Standard Code Library

Your TeamName

Your School

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;
    }
```

```
};
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<Segment> 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) {}
};
```

```
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

```
/**
 * 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<>> {
```

```
  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

```
// 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)

```cpp
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;
```

```
  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

```
//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

```
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);
            }
```

```
        }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

```
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

```
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() {
```

```
        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

```
        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);
```

```
      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

```
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

```
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]
};
```