数学

行列

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

template<typename T>
class Matrix{
private:
  using size_type = ::std::size_t;
  using Row = ::std::vector<T>;
  using Mat = ::std::vector<Row>;

  size_type R, C; // row, column
  Mat A;

  void add_row_to_another(size_type r1, size_type r2, const T k){ //
  ↪  Row(r1) += Row(r2)*k
    for(size_type i = 0;i < C;i++)
      A[r1][i] += A[r2][i]*k;
  }

  void scalar_multiply(size_type r, const T k){
    for(size_type i = 0;i < C;i++)
      A[r][i] *= k;
  }

  void scalar_division(size_type r, const T k){
    for(size_type i = 0;i < C;i++)
      A[r][i] /= k;
  }

public:
  Matrix(){}
  Matrix(size_type r, size_type c) : R(r), C(c), A(r, Row(c)) {}
  Matrix(const Mat &m) : R(m.size()), C(m[0].size()), A(m) {}
  Matrix(const Mat &&m) : R(m.size()), C(m[0].size()), A(m) {}
  Matrix(const Matrix<T> &m) : R(m.R), C(m.C), A(m.A) {}
  Matrix(const Matrix<T> &&m) : R(m.R), C(m.C), A(m.A) {}
  Matrix<T> &operator=(const Matrix<T> &m){
     R = m.R; C = m.C; A = m.A;
    return *this;
  }
  Matrix<T> &operator=(const Matrix<T> &&m){
     R = m.R; C = m.C; A = m.A;
    return *this;
  }
  static Matrix I(const size_type N){
    Matrix m(N, N);
    for(size_type i = 0;i < N;i++) m[i][i] = 1;
    return m;
  }

  const Row& operator[](size_type k) const& { return A.at(k); }
  Row& operator[](size_type k) & { return A.at(k); }
  Row operator[](size_type k) const&& { return ::std::move(A.at(k));
  ↪  }

  size_type row() const { return R; } // the number of rows
  size_type column() const { return C; }

  T determinant(){
    assert(R == C);
    Mat tmp = A;
    T res = 1;
    for(size_type i = 0;i < R;i++){
      for(size_type j = i;j < R;j++){ // satisfy A[i][i] > 0
        if (A[j][i] != 0) {
          if (i != j) res *= -1;
          swap(A[j], A[i]);
          break;
        }
      }
      if (A[i][i] == 0) return 0;
      res *= A[i][i];
      scalar_division(i, A[i][i]);
      for(size_type j = i+1;j < R;j++){
        add_row_to_another(j, i, -A[j][i]);
      }
    }
    swap(tmp, A);
    return res;
  }

  Matrix inverse(){
    assert(R == C);
    assert(determinant() != 0);
    Matrix inv(Matrix::I(R)), tmp(*this);
    for(size_type i = 0;i < R;i++){
      for(size_type j = i;j < R;j++){
        if (A[j][i] != 0) {
          swap(A[j], A[i]);
          swap(inv[j], inv[i]);
          break;
        }
      }
      inv.scalar_division(i, A[i][i]);
      scalar_division(i, A[i][i]);
      for(size_type j = 0;j < R;j++){
        if(i == j) continue;
        inv.add_row_to_another(j, i, -A[j][i]);
        add_row_to_another(j, i, -A[j][i]);
      }
    }
    (*this) = tmp;
    return inv;
  }

  Matrix& operator+=(const Matrix &B){
    assert(column() == B.column() && row() == B.row());
    for(size_type i = 0;i < R;i++)
      for(size_type j = 0;j < C;j++)
        (*this)[i][j] += B[i][j];
    return *this;
  }

  Matrix& operator-=(const Matrix &B){
    assert(column() == B.column() && row() == B.row());
    for(size_type i = 0;i < R;i++)
      for(size_type j = 0;j < C;j++)
        (*this)[i][j] -= B[i][j];
    return *this;
  }

  Matrix& operator*=(const Matrix &B){
    assert(column() == B.row());
    Matrix M(R, B.column());
    for(size_type i = 0;i < R;i++) {
      for(size_type j = 0;j < B.column();j++) {
        M[i][j] = 0;
        for(size_type k = 0;k < C;k++) {
          M[i][j] += (*this)[i][k] * B[k][j];
        }
      }
    }
    swap(M, *this);
    return *this;
  }

  Matrix& operator/=(const Matrix &B){
    assert(C == B.row());
    Matrix M(B);
    (*this) *= M.inverse();
    return *this;
  }

  Matrix operator+(const Matrix &B) const { return (Matrix(*this) +=
  ↪  B); }
  Matrix operator-(const Matrix &B) const { return (Matrix(*this) -=
  ↪  B); }
  Matrix operator*(const Matrix &B) const { return (Matrix(*this) *=
  ↪  B); }
  Matrix operator/(const Matrix &B) const { return (Matrix(*this) /=
  ↪  B); }

  bool operator==(const Matrix &B) const {
    if (column() != B.column() || row() != B.row()) return false;
    for(size_type i = 0;i < row();i++)
      for(size_type j = 0;j < column();j++)
        if ((*this)[i][j] != B[i][j]) return false;
    return true;
  }
  bool operator!=(const Matrix &B) const { return !((*this) == B); }

  Matrix pow(size_type k){
    assert(R == C);
    Matrix M(Matrix::I(R));
    while(k){
      if (k & 1) M *= (*this);
      k >>= 1;
      (*this) *= (*this);
    }
    A.swap(M.A);
    return *this;
  }

  friend ::std::ostream &operator<<(::std::ostream &os, Matrix &p){
    for(size_type i = 0;i < p.row();i++){
      for(size_type j = 0;j < p.column();j++){
        os << p[i][j] << " ";
      }
      os << ::std::endl;
    }
    return os;
  }
};

int main(){
}
```

データ構造

セグメントツリー

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

template<class ValueMonoid, template<class...> class
↪  Container=::std::vector>
class SegTree{
public:
  using value_structure = ValueMonoid;
  using value_type = typename value_structure::value_type;
  using const_reference = const value_type &;
  using container_type = Container<value_type>;
  using size_type = typename container_type::size_type;

private:
  ::std::vector<value_type> tree;
  size_type size_;

  static size_type getsize(const size_type x){
    size_type ret = 1;
    while(ret < x)
      ret <<= 1;
```

```
    return ret;
  }

  inline value_type calc(const value_type a, const value_type b){
    return value_structure::operation(a, b);
  }

  inline void calc_node(const size_type index){
    if(tree.size() <= (index << 1 | 1)) return;
    tree[index] = value_structure::operation(tree[index<<1],
    ↪  tree[index<<1 | 1]);
  }
public:
  SegTree() : size_(0), tree(){}
  SegTree(const size_type size)
      : size_(size), tree(size << 1, value_structure::identity()){}
  template<class InputIterator>
  SegTree(InputIterator first, InputIterator last)
      : size_(::std::distance(first, last)){
    tree = container_type(size_, value_structure::identity());
    tree.insert(tree.end(), first, last);
    for(size_type i = size_;i > 0;i--){
      calc_node(i);
    }
  }

  size_type size() const { return size_; }
  const_reference operator[](const size_type k) const {
    assert(k < size_);
    return tree[k+size_];
  }

  value_type query(size_type l, size_type r){
    assert(l <= r);
    assert(0 <= l && l < size_);
    assert(0 <= r && r <= size_);
    value_type retl = value_structure::identity(), retr =
    ↪  value_structure::identity();
    for(l += size_, r += size_; l < r ; l >>= 1, r >>= 1){
      if(l&1) retl = calc(retl, tree[l++]);
      if(r&1) retr = calc(tree[--r], retr);
    }
    return calc(retl, retr);
  }

  template<class F>
  void update(size_type index, const F& f){
    assert(0 <= index && index < size());
    index += size_;
    tree[index] = f(::std::move(tree[index]));
    while(index >>= 1)
      calc_node(index);
  }

  /*
  template<class F>
  size_type search(const F& f) const { // [0, result) is True and
↪   [0, result-1) is not.
    if(f(value_structure::identity()))
      return 0;
    if(!f(tree[1]))
      return size_+1;
    value_type acc = value_structure::identity();
    size_type i = 1;
    while(i <
  }
  */
};

/*
verify:
↪  http://judge.u-aizu.ac.jp/onlinejudge/review.jsp?rid=3162647#1
      http://judge.u-aizu.ac.jp/onlinejudge/review.jsp?rid=3162648
↪  #1

template<ValueMonoid, Container>
class SegTree

ValueMonoid
  - 要求
    - value_type
    - identity() -> value_type : 単位元を返す
    - operation(value_type, value_type) -> value_type : 演算結果を返す

SegTree
  - 提供
    - query(size_type l, size_type r) -> value_type
      - 計算量 O(log N)
      - [l, r)までの計算結果

    - update(size_type k, function f)
      - 計算量 O(log N)
      - kth elementをfを適用した結果に変更する

    * 未実装
    - search(function f) -> size_type
      - 計算量 O(log N)?
      - f([0, k)) is true and f([0, k+1)) is falseとなるkを返す
*/
```

遅延セグメントツリー

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

template<class ValueMonoid, class OperatorMonoid, class Modifier,
  template<class...> class Container=::std::vector>
class LazySegTree{
public:
  using value_structure = ValueMonoid;
  using value_type = typename value_structure::value_type;
  using operator_structure = OperatorMonoid;
  using operator_type = typename operator_structure::value_type;
  using modifier = Modifier;
  using const_reference = const value_type &;
  using container_value_type = Container<value_type>;
  using container_operator_type = Container<operator_type>;
  using size_type = typename container_value_type::size_type;

private:
  container_value_type tree;
  container_operator_type lazy;
  size_type size_, height;

  static size_type getsize(const size_type x){
    size_type ret = 1;
    while(ret < x)
      ret <<= 1;
    return ret;
  }

  static size_type getheight(const size_type x){
    size_type ret = 0;
    while((static_cast<size_type>(1) << ret) < x){
      ret++;
    }
    return ret;
  }

  inline static value_type calc(const value_type a, const value_type
  ↪  b){
    return value_structure::operation(a, b);
  }

  inline static void apply(operator_type &data, const operator_type
  ↪  a){
    data = operator_structure::operation(data, a);
  }

  inline static value_type reflect(const value_type v, const
  ↪  operator_type o){
    return modifier::operation(v, o);
  }

  void push(const size_type index){
    tree[index] = reflect(tree[index], lazy[index]);
    apply(lazy[index << 1], lazy[index]);
    apply(lazy[index << 1 | 1], lazy[index]);
    lazy[index] = operator_structure::identity();
  }

  void calc_node(const size_type index){
    if(tree.size() <= (index << 1 | 1)) return;
    assert(0 < index);
    tree[index] = calc(reflect(tree[index << 1],  lazy[index << 1]),
        reflect(tree[index << 1 | 1], lazy[index << 1 | 1]));
  }

  void build(size_type index){
    while(index >>= 1){
      calc_node(index);
    }
  }

  void propagate(const size_type index){
    for(size_type shift = height; shift ; --shift){
      push(index >> shift);
    }
  }

  void rebuild(){
    for(size_type i = size_-1;i > 0;--i){
      calc_node(i);
    }
  }
public:
  LazySegTree() : size_(0), height(0), tree(), lazy(){}
  LazySegTree(const size_type size)
      : size_(size), height(getheight(size)),
        tree(size << 1, value_structure::initializer()),
        lazy(size << 1, operator_structure::identity()){
    rebuild();
  }
  template<class InputIterator>
  LazySegTree(InputIterator first, InputIterator last)
      : size_(::std::distance(first, last)){
    height = getheight(size_);
    tree = container_value_type(size_, value_structure::identity());
    lazy = container_operator_type(size_ << 1,
    ↪  operator_structure::identity());
    tree.insert(tree.end(), first, last);
    rebuild();
  }

  size_type size() const { return size_; }
  const_reference operator[](const size_type k){
    assert(k < size_);
    propagate(k+size_);
    tree[k+size_] = reflect(tree[k+size_], lazy[k+size_]);
    lazy[k+size_] = operator_structure::identity();
    return tree[k+size_];
  }
```

```cpp
  value_type query(size_type l, size_type r){
    assert(l <= r);
    assert(0 <= l && l < size_);
    assert(0 <= r && r <= size_);
    value_type retl = value_structure::identity(),
               retr = value_structure::identity();
    l += size_;
    r += size_;
    propagate(l);
    propagate(r-1);
    for(; l < r ; l >>= 1, r >>= 1){
      if(l&1){
        retl = calc(retl, reflect(tree[l], lazy[l]));
        l++;
      }
      if(r&1){
        r--;
        retr = calc(reflect(tree[r], lazy[r]), retr);
      }
    }
    return calc(retl, retr);
  }

  void update(size_type l, size_type r, const operator_type& data){
    assert(l <= r);
    assert(0 <= l && l < size_);
    assert(0 <= r && r <= size_);
    l += size_;
    r += size_;
    propagate(l);
    propagate(r - 1);
    for(size_type l_ = l, r_ = r; l_ < r_ ; l_ >>= 1, r_ >>= 1){
      if(l_ & 1) apply(lazy[l_++], data);
      if(r_ & 1) apply(lazy[--r_], data);
    }
    build(l);
    build(r - 1);
  }

  template<class F>
  void update(size_type index, const F& f){
    assert(0 <= index && index < size());
    index += size_;
    propagate(index);
    tree[index] = f(::std::move(tree[index]));
    lazy[index] = operator_structure::identity();
    build(index);
  }

  /*
  template<class F>
  size_type search(const F& f) const { // [0, result) is True and [0, result-1) is not.
    if(f(value_structure::identity()))
      return 0;
    if(!f(tree[1]))
      return size_+1;
    value_type acc = value_structure::identity();
    size_type i = 1;
    while(i <
  }
  */
};


/*
verify: http://judge.u-aizu.ac.jp/onlinejudge/review.jsp?rid=3176153
        http://judge.u-aizu.ac.jp/onlinejudge/review.jsp?rid=3176158
        http://judge.u-aizu.ac.jp/onlinejudge/review.jsp?rid=3176164
        http://judge.u-aizu.ac.jp/onlinejudge/review.jsp?rid=3176248
        http://judge.u-aizu.ac.jp/onlinejudge/review.jsp?rid=3176296

template<ValueMonoid, OperatorMonoid, Modifier, Container>
class LazySegTree

ValueMonoid
  - 役割
    - 扱う要素の値
  -  要求
    - value_type
    - identity() -> value_type : 単位元を返す
    - initializer() -> value_type : 要素の初期値を返す
    - operation(value_type, value_type) -> value_type : 演算結果を返す

    - 必要時
      - size_type value_type::len : ノードの幅

OperatorMonoid
  - 役割
    - 扱う要素に適用させる値
  - 要求
    - value_type
    - identity() -> value_type : 単位元を返す
    - operation(value_type, value_type) -> value_type : 作用素を結合する

Modifier<ValueMonoid, OperatorMonoid>
  - 役割
    - OperatorMonoidをValueMonoidに適用させる
  - 要求
    - operation(value_type, operator_type) -> value_type : 作用素を適用させた結果を返す

LazySegTree
  - 提供
    - query(size_type l, size_type r) -> value_type
      - 計算量 O(log N)
      - [l, r)までの計算結果

    - update(size_type l, size_type r, operator_type x)
      - 計算量 O(log N)
      - [l, r)にxを適用させた結果に変更する

    - update(size_type k, function f)
      - 計算量 O(log N)
      - kth elementをfを適用した結果に変更する

    * 未実装
    - search(function f) -> size_type
      - 計算量 O(log N)?
      - f([0, k)) is true and f([0, k+1)) is falseとなるkを返す
*/
```