Standard Code Library

SDU-TCS

Shandong University

May 12, 2024

Contents

一切的开始

数据结构

ST 表

```cpp
struct ST{
    int n;
    std::vector<array<int,21>> st;
    ST(int n):n(n),st(n + 1) {}
    void init(vector<int>& a){
        for(int i = 1;i <= n;i ++)st[i][0] = a[i - 1];
        for(int j = 1;j <= 18;j ++){
            for(int i = 1;i + (1 << j) <= n + 1;i ++){
                st[i][j] = max(st[i][j - 1],st[i + (1 << (j - 1))][j - 1]);
            }
        }
    }
    int rmq(int l,int r){
        int j = log(r - l + 1)/log(2);
        return max(st[l][j],st[r - (1 << j) + 1][j]);
    }
};
```

线段树

```cpp
struct SegTree {
    int l, r;
    SegTree *ls, *rs;
    ll sum;
    ll plus;
    SegTree (const int L, const int R) : l(L), r(R) {
        plus = 0;
        if (L == R) {
            /*Initial*/
            ls = rs = nullptr;
        } else {
            int M = (L + R) >> 1;
            ls = new SegTree (L, M);
            rs = new SegTree (M + 1, R);
            pushup();
        }
    }
    void pushup() {
        sum = ls->sum + rs->sum;
        //  std::cerr << "AAA" << l << ' ' << r << ' ' << sum;
    }
    void make_tag(long long w) {
        sum += (r - l + 1) * w;
        plus += w;
    }
    void pushdown() {
        if (plus == 0) return;
        ls->make_tag(plus);
        rs->make_tag(plus);
        plus = 0;
    }
    void upd(const int L, const int R, const int w) {
        if ((L > r) || (l > R)) return;
        if ((L <= l) && (r <= R)) {
            make_tag(w);
        } else {
            pushdown();
            ls->upd(L, R, w);
            rs->upd(L, R, w);
            pushup();
        }
    }
};
```

树状数组

```cpp
template <typename T>
struct Fenwick {
    int n;
    std::vector<T> a;
    Fenwick(int n) : n(n), a(n) {}
    void add(int x, T v) {
        for (int i = x + 1; i <= n; i += i & -i) {
            a[i - 1] += v;
        }
    }
    T sum(int x) {
        T ans = 0;
        for (int i = x; i > 0; i -= i & -i) {
            ans += a[i - 1];
        }
        return ans;
    }
    T rangeSum(int l, int r) {
        return sum(r) - sum(l);
    }
    int kth(T k) {
        int x = 0;
        // 先从高位开始取，如果当前这一位可以取，那么就考虑下一位是取 1 还是 0
        // 到最后找到的就是最大的那个 pos 并且对应的 <=x 的
        for (int i = 1 << std::__lg(n); i; i /= 2) {
            if (x + i <= n && k >= a[x + i - 1]) {
                x += i;
                k -= a[x - 1];
            }
        }
        return x;
    }//树状数组上倍增本质上是通过倍增来快速找出对应的区间
};
```

DSU

```cpp
struct DSU {
    std::vector<int> f, siz;
    DSU(int n) : f(n), siz(n, 1) { std::iota(f.begin(), f.end(), 0); }
    int leader(int x) {
        while (x != f[x]) x = f[x] = f[f[x]];
        return x;
    }
    bool same(int x, int y) { return leader(x) == leader(y); }
    bool merge(int x, int y) {
        x = leader(x);
        y = leader(y);
        if (x == y) return false;
        siz[x] += siz[y];
        f[y] = x;
        return true;
    }
    int size(int x) { return siz[leader(x)]; }
};
```

Splay

```cpp
struct Node {
  int v, sz, sm;
  Node *ch[2], *fa;

  Node(const int V, Node *const f) : v(V), sz(1), sm(1), fa(f) {
    ch[0] = ch[1] = nullptr;
  }

  inline int GetRela(const int x) { return (v == x) ? -1 : (x > v); }

  void pushup() { sm = (ch[0] ? ch[0]->sm : 0) + (ch[1] ? ch[1]->sm : 0) + sz; }
```

```

  inline void rotate(const int x) {
    auto nrt = ch[x];
    ch[x] = nrt->ch[x ^ 1];
    nrt->ch[x ^ 1] = this;
    if (ch[x]) ch[x]->fa = this;
    nrt->fa = fa; fa = nrt;
    if (nrt->fa) nrt->fa->ch[nrt->fa->GetRela(nrt->v)] = nrt;
    pushup(); nrt->pushup();
  }

  void splay(const Node *p) {
    while (fa != p) {
      auto pa = fa->fa;
      if (pa == p) {
        fa->rotate(fa->GetRela(v));
      } else {
        int k1 = fa->GetRela(v), k2 = pa->GetRela(fa->v);
        if (k1 == k2) {
          pa->rotate(k1);
          fa->rotate(k1);
        } else {
          fa->rotate(k1);
          fa->rotate(k2);
        }
      }
    }
  }
};
```

LCT

```
struct Node {
  int v, s;
  bool tag;
  Node *ch[2], *fa;

  inline void maketag() {
    tag = !tag;
    std::swap(ch[0], ch[1]);
  }
  inline void pushup() {
    s = v;
    for (auto u : ch) if (u != nullptr) {
      s ^= u->s;
    }
  }
  inline void pushdown() {
    if (tag) {
      for (auto u : ch) if (u != nullptr) {
        u->maketag();
      }
      tag = false;
    }
  }

  inline int Getson() { return fa->ch[1] == this; }

  inline bool IsRoot() { return (fa == nullptr) || (fa->ch[Getson()] != this); }

  void rotate(const int x) {
    auto nt = ch[x];
    ch[x] = nt->ch[x ^ 1];
    nt->ch[x ^ 1] = this;
    if (ch[x]) ch[x]->fa = this;
    nt->fa = fa;
    if (!IsRoot()) { fa->ch[Getson()] = nt; }
    fa = nt;
    pushup(); nt->pushup();
  }

```

```
  void splay() {
    static Node* stk[maxn];
    int top = 0;
    stk[++top] = this;
    for (auto u = this; !u->IsRoot(); stk[++top] = u = u->fa);
    while (top) stk[top--]->pushdown();
    while (!IsRoot()) {
      if (fa->IsRoot()) {
        fa->rotate(Getson());
      } else {
        auto pa = fa->fa;
        int l1 = Getson(), l2 = fa->Getson();
        if (l1 == l2) {
          pa->rotate(l2);
          fa->rotate(l1);
        } else {
          fa->rotate(l1);
          fa->rotate(l2);
        }
      }
    }
  }
};
Node *node[maxn], Mem[maxn];

void Cut(const int x, const int y);
void Link(const int x, const int y);
void Query(const int x, const int y);
void Update(const int x, const int y);

void access(Node *u) {
  for (Node *v = nullptr; u; u = (v = u)->fa) {
    u->splay();
    u->ch[1] = v; u->pushup();
  }
}

void makeroot(Node *const u) {
  access(u);
  u->splay();
  u->maketag();
}

void Query(const int x, const int y) {
  auto u = node[x], v = node[y];
  makeroot(u);
  access(v);
  v->splay();
  qw(v->s, '\n');
}

void Link(const int x, const int y) {
  auto u = node[x], v = node[y];
  makeroot(u);
  access(v); v->splay();
  if (u->IsRoot() == false) return;
  u->fa = v;
}

void Cut(const int x, const int y) {
  auto u = node[x], v = node[y];
  makeroot(u); access(v); u->splay();
  if ((u->ch[1] != v) || (v->ch[0] != nullptr)) return;
  u->ch[1] = v->fa = nullptr;
  u->pushup();
}

// w[x] -> y
void Update(const int x, const int y) {
  auto u = node[x];
  u->splay();
```

```
  u->s ^= u->v;
  u->s ^= (u->v = a[x] = y);
}
```

扫描线

```
//二维数点
struct Segment{
    int l,r,h,add;
    bool operator <(const Segment a)const{
        return h < a.h;
    }
};
struct SegTree {
    int l, r;
    SegTree *ls, *rs;
    int mn,len;
    int plus;
    SegTree (const int L, const int R) : l(L), r(R) {
        plus = 0;len = 0;
        if (L == R) {
            ls = rs = nullptr;
        } else {
            int M = (L + R) >> 1;
            ls = new SegTree (L, M);
            rs = new SegTree (M + 1, R);
            pushup();
        }
    }
    void pushup() {
        if(plus) len = r - l + 1;
        else if(l == r)len = 0;
        else len = ls->len + rs->len;
    }
    void make_tag(int w) {
        plus += w;
    }
    void pushdown() {
        if (plus == 0) return;
        ls->make_tag(plus);
        rs->make_tag(plus);
        plus = 0;
    }
    void update(const int L, const int R, const int w) {
        if ((L > r) || (l > R)) {
            return;
        }
        if ((L <= l) && (r <= R)) {
            make_tag(w);
            pushup();
            return ;
        } else {
            ls->update(L, R, w);
            rs->update(L, R, w);
            pushup();
        }
    }
};
//矩形面积并
#include<bits/stdc++.h>

using namespace std;
typedef long long ll;
const double eps = 1e-8;
const int maxn = 2e5 + 7;
std::vector<int> x;
struct Segment{
    int l,r,h,add;
    bool operator <(const Segment a)const{
        return h < a.h;
    }
```

```cpp
};
struct SegTree {
    int l, r;
    SegTree *ls, *rs;
    int mn,len;
    int plus;
    SegTree (const int L, const int R) : l(L), r(R) {
        plus = 0;len = 0;
        if (L == R) {
            ls = rs = nullptr;
        } else {
            int M = (L + R) >> 1;
            ls = new SegTree (L, M);
            rs = new SegTree (M + 1, R);
            pushup();
        }
    }
    void pushup() {
        if(plus) len = x[r] - x[l - 1];
        else if(l == r)len = 0;
        else len = ls->len + rs->len;
    }
    void make_tag(int w) {
        plus += w;
    }
    void pushdown() {
        if (plus == 0) return;
        ls->make_tag(plus);
        rs->make_tag(plus);
        plus = 0;
    }
    void update(const int L, const int R, const int w) {
        if ((L >= x[r]) || (x[l - 1] >= R)) {
            return;
        }
        if ((L <= x[l - 1]) && (x[r] <= R)) {
            make_tag(w);
            pushup();
            return ;
        } else {
            //pushdown();
            ls->update(L, R, w);
            rs->update(L, R, w);
            pushup();
        }
    }
};
int main(){
    ios::sync_with_stdio(false);
    cin.tie(0);

    vector&lt;Segment&gt; s;
    int n;
    cin >> n;
    for(int i = 0;i < n;i ++){
        int xa,ya,xb,yb;
        cin >> xa >> ya >> xb >> yb;
        x.push_back(xa);
        x.push_back(xb);
        s.push_back({xa,xb,ya,1});
        s.push_back({xa,xb,yb,-1});
    }
    sort(s.begin(),s.end());
    sort(x.begin(),x.end());
    x.erase(unique(x.begin(),x.end()),x.end());
    int N = x.size();
    SegTree Seg(1,N - 1);
    ll ans = 0;
    if(s.size()){
        Seg.update(s[0].l,s[0].r,s[0].add);
        for(int i = 1;i < s.size();i ++){
```

```
            ans += 1ll * Seg.len * (s[i].h - s[i - 1].h);
            Seg.update(s[i].l,s[i].r,s[i].add);
        }
    }
    cout << ans << "\n";
    return 0;
}
```

Seg beats

本质上是维护了两棵线段树，A 树维护区间内最大值产生的贡献，B 树维护剩下树的贡献。注意 A 树某节点的孩子不一定全部能贡献到该节点，因为孩子的最大值不一定是父亲的最大值。所以要注意下传标记时，A 树的孩子下传的可能是 B 的标记。

beats 的部分是，每次让序列里每个数对另一个数 V 取 min，则直接暴力递归到 `inRange` 且 B 的最大值小于 V 的那些节点上，转化成对 A 那个节点的区间加法（加上 $V - val_A$）即可。这么做的均摊复杂度是 $O(\log n)$。

做区间历史最大值的方法是，维护两个标记 x, y，x 是真正的加标记，y 是 x 在上次下传结束并清零后的历史最大值。下传时注意先下传 y 再下传 x。实现历史最值是平凡的，不需要 beats。beats 解决的仅是取 min 的操作。

下面五个操作分别是：区间加，区间对 k 取 min，区间求和，区间最大值，区间历史最大值。

```
#include <array>
#include <iostream>
#include <algorithm>

typedef long long int ll;

const int maxn = 500005;

ll a[maxn];

const ll inf = 0x3f3f3f3f3f3f3f3fll;

struct Node {
  Node *ls, *rs;
  int l, r, maxCnt;
  ll v, add, maxAdd, sum, maxV, maxHistory;

  Node(const int L, const int R) :
      ls(nullptr), rs(nullptr), l(L), r(R), maxCnt(0),
      v(0), add(0), maxAdd(0), sum(0), maxV(-inf), maxHistory(-inf) {}

  inline bool inRange(const int L, const int R) {
    return L <= l && r <= R;
  }
  inline bool outRange(const int L, const int R) {
    return l > R || L > r;
  }

  void addVal(const ll t, int len) {
    add += t;
    sum += len * t;
    maxV += t;
  }

  void makeAdd(const ll t, int len) {
    addVal(t, len);
    maxHistory = std::max(maxHistory, maxV);
    maxAdd = std::max(maxAdd, add);
  }
};

void pushup(Node *x, Node *y) {
  y->maxV = std::max(y->ls->maxV, y->rs->maxV);
  y->sum = y->ls->sum + y->rs->sum;
  y->maxHistory = std::max({y->maxHistory, y->ls->maxHistory, y->rs->maxHistory});
  if (x->ls->maxV != x->rs->maxV) {
    bool flag = x->ls->maxV < x->rs->maxV;
    if (flag) std::swap(x->ls, x->rs);
    x->maxV = x->ls->maxV;
```

```
    x->maxCnt = x->ls->maxCnt;
    y->maxV = std::max(y->maxV, x->rs->maxV);
    y->sum += x->rs->sum;
    x->sum = x->ls->sum;
    if (flag) std::swap(x->ls, x->rs);
  } else {
    x->maxCnt = x->ls->maxCnt + x->rs->maxCnt;
    x->sum = x->ls->sum + x->rs->sum;
    x->maxV = x->ls->maxV;
  }
  x->maxHistory = std::max({x->ls->maxHistory, x->rs->maxHistory, x->maxHistory, y->maxHistory});
}

void New(Node *&u1, Node *&u2, int L, int R) {
  u1 = new Node(L, R);
  u2 = new Node(L, R);
  if (L == R) {
    u1->v = u1->sum = u1->maxV = u1->maxHistory = a[L];
    u1->maxCnt = 1;
  } else {
    int M = (L + R) >> 1;
    New(u1->ls, u2->ls, L, M);
    New(u1->rs, u2->rs, M + 1, R);
    pushup(u1, u2);
  }
}

void pushdown(Node *x, Node *y) {
  ll val = std::max(x->ls->maxV, x->rs->maxV);
  std::array<Node*, 2> aim({y, x});
  Node *curl = aim[x->ls->maxV == val], *curr = aim[x->rs->maxV == val];
  x->ls->maxAdd = std::max(x->ls->maxAdd, x->ls->add + curl->maxAdd);
  x->ls->maxHistory = std::max(x->ls->maxHistory, x->ls->maxV + curl->maxAdd);
  x->ls->addVal(curl->add, x->ls->maxCnt);
  x->rs->maxAdd = std::max(x->rs->maxAdd, x->rs->add + curr->maxAdd);
  x->rs->maxHistory = std::max(x->rs->maxHistory, x->rs->maxV + curr->maxAdd);
  x->rs->addVal(curr->add, x->rs->maxCnt);
  y->ls->maxAdd = std::max(y->ls->maxAdd, y->ls->add + y->maxAdd);
  y->rs->maxAdd = std::max(y->rs->maxAdd, y->rs->add + y->maxAdd);
  y->ls->addVal(y->add, x->ls->r - x->ls->l + 1 - x->ls->maxCnt);
  y->rs->addVal(y->add, x->rs->r - x->rs->l + 1 - x->rs->maxCnt);
  x->add = y->add = x->maxAdd = y->maxAdd = 0;
}

void addV(Node *x, Node *y, int L, int R, ll k) {
  if (x->inRange(L, R)) {
    x->makeAdd(k, x->maxCnt);
    y->makeAdd(k, x->r - x->l + 1 - x->maxCnt);
  } else if (!x->outRange(L, R)) {
    pushdown(x, y);
    addV(x->ls, y->ls, L, R, k);
    addV(x->rs, y->rs, L, R, k);
    pushup(x, y);
  }
}

std::array<ll, 3> qry(Node *x, Node *y, const int L, const int R) {
  if (x->inRange(L, R)) return {x->sum + y->sum * ((x->r - x->l + 1) != x->maxCnt), x->maxV, x->maxHistory};
  else if (x->outRange(L, R)) return {0, -inf, -inf};
  else {
    pushdown(x, y);
    auto A = qry(x->ls, y->ls, L, R), B = qry(x->rs, y->rs, L, R);
    return {A[0] + B[0], std::max(A[1], B[1]), std::max(A[2], B[2])};
  }
}

void minV(Node *x, Node *y, const int L, const int R, int k) {
  if (x->maxV <= k) return;
  if (x->inRange(L, R) && y->maxV < k) {
    ll delta = k - x->maxV;
    x->makeAdd(delta, x->maxCnt);
```

```
  } else if (!x->outRange(L, R)) {
    pushdown(x, y);
    minV(x->ls, y->ls, L, R, k);
    minV(x->rs, y->rs, L, R, k);
    pushup(x, y);
  }
}

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);
  int n, m;
  std::cin >> n >> m;
  for (int i = 1; i <= n; ++i) std::cin >> a[i];
  Node *rot1, *rot2;
  New(rot1, rot2, 1, n);
  for (int op, l, r; m; --m) {
    std::cin >> op >> l >> r;
    if (op == 1) {
      std::cin >> op;
      addV(rot1, rot2, l, r, op);
    } else if (op == 2) {
      std::cin >> op;
      minV(rot1, rot2, l, r, op);
    } else {
      std::cout << qry(rot1, rot2, l, r)[op - 3] << '\n';
    }
  }
}
```

珂朵莉树

```
auto getPos(int pos) {
  return --s.upper_bound({pos + 1, 0, 0});
}

void split(int pos) {
  auto it = getPos(pos);
  auto [l, r, v] = *it;
  s.erase(it);
  if (pos > l) s.insert({l, pos - 1, v});
  s.insert({pos, r, v});
}

void add(int l, int r, int v) {
  split(l); split(r + 1);
  for (auto x = getPos(l), y = getPos(r + 1); x != y; ++x) {
    x->v += v;
  }
}

void upd(int l, int r, int v) {
  split(l); split(r + 1);
  s.erase(getPos(l), getPos(r + 1));
  s.insert({l, r, v});
}
```

`getPos(pos)`：找到 pos 所在的迭代器 `split(pos)`：把 pos 所在的迭代器区间 [l, r] 分成 [l, pos - 1] 和 [pos, r] 两个

李超树

插入线段 kx + b 求某点最值

```
constexpr long long INF = 1'000'000'000'000'000'000;
constexpr int C = 100'000;
struct Line {
    int k;
    long long b;
    Line(int k, long long b) : k(k), b(b) {}
};
```

```cpp
long long f(const Line &line, int x) {
    return 1LL * line.k * x + line.b;
}
struct Node {
    Node *lc, *rc;
    Line line;
    Node(const Line &line) : lc(nullptr), rc(nullptr), line(line) {}
};
void modify(Node *&p, int l, int r, Line line) {
    if (p == nullptr) {
        p = new Node(line);
        return;
    }
    int m = (l + r) / 2;
    bool le = f(p -> line, l) < f(line, l);
    bool mi = f(p -> line, m) < f(line, m);
    if (!mi)
        std::swap(p -> line, line);
    if (r - l == 1)
        return;
    if (le != mi) {
        modify(p -> lc, l, m, line);
    } else {
        modify(p -> rc, m, r, line);
    }
}
Node *merge(Node *p, Node *q, int l, int r) {
    if (p == nullptr)
        return q;
    if (q == nullptr)
        return p;
    int m = (l + r) / 2;
    p -> lc = merge(p -> lc, q -> lc, l, m);
    p -> rc = merge(p -> rc, q -> rc, m, r);
    modify(p, l, r, q -> line);
    return p;
}
long long query(Node *p, int l, int r, int x) {
    if (p == nullptr)
        return INF;
    long long ans = f(p -> line, x);
    if (r - l == 1)
        return ans;
    int m = (l + r) / 2;
    if (x < m) {
        return std::min(ans, query(p -> lc, l, m, x));
    } else {
        return std::min(ans, query(p -> rc, m, r, x));
    }
}
```

动态维护凸壳

```cpp
/**
 * Author: Simon Lindholm
 * Date: 2017-04-20
 * License: CC0
 * Source: own work
 * Description: Container where you can add lines of the form kx+m, and query maximum values at points x.
 *  Useful for dynamic programming.
 * Time: O(\log N)
 * Status: tested
 */

struct Line {
  mutable ll k, m, p;
  bool operator<(const Line &o) const { return k < o.k; }
  bool operator<(ll x) const { return p < x; }
};

struct LineContainer: multiset<Line, less<>> {
```

```cpp
  const ll inf = LLONG_MAX;
  ll val_offset = 0;
  void offset(ll x) {
    val_offset += x;//整体加
  }
  ll div(ll a, ll b) {
    return a / b - ((a^b) < 0 && a%b);
  }
  bool isect(iterator x, iterator y) {
    if (y == end()) {
      x->p = inf;
      return 0;
    }
    if (x->k == y->k) {
      x->p = (x->m > y->m)? inf: -inf;
    } else {
      x->p = div(y->m - x->m, x->k - y->k);
    }
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m - val_offset, 0}), y = z++, x = y;//这里加减看情况
    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.k * x + l.m + val_offset;
  }
};

LineContainer* merge(LineContainer *S, LineContainer *T) {
  if (S->size() > T->size())
    swap(S, T);
  for (auto l: *S) {
    T->add(l.k, l.m + S->val_offset);
  }
  return T;
}
```

TODO

线段树合并和分裂

图论

树链剖分

```cpp
// 重链剖分
void dfs1(int x) {
  son[x] = -1;
  siz[x] = 1;
  for (auto v:e[x])
    if (!dep[v]) {
      dep[v] = dep[x] + 1;
      fa[v] = x;
      dfs1(v);
      siz[x] += siz[v];
      if (son[x] == -1 || siz[v] > siz[son[x]]) son[x] = v;
    }
}

void dfs2(int x, int t) {
  top[x] = t;
  dfn[x] = ++ cnt;
  rnk[cnt] = x;
  if (son[x] == -1) return;
  dfs2(son[x], t);
```

```
  for (auto v:e[x])
    if (v != son[x] && v != fa[x]) dfs2(v, v);
}
int lca(int u, int v) {
  while (top[u] != top[v]) {
    if (dep[top[u]] > dep[top[v]])
      u = fa[top[u]];
    else
      v = fa[top[v]];
  }
  return dep[u] > dep[v] ? v : u;
}
```

LCA

倍增求 LCA

```
void dfs(int x){
    for(int j = 1;j <= 19;j ++){
        f[x][j] = f[f[x][j - 1]][j - 1];
    }
    for(auto v:e[x]){
        if(v == f[x][0])continue;
        f[v][0] = x;
        dep[v] = dep[x] + 1;
        dfs(v);
    }
}
int lca(int u,int v){
    if(dep[u] < dep[v])swap(u,v);
    for(int i = 0;i <= 19;i ++){
        if((dep[u] - dep[v]) & (1 << i))u = f[u][i];
    }
    if(u == v)return u;
    for(int j = 19;j >= 0; j--){
        if(f[u][j] != f[v][j]){
            u = f[u][j];
            v = f[v][j];
        }
    }
    return f[u][0];
}
int kth(int x,int k){
    for(int i = 0;i <= 19;i ++){
        if(k & (1 << i))x = f[x][i];
    }
    return x;
}
```

dfn 求 LCA

```
int get(int x, int y) {return dfn[x] < dfn[y] ? x : y;}
void dfs(int id, int f) {
  mi[0][dfn[id] = ++dn] = f;
  for(int it : e[id]) if(it != f) dfs(it, id);
}
int lca(int u, int v) {
  if(u == v) return u;
  if((u = dfn[u]) > (v = dfn[v])) swap(u, v);
  int d = __lg(v - u++);
  return get(mi[d][u], mi[d][v - (1 << d) + 1]);
}
dfs(R, 0);
for(int i = 1; i <= __lg(n); i++)
  for(int j = 1; j + (1 << i) - 1 <= n; j++)
    mi[i][j] = get(mi[i - 1][j], mi[i - 1][j + (1 << i - 1)]);
```

树哈希

```cpp
typedef unsigned long long ull;
struct TreeHash{
    std::vector<int> hs;
    TreeHash(int n){
        hs.resize(n,0);
    }
    mt19937_64 rnd(chrono::steady_clock::now().time_since_epoch().count());
    ull bas = rnd();
    ull H(ull x){
        return x*x*x*19890535+19260817;
    }
    ull F(ull x){
        return H(x & ((1ll << 32) - 1)) + H(x >> 32);
    }
    int flag,n;
    void dfs(int u,int fa){
        hs[u] = bas;
        for(auto v:e[u]){
            if(v == fa) continue;
            dfs(v,u);
            hs[u] += F(hs[v]);
        }
    }
};
```

虚树

```cpp
void build_virtual_tree(vector<int> &h) {
  vector<int> a;
  sort(h.begin(), h.end(),[&](int &a,int &b){
      return dfn[a] < dfn[b];
  });  // 把关键点按照 dfn 序排序
  for (int i = 0; i < h.size(); ++i) {
    a.push_back(h[i]);
    if(i + 1 != h.size())a.push_back(lca(h[i], h[i + 1]));  // 插入 lca
  }
  sort(a.begin(), a.end(), [&](int &a,int &b){
      return dfn[a] < dfn[b];
  });  // 把所有虚树上的点按照 dfn 序排序
  a.erase(unique(a.begin(),a.end()),a.end());
  for (int i = 0; i < a.size() - 1; ++i) {
    int lc = lca(a[i], a[i + 1]);
    add(lc, a[i + 1]);  // 连边，如有边权 就是 distance(lc,a[i+1])
  }
}
```

Dijkstra

```cpp
void dijkstra(int s) {
    memset(dis, 0x3f, sizeof(dis));
    dis[s] = 0;
    priority_queue<pair<int,int>> q;
    q.push(make_pair(0, s));
    while(!q.empty()) {
        auto x = q.top().second;
        q.pop();
        if(vis[x]) continue;
        vis[x] = 1;
        for(auto [v,w] : e[x]) {
           if(dis[v] > dis[x] + w) {
               dis[v] = dis[x] + w;
               q.push({-dis[v],v});
           }
        }
    }
}
```

最小环

```cpp
//floyd 找最小环
//dijkstra 暴力删边跑最短路-
int floyd(const int &n) {
  for (int i = 1; i <= n; ++i)
    for (int j = 1; j <= n; ++j)
      dis[i][j] = f[i][j]; // 初始化最短路矩阵
  int ans = inf;
  for (int k = 1; k <= n; ++k) {
    for (int i = 1; i < k; ++i)
      for (int j = 1; j < i; ++j)
        ans = std::min(ans, dis[i][j] + f[i][k] + f[k][j]); // 更新答案
    for (int i = 1; i <= n; ++i)
      for (int j = 1; j <= n; ++j)
        dis[i][j] = std::min(dis[i][j], dis[i][k] + dis[k][j]); // 正常的 floyd 更新最短路矩阵
  }
  return ans;
}
```

差分约束

$x_i + C \geq x_j$

最短路-> 最大解

最长路-> 最小解

判负环或正环即可

```cpp
bool spfa(){
    queue<int> q;
    vector<int> vis(n + 1),cnt(n + 1),dis(n + 1,1e9);
    dis[1] = 0;
    cnt[1] = 1;
    q.push(1);
    while(!q.empty()){
        int u = q.front();
        q.pop();
        vis[u] = 0;
        if(cnt[u] >= n)return 1;
        for(auto v:e[u]){
            if(dis[v] > dis[u] + len[p]){
                dis[v] = dis[u] + len[p];
                if(vis[v] == 0){
                    vis[v] = 1;
                    q.push(v);
                    cnt[v] ++;
                }
            }
        }
    }
    return 0;
}
```

最大流

```cpp
struct Flow {
    static constexpr int INF = 1e9;
    int n;
    struct Edge {
        int to, cap;
        Edge(int to, int cap) : to(to), cap(cap) {}
    };
    vector<Edge> e;
    vector<vector<int>> g;
    vector<int> cur, h;
    Flow(int n) : n(n), g(n) {}
    void init(int n) {
        for (int i = 0; i < n; i++) g[i].clear();
        e.clear();
```

```
      }
    bool bfs(int s, int t) {
        h.assign(n, -1);
        queue<int> que;
        h[s] = 0;
        que.push(s);
        while (!que.empty()) {
            int u = que.front();
            que.pop();
            for (int i : g[u]) {
                int v = e[i].to;
                int c = e[i].cap;
                if (c > 0 && h[v] == -1) {
                    h[v] = h[u] + 1;
                    if (v == t)
                        return true;
                    que.push(v);
                }
            }
        }
        return false;
    }
    int dfs(int u, int t, int f) {
        if (u == t)
            return f;
        int r = f;
        for (int &i = cur[u]; i < int(g[u].size()); ++i) {
            int j = g[u][i];
            int v = e[j].to;
            int c = e[j].cap;
            if (c > 0 && h[v] == h[u] + 1) {
                int a = dfs(v, t, std::min(r, c));
                e[j].cap -= a;
                e[j ^ 1].cap += a;
                r -= a;
                if (r == 0)
                    return f;
            }
        }
        return f - r;
    }
    void addEdge(int u, int v, int c) {
        g[u].push_back(e.size());
        e.push_back({v, c});
        g[v].push_back(e.size());
        e.push_back({u, 0});
    }
    int maxFlow(int s, int t) {
        int ans = 0;
        while (bfs(s, t)) {
            cur.assign(n, 0);
            ans += dfs(s, t, INF);
        }
        return ans;
    }
};
```

最小费用最大流

```
using i64 = long long;

struct MCFGraph {
    struct Edge {
        int v, c, f;
        Edge(int v, int c, int f) : v(v), c(c), f(f) {}
    };
    const int n;
    std::vector<Edge> e;
    std::vector<std::vector<int>> g;
    std::vector<i64> h, dis;
    std::vector<int> pre;
```

```
    bool dijkstra(int s, int t) {
        dis.assign(n, std::numeric_limits<i64>::max());
        pre.assign(n, -1);
        priority_queue<pair<i64, int>, vector<pair<i64, int>>, greater<pair<i64, int>>> que;
        dis[s] = 0;
        que.emplace(0, s);
        while (!que.empty()) {
            i64 d = que.top().first;
            int u = que.top().second;
            que.pop();
            if (dis[u] < d) continue;
            for (int i : g[u]) {
                int v = e[i].v;
                int c = e[i].c;
                int f = e[i].f;
                if (c > 0 && dis[v] > d + h[u] - h[v] + f) {
                    dis[v] = d + h[u] - h[v] + f;
                    pre[v] = i;
                    que.emplace(dis[v], v);
                }
            }
        }
        return dis[t] != std::numeric_limits<i64>::max();
    }
    MCFGraph(int n) : n(n), g(n) {}
    void addEdge(int u, int v, int c, int f) {
        if (f < 0) {
            g[u].push_back(e.size());
            e.emplace_back(v, 0, f);
            g[v].push_back(e.size());
            e.emplace_back(u, c, -f);
        } else {
            g[u].push_back(e.size());
            e.emplace_back(v, c, f);
            g[v].push_back(e.size());
            e.emplace_back(u, 0, -f);
        }
    }
    std::pair<int, i64> flow(int s, int t) {
        int flow = 0;
        i64 cost = 0;
        h.assign(n, 0);
        while (dijkstra(s, t)) {
            for (int i = 0; i < n; ++i) h[i] += dis[i];
            int aug = std::numeric_limits<int>::max();
            for (int i = t; i != s; i = e[pre[i] ^ 1].v) aug = std::min(aug, e[pre[i]].c);
            for (int i = t; i != s; i = e[pre[i] ^ 1].v) {
                e[pre[i]].c -= aug;
                e[pre[i] ^ 1].c += aug;
            }
            flow += aug;
            cost += i64(aug) * h[t];
        }
        return std::make_pair(flow, cost);
    }
};
```

二分图最大匹配

```
auto dfs = [&](auto &&dfs, int u, int tag) -> bool {
    if (vistime[u] == tag) return false;
    vistime[u] = tag;
    for (auto v : e[u]) if (!mtch[v] || dfs(dfs, mtch[v], tag)) {
      mtch[v] = u;
      return true;
    }
    return false;
 };
```

KM(二分图最大权匹配)

```
template <typename T>
struct hungarian {  // km
  int n;
  vector<int> matchx;  // 左集合对应的匹配点
  vector<int> matchy;  // 右集合对应的匹配点
  vector<int> pre;     // 连接右集合的左点
  vector<bool> visx;   // 拜访数组 左
  vector<bool> visy;   // 拜访数组 右
  vector<T> lx;
  vector<T> ly;
  vector<vector<T> > g;
  vector<T> slack;
  T inf;
  T res;
  queue<int> q;
  int org_n;
  int org_m;

  hungarian(int _n, int _m) {
    org_n = _n;
    org_m = _m;
    n = max(_n, _m);
    inf = numeric_limits<T>::max();
    res = 0;
    g = vector<vector<T> >(n, vector<T>(n));
    matchx = vector<int>(n, -1);
    matchy = vector<int>(n, -1);
    pre = vector<int>(n);
    visx = vector<bool>(n);
    visy = vector<bool>(n);
    lx = vector<T>(n, -inf);
    ly = vector<T>(n);
    slack = vector<T>(n);
  }

  void addEdge(int u, int v, int w) {
    g[u][v] = max(w, 0);  // 负值还不如不匹配 因此设为 0 不影响
  }

  bool check(int v) {
    visy[v] = true;
    if (matchy[v] != -1) {
      q.push(matchy[v]);
      visx[matchy[v]] = true;  // in S
      return false;
    }
    // 找到新的未匹配点 更新匹配点 pre 数组记录着" 非匹配边" 上与之相连的点
    while (v != -1) {
      matchy[v] = pre[v];
      swap(v, matchx[pre[v]]);
    }
    return true;
  }

  void bfs(int i) {
    while (!q.empty()) {
      q.pop();
    }
    q.push(i);
    visx[i] = true;
    while (true) {
      while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v = 0; v < n; v++) {
          if (!visy[v]) {
            T delta = lx[u] + ly[v] - g[u][v];
            if (slack[v] >= delta) {
              pre[v] = u;
```

```
              if (delta) {
                slack[v] = delta;
              } else if (check(v)) {  // delta=0 代表有机会加入相等子图 找增广路
                                      // 找到就 return 重建交错树
                return;
              }
            }
          }
        }
      }
      // 没有增广路 修改顶标
      T a = inf;
      for (int j = 0; j < n; j++) {
        if (!visy[j]) {
          a = min(a, slack[j]);
        }
      }
      for (int j = 0; j < n; j++) {
        if (visx[j]) {  // S
          lx[j] -= a;
        }
        if (visy[j]) {  // T
          ly[j] += a;
        } else {  // T'
          slack[j] -= a;
        }
      }
      for (int j = 0; j < n; j++) {
        if (!visy[j] && slack[j] == 0 && check(j)) {
          return;
        }
      }
    }
  }

  void solve() {
    // 初始顶标
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        lx[i] = max(lx[i], g[i][j]);
      }
    }

    for (int i = 0; i < n; i++) {
      fill(slack.begin(), slack.end(), inf);
      fill(visx.begin(), visx.end(), false);
      fill(visy.begin(), visy.end(), false);
      bfs(i);
    }

    // custom
    for (int i = 0; i < n; i++) {
      if (g[i][matchx[i]] > 0) {
        res += g[i][matchx[i]];
      } else {
        matchx[i] = -1;
      }
    }
    cout << res << "\n";
    for (int i = 0; i < org_n; i++) {
      cout << matchx[i] + 1 << " ";
    }
    cout << "\n";
  }
};
```

一般图最大匹配

```
#include <bits/stdc++.h>
struct Graph {
    int n;
```

```
    std::vector<std::vector<int>> e;
    Graph(int n) : n(n), e(n + 1) {}
    void addEdge(int u, int v) {
        e[u].push_back(v);
        e[v].push_back(u);
    }
    std::vector<int> findMatching() {
        std::vector<int> match(n + 1, -1), vis(n + 1), link(n + 1), f(n + 1), dep(n + 1);

        // disjoint set union
        auto find = [&](int u) {
            while (f[u] != u)
                u = f[u] = f[f[u]];
            return u;
        };

        auto lca = [&](int u, int v) {
            u = find(u);
            v = find(v);
            while (u != v) {
                if (dep[u] < dep[v])
                    std::swap(u, v);
                u = find(link[match[u]]);
            }
            return u;
        };

        std::queue<int> q;
        auto blossom = [&](int u, int v, int p) {
            while (find(u) != p) {
                link[u] = v;
                v = match[u];
                if (vis[v] == 0) {
                    vis[v] = 1;
                    q.push(v);
                }
                f[u] = f[v] = p;
                u = link[v];
            }
        };

        // find an augmenting path starting from u and augment (if exist)
        auto augment = [&](int u) {

            while (!q.empty())
                q.pop();

            std::iota(f.begin(), f.end(), 0);

            // vis = 0 corresponds to inner vertices, vis = 1 corresponds to outer vertices
            std::fill(vis.begin(), vis.end(), -1);

            q.push(u);
            vis[u] = 1;
            dep[u] = 0;

            while (!q.empty()){
                int u = q.front();
                q.pop();
                for (auto v : e[u]) {
                    if (vis[v] == -1) {

                        vis[v] = 0;
                        link[v] = u;
                        dep[v] = dep[u] + 1;
                        // found an augmenting path
                        if (match[v] == -1) {
                            for (int x = v, y = u, temp; y != -1; x = temp, y = x == -1 ? -1 : link[x]) {
                                temp = match[y];
                                match[x] = y;
                                match[y] = x;
```

```
                            }
                            return;
                        }
                        vis[match[v]] = 1;
                        dep[match[v]] = dep[u] + 2;
                        q.push(match[v]);

                    } else if (vis[v] == 1 && find(v) != find(u)) {
                        // found a blossom
                        int p = lca(u, v);
                        blossom(u, v, p);
                        blossom(v, u, p);
                    }
                }
            }

        };

        // find a maximal matching greedily (decrease constant)
        auto greedy = [&]() {

            for (int u = 1; u <= n; ++u) {
                if (match[u] != -1)
                    continue;
                for (auto v : e[u]) {
                    if (match[v] == -1) {
                        match[u] = v;
                        match[v] = u;
                        break;
                    }
                }
            }
        };

        greedy();

        for (int u = 1; u <= n; ++u)
            if (match[u] == -1)
                augment(u);

        return match;
    }
};
int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    int n, m;
    std::cin >> n >> m;
    Graph g(n);
    for (int i = 0; i < m; ++i) {
        int u, v;
        std::cin >> u >> v;
        g.addEdge(u, v);
    }
    auto match = g.findMatching();
    int ans = 0;
    for (int u = 1; u <= n; ++u)
        if (match[u] != -1)
            ++ans;
    std::cout << ans / 2 << "\n";
    for (int u = 1; u <= n; ++u)
        if(match[u] != -1)std::cout << match[u] << " ";
        else std::cout << 0 << " ";
    return 0;
}
```

缩点 SCC

```
void dfs(const int u) {
  low[u] = dfn[u] = ++cnt;
  ins[stk[++top] = u] = true;
```

```cpp
  for (auto v : e[u]) if (dfn[v] == 0) {
    dfs(v);
    low[u] = std::min(low[u], low[v]);
  } else if (ins[v]) {
    low[u] = std::min(low[u], dfn[v]);
  }
  if (low[u] == dfn[u]) {
    ++scnt; int v;
    do {
      ins[v = stk[top--]] = false;
      w[bel[v] = scnt] += a[v];
    } while (u != v);
  }
}
```

割点与桥

```cpp
//割点
void tarjan(int u, int fa){
    dfn[u] = low[u] = ++cnt; int du = 0;
    for(for v:e[x]){
        if(v == fa) continue;
        if(!dfn[v]){ ++du;
            tarjan(v, u); low[u] = min(low[u], low[v]);
            if(low[v] >= dfn[u] && fa) vis[u] = 1;
        }
        else low[u] = min(low[u], dfn[v]);
    }
    if(!fa && du > 1) vis[u] = 1;
}
//桥
void tarjan(int u, int fa) {
  f[u] = fa;
  low[u] = dfn[u] = ++cnt;
  for (auto v:e[u]) {
    if (!dfn[v]) {
      tarjan(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > dfn[u]) {
        isbridge[v] = true;
        ++cnt_bridge;
      }
    } else if (dfn[v] < dfn[u] && v != fa) {
      low[u] = min(low[u], dfn[v]);
    }
  }
}
```

边双缩点

```cpp
void form(int x){
    std::vector<int> tmp;
    int now = 0;
    do{
        now = s[top --];
        tmp.push_back(now);
    }while(now != x);
    ans.push_back(tmp);
}
void tarjan(int x,int now){
    dfn[x] = low[x] = ++cnt;
    s[++ top] = x;
    for(auto [v,_]:e[x]){
        if(_ == now)continue;
        if(!dfn[v]){
            tarjan(v,_);
            low[x] = min(low[x],low[v]);
            if(low[v] > dfn[x]){
                form(v);
            }
```

```cpp

        }else low[x] = min(low[x],dfn[v]);
    }
}
for(int i = 1;i <= n;i ++){
    if(dfn[i] == 0){
        tarjan(i,0);
        form(i);
    }
}
cout << ans.size() << "\n";
for(auto A:ans){
    cout << A.size() << " ";
    for(auto x:A){
        cout << x << " ";
    }cout << "\n";
}
```

圆方树

```cpp
void dfs(int u) {
    static int cnt = 0;
    dfn[u] = low[u] = ++cnt;
    for (auto [v,w]:e[u]) {
        if (v == fa[u]) continue;
        if (!dfn[v]) {
            fa[v] = u; fr[v] = w;
            dfs(v); low[u] = min(low[u], low[v]);
        }
        else low[u] = min(low[u], dfn[v]);
        if (low[v] > dfn[u]) add(u, v, w); // 圆 - 圆
    }
    for (auto [v,w]:e[u]) {
        if (u == fa[v] || dfn[v] < dfn[u]) continue;
        add(u, v, w); // 圆 - 方
    }
}
```

广义圆方树

跟普通圆方树没有太大的区别，大概就是对于每个点双新建一个方点，然后将点双中的所有点向方点连边

需要注意的是我的写法中，两个点一条边也视为一个点双

性质

1. 树上的每一条边都连接了一个圆点和一个方点
2. 每个点双有唯一的方点
3. 一条从圆点到圆点的树上简单路径代表原图的中的一堆路径，其中圆点是必须经过的，而方点 (指的是与方点相连的点双）是可以随便走的，也可以理解成原图中两点简单路径的并

```cpp
void dfs(int x) {
    stk.push_back(x);
    dfn[x] = low[x] = cur++;

    for (auto y : adj[x]) {
        if (dfn[y] == -1) {
            dfs(y);
            low[x] = std::min(low[x], low[y]);
            if (low[y] == dfn[x]) {
                int v;
                do {
                    v = stk.back();
                    stk.pop_back();
                    edges.emplace_back(n + cnt, v);
                } while (v != y);
                edges.emplace_back(x, n + cnt);
                cnt++;
            }
        } else {
```

```
            low[x] = std::min(low[x], dfn[y]);
        }
    }
}
```

2-SAT

输出方案时可以通过变量在图中的拓扑序确定该变量的取值。如果变量 x 的拓扑序在 $\neg x$ 之后，那么取 x 值为真。应用到 Tarjan 算法的缩点，即 x 所在 SCC 编号在 $\neg x$ 之前时，取 x 为真。因为 Tarjan 算法求强连通分量时使用了栈，所以 Tarjan 求得的 SCC 编号相当于反拓扑序。

环计数

```
//三元环
  for (int u, v; m; --m) {
    u = A[m]; v = B[m];
    if (d[u] > d[v]) {
      std::swap(u, v);
    } else if ((d[u] == d[v]) && (u > v)) {
      std::swap(u, v);
    }
    e[u].push_back(v);
  }
  for (int u = 1; u <= n; ++u) {
    for (auto v : e[u]) vis[v] = u;
    for (auto v : e[u]) {
      for (auto w : e[v]) if (vis[w] == u) {
        ++ans;
      }
    }
  }
 // 四元环
  auto cmp = [&](int &a,int &b){
      if(d[a] != d[b])return d[a] > d[b];
      else return a < b;
  }
  for(int u = 1;u <= n;++ u) {
      for(auto v: G[u])//G 为原图
          for(auto w: e[v])
             if(cmp(u,w)) (ans += vis[w] ++)%=MOD;
      for(auto v: G[u])
          for(auto w: e[v])
             if(cmp(u,w)) vis[w] = 0;
  }
```

字符串

manacher

```
struct Manacher {
    int n, l, f[maxn * 2], Len;
    char s[maxn * 2];

    void init(char *c) {
        l = strlen(c + 1); s[0] = '~';
        for (int i = 1, j = 2; i <= l; ++i, j += 2)
            s[j] = c[i], s[j - 1] = '#';
        n = 2 * l + 1; s[n] = '#'; s[n + 1] = '\0';
    }
    void manacher() {
        int p = 0, mr = 0;
        for (int i = 1; i <= n; ++i) f[i] = 0;
        for (int i = 1; i <= n; ++i) {
            if (i < mr) f[i] = min(f[2 * p - i], mr - i);
            while (s[i + f[i]] == s[i - f[i]]) ++f[i]; --f[i];
            if (f[i] + i > mr) mr = i + f[i], p = i;
            Len = max(Len, f[i]);
        }
```

```
    }

    void solve() {
        for (int i = 1; i <= n; ++i) {
            // [1, l]
            int L = i - f[i] + 1 >> 1, R = i + f[i] - 1 >> 1;
            if (!f[i]) continue;

            // [1, 2 * l + 1]
            L = i - f[i], R = i + f[i];
        }
    }
} M;
```

SA

sa_i 表示排名为 i 的后缀。

rnk_i 表示 $[i, n]$ 这个后缀的排名（在 SA 里的下标）。

height_i 是 sa_i 和 sa_{i-1} 的 LCP 长度。换句话说，向求排名为 i 的后缀和排名为 $i-1$ 的后缀的 LCP 直接就是 height_i；求 $[i, n]$ 这个后缀和它在 sa 里前一个串的 LCP 就是 height_{rnk_i}

```
const int maxn = 1000005;

int sa[maxn], rnk[maxn], tax[maxn], tp[maxn], height[maxn];
void SA(string s) {
    int n = s.size();
    s = '#' + s;
    m = SIGMA_SIZE;
    vector<int> S(n + 1);
    auto RadixSort = [&]() {
        for (int i = 0; i <= m; ++i) tax[i] = 0;
        for (int i = 1; i <= n; ++i) ++tax[rnk[i]];
        for (int i = 1; i <= m; ++i) tax[i] += tax[i - 1];
        for (int i = n; i; --i) sa[tax[rnk[tp[i]]]--] = tp[i];
    };
    for (int i = 1; i <= n; ++i) {
        S[i] = s[i] - '0';
        tp[i] = i;
        rnk[i] = S[i];
    }
    RadixSort();
    for (int len = 1, p = 0; p != n; m = p, len <<= 1) {
        p = 0;
        for (int i = n - len + 1; i <= n; ++i) tp[++p] = i;
        for (int i = 1; i <= n; ++i) if (sa[i] > len) tp[++p] = sa[i] - len;
        RadixSort();
        std::swap(rnk, tp);
        p = 0;
        for (int i = 1; i <= n; ++i)
          rnk[sa[i]] = ((tp[sa[i]] == tp[sa[i-1]]) && (tp[sa[i] + len] == tp[sa[i-1] + len])) ? p : ++p;
    }
    for (int i = 1, p = 0; i <= n; ++i) {
        int pre = sa[rnk[i] - 1];
        if (p) --p;
        while (S[pre + p] == S[i + p]) ++p;
        h[0][rnk[i]] = height[rnk[i]] = p;
    }
    for (int i = 1; i <= 20; ++i) {
        memset(h[i], 0x3f, n * 4 + 4);
        for (int j = 1; j + (1 << i - 1) <= n; ++j)
            h[i][j] = min(h[i - 1][j], h[i - 1][j + (1 << i - 1)]);
    }
}
int Q(int l, int r) {
    if (l > r) swap(l, r);
    ++l;
    int k = __lg(r - l + 1);
    return min(h[k][l], h[k][r - (1 << k) + 1]);
```

```
}
int lcp(int i, int j) {
    if (i == j) return n - i + 1;
    return Q(rnk[i], rnk[j]);
}
```

PAM

```
struct PAM {
    static constexpr int ALPHABET_SIZE = 28;
    struct Node {
        int len; // 当前节点最长回文长度
        int fail;// 回文树边
        int scnt; // 当前节点表示的回文后缀的本质不同回文串个数
        int pcnt; // 当前节点回文串在字符串中出现次数, 每个点代表一个不同的回文串
        std::array<int, ALPHABET_SIZE> next;
        Node() : len{}, fail{}, scnt{}, next{}, pcnt{} {}
    };
    std::vector<Node> t;
    int last;
    std::string s;
    PAM() {
        init();
    }
    void init() {
        t.assign(2, Node());
        t[1].len = -1;
        last = 0;
        t[0].fail = 1;
        s = "$";
    }
    int newNode() {
        t.emplace_back();
        return t.size() - 1;
    }
    int get_fail(int x) {
        int pos = s.size() - 1;
        while(s[pos - t[x].len - 1] != s[pos]) x = t[x].fail;
        return x;
    }
    void add(char c, char offset = 'a') {
        s += c;
        int let = c - offset;
        int x = get_fail(last);
        if (!t[x].next[let]) {
            int now = newNode();
            t[now].len = t[x].len + 2;
            t[now].fail = t[get_fail(t[x].fail)].next[let];
            t[x].next[let] = now;
            t[now].scnt = t[t[now].fail].scnt + 1;
        }
        last = t[x].next[let];
        t[last].pcnt ++;
    }
};
```

SAM

```
struct SAM {
    static constexpr int ALPHABET_SIZE = 26,rt = 1;
    struct Node {
        int len,fa,siz;
        std::array<int, ALPHABET_SIZE> nxt;
        Node() : len{}, fa{}, siz{}, nxt{} {}
    };
    std::vector<Node> t;
    SAM() {
        init();
    }
    void init() {
```

```cpp
        t.assign(2, Node());
    }
    int newNode() {
        t.emplace_back();
        return t.size() - 1;
    }
    int getfa(int x){
        return t[x].fa;
    }
    int getlen(int x){
        return t[x].len;//表示该状态能够接受的最长的字符串长度。
    }
    int size(){
        return t.size();
    }
    int extend(int p, int ch) {
        int np = newNode();
        t[np].len = t[p].len + 1;t[np].siz = 1;
        while(p && !t[p].nxt[ch])t[p].nxt[ch] = np,p = t[p].fa;
        if(!p){t[np].fa = rt;return np;}
        int q = t[p].nxt[ch];
        if(t[q].len == t[p].len + 1){
            t[np].fa = q;
        }else {
            int nq = newNode();t[nq].len = t[p].len + 1;t[nq].fa = t[q].fa;
            for(int i = 0;i < 26;i ++)t[nq].nxt[i] = t[q].nxt[i];
            while(p && t[p].nxt[ch] == q)t[p].nxt[ch] = nq,p = t[p].fa;
            t[np].fa = t[q].fa = nq;
        }
        return np;
    }
    int extend_(int p, int ch) {//广义
        if(t[p].nxt[ch]){
            int q = t[p].nxt[ch];
            if(t[q].len == t[p].len + 1)return q;
            int nq = newNode();t[nq].len = t[p].len + 1;t[nq].fa = t[q].fa;
            for(int i = 0;i < 26;i ++)t[nq].nxt[i] = t[q].nxt[i];
            while(p && t[p].nxt[ch] == q)t[p].nxt[ch] = nq,p = t[p].fa;
            t[q].fa = nq;return nq;
        }
        int np = newNode();
        t[np].len = t[p].len + 1;
        while(p && !t[p].nxt[ch])t[p].nxt[ch] = np,p = t[p].fa;
        if(!p){t[np].fa = rt;return np;}
        int q = t[p].nxt[ch];
        if(t[q].len == t[p].len + 1){
            t[np].fa = q;
        }else {
            int nq = newNode();t[nq].len = t[p].len + 1;t[nq].fa = t[q].fa;
            for(int i = 0;i < 26;i ++)t[nq].nxt[i] = t[q].nxt[i];
            while(p && t[p].nxt[ch] == q)t[p].nxt[ch] = nq,p = t[p].fa;
            t[np].fa = t[q].fa = nq;
        }
        return np;
    }
    void build(vector<vector<int>> &e){
        e.resize(t.size());
        for(int i = 2;i < t.size();i ++){
            e[t[i].fa].push_back(i);
        }
    }
};
```

1. 本质不同的子串个数

   这个显然就是所有状态所对应的 endpos 集合的大小的和也等价于每个节点的 len 减去 parent 树上的父亲的 len

2. 求两个串的最长公共子串

```cpp
        int p = 1,len = 0,ans = 0;
        std::vector<int> l(m),L(m);
        for(int i = 0;i < m;i ++){
```

```
            int ch = s[i] - 'a';
            if(sam.t[p].nxt[ch]){
                p = sam.t[p].nxt[ch];len ++;
            }else {
                while(p && sam.t[p].nxt[ch] == 0){
                    p = sam.t[p].fa;
                }
                if(!p)p = 1,len = 0;
                else len = sam.t[p].len + 1,p = sam.t[p].nxt[ch];
            }//其中 p 为前缀最长能匹配到的后缀所在的节点
            l[i] = len;
            L[i] = i - len + 1;
        }
```

parent 树上每个节点维护了一个区间，若 p 是 q 的父节点则有 maxp = minq - 1

每个节点的 endpos 集合为该节点 parent 树上的子树 siz 大小

反串的 SAM 的 parent 树是原串的后缀树

ACAM

```
#define ch s[i] - 'a'
struct AC_automaton {
    int nxt[26], Nxt[26], cnt, fail;
} T[maxn]; int top = 1, rt = 1, id[maxn];
void insert(char *s, int k) {
    int now = rt, l = strlen(s);
    for (int i = 0; i < l; ++i) {
        if (!T[now].nxt[ch]) T[now].nxt[ch] = ++top;
        now = T[now].nxt[ch];
    } id[k] = now;
}

void init_fail() { // Trie 图
    queue<int> Q;
    for (int i = 0; i < 26; ++i) {
        int &u = T[rt].nxt[i];
        if (!u) { u = rt; continue; }
        T[u].fail = rt; Q.push(u);
    }
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int i = 0; i < 26; ++i) {
            int &v = T[u].nxt[i];
            if (!v) { v = T[T[u].fail].nxt[i]; continue; }
            T[v].fail = T[T[u].fail].nxt[i]; Q.push(v);
        }
    }
}

void init_fail() {
    queue<int> Q;
    for (int i = 0; i < 26; ++i) {
        int u = T[rt].nxt[i]; if (!u) { T[rt].Nxt[i] = rt; continue; }
        T[rt].Nxt[i] = u; T[u].fail = rt; Q.push(u);
    }
    while (!Q.empty()) {
        int u = Q.front(); Q.pop();
        for (int i = 0; i < 26; ++i) {
            int v = T[u].nxt[i];
            if (!v) { T[u].Nxt[i] = T[T[u].fail].Nxt[i]; continue; }
            T[u].Nxt[i] = v; T[v].fail = T[T[u].fail].Nxt[i]; Q.push(v);
        }
    }
}
```

KMP

```cpp
struct KMP{
    string s2;// add '#'
    std::vector<int> nxt;
    int m;
    KMP(string y) :s2(y){
        m = s2.size() - 1;
        nxt.resize(m + 1,0);
        for(int i = 2,p = 0;i <= m;i ++){
            while(p && s2[i] != s2[p + 1])p = nxt[p];
            if(s2[i] == s2[p + 1])p ++;
            nxt[i] = p;
        }
    }
    void match(string s1){
        int n = s1.size() - 1;
        for(int i = 1,p = 0;i <= n;i ++){
            while(p && s1[i] != s2[p + 1])p = nxt[p];
            if(s1[i] == s2[p + 1]){
                p ++;
                if(p == m){
                    //cout<<i - m + 1<<endl;
                    p = nxt[p];
                }
            }
        }
    }
    std::vector<int> find_border(){
        std::vector<int> v;
        for(int i = nxt[m];i;i = nxt[i])v.push_back(i);
        return v;
    }// 找该串所有的周期
    std::vector<int> calc_prefixes(){
        std::vector<int> cnt(m + 1,1);
        for(int i = m;i >= 1;i --)cnt[nxt[i]] += cnt[i];
        return cnt;
    }// 每个前缀出现次数
};
```

Z 函数

对于一个长度为 nn 的字符串 s，定义函数 $z[i]$ 表示和 $s[i, n-1]$（即以 $s[i]$ 开头的后缀）的最长公共前缀（LCP）的长度，特别地，$z[0] = 0$。

```cpp
std::vector<int> getZ(const std::string &s) {
  int n = s.size();
  std::vector<int> Z(n);
  Z[0] = n;
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r && Z[i - l] < r - i + 1) {
      Z[i] = Z[i - l];
    } else {
      Z[i] = std::max(0, r - i + 1);
      while (i + Z[i] < n && s[Z[i]] == s[i + Z[i]]) ++Z[i];
    }
    if (i + Z[i] - 1 > r) r = i + Z[l = i] - 1;
  }
  return Z;
}

std::vector<int> match(const std::string &s, const std::string &t) {
  auto Z = getZ(t);
  int n = s.size(), m = t.size();
  std::vector<int> ret(n);
  while (ret[0] < n && ret[0] < m && s[ret[0]] == t[ret[0]]) ++ret[0];
  for (int l = 0, r = ret[0] - 1, i = 1; i < n; ++i) {
    if (i <= r && Z[i - l] < r - i + 1) {
      ret[i] = Z[i - l];
    } else {
      ret[i] = std::max(0, r - i + 1);
```

```cpp
            while (i + ret[i] < n && s[i + ret[i]] == t[ret[i]]) ++ret[i];
        }
        if (i + ret[i] - 1 > r) r = i + ret[l = i] - 1;
    }
    return ret;
}
```

LCP

```cpp
for(int i = n;i >= 1;i --) {
    for(int j = n;j >= 1;j --) {
        if(s[i] == s[j]) {
            f[i][j] = f[i + 1][j + 1] + 1;// i-n 和 j-n 的 lcp
        }
    }
}
```

Hash

```cpp
struct Hash {
    string s;
    using ull = unsigned long long;
    ull P1 = 998255347;
    ull P2 = 1018253347;
    ull base = 131;
    vector<ull> hs1,hs2;
    vector<ull> ps1,ps2;
    Hash(string s): s(s) {
        int n = s.size();
        hs1.resize(n);
        hs2.resize(n);
        ps1.resize(n);
        ps2.resize(n);
        ps1[0] = ps2[0] = 1;
        hs1[0] = hs2[0] = (s[0] - 'a');
        for(int i = 1;i < n;i ++) {
            hs1[i] = hs1[i - 1] * base % P1 + (s[i] - 'a');
            hs2[i] = hs2[i - 1] * base % P2 + (s[i] - 'a');
            ps1[i] = (ps1[i - 1] * base) % P1;
            ps2[i] = (ps2[i - 1] * base) % P2;
        }
    }
    pair<ull,ull> query(int l,int r) {
        ull res1 = (hs1[r] - (l == 0 ? 0 : hs1[l - 1]) * ps1[r - l + 1] % P1 + P1) % P1;
        ull res2 = (hs2[r] - (l == 0 ? 0 : hs2[l - 1]) * ps2[r - l + 1] % P2 + P2) % P2;
        return {res1,res2};
    } // [l,r]
};
```

多项式

拉格朗日插值

$$f(x) = \sum_{i=1}^{n} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

```cpp
vector<int> lagrange(const vector<int> &x,const vector<int> y){
    assert(x.size() == y.size());
    int n = x.size();
    std::vector<int> a(n);
    for(int i = 0;i < n;i ++){
        int A = 1;
        for(int j = 0;j < n;j ++){
            if(i == j)continue;
            assert(x[i] - x[j]);
            A = 1ll * A * (x[i] - x[j] + mo) % mo;
        }
        a[i] = 1ll * y[i] * qp(A,mo - 2) % mo;
```

```
    }
    std::vector<int> b(n + 1),c(n),f(n);
    b[0] = 1;
    for(int i = 0;i < n;i ++){
        for(int j = i + 1;j >= 1;j --){
            b[j] = (1ll * b[j] * (mo - x[i]) % mo + b[j - 1]) % mo;
        }
        b[0] = 1ll * b[0] * (mo - x[i]) % mo;
    }
    for(int i = 0;i < n;i ++){
        int inv = qp(mo - x[i],mo - 2);
        if(!inv){
            for(int j = 0;j < n;j ++)c[j] = b[j + 1];
        }else {
            c[0] = 1ll * b[0] * inv % mo;
            for(int j = 1;j < n;j ++){
                c[j] = 1ll * (b[j] - c[j - 1] + mo) * inv % mo;
            }
        }
        for(int j = 0 ;j < n;j ++){
            f[j] = (f[j] + 1ll * a[i] * c[j] % mo) % mo;
        }
    }
    return f;
}
```

横坐标连续

$$f(x) = \sum_{i=1}^{n+1} y_i \cdot \frac{\prod_{j=1}^{n+1}(x-j)}{(x-i)\cdot(-1)^{n+1-i}\cdot(i-1)!\cdot(n+1-i)!}$$

普通幂与上升幂和下降幂

记上升阶乘幂 $x^{\overline{n}} = \prod_{k=0}^{n-1}(x+k)$。

则可以利用下面的恒等式将上升幂转化为普通幂:

$$x^{\overline{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^k$$

如果将普通幂转化为上升幂，则有下面的恒等式:

$$x^n = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} (-1)^{n-k} x^{\overline{k}}$$

记下降阶乘幂 $x^{\underline{n}} = \dfrac{x!}{(x-n)!} = \prod_{k=0}^{n-1}(x-k) = n!\binom{x}{n}$。

则可以利用下面的恒等式将普通幂转化为下降幂:

$$x^n = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} x^{\underline{k}}$$

如果将下降幂转化为普通幂，则有下面的恒等式:

$$x^{\underline{n}} = \sum_k \begin{bmatrix} n \\ k \end{bmatrix} (-1)^{n-k} x^k$$

多项式操作技巧

分治 FFT

$\prod_{i=1}^{n} f_i(x)$，分治两两合并即可

$f_i = \sum_{j=1}^{i} f_{i-j} g_j$，先求出左边 f_i 再将其与 g_i 相乘算出对右边 f_i 的贡献。

上式也可化为 $f(x) = (1 - g(x))^{-1}$

循环卷积

将其中一个序列复制一边，再做卷积

差卷积

$h_d = \sum f_i g_{i+d}$

将其中一个序列翻转，就转化成了加法卷积

乘法卷积

$h_k = \sum_{i \times j = k} f_i g_j$

对下标取离散对数，转成加法卷积

FWT

or，and，xor 卷积

```cpp
void FWT_or(ll *a,int N,int opt){
    for(int len = 2,M = 1;len <= N;M = len,len <<= 1){
        for(int L = 0,R = len - 1; R <= N;L += len,R += len){
            for(int k = L;k < L + M;k ++){
                if(opt == 1)a[k + M] = (a[k + M] + a[k]) % mo;
                else a[k + M] = (a[k + M] - a[k] + mo) % mo;
            }
        }
    }
}

void FWT_and(ll *a,int N,int opt){
    for(int len = 2,M = 1;len <= N;M = len,len <<= 1){
        for(int L = 0,R = len - 1; R <= N;L += len,R += len){
            for(int k = L;k < L + M;k ++){
                if(opt == 1)a[k] = (a[k] + a[k + M]) % mo;
                else a[k] = (a[k] - a[k + M] + mo) % mo;
            }
        }
    }
}
void FWT_xor(ll *a,int N,int opt){
    for(int len = 2,M = 1;len <= N;M = len,len <<= 1){
        for(int L = 0,R = len - 1; R <= N;L += len,R += len){
            for(int k = L;k < L + M;k ++){
                ll x = a[k],y = a[k + M];
                a[k] = (x + y) % mo;a[k + M] = (x - y + mo) % mo;
                if(opt == -1)a[k] = a[k] * inv2 % mo,a[k + M] = a[k + M] * inv2 % mo;
            }
        }
    }
}
```

生成函数

关键为形式幂级数与封闭形式互化

OGF

基本运算考虑两个序列 a, b 的普通生成函数，分别为 $F(x), G(x)$。那么有

$$F(x) \pm G(x) = \sum_n (a_n \pm b_n) x^n$$

因此 $F(x) \pm G(x)$ 是序列 $\langle a_n \pm b_n \rangle$ 的普通生成函数。考虑乘法运算，也就是卷积：

$$F(x)G(x) = \sum_n x^n \sum_{i=0}^{n} a_i b_{n-i}$$

因此 $F(x)G(x)$ 是序列 $\langle \sum_{i=0}^{n} a_i b_{n-i} \rangle$ 的普通生成函数。

常见互化手段求导，二项式定理展开

$\langle 1, p, p^2, p^3, p^4, \cdots \rangle$ 的生成函数 $F(x) = \sum_{n \geq 0} p^n x^n = \frac{1}{1-px}$

$\langle 1^{\underline{k}}, 2^{\underline{k}}, 3^{\underline{k}}, 4^{\underline{k}}, \cdots \rangle$ 的生成函数 $F(x) = \sum_{n \geq 0} (n+1)^{\underline{k}} x^n = \frac{k!}{(1-x)^{k+1}}$

$F(x) = \sum_{n \geq 0} \binom{m}{n} x^n = (1+x)^m$

$F(x) = \sum_{n \geq 0} \binom{m+n}{n} x^n = \frac{1}{(1-x)^{m+1}}$

斐波那契数列生成函数 $F(x) = \frac{x}{1-x-x^2} = \sum_{n \geq 0} (1 - 2^{n+1} + (n+1) \cdot 2^{n+1}) x^n$

EGF

在 OGF 的基础上考虑有序，形式上基本和泰勒展开等价

$$f(x) = f(x_0) + \frac{f'(x_0)}{1!}(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} \quad \forall x$$

$$\ln(1+x) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} x^n \quad \forall x \in (-1, 1]$$

- 三角函数：

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} \quad \forall x$$

$$\cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} \quad \forall x$$

$$\tan x = \sum_{n=1}^{\infty} \frac{B_{2n}(-4)^n(1-4^n)}{(2n)!} x^{2n-1} \quad \forall x : |x| < \frac{\pi}{2}$$

$$\sec x = \sum_{n=0}^{\infty} \frac{(-1)^n E_{2n}}{(2n)!} x^{2n} \quad \forall x : |x| < \frac{\pi}{2}$$

$$\arcsin x = \sum_{n=0}^{\infty} \frac{(2n)!}{4^n(n!)^2(2n+1)} x^{2n+1} \quad \forall x : |x| < 1$$

$$\arctan x = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1} \quad \forall x : |x| < 1$$

$$\arctan x = \frac{\pi \operatorname{sgn} x}{2} - \frac{1}{x} + \sum_{k=1}^{\infty} \frac{(-1)^k}{(2k+1)\, x^{2k+1}} \quad \forall x : |x| > 1$$

基本运算

指数生成函数的加减法与普通生成函数是相同的，也就是对应项系数相加。

考虑指数生成函数的乘法运算。对于两个序列 a, b，设它们的指数生成函数分别为 $\hat{F}(x), \hat{G}(x)$，那么

$$\begin{aligned}
\hat{F}(x)\hat{G}(x) &= \sum_{i \geq 0} a_i \frac{x^i}{i!} \sum_{j \geq 0} b_j \frac{x^j}{j!} && (1) \\
&= \sum_{n \geq 0} x^n \sum_{i=0}^{n} a_i b_{n-i} \frac{1}{i!(n-i)!} && (2) \\
&= \sum_{n \geq 0} \frac{x^n}{n!} \sum_{i=0}^{n} \binom{n}{i} a_i b_{n-i} && (3)
\end{aligned}$$

因此 $\hat{F}(x)\hat{G}(x)$ 是序列

$$\left\langle \sum_{i=0}^{n} \binom{n}{i} a_i b_{n-i} \right\rangle$$

的指数生成函数。

多项式 exp 组合意义：将 n 个互异元素分到若干**非空的无序集合**中，大小为 i 的集合内有 f_i 种方案，记最后的总方案数为 g_n。则两者的 EGF 满足 $G(x) = e^{F(x)}$。

集合幂级数

计算 $\prod(1 + x^{a_i})$ 这里为或卷积。

由于点值一定为 $1^x 2^{n-x}$ 的形式，所以可以通过一次 fwt 解出 x，然后再将点值序列 ifwt 回去解出答案。

计算 $\prod(x^U + x^{a_i})$ 这里为与卷积。

点值同样为 $1^x 2^{n-x}$ 的形式

计算 $\prod(1 + a_i x^i)$ 这里为异或卷积。

等价于对每个 x 求 $\prod(1 + (-1)^{x \oplus i} a_i)$，分治求解，最后再 ifwt 回去。$a_i$ 为定值则可套用上面的方式。

```
void solve(vector<int>&a,int N) {
    vector<int> A(N),B(N);
    for(int i = 0;i < N;i ++){
        A[i] = (1 + a[i]) % mo;
        B[i] = (1 - a[i] + mo) % mo;
    }
    for(int i = 1;i < N;i <<= 1)
        for(int len = i << 1,j = 0;j < N;j += len)
            for(int k = 0;k < i;k ++){
                int a0 = A[j + k],a1 = A[j + k + i];
                int b0 = B[j + k],b1 = B[j + k + i];
                A[j + k] = 1ll * a0 * a1 % mo;
                B[j + k] = 1ll * b0 * b1 % mo;
                A[j + k + i] = 1ll * a0 * b1 % mo;
                B[j + k + i] = 1ll * a1 * b0 % mo;
            }
    for(int i = 0;i < N;i ++)a[i] = A[i];
}
```

FFT

```
constexpr double PI = std::atan2(0, -1);
std::vector<int> rev;
std::vector<std::complex<double>> roots {0, 1};
void dft(std::vector<std::complex<double>> &a) {
    int n = a.size();
    if (int(rev.size()) != n) {
        int k = __builtin_ctz(n) - 1;
        rev.resize(n);
        for (int i = 0; i < n; ++i)
            rev[i] = rev[i >> 1] >> 1 | (i & 1) << k;
    }
    for (int i = 0; i < n; ++i)
        if (rev[i] < i)
            swap(a[i], a[rev[i]]);
    if (int(roots.size()) < n) {
        int k = __builtin_ctz(roots.size());
        roots.resize(n);
        while ((1 << k) < n) {
            std::complex<double> e = {cos(PI / (1 << k)), sin(PI / (1 << k))};
            for (int i = 1 << (k - 1); i < (1 << k); ++i) {
                roots[2 * i] = roots[i];
                roots[2 * i + 1] = roots[i] * e;
            }
            ++k;
        }
```

```cpp
    }
    for (int k = 1; k < n; k *= 2) {
        for (int i = 0; i < n; i += 2 * k) {
            for (int j = 0; j < k; ++j) {
                auto u = a[i + j], v = a[i + j + k] * roots[k + j];
                a[i + j] = u + v;
                a[i + j + k] = u - v;
            }
        }
    }
}
void idft(std::vector<std::complex<double>> &a) {
    int n = a.size();
    reverse(a.begin() + 1, a.end());
    dft(a);
    for (int i = 0; i < n; ++i)
        a[i] /= n;
}
std::vector<ll> operator*(std::vector<ll> a, std::vector<ll> b) {
    int sz = 1, tot = a.size() + b.size() - 1;
    while (sz < tot)
        sz *= 2;
    std::vector<std::complex<double>> ca(sz), cb(sz);
    //copy(a.begin(), a.end(), ca.begin());
    //copy(b.begin(), b.end(), cb.begin());
    for(int i = 0;i < sz;i ++){
        if(i < a.size())ca[i].real(a[i]);
        if(i < b.size())ca[i].imag(b[i]);
    }
    dft(ca);
    //dft(cb);
    for (int i = 0; i < sz; ++i)
        ca[i] *= ca[i];
    idft(ca);
    a.resize(tot);
    for (int i = 0; i < tot; ++i)
        a[i] = std::floor(ca[i].imag() / 2 + 0.5);
    return a;
}

```

多项式全家桶

```cpp
using namespace std;
using i64 = long long;
constexpr int P = 998244353;
int norm(int x) {
    if (x < 0) x += P;
    if (x >= P) x -= P;
    return x;
}
template<class T>
T qp(T a, int b) {
    T res = 1;
    for (; b; b /= 2, a *= a) {
        if (b % 2) {
            res *= a;
        }
    }
    return res;
}
struct Z{
    int x;
    Z(): x{} {}
    Z(int x) : x{norm(x)} {}
    Z(i64 x) : x{norm(int(x % P))} {}
    friend std::istream &operator>>(std::istream &is,Z &a) {
        i64 v;
        is >> v;
        a = Z(v);
        return is;
```

```
    }
    friend std::ostream &operator<<(std::ostream &os, const Z &a) {
        return os << a.x;
    }
    Z inv() const {
        return qp(Z(x),P - 2);
    }
};
bool operator==(const Z a,const Z b) { return a.x == b.x; }
bool operator!=(const Z a,const Z b) { return a.x != b.x; }
Z operator+(const Z a, const Z b) { return norm(a.x + b.x); }
Z operator-(const Z a, const Z b) { return norm(a.x + P - b.x); }
Z operator-(const Z x) { return x.x ? P - x.x : 0; }
Z operator*(const Z a, const Z b) { return i64(a.x) * b.x % P; }
Z operator/(const Z a, const Z b) { return a * b.inv(); }
Z &operator+=(Z &a, const Z b) { return a = a + b; }
Z &operator-=(Z &a, const Z b) { return a = a - b; }
Z &operator*=(Z &a, const Z b) { return a = a * b; }
Z &operator/=(Z &a, const Z b) { return a = a / b; }

std::vector<int> rev;
std::vector<Z> roots{0, 1};
void dft(std::vector<Z> &a) {
    int n = a.size();
    if (int(rev.size()) != n) {
        int k = __builtin_ctz(n) - 1;
        rev.resize(n);
        for (int i = 0; i < n; i++) {
            rev[i] = rev[i >> 1] >> 1 | (i & 1) << k;
        }
    }
    for (int i = 0; i < n; i++) if (rev[i] < i) swap(a[i], a[rev[i]]);
    if (int(roots.size()) < n) {
        int k = __builtin_ctz(roots.size());
        roots.resize(n);
        while ((1 << k) < n) {
            Z e = qp(Z(3), (P - 1) >> (k + 1));
            for (int i = 1 << (k - 1); i < (1 << k); i++) {
                roots[2 * i] = roots[i];
                roots[2 * i + 1] = roots[i] * e;
            }
            k++;
        }
    }
    for (int k = 1; k < n; k *= 2) {
        for (int i = 0; i < n; i += 2 * k) {
            for (int j = 0; j < k; j++) {
                Z u = a[i + j];
                Z v = a[i + j + k] * roots[k + j];
                a[i + j] = u + v;
                a[i + j + k] = u - v;
            }
        }
    }
}
void idft(std::vector<Z> &a) {
    int n = a.size();
    std::reverse(a.begin() + 1, a.end());
    dft(a);
    Z inv = (1 - P) / n;
    for (int i = 0; i < n; i++) {
        a[i] *= inv;
    }
}
struct Poly {
    std::vector<Z> a;
    Poly() {}
    Poly(const std::vector<Z> &a) : a(a) {}
    Poly(const std::initializer_list<Z> &a) : a(a) {}
    int size() const {
        return a.size();
```

```
    }
    void resize(int n) {
        a.resize(n);
    }
    Z operator[](int idx) const {
        if (idx < size()) {
            return a[idx];
        } else {
            return 0;
        }
    }
    Z &operator[](int idx) {
        return a[idx];
    }
    Poly mulxk(int k) const {
        auto b = a;
        b.insert(b.begin(), k, 0);
        return Poly(b);
    }
    Poly modxk(int k) const {
        k = std::min(k, size());
        return Poly(std::vector<Z>(a.begin(), a.begin() + k));
    }
    Poly divxk(int k) const {
        if (size() <= k) {
            return Poly();
        }
        return Poly(std::vector<Z>(a.begin() + k, a.end()));
    }
    friend Poly operator+(const Poly &a, const Poly &b) {
        std::vector<Z> res(std::max(a.size(), b.size()));
        for (int i = 0; i < int(res.size()); i++) {
            res[i] = a[i] + b[i];
        }
        return Poly(res);
    }
    friend Poly operator-(const Poly &a, const Poly &b) {
        std::vector<Z> res(std::max(a.size(), b.size()));
        for (int i = 0; i < int(res.size()); i++) {
            res[i] = a[i] - b[i];
        }
        return Poly(res);
    }
    friend Poly operator*(Poly a, Poly b) {
        if (a.size() == 0 || b.size() == 0) {
            return Poly();
        }
        int sz = 1, tot = a.size() + b.size() - 1;
        while (sz < tot) {
            sz *= 2;
        }
        a.a.resize(sz);
        b.a.resize(sz);
        dft(a.a);
        dft(b.a);
        for (int i = 0; i < sz; ++i) {
            a.a[i] = a[i] * b[i];
        }
        idft(a.a);
        a.resize(tot);
        return a;
    }
    friend Poly operator*(Z a, Poly b) {
        for (int i = 0; i < int(b.size()); i++) {
            b[i] *= a;
        }
        return b;
    }
    friend Poly operator*(Poly a, Z b) {
        for (int i = 0; i < int(a.size()); i++) {
            a[i] *= b;
```

```
        }
        return a;
    }
    Poly &operator+=(Poly b) {
        return (*this) = (*this) + b;
    }
    Poly &operator-=(Poly b) {
        return (*this) = (*this) - b;
    }
    Poly &operator*=(Poly b) {
        return (*this) = (*this) * b;
    }
    Poly deriv() const {
        if (a.empty()) {
            return Poly();
        }
        std::vector<Z> res(size() - 1);
        for (int i = 0; i < size() - 1; ++i) {
            res[i] = (i + 1) * a[i + 1];
        }
        return Poly(res);
    }//求导
    Poly integr() const {
        std::vector<Z> res(size() + 1);
        for (int i = 0; i < size(); ++i) {
            res[i + 1] = a[i] / (i + 1);
        }
        return Poly(res);
    }//积分
    Poly inv(int m) const {
        Poly x{a[0].inv()};
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x * (Poly{2} - modxk(k) * x)).modxk(k);
        }
        return x.modxk(m);
    }//求逆
    Poly log(int m) const {
        return (deriv() * inv(m)).integr().modxk(m);
    }
    Poly exp(int m) const {
        Poly x{1};
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x * (Poly{1} - x.log(k) + modxk(k))).modxk(k);
        }
        return x.modxk(m);
    }
    Poly pow(int k, int m) const {
        int i = 0;
        while (i < size() && a[i] == 0) {
            i++;
        }
        if (i == size() || 1LL * i * k >= m) {
            return Poly(std::vector<Z>(m));
        }
        Z v = a[i];
        auto f = divxk(i) * v.inv();
        return (f.log(m - i * k) * k).exp(m - i * k).mulxk(i * k) * qp(v, k);
//         Poly res = {1};
//         Poly base = *this;
//         while(k){
//             if(k & 1) res = res * base;
//             if(res.size() > m)res.modxk(m);
//             base = base * base;
//             if(base.size() > m)base.modxk(m);
//             k >>= 1;
//         }
//         return res;
```

```
    }
    Poly sqrt(int m) const {
        Poly x{1};
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x + (modxk(k) * x.inv(k)).modxk(k)) * ((P + 1) / 2);
        }
        return x.modxk(m);
    }
    Poly mulT(Poly b) const {
        if (b.size() == 0) {
            return Poly();
        }
        int n = b.size();
        std::reverse(b.a.begin(), b.a.end());
        return ((*this) * b).divxk(n - 1);
    }
    std::vector<Z> eval(std::vector<Z> x) const {
        if (size() == 0) {
            return std::vector<Z>(x.size(), 0);
        }
        const int n = std::max(int(x.size()), size());
        std::vector<Poly> q(4 * n);
        std::vector<Z> ans(x.size());
        x.resize(n);
        std::function<void(int, int, int)> build = [&](int p, int l, int r) {
            if (r - l == 1) {
                q[p] = Poly{1, -x[l]};
            } else {
                int m = (l + r) / 2;
                build(2 * p, l, m);
                build(2 * p + 1, m, r);
                q[p] = q[2 * p] * q[2 * p + 1];
            }
        };
        build(1, 0, n);
        std::function<void(int, int, int, const Poly &)> work = [&](int p, int l, int r, const Poly &num) {
            if (r - l == 1) {
                if (l < int(ans.size())) {
                    ans[l] = num[0];
                }
            } else {
                int m = (l + r) / 2;
                work(2 * p, l, m, num.mulT(q[2 * p + 1]).modxk(m - l));
                work(2 * p + 1, m, r, num.mulT(q[2 * p]).modxk(r - m));
            }
        };
        work(1, 0, n, mulT(q[1].inv(n)));
        return ans;
    }//多点求值
};
Poly S2_row;
void S2_row_init(int n) {
    vector<Z> f(n + 1), g(n + 1);
    for (int i = 0; i <= n; i ++) {
        f[i] = qp(Z(i), n) * inv[i];
        g[i] = Z(i & 1 ? -1 : 1) * inc[i];
    }
    S2_row = Poly(f) * Poly(g);
}
Poly S2_col;
void S2_col_init(int n, int k) {
    n ++;
    vector<Z> f(n);
    for (int i = 1; i < n; i ++) {
        f[i] = inv[i];
    }
    auto ans = Poly(f).pow(k, n);
    S2_col.resize(n + 1);
    for (int i = 0; i < n; i ++) {
```

```cpp
        S2_col[i] = ans[i] * fc[i] * inv[k];
    }
}
Poly Bell;
void Bell_init(int n) {
    vector<Z> f(n + 1);
    for (int i = 1; i <= n; i ++) {
        f[i] = inv[i];
    }
    auto ans = Poly(f).exp(n + 1);
    Bell.resize(n + 1);
    for (int i = 0; i <= n; i ++) {
        Bell[i] = ans[i] * fc[i];
    }
}
```

计算几何

tips:

直线上两点整点坐标范围在 $[-10^6, 10^6]$，直线交点范围在 $[-10^{18}, 10^{18}]$

Pick 定理：给定顶点均为整点的简单多边形，其面积 A 和内部格点数目 i、边上格点数目 b 的关系为 $A = i + \frac{b}{2} - 1$

曼哈顿转切比雪夫：(x, y) 变为 $(\frac{x+y}{2}, \frac{x-y}{2})$

二维计算几何

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
constexpr double eps = 1e-7;
constexpr double PI = acos(-1);
constexpr double inf = 1e9;
struct Point { double x, y; };        // 点
using Vec = Point;                    // 向量
struct Line { Point P; Vec v; };      // 直线（点向式），射线时为 A->B
struct Seg { Point A, B; };           // 线段（存两个端点）
struct Circle { Point O; double r; }; // 圆（存圆心和半径）
using Points = std::vector<Point>;
using ConvexHull = std::vector<Point>;
const Point O = {0, 0};                        // 原点
const Line Ox = {O, {1, 0}}, Oy = {O, {0, 1}}; // 坐标轴

bool eq(double a, double b) { return abs(a - b) < eps; } // ==
bool gt(double a, double b) { return a - b > eps; }      // >
bool lt(double a, double b) { return a - b < -eps; }     // <
bool ge(double a, double b) { return a - b > -eps; }     // >=
bool le(double a, double b) { return a - b < eps; }      // <=
Vec operator + (const Vec &a,const Vec &b){return (Vec){a.x + b.x,a.y + b.y};}
Vec operator - (const Vec &a,const Vec &b){return (Vec){a.x - b.x,a.y - b.y};}
Vec operator * (const Vec &a,const double &b){return (Vec){b * a.x,b * a.y};}
Vec operator * (const double &a,const Vec &b){return (Vec){a * b.x,a * b.y};}
Vec operator / (const Vec &a,const double &b){return (Vec){a.x / b,a.y / b};}
double operator * (const Point &a,const Point &b){return a.x * b.x + a.y * b.y;}// dot // 点乘
double operator ^ (const Point &a,const Point &b){return a.x * b.y - a.y * b.x;}// cross  // 叉乘
bool operator < (const Point& a, const Point& b) {return a.x < b.x || (a.x == b.x && a.y < b.y);}
double len(const Vec &a){return sqrt(a * a);}

ll cross(Point a,Point b){return (ll)a.x * (ll)b.y - (ll)a.y * (ll)b.x;}
ll dot(Point a,Point b){return (ll)a.x * (ll)b.x + (ll)a.y * (ll)b.y;}

double angle(const Vec &a,const Vec &b){return acos(a * b / len(a)/len(b));}

double Polar_angle(Vec &v){return atan2(v.y,v.x);}

int sgn(double x){
    if(fabs(x) < eps)
        return 0;
```

```cpp
    if(x < 0)
        return -1;
    return 1;
}

Vec r90a(Vec v) { return {-v.y, v.x}; }  // 逆时针旋转 90 度的向量
Vec r90c(Vec v) { return {v.y, -v.x}; }  // 顺时针旋转 90 度的向量

// 两向量的夹角余弦
// DEPENDS len, V*V
double cos_t(Vec u, Vec v) { return u * v / len(u) / len(v); }

// 归一化向量（与原向量方向相同的单位向量）
// DEPENDS len
Vec norm(Vec v) { return {v.x / len(v), v.y / len(v)}; }

// 与原向量平行且横坐标大于等于 0 的单位向量
// DEPENDS d*V, len
Vec pnorm(Vec v) { return (v.x < 0 ? -1 : 1) / len(v) * v; }

// 线段的方向向量
// DEPENDS V-V
// NOTE 直线的方向向量直接访问属性 v
Vec dvec(Seg l) { return l.B - l.A; }
//----------------------------------------------------------//
Line line(Point A, Point B) { return {A, B - A}; }

// 斜截式直线
Line line(double k, double b) { return {{0, b}, {1, k}}; }

// 点斜式直线
Line line(Point P, double k) { return {P, {1, k}}; }

// 线段所在直线
// DEPENDS V-V
Line line(Seg l) { return {l.A, l.B - l.A}; }

// 给定直线的横坐标求纵坐标
// NOTE 请确保直线不与 y 轴平行
double at_x(Line l, double x) { return l.P.y + (x - l.P.x) * l.v.y / l.v.x; }

// 给定直线的纵坐标求横坐标
// NOTE 请确保直线不与 x 轴平行
double at_y(Line l, double y) { return l.P.x - (y + l.P.y) * l.v.x / l.v.y; }

// 点到直线的垂足
// DEPENDS V-V, V*V, d*V
Point pedal(Point P, Line l) { return l.P - (l.P - P) * l.v / (l.v * l.v) * l.v; }

// 过某点作直线的垂线
// DEPENDS r90c
Line perp(Line l, Point P) { return {P, r90c(l.v)}; }

// 角平分线
// DEPENDS V+V, len, norm
Line bisec(Point P, Vec u, Vec v) { return {P, norm(u) + norm(v)}; }

//seg-----------------------------------------//

// 线段的方向向量
// DEPENDS V-V
// NOTE 直线的方向向量直接访问属性 v
//Vec dvec(Seg l) { return l.B - l.A; }

// 线段中点
Point midp(Seg l) { return {(l.A.x + l.B.x) / 2, (l.A.y + l.B.y) / 2}; }

// 线段中垂线
// DEPENDS r90c, V-V, midp
Line perp(Seg l) { return {midp(l), r90c(l.B - l.A)}; }
//------------------------------------------------------------------//
```

```cpp
// 向量是否互相垂直
// DEPENDS eq, V*V
bool verti(Vec u, Vec v) { return eq(u * v, 0); }

// 向量是否互相平行
// DEPENDS eq, V^V
bool paral(Vec u, Vec v) { return eq(u ^ v, 0); }

// 向量是否与 x 轴平行
// DEPENDS eq
bool paral_x(Vec v) { return eq(v.y, 0); }

// 向量是否与 y 轴平行
// DEPENDS eq
bool paral_y(Vec v) { return eq(v.x, 0); }

// 点是否在直线上
// DEPENDS eq
bool on(Point P, Line l) { return eq((P.x - l.P.x) * l.v.y, (P.y - l.P.y) * l.v.x); }


// 点是否在射线上
// DEPENDS eq
bool on_ray(Point P, Line l) { return on(P,l) && ((P - l.P) * l.v) >= 0; }

// 点是否在线段上
// DEPENDS eq, len, V-V
bool on(Point P, Seg l) { return eq(len(P - l.A) + len(P - l.B), len(l.A - l.B)); }

// 两个点是否重合
// DEPENDS eq
bool operator==(Point A, Point B) { return eq(A.x, B.x) && eq(A.y, B.y); }

// 两条直线是否重合
// DEPENDS eq, on(L)
bool operator==(Line a, Line b) { return on(a.P, b) && on(a.P + a.v, b); }

// 两条线段是否重合
// DEPENDS eq, P==P
bool operator==(Seg a, Seg b) { return (a.A == b.A && a.B == b.B) || (a.A == b.B && a.B == b.A); }

// 以横坐标为第一关键词、纵坐标为第二关键词比较两个点
// DEPENDS eq, lt
//bool operator<(Point A, Point B) { return lt(A.x, B.x) || (eq(A.x, B.x) && lt(A.y, B.y)); }

// 直线与圆是否相切
// DEPENDS eq, V^V, len
bool tangency(Line l, Circle C) { return eq(abs((C.O ^ l.v) - (l.P ^ l.v)), C.r * len(l.v)); }

// 圆与圆是否相切
// DEPENDS eq, V-V, len
bool tangency(Circle C1, Circle C2) { return eq(len(C1.O - C2.O), C1.r + C2.r); }
//-------------------------------------------------//
// 两点间的距离
// DEPENDS len, V-V
double dis(Point A, Point B) { return len(A - B); }

// 点到直线的距离
// DEPENDS V^V, len
double dis(Point P, Line l) { return abs((P ^ l.v) - (l.P ^ l.v)) / len(l.v); }

// 点到线段的距离
double dis(Point P,Seg l) {
    if(((P - l.A) * (l.B - l.A)) < 0 || ((P - l.B) * (l.A - l.B)) < 0){
        return min(dis(P,l.A),dis(P,l.B));
    }else {
        Line ll = line(l);
        return dis(P,l);
    }
}
// 平行直线间的距离
```

```cpp
// DEPENDS d*V, V^V, len, pnorm
// NOTE 请确保两直线是平行的
double dis(Line a, Line b) { return abs((a.P ^ pnorm(a.v)) - (b.P ^ pnorm(b.v))); }


// 平移
// DEPENDS V+V
Line operator+(Line l, Vec v) { return {l.P + v, l.v}; }
Seg operator+(Seg l, Vec v) { return {l.A + v, l.B + v}; }


// 旋转 逆时针
// DEPENDS V+V, V-V
Point rotate(Point P, double rad) { return {cos(rad) * P.x - sin(rad) * P.y, sin(rad) * P.x + cos(rad) * P.y}; }
Point rotate(Point P, double rad, Point C) { return C + rotate(P - C, rad); }                       // DEPENDS ^1
Line rotate(Line l, double rad, Point C = O) { return {rotate(l.P, rad, C), rotate(l.v, rad)}; }  // DEPENDS ^1, ^2
Seg rotate(Seg l, double rad, Point C = O) { return {rotate(l.A, rad, C), rotate(l.B, rad, C)}; } // DEPENDS ^1, ^2

// 直线与直线交点
// DEPENDS eq, d*V, V*V, V+V, V^V
Points inter(Line a, Line b){
    double c = a.v ^ b.v;
    if (eq(c, 0)) {return {};}
    Vec v = 1 / c * Vec{a.P ^ (a.P + a.v), b.P ^ (b.P + b.v)};
    return {{v * Vec{-b.v.x, a.v.x}, v * Vec{-b.v.y, a.v.y}}};
}

// 线段与线段是否相交
bool cross_seg(Seg A,Seg B){
    Point a = A.A,b = A.B,c = B.A,d = B.B;
    double c1 = (b - a) ^ (c - a),c2 = (b - a) ^ (d - a);
    double d1 = (d - c) ^ (a - c),d2 = (d - c) ^ (b - c);
    return sgn(c1) * sgn(c2) < 0 && sgn(d1) * sgn(d2) < 0;
}

// 直线与线段相交 => 直线与直线相交 + 点是否在线段上
// bool cross_line_seg(Line A,Seg B){
//     Line BB = {B.A,B.B};
//     Points tmp = inter(A,BB);
//     if(tmp.size() == 0)return false;
//     return on(tmp[0],B);
// }
bool cross_line_seg(Line A,Seg B){
    if(fabs(A.v ^ (B.A - B.B)) < eps)return false;// 平行
    Vec v1 = B.A - A.P,v2 = B.B - A.P;
    if((v2 ^ v1) < 0){
        swap(v1,v2);
    }else if(fabs(v2 ^ v1) < eps){
        if((v1 * v2) <= 0)return true;
        else return false;
    }// 保证 v2 在 v1 下面
    int d1 = sgn(A.v ^ v1);
    int d2 = sgn(A.v ^ v2);
    if(d1 * d2 <= 0)return true;
    return false;
}

// 射线与射线交
bool cross_ray_ray(Line A,Line B){
    Points tmp = inter(A,B);
    if(tmp.size() == 0)return false;//注意重合
    int d1 = sgn((tmp[0] - A.P) * A.v);
    int d2 = sgn((tmp[0] - B.P) * B.v);
    return d1 >= 0 && d2 >= 0;
}

// 射线与线段交
// bool cross_ray_seg(Line A,Seg B){
//     Line BB = {B.A,B.B};
//     Points tmp = inter(A,BB);
//     if(tmp.size() == 0)return false;//注意重合
```

```cpp
//     int d = sgn((tmp[0] - A.P) * A.v);
//     return on(tmp[0],B) && d >= 0;
// }
bool cross_ray_seg(Line A,Seg B){
    if(fabs(A.v ^ (B.A - B.B)) < eps)return false;// 平行
    Vec v1 = B.A - A.P,v2 = B.B - A.P;
    if((v2 ^ v1) < 0){
        swap(v1,v2);
    }else if(fabs(v2 ^ v1) < eps){
        if((v1 * v2) <= 0)return true;
        else return false;
    }// 保证 v2 在 v1 下面
    int d1 = sgn(A.v ^ v1);
    int d2 = sgn(A.v ^ v2);
    if(d1 >= 0 && d2 <= 0)return true;
    return false;
}

// 射线与直线交
bool cross_ray_line(Line A,Line B){ // A 为射线
    Points tmp = inter(A,B);
    if(tmp.size() == 0)return false;
    int d = sgn((tmp[0] - A.P) * A.v);
    return d >= 0;
}

// 直线与圆交点
// DEPENDS eq, gt, V+V, V-V, V*V, d*V, len, pedal
std::vector<Point> inter(Line l, Circle C){
    Point P = pedal(C.O, l);
    double h = len(P - C.O);
    if (gt(h, C.r)) return {};
    if (eq(h, C.r)) return {P};
    double d = sqrt(C.r * C.r - h * h);
    Vec vec = d / len(l.v) * l.v;
    return {P + vec, P - vec};
}

// 圆与圆的交点
// DEPENDS eq, gt, V+V, V-V, d*V, len, r90c
std::vector<Point> inter(Circle C1, Circle C2){
    Vec v1 = C2.O - C1.O, v2 = r90c(v1);
    double d = len(v1);
    if (gt(d, C1.r + C2.r) || gt(abs(C1.r - C2.r), d)) return {};
    if (eq(d, C1.r + C2.r) || eq(d, abs(C1.r - C2.r))) return {C1.O + C1.r / d * v1};
    double a = ((C1.r * C1.r - C2.r * C2.r) / d + d) / 2;
    double h = sqrt(C1.r * C1.r - a * a);
    Vec av = a / len(v1) * v1, hv = h / len(v2) * v2;
    return {C1.O + av + hv, C1.O + av - hv};
}



// 三角形的重心
Point barycenter(Point A, Point B, Point C){
    return {(A.x + B.x + C.x) / 3, (A.y + B.y + C.y) / 3};
}

// 三角形的外心
// DEPENDS r90c, V*V, d*V, V-V, V+V
// NOTE 给定圆上三点求圆，要先判断是否三点共线
Point circumcenter(Point A, Point B, Point C){
    double a = A * A, b = B * B, c = C * C;
    double d = 2 * (A.x * (B.y - C.y) + B.x * (C.y - A.y) + C.x * (A.y - B.y));
    return 1 / d * r90c(a * (B - C) + b * (C - A) + c * (A - B));
}

// 三角形的内心
// DEPENDS len, d*V, V-V, V+V
Point incenter(Point A, Point B, Point C){
    double a = len(B - C), b = len(A - C), c = len(A - B);
```

```
    double d = a + b + c;
    return 1 / d * (a * A + b * B + c * C);
}

// 三角形的垂心
// DEPENDS V*V, d*V, V-V, V^V, r90c
Point orthocenter(Point A, Point B, Point C){
    double n = B * (A - C), m = A * (B - C);
    double d = (B - C) ^ (A - C);
    return 1 / d * r90c(n * (C - B) - m * (C - A));
}


// Graham 扫描法

// DEPENDS eq, lt, cross, V-V, P<P

double theta(Point p) { return p == O ? -1 / 0. : atan2(p.y, p.x); } // 求极角
void psort(Points &ps, Point c = O) {                                  // 极角排序
    sort(ps.begin(), ps.end(), [&](auto a, auto b) {
        return lt(theta(a - c), theta(b - c));
    });
}

//极角排序
int qua(const Point &P){
  if(P.x == 0 && P.y == 0) return 0;
  if(P.x >= 0 && P.y >= 0) return 1;
  if(P.x < 0&& P.y >= 0) return 2;
  if(P.x<0 && P.y < 0) return 3;
  if(P.x >= 0 && P.y < 0) return 4;
  exit(-1);
}
void psort(Points &ps, Point c = O){                                // 极角排序
    stable_sort(ps.begin(), ps.end(), [&](auto p1, auto p2) {
        return qua(p1 - c) < qua(p2 - c) || qua(p1 - c) == qua(p2 - c) && gt((Point)(p1 - c) ^ (Point)(p2 - c), 0);
    });
}

// 检查向量夹角 acb 小于 180
bool check1(Point a,Point b,Point c){//
    ll d = (a - c) ^ (b - c);
    if(d > 0)return true;
    if(d < 0)return false;
    return (a - c) * (b - c) > 0;
}



bool check(Point p, Point q, Point r){ // 检查三个点组成的两个向量的旋转方向是否为逆时针
    return lt(0, (q - p) ^ (r - q));
}
ConvexHull Andrew(Points &ps){
    if(ps.size() == 1){
        return ps;
    }
    sort(ps.begin(), ps.end());
    std::vector<int> I{0}, used(ps.size());
    for (int i = 1; i < ps.size(); i++){
        //std::cout << ps[i].x << " " <<ps[i].y <<"\n";
        while (I.size() > 1 && !check(ps[I[I.size() - 2]], ps[I.back()], ps[i]))
            used[I.back()] = 0, I.pop_back();
        used[i] = 1, I.push_back(i);
    }
    int tmp = I.size();
    for (int i = ps.size() - 2; i >= 0; i--){
        if (used[i])
            continue;
        while (I.size() > tmp && !check(ps[I[I.size() - 2]], ps[I.back()], ps[i]))
            used[I.back()] = 0, I.pop_back();
```

```cpp
            used[i] = 1, I.push_back(i);
        }
        Points H;
        for (int i = 0; i < I.size() - 1; i++)
            H.push_back(ps[I[i]]);
        return H;
    }//逆时针
    ConvexHull chull(Points &ps){
        psort(ps, *min_element(ps.begin(), ps.end())); // 以最左下角的点为极角排序
        Points H{ps[0]};
        for (int i = 1; i < ps.size(); i++){
            while (H.size() > 1 && !check(H[H.size() - 2], H.back(), ps[i]))
                H.pop_back();
            H.push_back(ps[i]);
        }
        return H;
    }
    ConvexHull operator+(const ConvexHull &A,const ConvexHull B){
        int n = A.size();
        int m = B.size();
        std::vector<Point> v1(n),v2(m);
        for(int i = 0;i < n;i ++){
            v1[i] = A[(i + 1) % n] - A[i];
        }
        for(int i = 0;i < m;i ++){
            v2[i] = B[(i + 1) % m] - B[i];
        }
        ConvexHull C;
        C.push_back(A[0] + B[0]);
        int p1 = 0,p2 = 0;
        while(p1 < n && p2 < m){
            C.push_back(C.back() + ((v1[p1] ^ v2[p2]) >= 0 ? v1[p1 ++] : v2[p2 ++]));
        }// 对上凸壳做闵可夫斯基和时将 >= 改为 <= 并且合并凸包时不需要排序
        while(p1 < n)C.push_back(C.back() + v1[p1 ++]);
        while(p2 < m)C.push_back(C.back() + v2[p2 ++]);
        C = chull(C);
        return C;
    }
    void test(Points a,Point b){
        int n = a.size();
        int r = 0;
        for(int l = 0;l < n;l ++){
            auto nxt = [&](int x){
                return (x + 1) % n;
            };
            while(nxt(r) != l && check(a[l],a[nxt(r)],b)){
                // b 为轴点
                r = nxt(r);

                if(l == r)break;
            }
        }//极角排序 转半平面
    }

    // 半平面交
    int sgn(Point a) {
        return a.y > 0 || (a.y == 0 && a.x > 0) ? 1 : -1;
    }
    bool pointOnLineLeft(Point p, Line l) {
        return (l.v ^ (p - l.P)) > eps;
    }
    Point lineIntersection(Line l1, Line l2) {
        return l1.P + l1.v * (cross(l2.v, l1.P - l2.P) / cross(l2.v, O - l1.v));
    }
    std::vector<Point> hp(std::vector<Line> lines) {
        std::sort(lines.begin(), lines.end(), [&](auto l1, auto l2) {
            auto d1 = l1.v;
            auto d2 = l2.v;

            if (sgn(d1) != sgn(d2)) {
                return sgn(d1) == 1;
```

```
        }

        return cross(d1, d2) > 0;
    });
    std::deque<Line> ls;
    std::deque<Point> ps;
    for (auto l : lines) {
        if (ls.empty()) {
            ls.push_back(l);
            continue;
        }
        while (!ps.empty() && !pointOnLineLeft(ps.back(), l)) {
            ps.pop_back();ls.pop_back();
        }

        while (!ps.empty() && !pointOnLineLeft(ps[0], l)) {
            ps.pop_front();ls.pop_front();
        }
        if (fabs(cross(l.v, ls.back().v)) < eps) {

            if ((l.v * ls.back().v) > eps) {
                //continue;
                if (!pointOnLineLeft(ls.back().P, l)) {
                    assert(ls.size() == 1);
                    ls[0] = l;
                }
                continue;
            }
            return {};
        }
        auto now = inter(ls.back(),l);
        ps.push_back(now[0]);
        // ps.push_back(lineIntersection(ls.back(), l));
        ls.push_back(l);
    }

    while (!ps.empty() && !pointOnLineLeft(ps.back(), ls[0])) {
        ps.pop_back();ls.pop_back();
    }
    if (ls.size() <= 2) {
        return {};
    }
    auto now = inter(ls[0],ls.back());
    ps.push_back(now[0]);
    // ps.push_back(lineIntersection(ls[0], ls.back()));
    return std::vector(ps.begin(), ps.end());
}

// int sta[N], top;  // 将凸包上的节点编号存在栈里，第一个和最后一个节点编号相同
// bool is[N];

// ll pf(ll x) { return x * x; }

// ll dis(int p, int q) { return pf(a[p].x - a[q].x) + pf(a[p].y - a[q].y); }

// ll sqr(int p, int q, int y) { return abs((a[q] - a[p]) * (a[y] - a[q])); }

// ll mx;

// void get_longest() {  // 求凸包直径
//   int j = 3;
//   if (top < 4) {
//     mx = dis(sta[1], sta[2]);
//     return;
//   }
//   for (int i = 1; i <= top; ++i) {
//     while (sqr(sta[i], sta[i + 1], sta[j]) <=
//            sqr(sta[i], sta[i + 1], sta[j % top + 1]))
//       j = j % top + 1;
//     mx = max(mx, max(dis(sta[i + 1], sta[j]), dis(sta[i], sta[j])));
//   }
```

```
// }


```

动态凸包

```
struct Item {
    P p;
    mutable P vec;
    int q = 0;
};

bool operator<(const Item &a, const Item &b) {
    if (!b.q) {
        return a.p.x < b.p.x;
    }
    return dot(a.vec, b.p) > 0;
}

struct Hull {
    std::set<Item> s;
    i128 dx = 0;
    i128 dy = 0;
};

void print(const Hull &h) {
    for (auto it : h.s) {
        std::cerr << "(" << i64(it.p.x + h.dx) << ", " << i64(it.p.y + h.dy) << ") ";
    }
    std::cerr << "\n";
}

constexpr i64 inf = 2E18;

void insert(Hull &h, P p) {
    p.x -= h.dx;
    p.y -= h.dy;
    h.s.insert({p});
    auto it = h.s.lower_bound({p});
    if (it != h.s.end() && it->p.x == p.x) {
        if (it->p.y > p.y) {
            return;
        }
        it = h.s.erase(it);
    }
    if (it != h.s.begin() && it != h.s.end()
        && cross(p - std::prev(it)->p, it->p - p) >= 0) {
        return;
    }
    it = h.s.insert({p}).first;
    auto r = std::next(it);
    if (r != h.s.end()) {
        while (cross(r->p - p, r->vec) >= 0) {
            r = h.s.erase(r);
        }
        it->vec = r->p - p;
    } else {
        it->vec = P(0, -inf);
    }

    if (it != h.s.begin()) {
        auto l = std::prev(it);
        while (l != h.s.begin()) {
            auto a = std::prev(l);
            if (cross(a->vec, p - l->p) < 0) {
                break;
            }
            h.s.erase(l);
            l = a;
        }
        l->vec = p - l->p;
```

```
    }
}

i64 query(const Hull &h, i64 x) {
    if (h.s.empty()) {
        return 0LL;
    }
    auto it = h.s.lower_bound({P(x, 1), P{}, 1});
    assert(it != h.s.end());
    auto p = it->p;
    p.x += h.dx;
    p.y += h.dy;
    return p.x * x + p.y;
}
```

最小圆覆盖

```
int n;
double r;

struct point {
  double x, y;
} p[100005], o;

double sqr(double x) { return x * x; }

double dis(point a, point b) { return sqrt(sqr(a.x - b.x) + sqr(a.y - b.y)); }

bool cmp(double a, double b) { return fabs(a - b) < 1e-8; }

point geto(point a, point b, point c) {
  double a1, a2, b1, b2, c1, c2;
  point ans;
  a1 = 2 * (b.x - a.x), b1 = 2 * (b.y - a.y),
  c1 = sqr(b.x) - sqr(a.x) + sqr(b.y) - sqr(a.y);
  a2 = 2 * (c.x - a.x), b2 = 2 * (c.y - a.y),
  c2 = sqr(c.x) - sqr(a.x) + sqr(c.y) - sqr(a.y);
  if (cmp(a1, 0)) {
    ans.y = c1 / b1;
    ans.x = (c2 - ans.y * b2) / a2;
  } else if (cmp(b1, 0)) {
    ans.x = c1 / a1;
    ans.y = (c2 - ans.x * a2) / b2;
  } else {
    ans.x = (c2 * b1 - c1 * b2) / (a2 * b1 - a1 * b2);
    ans.y = (c2 * a1 - c1 * a2) / (b2 * a1 - b1 * a2);
  }
  return ans;
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) scanf("%lf%lf", &p[i].x, &p[i].y);
  for (int i = 1; i <= n; i++) swap(p[rand() % n + 1], p[rand() % n + 1]);
  o = p[1];
  for (int i = 1; i <= n; i++) {
    if (dis(o, p[i]) < r || cmp(dis(o, p[i]), r)) continue;
    o.x = (p[i].x + p[1].x) / 2;
    o.y = (p[i].y + p[1].y) / 2;
    r = dis(p[i], p[1]) / 2;
    for (int j = 2; j < i; j++) {
      if (dis(o, p[j]) < r || cmp(dis(o, p[j]), r)) continue;
      o.x = (p[i].x + p[j].x) / 2;
      o.y = (p[i].y + p[j].y) / 2;
      r = dis(p[i], p[j]) / 2;
      for (int k = 1; k < j; k++) {
        if (dis(o, p[k]) < r || cmp(dis(o, p[k]), r)) continue;
        o = geto(p[i], p[j], p[k]);
        r = dis(o, p[i]);
      }
    }
```

```
  }
  printf("%.10lf\n%.10lf %.10lf", r, o.x, o.y);
  return 0;
}
```

杂项

大质数和原根

$p = r \times 2^k + 1$

prime	r	k	g
3	1	1	2
5	1	2	2
17	1	4	3
97	3	5	5
193	3	6	5
257	1	8	3
7681	15	9	17
12289	3	12	11
40961	5	13	3
65537	1	16	3
786433	3	18	10
5767169	11	19	3
7340033	7	20	3
23068673	11	21	3
104857601	25	22	3
167772161	5	25	3
469762049	7	26	3
1004535809	479	21	3
2013265921	15	27	31
2281701377	17	27	3
3221225473	3	30	5
75161927681	35	31	3
77309411329	9	33	7
206158430209	3	36	22
2061584302081	15	37	7
2748779069441	5	39	3
6597069766657	3	41	5
39582418599937	9	42	5
79164837199873	9	43	5
263882790666241	15	44	7
1231453023109121	35	45	3
1337006139375617	19	46	3
3799912185593857	27	47	5
4222124650659841	15	48	19
7881299347898369	7	50	6
31525197391593473	7	52	3
180143985094819841	5	55	6
1945555039024054273	27	56	5
4179340454199820289	29	57	3

约瑟夫问题

```
//约瑟夫问题
int josephus(int n, int k) {
  int res = 0;
  for (int i = 1; i <= n; ++i) res = (res + k) % i;
  return res;
```

```
}
int josephus(int n, int k) {
  if (n == 1) return 0;
  if (k == 1) return n - 1;
  if (k > n) return (josephus(n - 1, k) + k) % n;  // 线性算法
  int res = josephus(n - n / k, k);
  res -= n % k;
  if (res < 0)
    res += n;  // mod n
  else
    res += res / (k - 1);  // 还原位置
  return res;
}
```

辛普森积分

```
const int N = 1000 * 1000;

double simpson_integration(double a, double b) {
  double h = (b - a) / N;
  double s = f(a) + f(b);
  for (int i = 1; i <= N - 1; ++i) {
    double x = a + h * i;
    s += f(x) * ((i & 1) ? 4 : 2);
  }
  s *= h / 3;
  return s;
}


//自适应
double simpson(double l, double r) {
  double mid = (l + r) / 2;
  return (r - l) * (f(l) + 4 * f(mid) + f(r)) / 6;  // 辛普森公式
}

double asr(double l, double r, double eps, double ans, int step) {
  double mid = (l + r) / 2;
  double fl = simpson(l, mid), fr = simpson(mid, r);
  if (abs(fl + fr - ans) <= 15 * eps && step < 0)
    return fl + fr + (fl + fr - ans) / 15;  // 足够相似的话就直接返回
  return asr(l, mid, eps / 2, fl, step - 1) +
         asr(mid, r, eps / 2, fr, step - 1);  // 否则分割成两段递归求解
}

double calc(double l, double r, double eps) {
  return asr(l, r, eps, simpson(l, r), 12);
}
```

unordered_map

```
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};
// pair
// 分别计算出内置类型的 Hash Value 然后对它们进行 Combine 得到一个哈希值
// 一般直接采用移位加异或（XOR）得到哈希值
struct HashFunc
{
```

```cpp
    template<typename T, typename U>
    size_t operator()(const std::pair<T, U>& p) const {
    return std::hash<T>()(p.first) ^ std::hash<U>()(p.second);
    }
};

// 键值比较，哈希碰撞的比较定义，需要直到两个自定义对象是否相等
struct EqualKey {
    template<typename T, typename U>
    bool operator ()(const std::pair<T, U>& p1, const std::pair<T, U>& p2) const {
    return p1.first == p2.first && p1.second == p2.second;
    }
};
```

位运算

1. `int __builtin_ffs(int x)`：返回 x 的二进制末尾最后一个 1 的位置，位置的编号从 1 开始（最低位编号为 1)。当 x 为 0 时返回 0。
2. `int __builtin_clz(unsigned int x)`：返回 x 的二进制的前导 0 的个数。当 x 为 0 时，结果未定义。
3. `int __builtin_ctz(unsigned int x)`：返回 x 的二进制末尾连续 0 的个数。当 x 为 0 时，结果未定义。
4. `int __builtin_clrsb(int x)`：当 x 的符号位为 0 时返回 0 的二进制的前导 0 的个数减一，否则返回 x 的二进制的前导 1 的个数减一。
5. `int __builtin_popcount(unsigned int x)`：返回 x 的二进制中 1 的个数。
6. `int __builtin_parity(unsigned int x)`：判断 x 的二进制中的个数的奇偶性。

模 2 的次幂

```cpp
int modPowerOfTwo(int x, int mod) { return x & (mod - 1); }
```

2 的次幂判断

```cpp
bool isPowerOfTwo(int n) { return n > 0 && (n & (n - 1)) == 0; }
```

子集枚举

```cpp
for (int i = 0; i < (1 << n);i ++)
  for (int s = i; s; s = (s - 1) & i)
```

int128 输出

```cpp
using i128 = __int128;

std::ostream &operator<<(std::ostream &os, i128 n) {
    std::string s;
    while (n) {
        s += '0' + n % 10;
        n /= 10;
    }
    std::reverse(s.begin(), s.end());
    return os << s;
}
```

随机生成质数

```cpp
bool isprime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

```cpp

int findPrime(int n) {
    while (!isprime(n)) {
        n++;
    }
    return n;
}
std::mt19937 rng(std::chrono::steady_clock::now().time_since_epoch().count());
const int P = findPrime(rng() % 900000000 + 100000000);
```

bitset

构造函数

- `bitset()`: 每一位都是 `false`。
- `bitset(unsigned long val)`: 设为 `val` 的二进制形式。
- `bitset(const string& str)`: 设为 01 串 `str`。

运算符

- `operator []`: 访问其特定的一位。
- `operator ==/!=`: 比较两个 `bitset` 内容是否完全一样。
- `operator &/&=/|/| =/^/^=/~`: 进行按位与/或/异或/取反操作。**`bitset` 只能与 `bitset` 进行位运算**，若要和整型进行位运算，要先将整型转换为 `bitset`。
- `operator <>/<<=/>>=`: 进行二进制左移/右移。
- `operator <>`: 流运算符，这意味着你可以通过 `cin/cout` 进行输入输出。

成员函数

- `count()`: 返回 `true` 的数量。
- `size()`: 返回 `bitset` 的大小。
- `test(pos)`: 它和 `vector` 中的 `at()` 的作用是一样的，和 `[]` 运算符的区别就是越界检查。
- `any()`: 若存在某一位是 `true` 则返回 `true`，否则返回 `false`。
- `none()`: 若所有位都是 `false` 则返回 `true`，否则返回 `false`。
- `all()`:**C++11**，若所有位都是 `true` 则返回 `true`，否则返回 `false`。
- 1. `set()`: 将整个 `bitset` 设置成 `true`。
  2. `set(pos, val = true)`: 将某一位设置成 `true`/`false`。
- 1. `reset()`: 将整个 `bitset` 设置成 `false`。
  2. `reset(pos)`: 将某一位设置成 `false`。相当于 `set(pos, false)`。
- 1. `flip()`: 翻转每一位。$0 \leftrightarrow 1$，相当于异或一个全是 1 的 `bitset`）
  2. `flip(pos)`: 翻转某一位。
- `to_string()`: 返回转换成的字符串表达。
- `to_ulong()`: 返回转换成的 `unsigned long` 表达 (`long` 在 NT 及 32 位 POSIX 系统下与 `int` 一样，在 64 位 POSIX 下与 `long long` 一样)。
- `to_ullong()`:**C++11**，返回转换成的 `unsigned long long` 表达。

一些文档中没有的成员函数:

- `_Find_first()`: 返回 `bitset` 第一个 `true` 的下标，若没有 `true` 则返回 `bitset` 的大小。
- `_Find_next(pos)`: 返回 `pos` 后面(下标严格大于 `pos` 的位置)第一个 `true` 的下标,若 `pos` 后面没有 `true` 则返回 `bitset` 的大小。

手写 bitset

```cpp
#include<vector>
using ull = unsigned long long;
struct Bit {
    ull mi[65];
```

```cpp
    // ull bit[15626];
    std::vector<ull> bit; int len;
    Bit() {
        len = 10;
        bit.resize(len);
        for(int i = 0;i <= 63;i ++) mi[i] = (1ull << i);
    }
    Bit(int len) : len(len){
        bit.resize(len);
        for(int i = 0;i <= 63;i ++) mi[i] = (1ull << i);
    }
    void reset() {bit.assign(len,0);}
    void set1(int x) { bit[x>>6] |= mi[x&63];}
    void set0(int x) { bit[x>>6] &= ~mi[x&63];}
    void flip(int x) { bit[x>>6] ^= mi[x&63];}
    bool operator [](int x) {
        return (bit[x>>6] >> (x&63)) & 1;
    }
    int count() {
        int s = 0;
        for(int i = 0;i < len;i ++)s += __builtin_popcountll(bit[i]);
        return s;
    }
    Bit operator ~ (void) const {
        Bit res;
        for (int i = 0;i < len; i++) res.bit[i] = ~bit[i];
        return res;
    }

    Bit operator & (const Bit &b) const {
        Bit res;
        for (int i = 0;i < len; i++) res.bit[i] = bit[i] & b.bit[i];
        return res;
    }

    Bit operator | (const Bit &b) const {
        Bit res;
        for (int i = 0;i < len; i++) res.bit[i] = bit[i] | b.bit[i];
        return res;
    }

    Bit operator ^ (const Bit &b) const {
        Bit res;
        for (int i = 0;i < len; i++) res.bit[i] = bit[i] ^ b.bit[i];
        return res;
    }

    void operator &= (const Bit &b)  {
        for (int i = 0;i < len; i++) bit[i] &= b.bit[i];
    }

    void operator |= (const Bit &b)  {
        for (int i = 0;i < len; i++) bit[i] |= b.bit[i];
    }

    void operator ^= (const Bit &b)  {
        for (int i = 0;i < len; i++) bit[i] ^= b.bit[i];
    }

    Bit operator << (const int t) const {
        Bit res; int high = t >> 6, low = t & 63;
        ull last = 0;
        for (int i = 0;i + high < len; i++) {
            res.bit[i + high] = (last | (bit[i] << low));
            if (low) last = (bit[i] >> (64 - low));
        }
        return res;
    }

    Bit operator >> (const int t) const {
        Bit res; int high = t >> 6, low = t & 63;
```

```
        ull last = 0;
        for (int i = len - 1;i >= high; i--) {
            res.bit[i - high] = last | (bit[i] >> low);
            if (low) last = bit[i] << (64 - low);
        }
        return res;
    }

    void operator <<= (const int t) {
        int high = t >> 6, low = t & 63;
        for (int i = len - high - 1; ~i; i--) {
            bit[i + high] = (bit[i] << low);
            if (low && i) bit[i + high] |= bit[i - 1] >> (64 - low);
        }
        for (int i = 0;i < high; i++) bit[i] = 0;
    }
};
```

string

转 char 数组

`string` 有两个成员函数能够将自己转换为 char 指针——`data()`/`c_str()`（它们几乎是一样的，但最好使用 `c_str()`，因为 `c_str()` 保证末尾有空字符，而 `data()` 则不保证）

寻找某字符（串）第一次出现的位置

`find(str,pos)` 函数可以用来查找字符串中一个字符/字符串在 `pos`（含）之后第一次出现的位置（若不传参给 `pos` 则默认为 `0`）。如果没有出现，则返回 `string::npos`（被定义为 `-1`，但类型仍为 `size_t`/`unsigned long`）。

截取子串

`substr(pos, len)` 函数的参数返回从 `pos` 位置开始截取最多 `len` 个字符组成的字符串（如果从 `pos` 开始的后缀长度不足 `len` 则截取这个后缀）。

插入/删除字符（串）

`insert(index,count,ch)` 和 `insert(index,str)` 是比较常见的插入函数。它们分别表示在 `index` 处连续插入 `count` 次字符串 `ch` 和插入字符串 `str`。

`erase(index,count)` 函数将字符串 `index` 位置开始（含）的 `count` 个字符删除（若不传参给 `count` 则表示删去 `count` 位置及以后的所有字符）。

替换字符（串）

`replace(pos,count,str)` 和 `replace(first,last,str)` 是比较常见的替换函数。它们分别表示将从 `pos` 位置开始 `count` 个字符的子串替换为 `str` 以及将以 `first` 开始（含）、`last` 结束（不含）的子串替换为 `str`，其中 `first` 和 `last` 均为迭代器。

STL

- `sort`：排序。`sort(v.begin(), v.end(), cmp)` 或 `sort(a + begin, a + end, cmp)`，其中 `end` 是排序的数组最后一个元素的后一位，`cmp` 为自定义的比较函数。
- `stable_sort`：稳定排序，用法同 `sort()`。
- `nth_element`：按指定范围进行分类，即找出序列中第 n 大的元素，使其左边均为小于它的数，右边均为大于它的数。`nth_element(v.begin(), v.begin() + mid, v.end(), cmp)` 或 `nth_element(a + begin, a + begin + mid, a + end, cmp)`。
- `binary_search`：二分查找。`binary_search(v.begin(), v.end(), value)`，其中 `value` 为需要查找的值。
- `merge`：将两个(已排序的)序列 **有序合并** 到第三个序列的 **插入迭代器** 上。`merge(v1.begin(), v1.end(), v2.begin(), v2.end() ,back_inserter(v3))`。
- `inplace_merge`：将两个（已按小于运算符排序的）：`[first,middle), [middle,last)` 范围 **原地合并为一个有序序列**。`inplace_merge(v.begin(), v.begin() + middle, v.end())`。
- `lower_bound`：在一个有序序列中进行二分查找，返回指向第一个 **大于等于** x 的元素的位置的迭代器。如果不存在这样的元素，则返回尾迭代器。`lower_bound(v.begin(),v.end(),x)`。

- upper_bound：在一个有序序列中进行二分查找，返回指向第一个 **大于** x 的元素的位置的迭代器。如果不存在这样的元素，则返回尾迭代器。upper_bound(v.begin(),v.end(),x)。
- next_permutation：将当前排列更改为 **全排列中的下一个排列**。如果当前排列已经是 **全排列中的最后一个排列**（元素完全从大到小排列），函数返回 false 并将排列更改为 **全排列中的第一个排列**（元素完全从小到大排列）；否则，函数返回 true。next_permutation(v.begin(), v.end()) 或 next_permutation(v + begin, v + end)。
- partial_sum: 求前缀和。设源容器为 x, 目标容器为 y, 则令 $y[i] = x[0] + x[1] + \cdots + x[i]$。partial_sum(src.begin(), src.end(), back_inserter(dst))。

pb_ds

__gnu_pbds :: tree

```cpp
#include <ext/pb_ds/assoc_container.hpp>  // 因为 tree 定义在这里 所以需要包含这个头文件
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
__gnu_pbds ::tree<Key, Mapped, Cmp_Fn = std::less<Key>, Tag = rb_tree_tag,
                  Node_Update = null_tree_node_update,
                  Allocator = std::allocator<char> >
```

模板形参

- Key: 储存的元素类型，如果想要存储多个相同的 Key 元素，则需要使用类似于 std::pair 和 struct 的方法，并配合使用 lower_bound 和 upper_bound 成员函数进行查找
- Mapped: 映射规则（Mapped-Policy）类型，如果要指示关联容器是 **集合**，类似于存储元素在 std::set 中，此处填入 null_type，低版本 g++ 此处为 null_mapped_type；如果要指示关联容器是 **带值的集合**，类似于存储元素在 std::map 中，此处填入类似于 std::map<Key, Value> 的 Value 类型
- Cmp_Fn: 关键字比较函子，例如 std::less<Key>
- Tag: 选择使用何种底层数据结构类型，默认是 rb_tree_tag。__gnu_pbds 提供不同的三种平衡树，分别是：
  - rb_tree_tag：红黑树，一般使用这个，后两者的性能一般不如红黑树
  - splay_tree_tag：splay 树
  - ov_tree_tag：有序向量树，只是一个由 vector 实现的有序结构，类似于排序的 vector 来实现平衡树，性能取决于数据想不想卡你
- Node_Update：用于更新节点的策略，默认使用 null_node_update，若要使用 order_of_key 和 find_by_order 方法，需要使用 tree_order_statistics_node_update
- Allocator：空间分配器类型

构造方式

```cpp
__gnu_pbds::tree<std::pair<int, int>, __gnu_pbds::null_type,
                 std::less<std::pair<int, int> >, __gnu_pbds::rb_tree_tag,
                 __gnu_pbds::tree_order_statistics_node_update>
    trr;
```

成员函数

- insert(x)：向树中插入一个元素 x，返回 std::pair<point_iterator, bool>。
- erase(x)：从树中删除一个元素/迭代器 x，返回一个 bool 表明是否删除成功。
- order_of_key(x)：返回 x 以 Cmp_Fn 比较的排名。
- find_by_order(x)：返回 Cmp_Fn 比较的排名所对应元素的迭代器。
- lower_bound(x)：以 Cmp_Fn 比较做 lower_bound，返回迭代器。
- upper_bound(x)：以 Cmp_Fn 比较做 upper_bound，返回迭代器。
- join(x)：将 x 树并入当前树，前提是两棵树的类型一样，x 树被删除。
- split(x,b)：以 Cmp_Fn 比较，小于等于 x 的属于当前树，其余的属于 b 树。
- empty()：返回是否为空。
- size()：返回大小。

示例

```cpp
// Common Header Simple over C++11
#include <bits/stdc++.h>
```

```cpp
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;
typedef pair<int, int> pii;
#define pb push_back
#define mp make_pair
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
__gnu_pbds ::tree<pair<int, int>, __gnu_pbds::null_type, less<pair<int, int> >,
                  __gnu_pbds::rb_tree_tag,
                  __gnu_pbds::tree_order_statistics_node_update>
    trr;

int main() {
  int cnt = 0;
  trr.insert(mp(1, cnt++));
  trr.insert(mp(5, cnt++));
  trr.insert(mp(4, cnt++));
  trr.insert(mp(3, cnt++));
  trr.insert(mp(2, cnt++));
  // 树上元素 {{1,0},{2,4},{3,3},{4,2},{5,1}}
  auto it = trr.lower_bound(mp(2, 0));
  trr.erase(it);
  // 树上元素 {{1,0},{3,3},{4,2},{5,1}}
  auto it2 = trr.find_by_order(1);
  cout << (*it2).first << endl;
  // 输出排名 0 1 2 3 中的排名 1 的元素的 first:1
  int pos = trr.order_of_key(*it2);
  cout << pos << endl;
  // 输出排名
  decltype(trr) newtr;
  trr.split(*it2, newtr);
  for (auto i = newtr.begin(); i != newtr.end(); ++i) {
    cout << (*i).first << ' ';
  }
  cout << endl;
  // {4,2},{5,1} 被放入新树
  trr.join(newtr);
  for (auto i = trr.begin(); i != trr.end(); ++i) {
    cout << (*i).first << ' ';
  }
  cout << endl;
  cout << newtr.size() << endl;
  // 将 newtr 树并入 trr 树，newtr 树被删除。
  return 0;
}
```

__gnu_pbds :: priority_queue

```cpp
#include <ext/pb_ds/priority_queue.hpp>
using namespace __gnu_pbds;
__gnu_pbds ::priority_queue<T, Compare, Tag, Allocator>
```

模板形参

- T: 储存的元素类型
- Compare: 提供严格的弱序比较类型
- Tag: 是 __gnu_pbds 提供的不同的五种堆，Tag 参数默认是 pairing_heap_tag 五种分别是：
  - pairing_heap_tag：配对堆官方文档认为在非原生元素（如自定义结构体/std :: string/pair) 中，配对堆表现最好
  - binary_heap_tag：二叉堆官方文档认为在原生元素中二叉堆表现最好，不过我测试的表现并没有那么好
  - binomial_heap_tag：二项堆二项堆在合并操作的表现要优于二叉堆，但是其取堆顶元素操作的复杂度比二叉堆高
  - rc_binomial_heap_tag：冗余计数二项堆
  - thin_heap_tag：除了合并的复杂度都和 Fibonacci 堆一样的一个 tag
- Allocator：空间配置器，由于 OI 中很少出现，故这里不做讲解

构造方式

要注明命名空间因为和 `std` 的类名称重复。

```
__gnu_pbds ::priority_queue<int> __gnu_pbds::priority_queue<int, greater<int> >
__gnu_pbds ::priority_queue<int, greater<int>, pairing_heap_tag>
__gnu_pbds ::priority_queue<int>::point_iterator id; // 点类型迭代器
// 在 modify 和 push 的时候都会返回一个 point_iterator，下文会详细的讲使用方法
id = q.push(1);
```

成员函数

- `push()`: 向堆中压入一个元素，返回该元素位置的迭代器。
- `pop()`: 将堆顶元素弹出。
- `top()`: 返回堆顶元素。
- `size()` 返回元素个数。
- `empty()` 返回是否非空。
- `modify(point_iterator, const key)`: 把迭代器位置的 key 修改为传入的 key，并对底层储存结构进行排序。
- `erase(point_iterator)`: 把迭代器位置的键值从堆中擦除。
- `join(__gnu_pbds :: priority_queue &other)`: 把 `other` 合并到 `*this` 并把 `other` 清空。

使用的 tag 决定了每个操作的时间复杂度：`pairing_heap_tag`

`push` : $O(1)$

`pop` : 最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$

`modify` : 最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$

`erase` : 最坏 $\Theta(n)$ 均摊 $\Theta(\log(n))$

`join` : $O(1)$

示例

```cpp
#include <algorithm>
#include <cstdio>
#include <ext/pb_ds/priority_queue.hpp>
#include <iostream>
using namespace __gnu_pbds;
// 由于面向 OIer, 本文以常用堆 : pairing_heap_tag 作为范例
// 为了更好的阅读体验，定义宏如下 :
#define pair_heap __gnu_pbds ::priority_queue<int>
pair_heap q1;  // 大根堆, 配对堆
pair_heap q2;
pair_heap ::point_iterator id;  // 一个迭代器

int main() {
  id = q1.push(1);
  // 堆中元素 :  [1];
  for (int i = 2; i <= 5; i++) q1.push(i);
  // 堆中元素 :  [1, 2, 3, 4, 5];
  std ::cout << q1.top() << std ::endl;
  // 输出结果 : 5;
  q1.pop();
  // 堆中元素 : [1, 2, 3, 4];
  id = q1.push(10);
  // 堆中元素 : [1, 2, 3, 4, 10];
  q1.modify(id, 1);
  // 堆中元素 :  [1, 1, 2, 3, 4];
  std ::cout << q1.top() << std ::endl;
  // 输出结果 : 4;
  q1.pop();
  // 堆中元素 : [1, 1, 2, 3];
  id = q1.push(7);
  // 堆中元素 : [1, 1, 2, 3, 7];
  q1.erase(id);
  // 堆中元素 : [1, 1, 2, 3];
```

```
  q2.push(1), q2.push(3), q2.push(5);
  // q1 中元素 : [1, 1, 2, 3], q2 中元素 : [1, 3, 5];
  q2.join(q1);
  // q1 中无元素, q2 中元素 : [1, 1, 1, 2, 3, 3, 5];
}
```

hash 表

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
const int RANDOM = chrono::high_resolution_clock::now().time_since_epoch().count();
struct chash {
    int operator()(int x) const { return x ^ RANDOM; }
};
typedef gp_hash_table<int, int, chash> hash_t;
```

rope

```
#include <ext/rope>
using namespace __gnu_cxx;
```

1）运算符：rope 支持 operator += , -= , + , - , <, ==

2）输入输出：可以用 << 运算符由输入输出流读入或输出。

3）长度/大小：调用 `length()`，`size()` 都可以

4）插入/添加等：

`push_back(x)`：在末尾添加 x

`insert(pos,x)`：在 pos 插入 x，自然支持整个 char 数组的一次插入

`erase(pos,x)`：从 pos 开始删除 x 个

`copy(pos,len,x)`：从 pos 开始到 pos+len 为止用 x 代替

`replace(pos,x)`：从 pos 开始换成 x

`substr(pos,x)`：提取 pos 开始 x 个

`at(x)/[x]`：访问第 x 个元素

对拍

```
#!/bin/bash
while true; do
    ./data > data.in
    ./std <data.in >std.out
    ./Todobe <data.in >Todobe.out
    if diff std.out Todobe.out; then
        printf "AC\n"
    else
        printf "Wa\n"
        exit 0
    fi
done
```

火车头

```
#pragma GCC optimize(3)
#pragma GCC target("avx")
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
#pragma GCC optimize("unroll-loops")
#pragma GCC optimize("Ofast")
#pragma GCC optimize("inline")
#pragma GCC optimize("-fgcse")
#pragma GCC optimize("-fgcse-lm")
#pragma GCC optimize("-fipa-sra")
#pragma GCC optimize("-ftree-pre")
```

```
#pragma GCC optimize("-ftree-vrp")
#pragma GCC optimize("-fpeephole2")
#pragma GCC optimize("-ffast-math")
#pragma GCC optimize("-fsched-spec")
#pragma GCC optimize("-falign-jumps")
#pragma GCC optimize("-falign-loops")
#pragma GCC optimize("-falign-labels")
#pragma GCC optimize("-fdevirtualize")
#pragma GCC optimize("-fcaller-saves")
#pragma GCC optimize("-fcrossjumping")
#pragma GCC optimize("-fthread-jumps")
#pragma GCC optimize("-funroll-loops")
#pragma GCC optimize("-fwhole-program")
#pragma GCC optimize("-freorder-blocks")
#pragma GCC optimize("-fschedule-insns")
#pragma GCC optimize("inline-functions")
#pragma GCC optimize("-ftree-tail-merge")
#pragma GCC optimize("-fschedule-insns2")
#pragma GCC optimize("-fstrict-aliasing")
#pragma GCC optimize("-fstrict-overflow")
#pragma GCC optimize("-falign-functions")
#pragma GCC optimize("-fcse-skip-blocks")
#pragma GCC optimize("-fcse-follow-jumps")
#pragma GCC optimize("-fsched-interblock")
#pragma GCC optimize("-fpartial-inlining")
#pragma GCC optimize("no-stack-protector")
#pragma GCC optimize("-freorder-functions")
#pragma GCC optimize("-findirect-inlining")
#pragma GCC optimize("-fhoist-adjacent-loads")
#pragma GCC optimize("-frerun-cse-after-loop")
#pragma GCC optimize("inline-small-functions")
#pragma GCC optimize("-finline-small-functions")
#pragma GCC optimize("-ftree-switch-conversion")
#pragma GCC optimize("-foptimize-sibling-calls")
#pragma GCC optimize("-fexpensive-optimizations")
#pragma GCC optimize("-funsafe-loop-optimizations")
#pragma GCC optimize("inline-functions-called-once")
#pragma GCC optimize("-fdelete-null-pointer-checks")
#pragma GCC optimize(2)
```

Sublime

```
{
    "encoding": "utf-8",
    "working_dir": "$file_path",
    "shell_cmd": "g++ -Wall -std=c++2a \"${file}\" -o \"${file_path}/${file_base_name}\"",
    "file_regex": "^(..[^:]*):([0-9]+):?([0-9]+)?:? (.*)$",
    "selector": "source.c++",

    "variants":
    [
        {
        "name": "Run",
        "shell_cmd": "g++ -Wall -std=c++2a \"${file}\" -o \"${file_base_name}\" && start cmd /c \"\"${file_
        }
    ]
}
```

卡常

vector 的调用空间和运算的效率并不低，主要是多次的加入元素过程效率很低。

减少申请空间的次数，例如不要循环内开 vector

```
// vector f(n,vector<int>(m,0));
f.assign(n, std::vector(m, 0));
```

数组访问尽量连续

被卡常时，不要爆交，先多想想剪枝

注意事项

- **相信所有题都是可做的。**
- **认真读题，模拟完样例再写程序。**
- 热身赛，测试机器速度，重点测 $O(n\log n), O(n\log^2 n), O(n^3), O(n^2\log n)$。
- 感觉不可做的，有较高多项式复杂度暴力的题，思考：**分治**、贪心、dp、线段树。
- 感觉不可做的，只有指数级复杂度暴力的最优化题，思考：贪心、dp、**流和割**、暴搜加优化。
- 感觉不可做的，只有指数级暴力的数数题，思考：dp、行列式、暴搜加优化、拉格朗日插值、**容斥**、造自动机。
- 构造、交互题，考虑：增量法、分治、暴搜策略。
- dp 优化：凸优化（wqs，闵可夫斯基和，李超树）、斜率优化，决策单调性、交换状态和值域、减少状态（包含常数上的）。
- 感觉不可做的题，考虑各个元素/集合之间有什么关系。
- 对于复杂度比较顶的做法，一定要充分沟通后再上机
- `int(v.size())` 切记不能 `ull` 减 `int`
- `__builtin_popcount` 和 `__builtin_popcountll`
- `sqrt` 和 `sqrtl`，`sqrtl` 返回 `long double`
- 几何题注意是不是可能返回 `nan`
- 不能 `x * 1ll` 而是 `1ll * x`
- 比较长的题，写一部分测一部分不要最后一块测
- 任何 n 较大的，可以快速算单项的东西考虑分段打表。

策略

签到题不会做，先确认题面，题面无误看看是不是想难了或者暴力很有道理。

沟通题意前切记确认题面，对着题面和队友讲题意

长时间陷入无效思考时，优先读题。

思路堵死时，要及时跳出来误区或及时找队友沟通。

榜上有简单题做不出来的时候，切记转换一下思路或立即拉另一个人过来重新想（先不要交流思路）

数学

数论

扩展欧几里得（线性同余方程，斐蜀定理）

扩展欧几里得：$\gcd(a,b)=\gcd(b,a\%b)$，$ax+by=bx+(a-\lfloor\frac{a}{b}\rfloor)y$

斐蜀定理：$ax+by=c$ 若有解，则有 $(a,b)|c$

线性同余方程：$ax\equiv c \pmod b \Rightarrow ax+by=c$

```cpp
ll exgcd(ll a,ll b,ll &x,ll &y){
    if(b == 0){
        x = 1 ,y = 0;return a;
    }
    ll d = exgcd(b,a % b,x,y);
    ll tmp = x;
    x = y;
    y = tmp - (a / b) * y;
    return d;
}
void solve(){
    ll a,b,c;
    cin >> a >> b >> c;
    ll x0,y0;
    ll d = exgcd(a,b,x0,y0);
    if(c % d){
```

```cpp
        cout << -1 << "\n";
        return ;
    }
    ll p = a / d,q = b / d;
    ll x = ((c / d) % q * x0 % q + q) % q;
    if(x == 0)x = q;
    ll y = (c - a * x) / b;
    if(y <= 0){
        y = ((c / d) % p * y0 % p + p) % p;
        cout << (x == 0 ? q : x) << " " << (y == 0 ? p : y) << "\n";
        return ;
    }
    ll ans_x_mn = x;
    ll ans_y_mx = y;
    y = ((c / d) % p * y0 % p + p) % p;
    if(y == 0)y = p;
    x = (c - b * y) / a;
    ll ans_x_mx = x;
    ll ans_y_mn = y;
    ll sum = min((ans_x_mx - ans_x_mn) / q,(ans_y_mx - ans_y_mn) / p);
    cout << sum + 1 << " " << ans_x_mn << " " << ans_y_mn << " " << ans_x_mx << " " << ans_y_mx << "\n";
    // 正整数解总数
}
```

费马小定理（逆元）

若 p 为素数，$\gcd(a,p)=1$，则 $a^{p-1}\equiv 1 \pmod p$

线性求逆元

```cpp
inv[0] = inv[1] = 1;
for(int i = 2 ;i <= n ;i++) inv[i] = (p - p/i) * inv[p % i] % p;
```

CRT（中国剩余定理）

$$\begin{cases} x = b_1(\mathrm{mod} a_1) \\ x = b_2(\mathrm{mod} a_2) \\ \ldots \\ x = b_n(\mathrm{mod} a_n) \end{cases}$$

若 $ a_{1} ,a_{2},...,a_{n}$ 两两互质:

令 $M=\prod_1^n a_i, m_i'=\frac{M}{a_i}, t_i\times m_i'\equiv 1(\mathrm{mod}\ a_i)$ 则有 $x=\sum_{i=1}^n b_i\times m_i'\times t_i$ （此解为唯一解）

若 $ a_{1} ,a_{2},...,a_{n}$ 两两不互质:

合并两个方程组 $x=a_1p+b_1=a_2q+b_2$

则可将方程依次两两合并为 $x\equiv a_1p+b_1(\mathrm{mod}\mathrm{lcm}(a_1,a_2))$，其中先求解 p，再带入求 x。

```cpp
ll r1 = B[1], m1 = A[1],r2,m2;
for(int i = 1;i < n ;i ++) {
    r2 = B[i + 1],m2 = A[i + 1];
    ll a = m1,b = m2,c = r2 - r1;
    ll d = exgcd(a,b,x,y);
    if(c % d) {
        cout<<0;return 0;
    }
    ll p = a / d,q = b / d;
    x = ((x * c / d) + q) % q;
    ll mod = lcm(m2,m1);
    ll x0 = (m1 * x + r1) % mod;
    r1 = x0 < 0 ? x0 + mod : x0;
    m1 = mod;
}
cout << r1 % m1 << "\n";
```

卢卡斯定理

$C_n^m = C_{n \bmod p}^{m \bmod p} \cdot C_{\lfloor n/p \rfloor}^{\lfloor m/p \rfloor}$，其中 p 为质数。

原根

设 $m \in \mathbf{N}^*$，$g \in \mathbf{Z}$. 若 $(g, m) = 1$，且 $\delta_m(g) = \varphi(m)$，则称 g 为模 m 的原根。

即 g 满足 $\delta_m(g) = |\mathbf{Z}_m^*| = \varphi(m)$. 当 m 是质数时，我们有 $g^i \bmod m,\ 0 < i < m$ 的结果互不相同。

原根判定定理:

设 $m \geq 3, (g, m) = 1$，则 g 是模 m 的原根的充要条件是，对于 $\varphi(m)$ 的每个素因数 p，都有 $g^{\frac{\varphi(m)}{p}} \not\equiv 1 \pmod m$。

若一个数 m 有原根，则它原根的个数为 $\varphi(\varphi(m))$，每一个原根都形如 g^k 的形式，要求满足 $\gcd(k, \varphi(n)) = 1$。