -1 && st[p].next[c] == q) {
      st[p].next[c] = clone;
      p = st[p].link;
    }
    st[q].link = st[cur].link = clone;
  }
```

```cpp
  void mark_terminals() {
    for (int cur = last; cur > 0; cur = st[cur].link)
      st[cur].terminal = true;
  }
  void calc_number_of_occurrences() {
    vector<vector<int>> lvl(st[last].len + 1);
    for (int i = 1; i < st.size(); i++)
      lvl[st[i].len].push_back(i);
    for (int i = st[last].len; i >= 0; i--)
      for (auto cur : lvl[i])
        st[st[cur].link].cnt += st[cur].cnt;
  }
  vector<ll> dp;
  ll Count(int cur) { //count number of paths
    ll &rt = dp[cur];
    if (rt)
      return rt;
    rt = 1;
    for (auto ch : st[cur].next)
      rt += Count(ch.second);
    return rt;
  }
  //1-based, different substring, 0 = ""
  string kth_substring(ll k) {
    assert(k <= Count(0));
    string rt;
    int cur = 0;
    while (k > 0) {
      for (auto ch : st[cur].next) {
        if (Count(ch.second) < k)
          k -= Count(ch.second);
        else {
          rt += ch.first;
          cur = ch.second;
          k--;
          break;
        }
      }
    }
    return rt;
  }
  string longest_common_substring(const string &t) {
    int cur = 0, l = 0, mx = 0, idx = 0;
    for (int i = 0; i < t.size(); i++) {
      while (cur > 0 && !st[cur].have_next(t[i])) {
        cur = st[cur].link;
        l = st[cur].len;
      }
      if (st[cur].have_next(t[i])) {
        cur = st[cur].next[t[i]];
        l++;
      }
      if (l > mx) {
        mx = l;
        idx = i;
      }
    }
    return t.substr(idx - mx + 1, mx);
  }
};
```

## 3.4 Manacher Hash

Manacher.h — 9df19f, 32 lines

```cpp
void Manacher(string& s) {
  int n = sz(s);
  vector<int> d1(n);
  //d1[i] and d2[i], denoting the number of
  //palindromes accordingly with odd and even lengths with
  //centers in the position  i
  // with d2 the center of aa is pos 1
  // 0-based
  for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 1 : min(d1[l + r - i], r - i + 1);
    while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
      k++;
    }
    d1[i] = k--;
    if (i + k > r) {
      l = i - k;
      r = i + k;
    }
  }
  vector<int> d2(n);
  for (int i = 0, l = 0, r = -1; i < n; i++) {
    int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
    while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s[i +
         k]) {
      k++;
    }
    d2[i] = k--;
    if (i + k > r) {
      l = i - k - 1;
      r = i + k;
    }
  }
}
```

# Math (4)

## 4.1 Prime

Prime.h — 88bb5c, 69 lines

```cpp
//linear sieve
const int N = 1e7;
int lpf[N + 1];
vector<int> prime;
void sieve() {
  for (int i = 2; i <= N; i++) {
    if (lpf[i] == 0) {
      lpf[i] = i;
      prime.push_back(i);
    }
    for (int j : prime) {
      if (j > lpf[i] || 1LL * i * j > N)break;
      lpf[i * j] = j;
    }
  }
}

// return number of Divisors(n) using prime factorization
ll numOfDivisors(primeFactors mp) {
  ll cnt = 1;
  for (auto it : mp) cnt *= (it.second + 1);
  return cnt;
}
// return sum of Divisors(n) using prime factorization
ll sumOfDivisors(primeFactors mp) {
  ll sum = 1;
  for (auto it : mp) sum *= sumPower(it.first, it.second);
  return sum;
}

ll phi_function(ll n) {
  ll result = n;
  primeFactors pf = prime_factors(n);
  for (auto &it : pf) {
    ll p = it.first;
    result -= (result / p);
  }
  return result;
}
void phi_1_to_n(int n) {
  for (int i = 0; i <= n; i++)
    phi[i] = i;
  for (int i = 2; i <= n; i++)
    if (phi[i] == i)
      for (int j = i; j <= n; j += i)
        phi[j] -= phi[j] / i;
}

char mob[N];
bool prime[N];
void moebius() {
  memset(mob, 1, sizeof mob);
  memset(prime + 2, 1, sizeof(prime) - 2);
  mob[0] = 0;
  mob[2] = -1;
  for (int i = 4; i < N; i += 2) {
    mob[i] *= (i & 3) ? -1 : 0;
    prime[i] = 0;
  }
  for (int i = 3; i < N; i += 2)
    if (prime[i]) {
      mob[i] = -1;
      for (int j = 2 * i; j < N; j += i) {
        mob[j] *= j % (1LL * i * i) ? -1 : 0;
        prime[j] = 0;
      }
    }
}
```

MillerRabinPrimalityTest.h — 25c8aa, 47 lines

```cpp
const int ITER = 4;
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
     count());

__int128 power(__int128 x, __int128 y, ll mod) {
    if (y == 0)
        return 1;
    if (y & 1)return (x * power(x, y - 1, mod)) % mod;
    __int128 r = power(x, y >> 1, mod);
    return (r * r) % mod;
}

bool millerTest(ll n, ll d) {
    __int128 a = uniform_int_distribution<ll>(2, n - 2)(rng);
    a = power(a, d, n);
    if (a == 1 || a == n - 1)
        return true;
    d <<= 1;
    while (d != n - 1) {
        a = a * a % n;
        if (a == 1) return false;
        if (a == n - 1) return true;
        d <<= 1;
    }
    return false;
}

bool is_prime(ll n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (!(n & 1)) return false;
```

```
    ll d = n - 1;
    while (!(d & 1))
        d >>= 1;
    for (int i = 0; i < ITER; i++)
        if (!millerTest(n, d))
            return false;
    return true;
}

bool isPrimeSquare(ll n) {
    ll sq = sqrt(n);
    if (sq * sq < n) {
        sq++;
        if (sq * sq != n)return false;
    }
    return is_prime(sq);
}
```

## 4.2 ModInverse

### ModInverse.h

1fc53a, 48 lines

```
#define ll long long

ll power(ll x, ll y, int mod) {
  if (y == 0)
    return 1;
  if (y == 1)
    return x % mod;
  ll r = power(x, y >> 1, mod);
  return (((r * r) % mod) * power(x, y & 1, mod)) % mod;
}

// return a ^ 1 + a ^ 2 + a ^ 3 + .... a ^ k
ll sumPower(ll a, ll k, int mod) {
  if (k == 1) return a % mod;
  ll half = sumPower(a, k / 2, mod);
  ll p = half * power(a, k / 2, mod) % mod;
  p = (p + half) % mod;
  if (k & 1) p = (p + power(a, k, mod)) % mod;
  return p;
}

ll modInverse(ll b, ll mod) { // if mod is Prime
  return power(b, mod - 2, mod);
}

ll modInverse(ll b, ll mod) { // if mod is not Prime,gcd(a,b)
     must be equal 1
  return power(b, phi_function(mod) - 1, mod);
}

// (a^n)%p=result,return n
int getPower(int a, int result, int mod) {
  int sq = sqrt(mod);
  map<int, int> mp;
  ll r = 1;
  for (int i = 0; i < sq; i++) {
    if (mp.find(r) == mp.end())
      mp[r] = i;
    r = (r * a) % mod;
  }
  ll tmp = modInverse(r, mod);
  ll cur = result;
  for (int i = 0; i <= mod; i += sq) {
    if (mp.find(cur) != mp.end())
      return i + mp[cur];
    cur = (cur * tmp) % mod;//val/(a^sq)
  }
  return INF;
}
```

## 4.3 Number Theory

### ExtendedEuclidean.h

fb76b3, 84 lines

```
ll egcd(ll a, ll b, ll& x, ll& y) {
    if (a < 0) {
        auto g = egcd(-a, b, x, y);
        x *= -1;
        return g;
    }
    if (b < 0) {
        auto g = egcd(a, -b, x, y);
        y *= -1;
        return g;
    }

    if (!b) {
        x = 1;
        y = 0;
        return a;
    }
    ll x1, y1;
    ll g = egcd(b, a % b, x1, y1);
    x = y1, y = x1 - y1 * (a / b);
    return g;
}

//O(n * log(m)) Memory & Time; coefficients.size() <= n,
     coefficients[i] <= m
// 0-based implementation
template<typename T>
T extended_euclidean(const deque<T>& cof, deque<T>& var) {
    int n = cof.size();
    if (!cof.back()) {
        int cnt = 0, id = 0;
        for (int i = 0; i < n; i++)
            if (!cof[i]) {
                cnt++;
                var[i] = 0;
            } else id = i;
        if (cnt >= n - 1) {
            var[id] = 1;
            return cof[id];
        }
        deque<T> new_cof, new_var;
        for (int i = 0; i < n; i++)
            if (cof[i]) {
                new_cof.push_back(cof[i]);
                new_var.push_back(var[i]);
            }
        T g = extended_euclidean(new_cof, new_var);
        for (int i = 0; !new_var.empty(); i++)
            if (cof[i]) {
                var[i] = new_var.front();
                new_var.pop_front();
            }
        return g;
    }
    deque<T> new_cof = cof;
    for (int i = 0; i < n - 1; i++)
        new_cof[i] %= new_cof.back();
    new_cof.push_front(new_cof.back());
    new_cof.pop_back();
    var.push_front(var.back());
    var.pop_back();
    T g = extended_euclidean(new_cof, var);
    var.push_back(var.front());
    var.pop_front();
    for (int i = 0; i < n - 1; i++)
        var.back() -= cof[i] / cof.back() * var[i];
    return g;
}
template<typename T>
vector<T> find_any_solution(const vector<T>& cof, T rhs) {
    int n = cof.size();
    if (!n)
        return vector<T>();
    deque<T> deque_cof(cof.begin(), cof.end()), deque_var(n);
    T g = extended_euclidean(deque_cof, deque_var);
    if (g && rhs % g)
        return vector<T>();
    vector<T> var(deque_var.begin(), deque_var.end());
    if (g) {
        rhs /= g;
        for (auto& it : var)
            it *= rhs;
    }
    return var;
}
```

### Diophantine.h

70d1ad, 96 lines

```
// return false if there is no solution
// return true if there exist a solution
// x, y are the solutions and g is the gcd between a and b
bool Diophantine_Solution(ll a, ll b, ll c, ll& x, ll& y, ll& g
     ) {
    if (!a && !b) {
        if (c) return false;
        x = y = g = 0;
        return true;
    }
    g = egcd(a, b, x, y);
    if (c % g) return false;
    x *= c / g;
    y *= c / g;
    return true;
}
void shift_solution(ll& x, ll& y, ll a, ll b, ll cnt) {
    x += b * cnt;
    y -= a * cnt;
}

// find all number of solutions of ax + by = c
// x in range {minx, maxx}
// y in range {miny, maxy}
ll Diophantine_Solutions(ll a, ll b, ll c, ll minx, ll maxx, ll
      miny, ll maxy) {
    if (minx > maxx || miny > maxy) return 0;
    if (!a && !b && !c)
        return (maxx - minx + 1) * (maxy - miny + 1);
    if (!a && !b) return 0;
    if (!a) {
        if (c % b) return 0;
        ll num = c / b;
        return (num >= miny && num <= maxy) * (maxx - minx + 1)
            ;
    }
    if (!b) {
        if (c % a) return 0;
        ll num = c / a;
        return (num >= minx && num <= maxx) * (maxy - miny + 1)
            ;
    }
    ll x, y, g;
    if (!Diophantine_Solution(a, b, c, x, y, g))
        return 0;
```

```
    ll lx1, lx2, rx1, rx2;
    // a * x + b * y = c
    // (a / g) * x + (b / g) * y = c / g
    a /= g, b /= g, c /= g;
    g = 1;
    int sign_a = (a > 0 ? 1 : -1);
    int sign_b = (b > 0 ? 1 : -1);
    // x + k * b >= minx
    // k * b >= minx - x
    // k >= (minx - x) / b
    // k >= ceil((minx - x) / b)
    shift_solution(x, y, a, b, (minx - x) / b);
    // if x is less than minx so we need to increase it by one
        step only
    // from the upove equation x + k * b >= minx
    // if b is positive so choose k equal to 1 to increase x
        one
    // if b is negattive so choose k equal to -1 because -1 *
        -1 = 1 ans also increase x
    if (x < minx)
        shift_solution(x, y, a, b, sign_b);
    if (x > maxx) return 0;
    lx1 = x;

    // x + k * b <= maxx
    // k * b <= maxx - x
    // k <= (maxx - x) / b
    shift_solution(x, y, a, b, (maxx - x) / b);
    if (x > maxx)
        shift_solution(x, y, a, b, -sign_b);
    rx1 = x;
    // y - k * a >= miny
    // y - miny >= k * a
    // k * a <= y - miny
    // k <= (y - miny) / a
    shift_solution(x, y, a, b, (y - miny) / a);
    if (y < miny)
        shift_solution(x, y, a, b, -sign_a);
    if (y > maxy) return 0;
    lx2 = x;
    // y - k * a <= maxy
    // y - maxy <= k * a
    // k * a >= y - maxy
    // k >= (y - maxy) / a
    shift_solution(x, y, a, b, (y - maxy) / a);
    if (y > maxy)
        shift_solution(x, y, a, b, sign_a);
    rx2 = x;
    if (lx2 > rx2) swap(lx2, rx2);
    // becuase we calculate the equations lx2, rx2 from
        shifting y
    // not from shifting x directly
    ll lx = max(lx1, lx2);
    ll rx = min(rx1, rx2);
    if (lx > rx) return 0;
    return (rx - lx) / abs(b) + 1;
}
```

CRT.h

b07699, 15 lines

```
ll CRT(vector<ll>& a, vector<ll>& m){
  ll lcm = m[0], rem = a[0];
  int n = a.size();
  for(int i = 1; i < n; i++){
    ll x, y;
    ll gcd = extended_euclidean(lcm, m[i], x, y);
    if((a[i] - rem) % gcd) return -1;
    ll tmp = m[i] / gcd, f = (a[i] - rem) / gcd;
        x = ((x % tmp) * (f % tmp)) % tmp;
    rem += lcm * x;
    lcm = lcm * m[i] / gcd;
    rem = (rem % lcm + lcm) % lcm;
  }
  return rem;
}
```

## 4.4 Combinatorics

Combinatorics.h

4c657c, 77 lines

```
typedef unsigned long long ull;

// nCr = n!/((n-r)! * r!)
// nCr(n,r) = nCr(n,n-r)
// nPr = n!/(n-r)!
// nPr_circle = nPr/r
// nCr(n,r) = pascal[n][r]
// catalan[n] = nCr(2n,n)/(n+1)

ull nCr(int n, int r) {
  if (r > n)
    return 0;
  r = max(r, n - r);
  ull ans = 1, div = 1, i = r + 1;
  while (i <= n) {
    ans *= i++;
    ans /= div++;
  }
  return ans;
}

ull nPr(int n, int r) {
  if (r > n)
    return 0;
  ull p = 1, i = n - r + 1;
  while (i <= n)
    p *= i++;
  return p;
}

vector<vector<ull>> pascalTriangle(int n) {
  vector<vector<ull>> pascal(n + 1, vector<ull>(n + 1));
  for (int i = 0; i <= n; i++) {
    pascal[i][i] = pascal[i][0] = 1;
    for (int j = 1; j < n; j++)
      pascal[i][j] = pascal[i - 1][j] + pascal[i - 1][j - 1];
  }
  return pascal;
}

// return catalan number n-th using dp O(n^2)//max = 35 then
     overflow
vector<ull> catalanNumber(int n) {
  vector<ull> catalan(n + 1);
  catalan[0] = catalan[1] = 1;
  for (int i = 2; i <= n; i++) {
    ull &rt = catalan[i];
    for (int j = 0; j < n; j++)
      rt += catalan[j] * catalan[n - j - 1];
  }
  return catalan;
}

// count number of paths in matrix n*m
// go to right or down only
ull countNumberOfPaths(int n, int m) {
  return nCr(n + m - 2, n - 1);
}
```

```
const int N = 1e5 + 100;
const int MOD = 1e9 + 7;
ll fact[N];
ll inv[N]; //mod inverse for i
ll invfact[N]; //mod inverse for i!
void factInverse() {
  fact[0] = inv[1] = fact[1] = invfact[0] = invfact[1] = 1;
  for (long long i = 2; i < N; i++) {
    fact[i] = (fact[i - 1] * i) % MOD;
    inv[i] = MOD - (inv[MOD % i] * (MOD / i) % MOD);
    invfact[i] = (inv[i] * invfact[i - 1]) % MOD;
  }
}

ll nCr(int n, int r) {
  if (r > n)
    return 0;
  return (((fact[n] * invfact[r]) % MOD) * invfact[n - r]) %
       MOD;
}
```

## 4.5 Matrix

Matrix.h

c238a0, 104 lines

```
#define ll long long
#define sz(s) (int)s.size()
#define REP(i,n) for(int i = 0;i<n;i++)
struct matrix {
  using T = int;
  using row = vector<T>;
  vector<vector<T>> v;
  matrix() {

  }
  matrix(int n, int m, T val = 0) :
      v(n, row(m, val)) {
  }
  int size() const {
    return v.size();
  }
  int cols() const {
    return v[0].size();
  }
  matrix operator*(T a) const {
    matrix rt = *this;
    REP(i,rt.size())
      REP(j,rt.cols())
        rt.v[i][j] *= a;
    return rt;
  }
  friend matrix operator*(T a, const matrix &b) {
    return (b * a);
  }
  friend matrix operator+(const matrix &a, const matrix &b) {
    assert(a.size() == b.size() && a.cols() == b.cols());
    matrix rt(a.size(), a.cols());
    REP(i,rt.size())
      REP(j,rt.cols())
        rt.v[i][j] = a.v[i][j] + b.v[i][j];
    return rt;
  }
  friend matrix operator*(const matrix &a, const matrix &b) {
    assert(a.cols() == b.size());
    matrix rt(a.size(), b.cols());
    REP(i,rt.size())
      REP(k,a.cols())
      {
        if (a.v[i][k] == 0)
          continue;
```

```
        REP(j,rt.cols())
          rt.v[i][j] += a.v[i][k] * b.v[k][j];
      }
    return rt;
  }
};

matrix identity(int n) {
  matrix r(n, n);
  for (int i = 0; i < n; i++)
    r.v[i][i] = 1;
  return r;
}

matrix addIdentity(const matrix &a) {
  matrix rt = a;
  REP(i,a.size())
    rt.v[i][i]++;
  return rt;
}

matrix power(matrix a, long long y) {
  assert(y >= 0 && a.size() == a.cols());
  matrix rt = identity(a.size());
  while (y > 0) {
    if (y & 1)
      rt = rt * a;
    a = a * a;
    y >>= 1;
  }
  return rt;
}

matrix sumPower(const matrix &a, ll k) {
  if (k == 0)
    return matrix(sz(a), sz(a));
  if (k & 1)
    return a * addIdentity(sumPower(a, k - 1));
  return (sumPower(a, k >> 1) * addIdentity(power(a, k >> 1)));
}

/* return matrix contains
  a^k         0
a^1+a^2.. a^k    I
*/

matrix sumPowerV2(const matrix &a, ll k) {
  int n = sz(a);
  matrix rt(2 * n, 2 * n);
  REP(i,n)
    REP(j,n)
    {
      rt.v[i][j] = a.v[i][j];
      rt.v[i + n][j] = a.v[i][j];
    }
  for (int i = n; i < 2 * n; i++)
    rt.v[i][i] = 1;
  return power(rt, k);
}
```

## 4.6 FFT

### FFT.h

71d3fc, 96 lines

```
typedef valarray<complex<double>> polynomial;
const int LGN = 20;
vector<complex<double>> CM1[3][LGN + 1];
const double PI = acos(-1);
void prepare() {
    for (int sign = -1; sign <= 1; sign += 2) {
        for (int i = 0; i <= LGN; i++) {
            int N = 1 << i;
            double theta = sign * 2 * PI / N;
            complex<double> cm1 = 1;
            complex<double> cm2(cos(theta), sin(theta));
            for (int j = 0; j < N / 2; j++) {
                CM1[sign + 1][i].push_back(cm1);
                cm1 *= cm2;
            }
        }
    }
}
void fft(polynomial &a, int sign = -1) {
    int N = a.size();
    int lgn = log2(N);
    for (int m = N; m >= 2; m >>= 1, lgn--) {
        int mh = m >> 1;
        for (int i = 0; i < mh; i++) {
            const complex<double> &w = CM1[sign + 1][lgn][i];
            for (int j = i; j < N; j += m) {
                int k = j + mh;
                complex<double> x = a[j] - a[k];
                a[j] += a[k];
                a[k] = w * x;
            }
        }
    }
    int i = 0;
    for (int j = 1; j < N - 1; j++) {
        for (int k = N >> 1; k > (i ^= k); k >>= 1);
        if (j < i)
            swap(a[i], a[j]);
    }
}
valarray<ll> inv_fft(polynomial&& a) {
    complex<double> N = a.size();
    fft(a, 1);
    a /= N;
    valarray<ll> rt(a.size());
    for (int i = 0; i < a.size(); i++)
        rt[i] = round(a[i].real());
    return rt;
}
valarray<int> mul(const valarray<int> &a, const valarray<int> &
     b) {
  int adeg = (int) a.size() - 1, bdeg = (int) b.size() - 1;
  int N = 1;
  while (N <= adeg + bdeg)
    N <<= 1;
  polynomial A(N), B(N);
  for (int i = 0; i < a.size(); i++)
    A[i] = a[i];
  for (int i = 0; i < b.size(); i++)
    B[i] = b[i];
  fft(A);
  fft(B);
  polynomial m = A * B;
  inv_fft(m);
  return rt;
}
valarray<int> mul_with_mod(const vector<int> & a, const vector<
     int>& b, int MOD = 1e9 + 7) {
    int adeg = (int) a.size() - 1, bdeg = (int) b.size() - 1;
    int N = 1;
    while (N <= adeg + bdeg)
        N <<= 1;
    int C = sqrt(MOD);
    polynomial a1(N), a2(N);
    polynomial b1(N), b2(N);
    for (int i = 0; i < a.size(); ++i) {
        a1[i] = a[i] % C;
        a2[i] = a[i] / C;
    }
    for (int i = 0; i < b.size(); ++i) {
        b1[i] = b[i] % C;
        b2[i] = b[i] / C;
    }
    fft(a1), fft(a2);
    fft(b1), fft(b2);
    valarray<ll> m11 = inv_fft(a1 * b1) % MOD;
    valarray<ll> m12 = inv_fft(a1 * b2) % MOD;
    valarray<ll> m21 = inv_fft(a2 * b1) % MOD;
    valarray<ll> m22 = inv_fft(a2 * b2) % MOD;
    valarray<ll> res = m11 % MOD;
    res += C * (m12 + m21) % MOD;
    res += C * (C * m22 % MOD) % MOD;
    res %= MOD;
    valarray<int> rt(res.size());
    for (int i = 0; i < res.size(); i++)
        rt[i] = res[i];
    return rt;
}
```

### NTT.h

d7f953, 178 lines

```
const int LGN = 20;
struct modint {
#define CUR (*this)
  int val;
  modint(const long long& a = 0) {
    val = a % MOD;
    if (val < 0)
      val += MOD;
  }
  modint& operator+=(const modint& a) {
    if ((val += a.val) >= MOD)
      val -= MOD;
    return CUR;
  }
  modint operator+(const modint& a) const {
    modint c = CUR;
    c += a;
    return c;
  }
  modint& operator-=(const modint& a) {
    if ((val -= a.val) < 0)
      val += MOD;
    return CUR;
  }
  modint operator-(const modint& a) const {
    modint c = CUR;
    c -= a;
    return c;
  }
  modint operator*(const modint& a) const {
    return modint((1LL * this->val * a.val) % MOD);
  }
  modint& operator*=(const modint& a) {
    CUR = CUR * a;
    return CUR;
  }
  modint operator/(const modint& a) {
    return CUR * power(a, MOD - 2);
  }
  modint& operator/=(const modint& a) {
    CUR = CUR / a;
    return CUR;
  }
```

```
  static modint power(modint x, long long y) {
    modint res = 1;
    while (y > 0) {
      if (y & 1)
        res *= x;
      x *= x;
      y >>= 1;
    }
    return res;
  }
  friend ostream& operator<<(ostream& out, const modint& a) {
    out << a.val;
    return out;
  }
#undef CUR
};

typedef valarray<modint> polynomial;
vector<modint> CM1[2][LGN + 1];
bool validRoot(modint root) {
  modint rootinv = modint(1) / root;
  for (int invert = 0; invert <= 1; invert++) {
    for (int i = 1; i <= LGN; i++) {
      int N = 1 << i;
      assert((MOD - 1) % N == 0);
      int C = (MOD - 1) / N;
      modint cm2 = modint::power(invert ? root : rootinv, C);
      if (cm2.val <= 1)
        return false;
    }
  }
  return true;
}

void prepare() {
  modint root = 2;
  while (!validRoot(root))
    root += 1;
  modint rootinv = modint(1) / root;
  for (int invert = 0; invert <= 1; invert++) {
    for (int i = 0; i <= LGN; i++) {
      int N = 1 << i;
      int C = (MOD - 1) / N;
      modint cm2 = modint::power(invert ? root : rootinv, C);
      modint cm1 = 1;
      set<int> st;
      for (int j = 0; j < N / 2; j++) {
        CM1[invert][i].push_back(cm1);
        cm1 *= cm2;
      }
    }
  }
}

void fft(polynomial& a, bool invert = 0) {
  int N = a.size();
  int lgn = log2(N);

  for (int m = N; m >= 2; m >>= 1, lgn--) {
    int mh = m >> 1;
    for (int i = 0; i < mh; i++) {
      const modint& w = CM1[invert][lgn][i];
      for (int j = i; j < N; j += m) {
        int k = j + mh;
        modint x = a[j] - a[k];
        a[j] += a[k];
        a[k] = w * x;
      }
    }
  }
  int i = 0;
  for (int j = 1; j < N - 1; j++) {
    for (int k = N >> 1; k > (i ^= k); k >>= 1)
      ;
    if (j < i)
      swap(a[i], a[j]);
  }
}

void inv_fft(polynomial& a) {
  int N = a.size();
  fft(a, 1);
  a /= N;
}

valarray<modint> mul(const polynomial& a, const polynomial& b)
    {
  int adeg = (int)a.size() - 1, bdeg = (int)b.size() - 1;
  int N = 1;
  while (N <= adeg + bdeg)
    N <<= 1;
  polynomial A(N), B(N);
  for (int i = 0; i < a.size(); i++)
    A[i] = a[i];
  for (int i = 0; i < b.size(); i++)
    B[i] = b[i];
  fft(A);
  fft(B);
  polynomial rt = A * B;
  inv_fft(rt);
  return rt;
}

int main() {
  run();
  prepare();
  int t;
  cin >> t;
  while (t--) {
    int n;
    cin >> n;
    polynomial f1(n + 1), f2(n + 1);
    for (int i = 0; i < n; i++) {
      int x, f;
      cin >> x >> f;
      f1[x] += f;
      f2[n - x] += f;
    }
    polynomial res = mul(f1, f2);
    vector<modint> out(res.size());
    out[0] = res[0];
    for (int i = 1; i < sz(res); i++)
      out[i] = res[i] + out[i - 1];
    modint inv2 = modint(1) / 2;
    int q;
    cin >> q;
    while (q--) {
      int l, r; cin >> l >> r;
      modint res = out[n + r] - out[n + l - 1];
      res += out[n - 1];
      if (n - r - 1 >= 0)res -= out[n - r - 1];
      cout << res * inv2 << endl;
    }
  }
}
```

# Geometry (5)

## 5.1 Basic Geometry

### Point.h

<bits/stdc++.h> 6653eb, 46 lines

```
using namespace std;
#define ll long long
typedef complex<double> point;  // it can be long long not
    double
template<class T>
istream& operator>>(istream& is, complex<T>& p) {
  T value;
  is >> value;
  p.real(value);
  is >> value;
  p.imag(value);
  return is;
}
#define PI acos(-1.0)
#define EPS 1e-8
#define X real()
#define Y imag()
#define angle(a)  (atan2((a).imag(), (a).real())) // angle with
    orignial
#define length(a)   (hypot((a).imag(), (a).real()))
#define vec(a,b)  ((b)-(a)) // return diff x1-x2 , y1-y2
#define dp(a,b)   ( (conj(a)*(b)).real() )
// a*b cos(T), if zero -> prep dot product A.B
#define cp(a,b)   ( (conj(a)*(b)).imag() )
// a*b sin(T), if zero -> parllel cross product = area of
    parllelogram
#define normalize(a)    (a)/length(a)
// norm(a)  // return x^2 + y^2 //a is point //can use dp(a,a)

bool same(point p1, point p2) {// check to points same or not
  return dp(vec(p1, p2), vec(p1, p2)) < EPS;
}

point rotate(point p, double angle, point around = point(0, 0))
    {
  p -= around;
  return (p * exp(point(0, angle))) + around;
}

// Refelect v around m
point reflect0(point v, point m) {
  return conj(v / m) * m;
}

// Refelect point p around l1-l2
point reflect(point p, point l1, point l2) {
  point z = p - l1, w = l2 - l1;
  return conj(z / w) * w + l1;
}
```

### Triangles.h

"point.h" 4cbaba, 71 lines

```
// sin(A)/a = sin(B)/b = sin(C)/c
// a^2 = b^2 + c^2 - 2b*c*cos(A
// sin(A+B) = sin(A) * cos(B) + sin(B) * cos(A)
// sin(A-B) = sin(A) * cos(B) - sin(B) * cos(A)
// cos(A+B) = cos(A) * cos(B) - sin(A) * sin(B)
// cos(A-B) = cos(A) * cos(B) + sin(A) * sin(B)
// tan(A+B) = (tan(A) + tan(B))/(1 - tan(A) * tan(B))
// tan(A-B) = (tan(A) - tan(B))/(1 - tan(A) * tan(B))

double fixAngle(double A) {
```

```
  return A > 1 ? 1 : (A < -1 ? -1 : A);
}
// return min angle: aOb / bOa
// dp(v1, v2) = |v1|*|v2|*cos(theta)
double angleO(point a, point O, point b) {
  point v1(a - O), v2(b - O);
  return acos(fixAngle(dp(v1, v2) / dist(v1) / dist(v2)));
}

double getSide_a_bAB(double b, double A, double B) {
  return (sin(A) * b) / sin(B);
}

double getAngle_A_abB(double a, double b, double B) {
  return asin(fixAngle((a * sin(B)) / b));
}

// wr answer in team formation :D
double getAngle_A_abc(double a, double b, double c) {
  return acos(fixAngle((b * b + c * c - a * a) / (2 * b * c)));
}

double triangleArea(double a, double b, double c) {
  double s = (a + b + c) / 2.0;
  return sqrt((s - a) * (s - b) * (s - c) * s);
}

double triangleArea(point p0, point p1, point p2) {
  double a = length(vec(p1, p0)), b = length(vec(p2, p0)),
    c = length(vec(p2, p1));
  return triangleArea(a, b, c);
}

double triangleArea3points(const point &a, const point &b,
    const point &c) {
  return fabs(cross(a,b) + cross(b,c) + cross(c,a)) / 2;
}

bool pointInTriangle(point a, point b, point c, point pt) {
  ll s1 = fabs(cp(vec(a,b), vec(a,c)));
  ll s2 = fabs(cp(vec(pt,a), vec(pt,b)))
    + fabs(cp(vec(pt, b), vec(pt, c)))
    + fabs(cp(vec(pt, a), vec(pt, c)));
  return s1 == s2;
}

// largest circle inside a triangle
//A triangle with area A and semi-perimeter s has an inscribed
    circle (incircle) with
//radius r = A/s
bool circleInTriangle(point a, point b, point c, point& ctr,
    double& r) {
  double ab = length(a - b), bc = length(b - c),
    ca = length(c - a);
  double s = 0.5 * (ab + bc + ca);
  r = triangleArea(ab, bc, ca) / s;

  if (fabs(r) < EPS) return 0; // no inCircle center
  double ratio = length(a - b) / length(a - c);
  point p1 = b + (vec(b, c) * (ratio / (1 + ratio)));
  ratio = length(b - a) / length(b - c);
  point p2 = a + (vec(a, c) * (ratio / (1 + ratio)));
  return intersectSegments(a, p1, b, p2, ctr); // get their
      intersection point
}
```

Lines.h

"point.h" a6d9ad, 106 lines

```
//o = anlge(a) - angle(b)
  //if o = 0 || o = 180 isCollinear
  //else not
bool isCollinear(point a, point b, point c) {
  return fabs(cp(vec(a, b), vec(a, c))) < EPS;
}
//o = anlge(a) - angle(b)
//if o = 0 isPointOnRay a->b
//else not
bool isPointOnRay(point a, point b, point c) {
  if (!isCollinear(a, b, c))
    return false;
  return dcmp(dp(vec(a, b), vec(a, c)), 0) == 1;
}

bool isPointOnRay(point a, point b, point c) {
  if (length(vec(a, c)) < EPS) return true;
  return same(normalize(vec(a, b)), normalize(vec(a, c)));
}

bool isPointOnSegment(point a, point b, point c) {
  return isPointOnRay(a, b, c) && isPointOnRay(b, a, c);
}

bool isPointOnSegment(point a, point b, point c) {
  double acb = length(vec(b, a)), ac = length(vec(c, a)), cb =
      length(vec(c, b));
  return dcmp(acb - (ac + cb), 0) == 0;
}

// dist point p2 to line p0-p1
double distToLine(point p0, point p1, point p2) {
  return fabs(cp(vec(p0, p1), vec(p0, p2)) / length(vec(p1, p0)
      )); // area = 0.5*b*h
}

//minimum distance from point p2 to segment p0-p1
double distToSegment(point p0, point p1, point p2) {
  double d1, d2;
  point v1 = p1 - p0, v2 = p2 - p0;
  if ((d1 = dp(v1, v2)) <= 0) return length(vec(p0, p2));
  if ((d2 = dp(v1, v1)) <= d1) return length(vec(p1, p2));
  double t = d1 / d2;
  return length(vec((p0 + v1 * t), p2));
}

// minimum point in segment po-p1 to point p2
point pointToSegment(point p0, point p1, point p2) {
  double d1, d2;
  point v1 = p1 - p0, v2 = p2 - p0;
  if ((d1 = dp(v1, v2)) <= 0) return p0;
  if ((d2 = dp(v1, v1)) <= d1) return p1;
  double t = d1 / d2;
  return (p0 + v1 * t);
}

// return point intersect in line a-b with c-d using parametric
    equations
bool intersectSegments(point a, point b, point c, point d,
    point& intersect) {
  double d1 = cp(vec(b, a), vec(c, d)),
    d2 = cp(vec(c, a), vec(c, d)),
    d3 = cp(vec(b, a), vec(c, a));
  if (fabs(d1) < EPS)
    return false; // Parllel || identical

  double t1 = d2 / d1, t2 = d3 / d1;
  intersect = a + (b - a) * t1;

  if (t1 < -EPS || t2 < -EPS || t2 > 1 + EPS)
    return false; //e.g ab is ray, cd is segment ... change to
        whatever
  return true;
}

// return 1 if point c is counter-clockwise about segment a-b
// -1 if point c is clockwise about segment a-b
// 0 if c is isCollinear about a-b
int ccw(point a, point b, point c) {
  point v1(b - a), v2(c - a);
  double t = cp(v1, v2);

  if (t > EPS)
    return 1;
  if (t < -EPS)
    return -1;
  if (v1.X * v2.X < -EPS || v1.Y * v2.Y < -EPS)
    return -1;
  if (norm(v1) < norm(v2) - EPS)
    return 1;
  return 0;
}

bool intersect(point p1, point p2, point p3, point p4) {
  // special case handling if a segment is just a point
  bool x = (p1 == p2), y = (p3 == p4);
  if (x && y)  return p1 == p3;
  if (x)   return ccw(p3, p4, p1) == 0;
  if (y)   return ccw(p1, p2, p3) == 0;
  return  ccw(p1, p2, p3) * ccw(p1, p2, p4) <= 0 &&
    ccw(p3, p4, p1) * ccw(p3, p4, p2) <= 0;
}

bool lineInsideRectangle(double x1, double x2, double y1,
    double y2, point st, point ed) {
  if (x2 < x1) swap(x1, x2);
  if (y2 < y1) swap(y1, y2);
  double mnX = min(st.X, ed.X), mxX = max(st.X, ed.X),
    mnY = min(st.Y, ed.Y), mxY = (st.Y, ed.Y);
  return dcmp(x1, mnX) <= 0 && dcmp(x2, mxX) >= 0
  && dcmp(y1, mnY) <= 0 && dcmp(y2, mxY) >= 0;
}
```

## 5.2 Circles

Circles.h

"point.h", "triangles.h" //getAngle_A_abc, "lines.h" //intersectSegments 6e1e8d, 72 lines

```
// 2 points has infinite circles
// Find circle passes with 3 points, some times, there is no
    circle! (in case colinear)
// Draw two perpendicular lines and intersect them
pair<double, point> findCircle(point a, point b, point c) {
  //create median, vector, its prependicular
  point m1 = (b + a) * 0.5, v1 = b - a, pv1 = point(v1.Y, -v1.X
      );
  point m2 = (b + c) * 0.5, v2 = b - c, pv2 = point(v2.Y, -v2.X
      );
  point end1 = m1 + pv1, end2 = m2 + pv2, center;
  intersectSegments(m1, end1, m2, end2, center);
  return make_pair(length(vec(center, a)), center);
}

// If line intersect cirlce at point p, and p = p0 + t(p1-p0)
// Then (p-c)(p-c) = r^2 substitute p and rearrange
// (p1-p0)(p1-p0)t^2 + 2(p1-p0)(p0-C)t + (p0-C)(p0-C) = r*r; ->
      Quadratic
```

```
vector<point> intersectLineCircle(point p0, point p1, point C,
    double r) {
  double a = dp(vec(p0, p1), vec(p0, p1)),
    b = 2 * dp(vec(p0, p1), vec(C, p0)),
    c = dp(vec(C, p0), vec(C, p0)) - r * r;
  double f = b * b - 4 * a * c;
  vector<point> v;
  if (dcmp(f, 0) >= 0) {
    if (dcmp(f, 0) == 0)  f = 0;
    double t1 = (-b + sqrt(f)) / (2 * a);
    double t2 = (-b - sqrt(f)) / (2 * a);
    v.push_back(p0 + t1 * (p1 - p0));
    if (dcmp(f, 0) != 0)
      v.push_back(p0 + t2 * (p1 - p0));
  }
  return v;
}

vector<point> intersectCircleCircle(point c1, double r1, point
    c2, double r2) {
  // Handle infinity case first: same center/radius and r > 0
  if (same(c1, c2) && dcmp(r1, r2) == 0 && dcmp(r1, 0) > 0)
    return vector<point>(3, c1);    // infinity 2 same circles
        (not points)

    // Compute 2 intersection case and handle 0, 1, 2 cases
  double ang1 = angle(vec(c1, c2)),
     ang2 = getAngle_A_abc(r2, r1, length(vec(c1, c2)));

  if (::isnan(ang2)) // if r1 or d = 0 => nan in getAngle_A_abc
        (/0)
    ang2 = 0; // fix corruption

  vector<point> v(1, polar(r1, ang1 + ang2) + c1);

  // if point NOT on the 2 circles = no intersection
  if (dcmp(dp(vec(c1, v[0]), vec(c1, v[0])), r1 * r1) != 0 ||
    dcmp(dp(vec(c2, v[0]), vec(c2, v[0])), r2 * r2) != 0)
    return vector<point>();

  v.push_back(polar(r1, ang1 - ang2) + c1);
  if (same(v[0], v[1]))  // if same, then 1 intersection only
    v.pop_back();
  return v;
}

ld circleCircleIntersectionArea(point cen1, ld r1, point cen2,
    ld r2) {
  ld dis = hypot(cen1.X - cen2.X, cen1.Y - cen2.Y);
  if (dis > r1 + r2)return 0;
  if (dis <= fabs(r2 - r1) && r1 >= r2)
    return PI * r2 * r2;
  if (dis <= fabs(r2 - r1) && r1 < r2)
    return PI * r1 * r1;
  ld a = r1 * r1, b = r2 * r2;
  ld ang1 = acos((a + dis * dis - b) / (2 * r1 * dis)) * 2;
  ld ang2 = acos((b + dis * dis - a) / (2 * r2 * dis)) * 2;
  ld ret1 = .5 * b * (ang2 - sin(ang2));
  ld ret2 = .5 * a * (ang1 - sin(ang1));
  return ret1 + ret2;
}
```

## MinimumEnclosingCircle.h

f9af54, 71 lines

```
//init p array with the points and ps with the number of points
//cen and rad are result circle
//you must call random_shuffle(p,p+ps); before you call mec
typedef complex<double> point;
#define perp(a)   (point(-(a).Y, (a).X))
#define vec(a,b)  ((b) - (a))
#define mid(a,b)  (((a) + (b)) / point(2, 0))

enum STATE {
    IN, OUT, BOUNDRY
};

STATE circlePoint(const point &cen, const double &r, const
     point &p) {
    double lensqr = lengthSqr(vec(cen,p));
    if (fabs(lensqr - r * r) < EPS)
        return BOUNDRY;
    if (lensqr < r * r)
        return IN;
    return OUT;
}

void circle2(const point &p1, const point &p2, point &cen,
     double &r) {
    cen = mid(p1, p2);
    r = length(vec(p1, p2)) / 2;
}

bool circle3(const point &p1, const point &p2, const point &p3,
                point& cen, double& r) {
    point m1 = mid(p1, p2);
    point m2 = mid(p2, p3);
    point perp1 = perp(vec(p1, p2));
    point perp2 = perp(vec(p2, p3));
    bool res = intersect(m1, m1 + perp1, m2, m2 + perp2, cen);
    r = length(vec(cen,p1));
    return res;
}

#define MAXPOINTS 100000
point p[MAXPOINTS], r[3], cen;
int ps, rs;
double rad;
//init p array with the points and ps with the number of points
//cen and rad are result circle
//you must call random_shuffle(p,p+ps); before you call mec
void mec() {
    if (rs == 3) {
        circle3(r[0], r[1], r[2], cen, rad);
        return;
    }
    if (rs == 2 && ps == 0) {
        circle2(r[0], r[1], cen, rad);
        return;
    }
    if (!ps) {
        cen = r[0];
        rad = 0;
        return;
    }
    ps--;
    mec();
    if (circlePoint(cen, rad, p[ps]) == OUT) {
        r[rs++] = p[ps];
        mec();
        rs--;
    }
    ps++;
}
```

## 5.3 Polygons

### Polygon.h

"Lines.h" //isPointOnSegment, CCW — ed3be7, 118 lines

```
struct cmp {
  point about;
  cmp(point c) {
    about = c;
  }
  bool operator()(const point &p, const point &q) const {
    double cr = cp(vec(about, p), vec(about, q));
    if (fabs(cr) < EPS)
      return make_pair(p.Y, p.X) < make_pair(q.Y, q.X);
    return cr < 0;
  }
};
void sortAntiClockWise(vector<point> &pnts) {
  point mn = pnts[0];
  for (int i = 0; i < sz(pnts); i++)
    if (make_pair(pnts[i].Y, pnts[i].X) < make_pair(mn.Y, mn.X)
        )
      mn = pnts[i];
  sort(all(pnts), cmp(mn));
}

// CCW function must return 0 if the 3 points are collinear
bool isConvex(vector<point>& v) {
  int n = v.size(), m = n, sum = 0;
  v.push_back(v[0]);
  v.push_back(v[1]);
  char tmp;
  for (int i = 0; i < n; i++) {
    tmp = ccw(v[i], v[i + 1], v[i + 2]);
    if (tmp) sum += tmp;
    else m--;
  }
  v.pop_back();
  v.pop_back();
  return abs(sum) == m;
}

//Area(p) = interal_points + (boundry_point/2) - 1
//2*interal_points = 2*Area(p) - 2*(boundry_point/2) + 2
ll picksTheorm(vector<point> &p) { //point = complex<int>
  ll area = 0;
  ll bound = 0;
  for (int i = 0; i < sz(p); i++) {
    int j = (i + 1 < sz(p) ? i + 1 : 0);
    area += cp(p[i], p[j]);
    point v = vec(p[i], p[j]);
    bound += abs(__gcd(v.X, v.Y));
  }
  return (abs(area) - 2 * (bound / 2) + 2) / 2;
}

bool pointInPolygon(const vector<point> &p, point p0) {
  int wn = 0;  // the winding number counter

  for (int i = 0; i < sz(p); i++) {
    point cur = p[i], nxt = p[(i + 1) % sz(p)];
    if (isPointOnSegment(cur, nxt, p0))
      return true;
    if (cur.Y <= p0.Y) {    // Upward edge
      if (nxt.Y > p0.Y && cp(nxt - cur, p0 - cur) > EPS)
        ++wn;
    } else {                // Downward edge
      if (nxt.Y <= p0.Y && cp(nxt - cur, p0 - cur) < -EPS)
        --wn;
    }
```

```
  }
  return wn != 0;
}

//to check if the points are sorted anti-clockwise or clockwise
//remove the fabs at the end and it will return -ve value if
    clockwise
double polygonArea(const vector<point> &p) {
  double res = 0;
  for (int i = 0; i < sz(p); i++) {
      int j = (i + 1) % sz(p);
      res += cp(p[i],p[j]);
  }
  return fabs(res) / 2;
}

// return the centroid point of the polygon
// The centroid is also known as the "centre of gravity" or the
    "center of mass". The position of the centroid
// assuming the polygon to be made of a material of uniform
    density.
point polygonCentroid(vector<point> &polygon) {
    point res(0, 0);
    double a = 0;
    for (int i = 0; i < (int) polygon.size(); i++) {
        int j = (i + 1) % polygon.size();
        res = res + (polygon[i] + polygon[j]) * cp(polygon[i],
            polygon[j]);
        a += cp(polygon[i], polygon[j]);
    }
    return res / 3.0 / a;
}

// P need to be counterclockwise convex polygon
pair<vector<point>,vector<point>> polygonCut(vector<point> &p,
    point A, point B) {

    vector<point> left, right;
    point intersect;

    for (int i = 0; i < sz(p); ++i) {
        point cur = p[i], nxt = p[(i + 1) % sz(p)];

    bool in1 = cp(B-A, cur-A) >= 0;
    bool in2 = cp(B-A, nxt-A) >= 0;

        if (in1) right.push_back(cur);

        //NOTE adust intersectSegments should handled AB as
            line
        if (intersectSegments(A, B, cur, nxt, intersect)) {
            right.push_back(intersect);
            left.push_back(intersect);
        }

        if (in2)left.push_back(cur);
    }
    return make_pair(left, right);
}
```

## ConvexHull.h

718c97, 48 lines

```
bool cmp(point a, point b) {
  return a.X < b.X || (a.X == b.X && a.Y < b.Y);
}

ll cross(point a, point b, point c) {
  return cp(vec(a,b), vec(a,c));
}

bool cw(point a, point b, point c) {
  return cp(vec(a,b), vec(b,c)) < 0;
}

bool ccw(point a, point b, point c) {
  return cp(vec(a,b), vec(b,c)) > 0;
}

//with collinear points, to remove collinears check if cross ==
    0 when pop
vector<point> convex_hull(vector<point> &p) {
  if (p.size() == 1)
    return p;

  sort(p.begin(), p.end(), &cmp);
  point p1 = p[0], p2 = p.back();
  vector<point> up, down;
  up.push_back(p1);
  down.push_back(p1);
  for (int i = 1; i < (int) p.size(); i++) {
    if (i == p.size() - 1 || cw(p1, p[i], p2)) {
      while (up.size() >= 2
          && !cw(up[up.size() - 2], up[up.size() - 1], p[i]))
        up.pop_back();
      up.push_back(p[i]);
    }
    if (i == p.size() - 1 || ccw(p1, p[i], p2)) {
      while (down.size() >= 2
          && !ccw(down[down.size() - 2], down[down.size() - 1],
              p[i]))
        down.pop_back();
      down.push_back(p[i]);
    }
  }

  vector<point> convex;
  for (int i = 0; i < (int) down.size(); i++)
    convex.push_back(down[i]);
  for (int i = up.size() - 2; i > 0; i--)
    convex.push_back(up[i]);
  return convex;
}
```

## PointInPolygon.h

2ecc17, 38 lines

```
ll cross(point a, point b, point c) {
  return cp(vec(a,b), vec(a,c));
}

void prepare(vector<point> &polygon) {
  int pos = 0;
  for (int i = 0; i < sz(polygon); i++) {
    if (make_pair(polygon[i].X, polygon[i].Y)
        < make_pair(polygon[pos].X, polygon[pos].Y))
      pos = i;
  }
  rotate(polygon.begin(), polygon.begin() + pos, polygon.end())
      ;
}

bool isPointOnSegment(point a, point b, point c) {
  double acb = length(a - b), ac = length(a - c), cb = length(b
      - c);
  return dcmp(acb - (ac + cb), 0) == 0;
}

bool pointInConvexPolygon(const vector<point> &polygon, point
    pt) {
  if (isPointOnSegment(polygon[0], polygon.back(), pt))
    return true;
  if (cross(polygon[0], polygon.back(), pt) > 0)
    return false;
  if (cross(polygon[0], polygon[1], pt) < 0)
    return false;
  if (polygon.size() == 2)
    return false;
  int st = 2, ed = sz(polygon) - 2, ans = 1;
  while (st <= ed) {
    int md = st + ed >> 1;
    if (cross(polygon[0], polygon[md], pt) >= 0)
      st = md + 1, ans = md;
    else
      ed = md - 1;
  }
  return cross(polygon[ans], polygon[ans + 1], pt) >= 0;
}
```

# Misc (6)

## LIS.h

9beb7b, 47 lines

```
//without build
//make upper_bound if can take equal elements
int LIS(const vector<int> &v) {
  vector<int> lis(v.size());//put value less than zero if
      needed
  int l = 0;
  for (int i = 0; i < sz(v); i++) {
    int idx = lower_bound(lis.begin(), lis.begin() + l, v[i]) -
        lis.begin();
    if (idx == l) l++;
    lis[idx] = v[i];
  }
  return l;
}

void LIS_binarySearch(vector<int> v) {
  int n = v.size();
  vector<int> last(n), prev(n, -1);
  int length = 0;
  auto BS = [&](int val) {
    int st = 1, ed = length, md, rt = length;
    while (st <= ed) {
      md = st + ed >> 1;
      if (v[last[md]] >= val)
        ed = md - 1, rt = md;
      else
        st = md + 1;
    }
    return rt;
  };
  for (int i = 1; i < n; i++) {
    if (v[i] < v[last[0]])
      last[0] = i;
    else if (v[i] > v[last[length]]) {
      prev[i] = last[length];
      last[++length] = i;
    } else {
      int index = BS(v[i]);
      prev[i] = last[index - 1];
      last[index] = i;
    }
  }
  cout << length + 1 << "\n";
  vector<int> out;
  for (int i = last[length]; i >= 0; i = prev[i])
```

```
    out.push_back(v[i]);
  reverse(out.begin(), out.end());
  for (auto it : out)cout << it << endl;
}
```

Random.h 522d3a, 9 lines

```
#include <chrono> // keep-include
#include <random> // keep-include
//write this line once in top
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().
    count() * ((uint64_t) new char | 1));
// use this instead of rand()
template<typename T>
T Rand(T low, T high) {
  return uniform_int_distribution<T>(low, high)(rng);
}
```

CustomHash.h 6d7dae, 17 lines

```
struct custom_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }
  size_t operator()(pair<uint64_t, uint64_t> x) const { // for
      pair
    static const uint64_t FIXED_RANDOM = chrono::steady_clock::
        now().time_since_epoch().count();
    return splitmix64(x.first + FIXED_RANDOM) ^ (splitmix64(x.
        second + FIXED_RANDOM) >> 1);
  }

  size_t operator()(uint64_t x) const { // for single element
    static const uint64_t FIXED_RANDOM = chrono::steady_clock::
        now().time_since_epoch().count();
    return splitmix64(x + FIXED_RANDOM);
  }
};
```

# Mathematics Notes (7)

## 7.1 Sums

- To get the summation of $a^n$ sequence from $x$ to $y$, transform $a^n$ form to $a^{\{n\}}$ form (factorial polynomial $= a \times (a-1) \times (a-2) \times \ldots \times (a-n+1)$), then integrate the equation with limits from $x$ to $(y+1)$
- To get the summation law from known values substitute and compute the equation:

$$S(0,n) + x[n+1] = x[0] + S(1,n+1)$$

, then take a factor from $S(1,n+1)$ to transform it to $S(0,n)$, so you can substitute and compute $S(0,n)$ easily. You can use replacement in the original formula and the bounds of the summation (like replacing every $(k-j)$ by $j$). You can also multiply the summand by $k$ - it might help. For example:

$$S(0,n) = 2^0 + 2^1 + \ldots + 2^n$$

$$S(0,n) + 2^{n+1} = 2^0 + (2^1 + 2^2 + \ldots + 2^{n+1})$$

$$S(0,n) + 2^{n+1} = 1 + 2 \times S(0,n)$$

$$S(0,n) = 2^{n+1} - 1$$

- $$\binom{n}{0}^2 + \binom{n}{1}^2 + \binom{n}{2}^2 + \ldots + \binom{n}{n}^2 = \binom{2n}{n}$$

## 7.2 Congruence

- IF $ax = by \mod n$, THEN $x = y \mod n$ IFF $gcd(a,n) = 1$ (Division Rule)
- IF $a = b \mod m$, THEN $a^n = b^n \mod m$ (Powers Rule)
- $x^n = x^{n\%phi(m)} \mod m$ IFF $n \geq \log_2 m$... otherwise calculate without a modulo (Euler Totient)
- ALL ${}^nC_r$ for $n =$ prime, $r \neq 0$, $r \neq n$ is divisible by $n$ (${}^nC_r$ Rule)
- Distribution is ok IFF $p$ is prime: $(x+y)^p = x^p + y^p \mod p$ (Powers Distribution Rule)
- IF $a = b \mod m$, THEN $a^{m^k} = b^{m^k} \mod m^{k+1}; k = 0, 1, 2, 3, \ldots$

$$\phi(n!) = (n - \text{isPrime}(n)) \times \phi((n-1)!)$$

$$\sum_{d|n} \phi(d) = n$$

$$\sum_{i=1}^{n}\sum_{j=1}^{n} gcd(i,j) = \sum_{d=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n} \phi(d)\ [d|i]\ [d|j]$$

$$= \sum_{d=1}^{n} \phi(d) \sum_{i=1}^{\lfloor \frac{n}{d} \rfloor}\sum_{j=1}^{\lfloor \frac{n}{d} \rfloor} 1 = \sum_{d=1}^{n} \phi(d)\ \lfloor \frac{n}{d} \rfloor^2$$

## 7.3 $E(x^2)$

- To get the expected value of $x^2$, you need to represent the random variable as a **length** variable → get the product of probabilities of every pair → Convert every loop to an equation.
- Some problems require going through all pairs using *DP* (if possible) → you can compute the result for every individual variable separately like the problem of bears and humans.

# KACTL Maths Notes (8)

## 8.1 Equations

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The extremum is given by $x = -b/2a$.

$$\begin{matrix} ax + by = e \\ cx + dy = f \end{matrix} \Rightarrow \begin{matrix} x = \dfrac{ed - bf}{ad - bc} \\ y = \dfrac{af - ec}{ad - bc} \end{matrix}$$

In general, given an equation $Ax = b$, the solution to a variable $x_i$ is given by

$$x_i = \frac{\det A_i'}{\det A}$$

where $A_i'$ is $A$ with the $i$'th column replaced by $b$.

## 8.2 Recurrences

If $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$, and $r_1, \ldots, r_k$ are distinct roots of $x^k - c_1 x^{k-1} - \cdots - c_k$, there are $d_1, \ldots, d_k$ s.t.

$$a_n = d_1 r_1^n + \cdots + d_k r_k^n.$$

Non-distinct roots $r$ become polynomial factors, e.g. $a_n = (d_1 n + d_2) r^n$.

## 8.3 Trigonometry

$$\sin(v+w) = \sin v \cos w + \cos v \sin w$$
$$\cos(v+w) = \cos v \cos w - \sin v \sin w$$

$$\tan(v+w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$$
$$\sin v + \sin w = 2 \sin \frac{v+w}{2} \cos \frac{v-w}{2}$$
$$\cos v + \cos w = 2 \cos \frac{v+w}{2} \cos \frac{v-w}{2}$$

$$(V+W)\tan(v-w)/2 = (V-W)\tan(v+w)/2$$

where $V, W$ are lengths of sides opposite angles $v, w$.

$$a \cos x + b \sin x = r \cos(x - \phi)$$
$$a \sin x + b \cos x = r \sin(x + \phi)$$

where $r = \sqrt{a^2 + b^2}, \phi = \operatorname{atan2}(b, a)$.

## 8.4 Geometry

### 8.4.1 Triangles

Side lengths: $a, b, c$

Semiperimeter: $p = \dfrac{a+b+c}{2}$

Area: $A = \sqrt{p(p-a)(p-b)(p-c)}$

Circumradius: $R = \dfrac{abc}{4A}$

Inradius: $r = \dfrac{A}{p}$

Length of median (divides triangle into two equal-area triangles): $m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$

Length of bisector (divides angles in two):

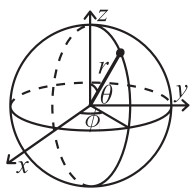
$$s_a = \sqrt{bc\left[1 - \left(\frac{a}{b+c}\right)^2\right]}$$

Law of sines: $\dfrac{\sin\alpha}{a} = \dfrac{\sin\beta}{b} = \dfrac{\sin\gamma}{c} = \dfrac{1}{2R}$

Law of cosines: $a^2 = b^2 + c^2 - 2bc\cos\alpha$

Law of tangents: $\dfrac{a+b}{a-b} = \dfrac{\tan\dfrac{\alpha+\beta}{2}}{\tan\dfrac{\alpha-\beta}{2}}$

### 8.4.2 Spherical coordinates




$$\begin{aligned} x &= r\sin\theta\cos\phi & r &= \sqrt{x^2+y^2+z^2} \\ y &= r\sin\theta\sin\phi & \theta &= \mathrm{acos}(z/\sqrt{x^2+y^2+z^2}) \\ z &= r\cos\theta & \phi &= \mathrm{atan2}(y, x) \end{aligned}$$

## 8.5 Derivatives/Integrals

$$\frac{d}{dx}\arcsin x = \frac{1}{\sqrt{1-x^2}} \qquad \frac{d}{dx}\arccos x = -\frac{1}{\sqrt{1-x^2}}$$

$$\frac{d}{dx}\tan x = 1 + \tan^2 x \qquad \frac{d}{dx}\arctan x = \frac{1}{1+x^2}$$

$$\int \tan ax = -\frac{\ln|\cos ax|}{a} \qquad \int x\sin ax = \frac{\sin ax - ax\cos ax}{a^2}$$

$$\int e^{-x^2} = \frac{\sqrt{\pi}}{2}\mathrm{erf}(x) \qquad \int xe^{ax}dx = \frac{e^{ax}}{a^2}(ax-1)$$

Integration by parts:

$$\int_a^b f(x)g(x)dx = [F(x)g(x)]_a^b - \int_a^b F(x)g'(x)dx$$

## 8.6 Sums

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1} - c^a}{c-1}, c \neq 1$$

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(2n+1)(n+1)}{6}$$

$$1^3 + 2^3 + 3^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$

$$1^4 + 2^4 + 3^4 + \cdots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

## 8.7 Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots, (-\infty < x < \infty)$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots, (-1 < x \leq 1)$$

$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \ldots, (-1 \leq x \leq 1)$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots, (-\infty < x < \infty)$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots, (-\infty < x < \infty)$$

## 8.8 Probability theory

Let $X$ be a discrete random variable with probability $p_X(x)$ of assuming the value $x$. It will then have an expected value (mean) $\mu = \mathbb{E}(X) = \sum_x xp_X(x)$ and variance $\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x - \mathbb{E}(X))^2 p_X(x)$ where $\sigma$ is the standard deviation. If $X$ is instead continuous it will have a probability density function $f_X(x)$ and the sums above will instead be integrals with $p_X(x)$ replaced by $f_X(x)$.

Expectation is linear:

$$\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$$

For independent $X$ and $Y$,

$$V(aX + bY) = a^2V(X) + b^2V(Y).$$

### 8.8.1 Discrete distributions

#### Binomial distribution

The number of successes in $n$ independent yes/no experiments, each which yields success with probability $p$ is $\mathrm{Bin}(n, p)$, $n = 1, 2, \ldots, 0 \leq p \leq 1$.

$$p(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$\mu = np,\ \sigma^2 = np(1-p)$$

$\mathrm{Bin}(n, p)$ is approximately $\mathrm{Po}(np)$ for small $p$.

#### First success distribution

The number of trials needed to get the first success in independent yes/no experiments, each which yields success with probability $p$ is $\mathrm{Fs}(p)$, $0 \leq p \leq 1$.

$$p(k) = p(1-p)^{k-1},\ k = 1, 2, \ldots$$

$$\mu = \frac{1}{p},\ \sigma^2 = \frac{1-p}{p^2}$$

#### Poisson distribution

The number of events occurring in a fixed period of time $t$ if these events occur with a known average rate $\kappa$ and independently of the time since the last event is $\mathrm{Po}(\lambda)$, $\lambda = t\kappa$.

$$p(k) = e^{-\lambda}\frac{\lambda^k}{k!}, k = 0, 1, 2, \ldots$$

$$\mu = \lambda,\ \sigma^2 = \lambda$$

### 8.8.2 Continuous distributions

#### Uniform distribution

If the probability density function is constant between $a$ and $b$ and 0 elsewhere it is $\mathrm{U}(a, b)$, $a < b$.

$$f(x) = \begin{cases} \frac{1}{b-a} & a < x < b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{a+b}{2},\ \sigma^2 = \frac{(b-a)^2}{12}$$

#### Exponential distribution

The time between events in a Poisson process is $\mathrm{Exp}(\lambda)$, $\lambda > 0$.

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

$$\mu = \frac{1}{\lambda},\ \sigma^2 = \frac{1}{\lambda^2}$$

#### Normal distribution

Most real random values with mean $\mu$ and variance $\sigma^2$ are well described by $\mathcal{N}(\mu, \sigma^2)$, $\sigma > 0$.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

If $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$ then

$$aX_1 + bX_2 + c \sim \mathcal{N}(\mu_1 + \mu_2 + c, a^2\sigma_1^2 + b^2\sigma_2^2)$$

# Appendix (A)

## thinking.txt

174 lines

```
Reading problem statement
1. Read carefully, make sure no statements conflicts.
2. Rewrite important details /mark it.
3. If text is not small, Re-read the problem statement again.
     Make sure you have the full picture.
4. Extract constraints info.
    Never ignore any constraints, especially unusual one (e.g.
         2*(a+b) < c).
    Sometimes constraints are not direct. Find triangle angle
         with 2 precision -> 360 * 10^2 //brute force
5. Trace Samples as long as they are traceable.
6. Think in Missed cases, smallest boundaries & largest
     boundaries & Especial cases.
7. Write it on paper (to test it in your idea).
8. don't make assumptions.
9. Revise carefully output section and output formats.
Investigate
Analysis
  Problem Constraints
  Problem domain(s)
  Search Space Size (size of unique solutions)
  Nature of target Function
  Output Bounding
Problem Abstraction
    never to drop the original problem, sometimes your
         abstraction drop some important domain consideration
Problem Simplification
    Adhoc
    Problem to Sub-Problems: you are JUST solving a sub-problem
          you invented.
    Incrementally: think in a special problem/case, and then
         try to update the solution for general problem/case.
    Simplification by Assumptions
    think in general case
Problem Reverse
    Adhoc
    f(x) = y , search (x)
    atmost vs exect
    Property Reverse
        probability(x) = 1 - probability(!x)
        Subset with x = total - (subset without x)
        Min/Max
Problem Domain re-interpretation
Thinking
    Think on papers not in pc
    The Brute Force solution
        Think in BF ITERATIVE and RECURSIVE.
        Think about Search space & Search State
    Problem Domain re-interpretation
    Concretely, Symbolically, Pictorially
    Forward and backward
    Brainstorm - Rank - Approach
    Divide & Conquer problem
Observations Discovery
    used pc if better (SIMPLE code only, don't take a lot of
         time)
Some popular properties:
    number of states
    Symmetry
    Inference
    Redundancy
    Independency
    IO re-representation
    Graph
        Think in bipartite graph
        DAG: topological sort
        Tree is a bipartite graph
    Canonical Form
    Cycle tricks
    Input Function Nature
    Tricks
    Dp
        Inference value
        convert to table:
        Applied data structures
        Adhoc trick
    Dp optimizations (D&C, Knuth, Convex Hull)
    Patterns
    Cyclic Function (repeated after X step)
Stuck?
1. Try to re-state the problem definition in 2 or 3 different
     ways and see if this helps.
2. Still Stuck? Do BF & Observe. Write the impractical BF
     solution, and try to find a pattern for answer / useful
     fact
3. Still Stuck? Iterate on your algorithms list, see if it
     could be the solution.
4. Be careful in analysis for solutions. You may discard a
     correct solution
    E.g., Calculated O(n^3) although it is O(n^2logn)
    E.g., Calculated recursion depth wrongly.
BIG order
    Check Exact # of operation
    Check for Reduced variables & Constrained input
         combinations
    Check duplicates and unique values
    SQRT tricks
    Preprocessing
        Think in all kinds of precomputation and try to utilize
              any of them.
        Next array
        Calc on machine a small temp array that help you in run
              time.
    In query problems, solve them offline if that help
    Order of loops and data such that loops break so early as
         much as possible.
    Reference of locality tricks
        Order loops such that: loops don't pass over arrays is
             in depth. Watch out from Col-order accessing
        Maybe duplicate some memory to switch some col order to
              row order.

Solution Verification
But before you verify, you should remember (keep it simple)!
     Could we find something simpler!
verifying the solution requires:
1. Test cases Verification (Ones in statement, boundaries,
     yours)
2. Solution Order: Time & Memory
3. Full logic & intuitive Revision
4. Is valid (BS/TS)?
5. Correctness, at least good intuitive - Assumption's
     validations
6. In case recursive code, does depth fit?
7. In case (*+^) operations, Any overflow (intermediate &
     output)
8. In case Double /, Is precision fine? Using long double
     instead of double?
9. In case Double /, can we not use it??

After Implementation
1. Revise code order & logic. Make sure it matches what you
     intended
2. Challenge every block of code. Never to read in a way that
     drop even ONE character
3. Think how this block of code maybe fail.
4. Revise data types
5. Double comparison, precision of +- numbers [(int)(a +/- EPS)
     ]
6. Return statement in functions
Test special cases. Testing the boundaries + revise SPECIAL
     CASES.
Failed? Check Error Inspection List

Error Inspection
General
  Do you read all input file?
  Initialization (between test cases)
  TYPO, variable names
  Conditions \function base case\return statement in function
  Overflow
  avoid double operations if possible
  Corner cases
  Arrays boundary
Wrong Answers
  Review constrains
  Review code again
  Re-read the problem statement
    Tricky text description
  Geometry
    Double precision
    Are there duplicate points? Does it matter? Co-linearity?
    Polygon: convex? concave?
  Graph
    Connected or disconnected?
    Directed or Undirected?
    Self Loops?
    Multiple edges
Precision
    Watchout -0.0
    int x = (int)(a +/- EPS) depends on a > 0 | a < 0.

Time Limit Exceed
    May be bug and just infinite loop
    Can we precompute the results?
    Function calls may need reference variables.
    % is used extensively?
        If mod is 2^p-1, use bitwise
    What is blocks of code that represent order? Do we just
         need to optimize it?
    Big Input file
        Need scanf & printf?
        Optimize code operations
        Switch to arrays and char[]
    DP Problems
        Do you really need to clear each time?
        The base case order is not O(1)
        Use effective base conditions E.g. If you are sure Dp
             (0, M) is X, do not wait until Dp(0,0)
    Cyclic recurrence?
    Backtracking If you have different ways to do it, try to do
          what minimize stack depth

Run time error
    Make sure to have correct array size.
    Make sure no wrong indexing < 0 || x >= n
    In DP, check you access dimensions correctly
    Stack overflow
    /0, %0
    Using incorrect compare function (e.g. return that return (
         A, B) same answer as (B, A))
```

## techniques.txt

109 lines

```
Data structures
  STL
  bitmask
  bitset
  monoqueue
  BIT
    update range
    2d
    Persistent
  Segment tree
    lazy
    dynamic
    merge sort
    persistent
    wavelet tree
  sparse table
  SQRT Decomposition
  ordered set
  MO
  DSU
  Treaps
Strings
  KMP
  trie
  aho-corasick
  suffix array
  rolling hash
  suffix automaton
Recursion
Divide and conquer
Greedy algorithm
Graph theory
  DFS
  BFS
  bipartite graph
  Topological sorting
  Strongly connected components
    Aritculation Points and Bridges
    biconnected components
  2-SAT
  MST
    Prim
    Kruskal
  bfs
  dijkstra
  Bellman-Ford's
  floyd
  Flow
    Augmenting paths
    Edmonds-Karp
    Bipartite matching
    Dinic
    Min-cost max flow
    Hopcroft-Karp
  Euler cycles
  Trees
    bipartite graph
    Diameter
    DP on tress
    SACK
  Data structues
    LCA
    HLD
    Centroid Decomposition
    Virtual trees
Dynamic programming
  Knapsack
  Knapsack SQRT trick
  Coin change
  Longest common subsequence
  Longest increasing subsequence
  Number of paths in a dag
  Shortest path in a dag
  Max 1D,2D Range Sum
  DP consecutive Ranges
  DP nested Ranges
  Push loop to paramter
  Push parameter to loop
  DP General Ranges
  Counting
  Building output
  DP Bitmask (TSP)
  Bitonic cycle
  DP on trees
  DP SOS (sum of submasks)
  Speed-up: Convex Hull Optimization
  Speed-up: Divide and Conquer Optimization
  Speed-up: Knuth Optimization
Techniques
  Binary search
  Ternary search
  Sliding Window
  State-Space Search
  Meet-in-the-middle
  Backtracking
  Binary Search
  ternary search
  divide and conquer
  Matrix power
Game theory
  Nim
  Grundy numbers
  Mini-max
  Alpha-beta pruning
  cycles and patterns
  mirror strategy
Combinatorics
  Inclusion/exclusion
  Catalan number
```