Contents

1 Algorithms

1.1 Mo.cc

```
/* Determining the number of distinct numbers in a subsequence
 * Time: O(M sqrt(N))
 * Memory: O(M + N + MAX_VALUE)
 */

#include <bits/stdc++.h>

#define SIZE 30010
#define MAX_VALUE 1000010
#define QUERIES 200010

using namespace std;

int N, M, sz, res, cnt[MAX_VALUE], a[SIZE], ans[QUERIES];

struct Query {
  int l, r, index;
  Query () {}
  Query (int l, int r, int index): l(l), r(r), index(index) {}
  bool operator < (const Query& q) const {
    if ((l - 1) / sz != (q.l - 1) / sz)
      return (l - 1) / sz > (q.l - 1) / sz;
    return r < q.r;
  }
} q[QUERIES];

void update (int i) {
  if (!cnt[i]++)
    res++;
}

void remove (int i) {
  if (!--cnt[i])
    res--;
}


int main () {
  scanf("%d", &N);
  sz = (int)sqrt(N);

  for (int i = 1; i <= N; i++)
    scanf("%d", &a[i]);

  scanf("%d", &M);
  for (int i = 0; i < M; i++) {
    int l, r;
    scanf("%d%d", &l, &r);
    q[i] = Query(l, r, i);
  }

  sort(q, q + M);
  int l = 1, r = 0;
  for (int i = 0; i < M; i++) {
    while (r > q[i].r)
      remove(a[r--]);
    while (r < q[i].r)
```

```
      update(a[++r]);
    while (l < q[i].l)
      remove(a[l++]);
    while (l > q[i].l)
      update(a[--l]);
    ans[q[i].index] = res;
  }

  for (int i = 0; i < M; i++)
    printf("%d\n", ans[i]);
  return 0;
}
```

2 Dynamic Programming

2.1 Convex_Hull_Trick.cc

```
/* Time: O(N) for any interlaced sequence of addLine and getMax calls
 * Time: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

typedef long long LL;

struct ConvexHullTrick {
  vector<LL> M, B;
  int ptr = 0;

  void addLine (LL m, LL b) {
    int len = M.size();
    while (len > 1 && (B[len - 2] - B[len - 1]) * (m - M[len - 1]) >= (B[len - 1]
        - b) * (M[len - 1] - M[len - 2]))
      len--;
    M.resize(len);
    B.resize(len);
    M.push_back(m);
    B.push_back(b);
  }

  LL getMax (LL x) {
    if (ptr >= (int)M.size())
      ptr = (int)M.size() - 1;
    while (ptr < (int)M.size() - 1 && M[ptr + 1] * x + B[ptr + 1] >= M[ptr] * x +
        B[ptr])
      ptr++;
    return M[ptr] * x + B[ptr];
  }
};
```

2.2 Convex_Hull_Trick_Dynamic.cc

```
/* Time: O(N) for any interlaced sequence of addLine and getBest calls
 * Time: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef long double ld;

struct Line {
  ll m, b, val;
  ld xVal;
  bool isQuery;

  Line (ll m, ll b, ll val, bool isQuery): m(m), b(b), val(val), xVal(-
      numeric_limits<double>::max()), isQuery(isQuery) {}

  bool isParallel (const Line& l) const {
    return m == l.m;
  }

  ld intersect (const Line& l) const {
    if (isParallel(l))
      return numeric_limits<double>::max();
    return (ld)(l.b - b) / (m - l.m);
  }

  bool operator < (const Line& l) const {
    if (l.isQuery)
      return l.val < xVal;
    return m < l.m;
  }
};

typedef set<Line>::iterator iter;

struct ConvexHullTrick {
  set<Line> hull;

  bool hasPrev (iter it) {
    return it != hull.begin();
  }

  bool hasNext (iter it) {
    return (it != hull.end()) && (++it != hull.end());
  }

  bool isIrrelevant (iter it) {
    if (!hasPrev(it) || !hasNext(it))
      return false;
    iter prev = it, next = it;
    prev--, next++;
    return next->intersect(*prev) <= next->intersect(*it);
  }

  iter updateIntersections (iter it) {
    if (!hasNext(it))
      return it;
    iter it2 = it;
    double val = it->intersect(*++it2);
    Line l(*it);
    l.xVal = val;
    hull.erase(it++);
    return hull.insert(++it, l);
  }

  void addLine (ll m, ll b) {
    Line l(m, b, 0, false);
    iter it = hull.lower_bound(l);

    if (it != hull.end() && it->isParallel(l)) {
      if (b < it->b)
        hull.erase(it);
      else
        return;
    }

    it = hull.insert(it, l);
    if (isIrrelevant(it)) {
      hull.erase(it);
      return;
    }

    while (hasPrev(it) && isIrrelevant(--it))
      hull.erase(it++);
    while (hasNext(it) && isIrrelevant(++it))
      hull.erase(it--);

    it = updateIntersections(it);
    if (hasPrev(it))
      updateIntersections(--it);
    if (hasNext(++it))
      updateIntersections(++it);
  }

  ll getBest (ll x) const {
    Line q(0, 0, x, true);
```

```
    iter it = hull.lower_bound(q);
    return it->m * x + it->b;
  }
};
```

2.3 Divide_And_Conquer.cc

```
/* Requirements:
 *  - dp[i][j] = min(dp[i - 1][k] + C[k][j]) where k < j
 *  - min[i][j] <= min[i][j + 1], min[i][j] = smallest k for optimal ans
 *
 * There are N people at an amusement park who are in a queue for a ride.
 * Each pair of people has a measured level of unfamiliarity. The people
 * will be divided into K non-empty contiguous groups. Each division has
 * a total unfamiliarty value which is the sum of the levels of
 * unfamiliarty between any pair of people for each group
 *
 * Time Improvement: O(KN^2) -> O(KN log N)
 * Memory: O(KN)
 */

#include <bits/stdc++.h>

#define MAX_N 4001
#define MAX_K 801
#define scan(x) do{while((x=getchar())<'0'); for(x-='0'; '0'<=(_=getchar()); x=(x
    <<3)+(x<<1)+_-'0');}while(0)
char _;
using namespace std;

int N, K;
int A[MAX_N][MAX_N];
int dp[MAX_K][MAX_N];

void compute (int g, int i, int j, int l, int r) {
  if (i > j)
    return;
  int mid = (i + j) >> 1;
  int bestIndex = l;
  for (int k = l; k <= min(mid, r); k++) {
    int val = dp[g - 1][k - 1] + (A[mid][mid] - A[mid][k - 1] - A[k - 1][mid] + A[
        k - 1][k - 1]);
    if (val < dp[g][mid]) {
      dp[g][mid] = val;
      bestIndex = k;
    }
  }
  compute(g, i, mid - 1, l, bestIndex);
  compute(g, mid + 1, j, bestIndex, r);
}

int main () {
  scan(N);
  scan(K);

  for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
      scan(A[i][j]);
      A[i][j] += A[i - 1][j] + A[i][j - 1] - A[i - 1][j - 1];
    }
  }

  for (int i = 0; i <= K; i++)
    for (int j = 0; j <= N; j++)
      dp[i][j] = 1 << 30;
  dp[0][0] = 0;

  for (int i = 1; i <= K; i++)
    compute(i, 1, N, 1, N);

  printf("%d\n", dp[K][N] / 2);
  return 0;
}
```

2.4 Knuth.cc

```
/* Requirements:
 *  - dp[i][j] = min(dp[i][k] + dp[k][j] + C[i][j]) where i < k < j
 *  - min[i][j - 1] <= min[i][j] <= min[i + 1][j],
 *    min[i][j] = smallest k for optimal answer
 *
 * A certain string-processing language allows the programmer to
 * break a string into two pieces. Since this involves copying the
 * old string, it cost N units of time to break a string of N
 * characters into two pieces. Suppose you want to break a string into
 * many pieces. The order in which the breaks are made can affect the
 * total amount of time.
 *
 * Given the length of the string N, and M places to break the string at,
 * what is the minimum amount of time to break the string?
 *
 * Time Improvement: O(N^3) -> O(N^2)
 * Memory: O(N^2)
 */

#include <bits/stdc++.h>

#define SIZE 1005

typedef long long ll;

ll dp[SIZE][SIZE];
int mid[SIZE][SIZE];
int pos[SIZE];
int N, M;

int main () {
  while (scanf("%d%d", &N, &M) != EOF) {
    for (int i = 1; i <= M; i++)
      scanf("%d", &pos[i]);
    pos[0] = 0;
    pos[M + 1] = N;

    for (int i = 0; i <= M + 1; i++) {
      for (int j = 0; j + i <= M + 1; j++) {
        if (i < 2) {
          dp[j][j + i] = 0LL;
          mid[j][j + i] = j;
          continue;
        }
        dp[j][j + i] = 1LL << 60;
        for (int k = mid[j][i + j - 1]; k <= mid[j + 1][i + j]; k++) {
          ll next = dp[j][k] + dp[k][j + i] + pos[j + i] - pos[j];
          if (next < dp[j][j + i]) {
            dp[j][j + i] = next;
            mid[j][j + i] = k;
          }
        }
      }
    }
    printf("%lld\n", dp[0][M + 1]);
  }
  return 0;
}
```

3 Data Structures

3.1 BIT.cc

```
/* Time: O(log N)
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

struct BIT {
  int N;
```


3

```
  vector<int> val;
  BIT (int N) : N(N), val(N) {}

  void update (int idx, int v) {
    for (int x = idx; x < N; x += (x & -x))
      val[x] += v;
  }

  int query (int idx) {
    int ret = 0;
    for (int x = idx; x > 0; x -= (x & -x))
      ret += val[x];
    return ret;
  }
};
```

3.2 BIT_Range.cc

```
/* Time: O(log N)
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

struct BIT_Range {
  int N;
  vector<int> val1, val2;
  BIT_Range (int N): N(N), val1(N), val2(N) {}

  void update (vector<int> &val, int idx, int v) {
    for (int x = idx; x < N; x += (x & -x))
      val[x] += v;
  }

  void update (int x1, int x2, int val) {
    update(val1, x1, val);
    update(val1, x2 + 1, -val);
    update(val2, x1, val * (x1 - 1));
    update(val2, x2 + 1, -val * x2);
  }

  int query (vector<int> &val, int idx) {
    int ret = 0;
    for (int x = idx; x > 0; x -= (x & -x))
      ret += val[x];
    return ret;
  }

  int query (int x) {
    return query(val1, x) * x - query(val2, x);
  }

  int query (int x1, int x2) {
    return query(x2) - query(x1 - 1);
  }
};
```

3.3 Treap.cc

```
/* Time: O(log N) query
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

struct Treap {
  struct Node {
    int val, p;
    Node *left, *right;
    Node (int val): val(val), p(randomPriority()) {
      left = nullptr;
      right = nullptr;
    }
  };

  static int randomPriority () {
    return rand() * 65536 + rand();
  }

  Node* root;

  Treap (): root(nullptr) {}

  // precondition: all values of u are smaller than all values of v
  static Node* join (Node* u, Node* v) {
    if (u == nullptr)
      return v;
    if (v == nullptr)
      return u;
    if (u->p > v->p) {
      u->right = join(u->right, v);
      return u;
    }
    v->left = join(u, v->left);
    return v;
  }

  static pair<Node*, Node*> split (Node* u, int k) {
    if (u == nullptr)
      return make_pair(nullptr, nullptr);
    if (u->val < k) {
      auto res = split(u->right, k);
      u->right = res.first;
      res.first = u;
      return res;
    } else if (u->val > k) {
      auto res = split(u->left, k);
      u->left = res.second;
      res.second = u;
      return res;
    } else {
      return make_pair(u->left, u->right);
    }
  }
  bool contains (int val) {
    Node *curr = root;
    while (curr != nullptr) {
      if (curr->val < val)
        curr = curr->right;
      else if (curr->val > val)
        curr = curr->left;
      else
        return true;
    }
    return false;
  }

  void insert (int val) {
    if (contains(val))
      return;
    auto nodes = split(root, val);
    root = join(nodes.first, join(new Node(val), nodes.second));
  }
  void remove (int val) {
    if (root == nullptr)
      return;
    auto nodes = split(root, val);
    root = join(nodes.first, nodes.second);
  }
};
```

3.4 Treap_Implicit.cc

```
/* The key of each element is the array index of the element.
 * Time: O(log N)
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;
```

```
struct Treap {
  struct Node {
    int val, p, sz;
    Node *left, *right;
    Node (int val): val(val), p(randomPriority()), sz(1), left(nullptr), right(
        nullptr) {}
  };

  static int randomPriority () {
    return rand() * 65536 + rand();
  }

  static int getSize (Node* u) {
    return u == nullptr ? 0 : u->sz;
  }

  static void update (Node* u) {
    if (u) u->sz = 1 + getSize(u->left) + getSize(u->right);
  }

  Node* root;

  Treap (): root(nullptr) {}

  // precondition: all values of u are smaller than all values of v
  static Node* join (Node* u, Node* v) {
    if (u == nullptr)
      return v;
    if (v == nullptr)
      return u;
    if (u->p > v->p) {
      u->right = join(u->right, v);
      update(u);
      return u;
    }
    v->left = join(u, v->left);
    update(v);
    return v;
  }

  static pair<Node*, Node*> split (Node* u, int k) {
    if (u == nullptr)
      return make_pair(nullptr, nullptr);
    if (getSize(u->left) + 1 > k) {
      auto res = split(u->left, k);
      u->left = res.second;
      res.second = u;
      update(res.first);
      update(res.second);
      return res;
    } else {
      auto res = split(u->right, k - getSize(u->left) - 1);
      u->right = res.first;
      res.first = u;
      update(res.first);
      update(res.second);
      return res;
    }
  }

  void modify (int index, int val) {
    Node *curr = root;
    while (curr != nullptr) {
      if (getSize(curr->left) + 1 < index)
        index -= getSize(curr->left) + 1, curr = curr->right;
      else if (getSize(curr->left) + 1 > index)
        curr = curr->left;
      else {
        curr->val = val;
        return;
      }
    }
  }

  int get (int index) {
    Node *curr = root;
    while (curr != nullptr) {
      if (getSize(curr->left) + 1 < index)
        index -= getSize(curr->left) + 1, curr = curr->right;
      else if (getSize(curr->left) + 1 > index)
        curr = curr->left;
      else
        return curr->val;
    }
    return -1;
  }

  void push_back (int val) {
    root = join(root, new Node(val));
  }

  void insert (int index, int val) {
    auto res = split(root, index);
    root = join(res.first, join(new Node(val), res.second));
  }

  void remove (int index) {
    auto nodes = split(root, index);
    root = join(nodes.first->left, join(nodes.first->right, nodes.second));
  }
};
```

3.5 Splay.cc

```
/* Recently accessed elements are quick to access again.
 * Time: O(log N)
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

struct Splay {
  struct Node {
    int val;
    Node *child[2], *par;
    Node (int val): val(val) {
      child[0] = this;
      child[1] = this;
      par = this;
    }
  };

  static Node *null;
  Node *root;

  Splay (): root(null) {}

  static void connect (Node* u, Node* v, int dir) {
    u->child[dir] = v;
    v->par = u;
  }

  // 0 = left, 1 = right;
  static Node* rotate (Node* u, int dir) {
    Node *c = u->child[dir ^ 1], *p = u->par, *pp = p->par;
    connect(p, c, dir);
    connect(u, p, dir ^ 1);
    connect(pp, u, getDir(p, pp));
    return u;
  }

  static int getDir (Node* u, Node* p) {
    return p->child[0] == u ? 0 : 1;
  }

  static Node* splay (Node* u) {
    while (u->par != null) {
      Node *p = u->par, *pp = p->par;
      int dp = getDir(u, p), dpp = getDir(p, pp);
      if (pp == null) rotate(u, dp);
      else if (dp == dpp) rotate(p, dpp), rotate(u, dp);
      else rotate(u, dp), rotate(u, dpp);
```

```
  }
  return u;
}

// closest node to val
static Node* nodeAt (Node* u, int val) {
  if (u == null) return u;
  Node* ret = u;
  while (u != null) {
    ret = u;
    if (u->val < val) u = u->child[1];
    else if (u->val > val) u = u->child[0];
    else return u;
  }
  return ret;
}

// precondition: all values of u are smaller than all values of v
static Node* join (Node* u, Node* v) {
  if (u == null) return v;
  while (u->child[1] != null)
    u = u->child[1];
  splay(u);
  u->child[1] = v;
  if (v != null)
    v->par = u;
  return u;
}

static pair<Node*, Node*> split (Node* u, int val) {
  if (u == null) return {null, null};
  splay(u = nodeAt(u, val));

  if (u->val == val) {
    u->child[0]->par = u->child[1]->par = null;
    return {u->child[0], u->child[1]};
  } else if (u->val < val) {
    Node *ret = u->child[1];
    u->child[1] = (u->child[1]->par = null);
    return {u, ret};
  } else {
    Node *ret = u->child[0];
    u->child[0] = (u->child[0]->par = null);
    return {ret, u};
  }
}

bool contains (int val) {
  Node *curr = root;
  while (curr != null) {
    if (curr->val < val)
      curr = curr->child[1];
    else if (curr->val > val)
      curr = curr->child[0];
    else
      return true;
  }
  return false;
}

void insert (int val) {
  if (contains(val))
    return;
  auto res = split(root, val);
  root = new Node(val);
  root->par = null;
  root->child[0] = res.first, root->child[1] = res.second;
  if (root->child[0] != null)
    root->child[0]->par = root;
  if (root->child[1] != null)
    root->child[1]->par = root;
}

void remove (int val) {
  Node *curr = nodeAt(root, val);
  splay(curr);
  curr->child[0]->par = curr->child[1]->par = null;
  root = join(curr->child[0], curr->child[1]);
  }
};
```

3.6 Splay_Implicit.cc

```
/* The key of each element is the array index of the element.
 * Time: O(log N)
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

struct Splay {
  struct Node {
    int val, sz;
    Node *child[2], *par;
    Node () {}
    Node (int val, int sz = 1): val(val), sz(sz) {
      child[0] = child[1] = par = this;
    }
  };

  Node *root;
  static Node *null;

  Splay () {
    null = new Node();
    null->child[0] = null->child[1] = null->par = null;
    null->sz = 0;
    root = null;
  }

  static void connect (Node* u, Node* v, int dir) {
    u->child[dir] = v;
    v->par = u;
  }

  static void update (Node* u) {
    if (u != null) u->sz = 1 + u->child[0]->sz + u->child[1]->sz;
  }

  // 0 = left, 1 = right;
  static Node* rotate (Node* u, int dir) {
    Node *c = u->child[dir ^ 1], *p = u->par, *pp = p->par;
    connect(p, c, dir);
    connect(u, p, dir ^ 1);
    connect(pp, u, getDir(p, pp));
    update(p);
    update(u);
    update(pp);
    return u;
  }

  static int getDir (Node* u, Node* p) {
    return p->child[0] == u ? 0 : 1;
  }

  static Node* splay (Node* u) {
    while (u->par != null) {
      Node *p = u->par, *pp = p->par;
      int dp = getDir(u, p), dpp = getDir(p, pp);
      if (pp == null) rotate(u, dp);
      else if (dp == dpp) rotate(p, dpp), rotate(u, dp);
      else rotate(u, dp), rotate(u, dpp);
    }
    return u;
  }

  static Node* nodeAt (Node* u, int index) {
    if (u == null) return u;
    Node* ret = u;
    while (u != null) {
      ret = u;
      if (u->child[0]->sz + 1 < index) index -= u->child[0]->sz + 1, u = u->child
          [1];
```

```
    else if (u->child[0]->sz + 1 > index) u = u->child[0];
    else return u;
  }
  return ret;
}

// precondition: all values of u are smaller than all values of v
static Node* join (Node* u, Node* v) {
  if (u == null) return v;
  while (u->child[1] != null)
    u = u->child[1];
  splay(u);
  u->child[1] = v;
  update(u);
  if (v != null)
    v->par = u;
  return u;
}

static pair<Node*, Node*> split (Node* u, int index) {
  if (u == null) return {null, null};
  splay(u = nodeAt(u, index));

  if (u->child[0]->sz + 1 <= index) {
    Node *ret = u->child[1];
    u->child[1] = (u->child[1]->par = null);
    update(u);
    update(ret);
    return {u, ret};
  } else {
    Node *ret = u->child[0];
    u->child[0] = (u->child[0]->par = null);
    update(u);
    update(ret);
    return {ret, u};
  }
}

void modify (int index, int val) {
  Node *curr = root;
  while (curr != null) {
    if (curr->child[0]->sz + 1 < index)
      index -= curr->child[0]->sz + 1, curr = curr->child[1];
    else if (curr->child[0]->sz + 1 > index)
      curr = curr->child[0];
    else {
      curr->val = val;
      return;
    }
  }
}

void push_back (int val) {
  Node *u = new Node(val);
  u->child[0] = u->child[1] = u->par = null;
  root = join(root, u);
}

void insert (int index, int val) {
  auto res = split(root, index);
  root = new Node(val);
  root->par = null;
  root->child[0] = res.first, root->child[1] = res.second;
  update(root);
  if (root->child[0] != null)
    root->child[0]->par = root;
  if (root->child[1] != null)
    root->child[1]->par = root;
}

void remove (int index) {
  Node *curr = nodeAt(root, index);
  splay(curr);
  curr->child[0]->par = curr->child[1]->par = null;
  root = join(curr->child[0], curr->child[1]);
}

  int get (int index) {
    return nodeAt(root, index)->val;
  }
};

Splay::Node* Splay::null = new Node(0, 0);
```

3.7 Link_Cut_Tree.cc

```
/* Time: O(log N) for all operations
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

struct LinkCut {
  struct Node {
    int index, sz;
    Node *child[2], *par, *pathPar;
    Node (int index, int sz): index(index), sz(sz) {
      child[0] = this;
      child[1] = this;
      par = this;
      pathPar = this;
    }
  };

  static Node *null;
  vector<Node*> nodes;

  LinkCut (int N): nodes(N) {
    for (int i = 0; i < N; i++) {
      nodes[i] = new Node(i, 1);
      nodes[i]->child[0] = nodes[i]->child[1] = nodes[i]->par = nodes[i]->pathPar
          = null;
    }
  }

  static void update (Node *u) {
    if (u == null)
      return;
    u->sz = 1 + u->child[0]->sz + u->child[1]->sz;
  }

  static int getDir (Node *u, Node *p) {
    return p->child[0] == u ? 0 : 1;
  }

  static void connect (Node* u, Node* v, int dir) {
    u->child[dir] = v;
    v->par = u;
  }

  static Node* rotate (Node* u, int dir) {
    Node *c = u->child[dir ^ 1], *p = u->par, *pp = p->par;
    connect(p, c, dir);
    connect(u, p, dir ^ 1);
    connect(pp, u, getDir(p, pp));

    u->pathPar = p->pathPar;
    p->pathPar = null;
    update(p);
    update(u);
    update(pp);
    return u;
  }

  static void splay (Node *u) {
    while (u->par != null) {
      Node *p = u->par, *pp = p->par;
      int dp = getDir(u, p), dpp = getDir(p, pp);
      if (pp == null) rotate(u, dp);
      else if (dp == dpp) rotate(p, dpp), rotate(u, dp);
      else rotate(u, dp), rotate(u, dpp);
    }
```

```
  }

  static Node* access (Node* u) {
    Node *prev = null;
    for (Node *v = u; v != null; v = v->pathPar) {
      splay(v);
      if (v->child[1] != null) {
        v->child[1]->pathPar = v;
        v->child[1]->par = null;
        v->child[1] = null;
      }
      v->child[1] = prev;
      update(v);
      if (prev != null) {
        prev->par = v;
        prev->pathPar = null;
      }
      prev = v;
    }
    splay(u);
    return prev;
  }

  // precondition: n must be a root node, and n and m must be in different trees
  static void link (Node *n, Node *m) {
    access(n);
    access(m);
    n->child[0] = m;
    m->par = n;
    update(n);
  }

  // precondition: n must not be a root node
  static void cut (Node *n) {
    access(n);
    if (n->child[0] != null) {
      n->child[0]->par = null;
      n->child[0] = null;
    }
    update(n);
  }

  static Node* getRoot (Node *n) {
    access(n);
    while (n->child[0] != null)
      n = n->child[0];
    access(n);
    return n;
  }

  static int getHeight (Node *n) {
    access(n);
    return n->child[0]->sz + 1;
  }

  static int lca (Node *u, Node *v) {
    access(u);
    return access(v)->index;
  }
};
```

3.8 Persistent_Segment_Tree.cc

```
/* What would be the k-th number in (A[i], A[i + 1], ..., A[j]) if this segment
     was sorted?
 * Time: O(log N) per query; O(N log N) for construction
 * Memory: O(N log N)
 */

#include <bits/stdc++.h>

#define SIZE 100001

using namespace std;

struct Node {
  int cnt;
  Node *left, *right;
  Node (int cnt): cnt(cnt) {}
  Node (int cnt, Node *left, Node *right): cnt(cnt), left(left), right(right) {}
};

struct Tree {
  int N;
  vector<Node*> val;
  Tree () {}
  Tree (int N): N(N), val(N + 1) {
    val[0] = new Node(0);
    val[0]->left = val[0]->right = val[0];
  }

  Node* update (Node* prev, int l, int r, int val) {
    if (l <= val && val <= r) {
      if (l == r)
        return new Node(prev->cnt + 1);
      int mid = (l + r) >> 1;
      return new Node(prev->cnt + 1, update(prev->left, l, mid, val), update(prev
          ->right, mid + 1, r, val));
    }
    return prev;
  }

  int query (Node* lo, Node* hi, int l, int r, int val) {
    if (l == r)
      return l;
    int mid = (l + r) >> 1;
    int cnt = hi->left->cnt - lo->left->cnt;
    if (val <= cnt)
      return query(lo->left, hi->left, l, mid, val);
    else
      return query(lo->right, hi->right, mid + 1, r, val - cnt);
  }
};

int N, Q;
set<int> ts;
int toVal[SIZE], a[SIZE];
unordered_map<int, int> toIndex;
Tree t(SIZE);

int main () {
  scanf("%d%d", &N, &Q);

  for (int i = 1; i <= N; i++) {
    scanf("%d", &a[i]);
    ts.insert(a[i]);
  }
  int cnt = 0;
  for (int val : ts) {
    toIndex[val] = ++cnt;
    toVal[cnt] = val;
  }

  for (int i = 1; i <= N; i++)
    t.val[i] = t.update(t.val[i - 1], 1, cnt, toIndex[a[i]]);

  for (int i = 0; i < Q; i++) {
    int l, r, k;
    scanf("%d%d%d", &l, &r, &k);
    printf("%d\n", toVal[t.query(t.val[l - 1], t.val[r], 1, cnt, k)]);
  }
}
```

4 Geometry

4.1 Convex_Hull.cc

```
/* Time: O(N log N)
 * Memory: O(N)
 */

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

```
using namespace std;

struct Point {
  int x, y;
  Point (int x, int y): x(x), y(y) {}
  bool operator < (const Point& p) const {
    return make_pair(x, y) < make_pair(p.x, p.y);
  }
};

int ccw (Point p1, Point p2, Point p3) {
  return (p2.x - p1.x) * (p3.y - p1.y) - (p2.y - p1.y) * (p3.x - p1.x);
}

vector<Point> convexHull (vector<Point> pts) {
  vector<Point> u, l;
  sort(pts.begin(), pts.end());

  for (int i = 0; i < (int)pts.size(); i++) {
    int j = (int)l.size();
    while (j >= 2 && ccw(l[j - 2], l[j - 1], pts[i]) <= 0) {
      l.erase(l.end() - 1);
      j = (int)l.size();
    }
    l.push_back(pts[i]);
  }
  for (int i = (int)pts.size() - 1; i >= 0; i--) {
    int j = (int)u.size();
    while (j >= 2 && ccw(u[j - 2], u[j - 1], pts[i]) <= 0) {
      u.erase(u.end() - 1);
      j = (int)u.size();
    }
    u.push_back(pts[i]);
  }

  u.erase(u.end() - 1);
  l.erase(l.end() - 1);
  l.reserve(l.size() + u.size());
  l.insert(l.end(), u.begin(), u.end());

  return l;
}
```

4.2 Delaunay.cc

```
/* Time: O(N^4)
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

// input: (x, y) coordinates
// output: M by 3 matrix containing tuple of indices corresponding to vertices
vector<vector<int>> triangulate (vector<int> x, vector<int> y) {
  int N = x.size();
  vector<int> z (N);
  vector<vector<int>> ret;

  for (int i = 0; i < N; i++)
    z[i] = x[i] * x[i] + y[i] + y[i];

  for (int i = 0; i < N - 2; i++) {
    for (int j = i + 1; j < N; j++) {
      for (int k = i + 1; k < N; k++) {
        if (j == k)
          continue;
        int xn = (y[j]-y[i])*(z[k]-z[i])-(y[k]-y[i])*(z[j]-z[i]);
        int yn = (x[k]-x[i])*(z[j]-z[i])-(x[j]-x[i])*(z[k]-z[i]);
        int zn = (x[j]-x[i])*(y[k]-y[i])-(x[k]-x[i])*(y[j]-y[i]);
        bool flag = zn < 0;
        for (int m = 0; flag && m < N; m++)
          flag &= ((x[m]-x[i])*xn+(y[m]-y[i])*yn+(z[m]-z[i])*zn <= 0);
        if (flag)
          ret.push_back({i, j, k});
      }
    }
  }
  return ret;
}
```

5 Graph Theory

5.1 Eulerian.cc

```
/* Time: O(N)
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

struct Edge {
  int dest, index;
  bool used;
};

struct Euler {
  int N;
  vector<vector<Edge>> adj;
  vector<int> used;
  Euler (int N): N(N), adj(N), used(N) {}

  void addEdge (int u, int v) {
    adj[u].push_back({v, (int)adj[v].size(), 0});
    adj[v].push_back({u, (int)adj[u].size() - 1, 0});
  }

  // precondition: all vertices are connected
  int getEuler () {
    int odd = 0;
    for (int i = 0; i < N; i++)
      if ((int)adj[i].size() & 1)
        odd++;
    if (odd > 2)
      return -1;
    return odd == 0 ? 0 : 1;
  }

  bool isEulerianPath () {
    return getEuler() != -1;
  }

  bool isEulerianCycle () {
    return getEuler() == 0;
  }

  void printEulerianPath () {
    if (!isEulerianPath()) {
      printf("No Eulerian Path Exists.");
      return;
    }
    stack<int> order;
    int curr = 0;
    for (int i = 0; i < N; i++)
      if ((int)adj[i].size() & 1)
        curr = i;
    while (true) {
      if ((int)adj[curr].size() - used[curr] == 0) {
        printf("%d ", curr);
        if (order.size() == 0)
          break;
        curr = order.top();
        order.pop();
      } else {
        order.push(curr);
        for (int i = 0; i < (int)adj[curr].size(); i++) {
          if (!adj[curr][i].used) {
            int dest = adj[curr][i].dest;
```

```
            int index = adj[curr][i].index;
            adj[curr][i].used = true;
            adj[dest][index].used = true;
            used[curr]++;
            used[dest]++;
            curr = dest;
            break;
          }
        }
      }
    }
  }
};
```

5.2 SCC.cc

```
/* Time: O(V)
 * Memory: O(V + E)
 */

#include <bits/stdc++.h>

using namespace std;

struct SCC {
  int N, cnt, idCnt;
  vector<int> disc, lo, id;
  vector<bool> inStack;
  vector<vector<int>> adj;
  stack<int> s;

  SCC (int N): N(N), disc(N), lo(N), id(N), inStack(N), adj(N) {}

  void addEdge (int u, int v) {
    adj[u].push_back(v);
  }

  void dfs (int i) {
    disc[i] = lo[i] = ++cnt;
    inStack[i] = true;
    s.push(i);
    for (int j : adj[i]) {
      if (disc[j] == 0) {
        dfs(j);
        lo[i] = min(lo[i], lo[j]);
      } else if (inStack[j]) {
        lo[i] = min(lo[i], disc[j]);
      }
    }
    if (disc[i] == lo[i]) {
      while (s.top() != i) {
        inStack[s.top()] = false;
        id[s.top()] = idCnt;
        s.pop();
      }
      inStack[s.top()] = false;
      id[s.top()] = idCnt++;
      s.pop();
    }
  }

  void compute () {
    for (int i = 0; i < N; i++)
      if (disc[i] == 0)
        dfs(i);
  }
};
```

5.3 Biconnected_Components.cc

```
/* Time: O(N)
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

typedef pair<int, int> edge;

struct BiconnectedComponents {
  int N, cnt = 0;
  vector<edge> bridges;
  vector<vector<edge>> components;
  vector<vector<int>> adj;
  stack<edge> s;
  vector<int> lo, disc;
  vector<bool> vis, cutVertex;

  BiconnectedComponents (int N): N(N), adj(N), lo(N), disc(N), vis(N), cutVertex(N
      ) {}

  void addEdge (int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }

  void dfs (int u, int prev) {
    disc[u] = lo[u] = cnt++;
    vis[u] = true;
    int children = 0;
    for (int v : adj[u]) {
      if (!vis[v]) {
        children++;
        s.push({u, v});
        dfs(v, u);
        lo[u] = min(lo[u], lo[v]);
        if ((disc[u] == 0 && children > 1) || (disc[u] > 0 && lo[v] >= disc[u])) {
          cutVertex[u] = true;
          components.push_back(vector<edge>());
          while (s.top().first != u && s.top().second != v) {
            components.back().push_back(edge(s.top().first, s.top().second));
            s.pop();
          }
          components.back().push_back(edge(s.top().first, s.top().second));
          s.pop();
        }
        if (lo[v] > disc[u])
          bridges.push_back(edge(s.top().first, s.top().second));
      } else if (v != prev && disc[v] < lo[u]) {
        lo[u] = disc[v];
        s.push({u, v});
      }
    }
  }

  void compute () {
    for (int i = 0; i < N; i++)
      if (!vis[i])
        dfs(i, -1);
  }
};
```

5.4 Max_Flow.cc

```
/* Time: O(EV^2); O(min(V^(2/3), E^(1/2))E) for unit capacities
 * Memory: O(V + E)
 */

#include <bits/stdc++.h>

using namespace std;

struct Edge {
  int dest, cost, next;
  Edge (int dest, int cost, int next): dest(dest), cost(cost), next(next) {}
};

struct Network {
  int N, src, sink;
  vector<int> last, dist;
  vector<Edge> e;
```

```
  Network (int N, int src, int sink): N(N), src(src), sink(sink), last(N), dist(N)
       {
    fill(last.begin(), last.end(), -1);
  }

  void addEdge (int x, int y, int xy, int yx) {
    e.push_back(Edge(y, xy, last[x]));
    last[x] = (int)e.size() - 1;
    e.push_back(Edge(x, yx, last[y]));
    last[y] = (int)e.size() - 1;
  }

  bool getPath () {
    fill(dist.begin(), dist.end(), -1);
    queue<int> q;
    q.push(src);
    dist[src] = 0;

    while (!q.empty()) {
      int curr = q.front(); q.pop();
      for (int i = last[curr]; i != -1; i = e[i].next) {
        if (e[i].cost > 0 && dist[e[i].dest] == -1) {
          dist[e[i].dest] = dist[curr] + 1;
          q.push(e[i].dest);
        }
      }
    }

    return dist[sink] != -1;
  }

  int dfs (int curr, int flow) {
    if (curr == sink)
      return flow;
    int ret = 0;
    for (int i = last[curr]; i != -1; i = e[i].next) {
      if (e[i].cost > 0 && dist[e[i].dest] == dist[curr] + 1) {
        int res = dfs(e[i].dest, min(flow, e[i].cost));
        ret += res;
        e[i].cost -= res;
        e[i ^ 1].cost += res;
        flow -= res;
        if (flow == 0)
          break;
      }
    }
    return ret;
  }

  int getFlow () {
    int res = 0;
    while (getPath())
      res += dfs(src, 1 << 30);
    return res;
  }
};
```

5.5 Max_Flow_Min_Cost.cc

```
/* Time: O(VEB log V) where B is upper bound on the largest supply of any node
 * Memory: O(V + E)
 */

#include <bits/stdc++.h>

using namespace std;

struct Edge {
  int orig, dest, origCost, cost, flow, last;
  Edge (int orig, int dest, int cost, int flow, int last): orig(orig), dest(dest),
       origCost(cost), cost(cost), flow(flow), last(last) {}
};

struct Vertex {
  int index, cost;
  Vertex (int index, int cost): index(index), cost(cost) {}
  bool operator < (const Vertex& v) const {
    return cost < v.cost;
  }
};

struct MaxFlowMinCost {
  int N, src, sink, cnt = 0;
  vector<Edge> e;
  vector<int> last, phi, prev, dist, index;
  MaxFlowMinCost (int N, int src, int sink): N(N), src(src), sink(sink), last(N),
       phi(N), prev(N), dist(N), index(N) {
    fill(last.begin(), last.end(), -1);
  }

  void addEdge (int u, int v, int flow, int cost) {
    e.push_back({u, v, cost, flow, last[u]});
    last[u] = (int)e.size() - 1;
    e.push_back({v, u, -cost, 0, last[v]});
    last[v] = (int)e.size() - 1;
  }

  void reduceCost () {
    for (int i = 0; i < (int)e.size(); i += 2) {
      e[i].cost += phi[e[i].orig] - phi[e[i].dest];
      e[i ^ 1].cost = 0;
    }
  }

  void bellmanFord () {
    fill(phi.begin(), phi.end(), 1 << 25);
    phi[src] = 0;
    for (int j = 0; j < N - 1; j++)
      for (int i = 0; i < (int)e.size(); i++)
        if (e[i].flow > 0)
          phi[e[i].dest] = min(phi[e[i].dest], phi[e[i].orig] + e[i].cost);
  }

  bool dijkstra () {
    fill(dist.begin(), dist.end(), 1 << 30);
    fill(prev.begin(), prev.end(), -1);
    fill(index.begin(), index.end(), -1);
    dist[src] = 0;
    priority_queue<Vertex> pq;
    pq.push({src, 0});
    while (!pq.empty()) {
      Vertex curr = pq.top();
      pq.pop();
      for (int next = last[curr.index]; next != -1; next = e[next].last) {
        if (e[next].flow == 0 || dist[e[next].dest] <= dist[curr.index] + e[next].
            cost)
          continue;
        dist[e[next].dest] = dist[curr.index] + e[next].cost;
        prev[e[next].dest] = curr.index;
        index[e[next].dest] = next;
        pq.push({e[next].dest, dist[e[next].dest]});
      }
    }
    return dist[sink] != 1 << 30;
  }

  pair<int, int> getMaxFlowMinCost () {
    int flow = 0;
    int cost = 0;
    bellmanFord();
    reduceCost();
    while (dijkstra()) {
      for (int i = 0; i < N; i++)
        phi[i] = dist[i];
      reduceCost();
      int aug = 1 << 30;
      int curr = sink;
      while (prev[curr] != -1) {
        aug = min(aug, e[index[curr]].flow);
        curr = prev[curr];
      }
      flow += aug;
      curr = sink;
```

```
      while (prev[curr] != -1) {
        e[index[curr]].flow -= aug;
        e[index[curr] ^ 1].flow += aug;
        cost += aug * e[index[curr]].origCost;
        curr = prev[curr];
      }
    }
    return {flow, cost};
  }
};
```

5.6 Max_Matching.cc

```
/* Time: O(V^4)
 * Memory: O(V + E)
 */

#include <bits/stdc++.h>

using namespace std;

struct MaxMatching {
  int N;
  vector<vector<int>> adj;
  vector<bool> mark, used;
  vector<int> match, par, id;
  MaxMatching (int N): N(N), adj(N), mark(N), used(N), match(N), par(N), id(N) {}

  void addEdge (int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }

  void markPath (vector<bool>& blossom, int i, int b, int j) {
    for (; id[i] != b; i = par[match[i]]) {
      blossom[id[i]] = blossom[id[match[i]]] = true;
      par[i] = j;
      j = match[i];
    }
  }

  int lca (int i, int j) {
    vector<bool> v(N);
    while (true) {
      i = id[i];
      used[i] = true;
      if (match[i] == -1)
        break;
      i = par[match[i]];
    }
    while (true) {
      j = id[j];
      if (v[j])
        return j;
      j = par[match[j]];
    }
  }

  int getAugmentingPath (int src) {
    fill(par.begin(), par.end(), -1);
    fill(used.begin(), used.end(), 0);
    for (int i = 0; i < N; i++)
      id[i] = i;
    used[src] = true;
    queue<int> q;
    q.push(src);

    while (!q.empty()) {
      int curr = q.front();
      q.pop();
      for (int next : adj[curr]) {
        if (id[curr] == id[next] || match[curr] == next)
          continue;
        if (next == src || (match[next] != -1 && par[match[next]] != -1)) {
          int newBase = lca(curr, next);
          vector<bool> blossom(N);
          markPath(blossom, curr, newBase, next);
          markPath(blossom, next, newBase, curr);
          for (int i = 0; i < N; i++) {
            if (blossom[id[i]]) {
              id[i] = newBase;
              if (!used[i]) {
                used[i] = true;
                q.push(i);
              }
            }
          }
        } else if (par[next] == -1) {
          par[next] = curr;
          if (match[next] == -1)
            return next;
          next = match[next];
          used[next] = true;
          q.push(next);
        }
      }
    }
  }

  int getMaxMatching () {
    fill(match.begin(), match.end(), -1);
    fill(par.begin(), par.end(), 0);
    fill(id.begin(), id.end(), 0);
    fill(used.begin(), used.end(), 0);

    for (int i = 0; i < N; i++) {
      if (match[i] == -1){
        int v = getAugmentingPath(i);
        while (v != -1) {
          int pv = par[v];
          int ppv = match[pv];
          match[v] = pv;
          match[pv] = v;
          v = ppv;
        }
      }
    }

    int res = 0;
    for (int i = 0; i < N; i++)
      if (match[i] != -1)
        res++;
    return res / 2;
  }
};
```

5.7 Min_Cut.cc

```
/* Time: O(V^3)
 * Memory: O(V^2)
 */

#include <bits/stdc++.h>

using namespace std;

struct MinCut {
  int N;
  vector<vector<int>> adj;
  vector<int> weight;
  vector<bool> inContraction, used;
  MinCut (int N): N(N), adj(N, vector<int>(N)), weight(N, 0), inContraction(N, 0),
        used(N, 0) {}

  void addEdge (int u, int v, int c) {
    adj[u][v] = c;
    adj[v][u] = c;
  }

  int getMinCut () {
    int minCut = 1 << 30;
    for (int v = N - 1; v >= 0; v--) {
      for (int i = 1; i < N; i++) {
```

```
        used[i] = inContraction[i];
        weight[i] = adj[0][i];
      }
      int prev = 0, curr = 0;
      for (int sz = 1; sz <= v; sz++) {
        prev = curr;
        curr = -1;
        for (int i = 1; i < N; i++)
          if (!used[i] && (curr == -1 || weight[i] > weight[curr]))
            curr = i;
        if (sz != v) {
          for (int i = 0; i < N; i++)
            weight[i] += adj[curr][i];
          used[curr] = true;
        } else {
          for (int i = 0; i < N; i++)
            adj[prev][i] = adj[i][prev] += adj[i][curr];
          inContraction[curr] = true;
          minCut = min(minCut, weight[curr]);
        }
      }
    }
    return minCut;
  }
};
```

5.8 LCA.cc

```
/* Time: O(log N) query; O(N log N) construction
 * Memory: O(N log N)
 */

#include <bits/stdc++.h>

using namespace std;

struct LCA {
  int N, LN;
  vector<int> depth;
  vector<vector<int>> pa;
  vector<vector<int>> adj;
  LCA (int N): N(N), LN(ceil(log(N) / log(2) + 1)), depth(N), pa(N, vector<int>(LN
       )), adj(N) {
    for (auto &x : pa)
      fill(x.begin(), x.end(), -1);
  }

  void addEdge (int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }

  void dfs (int u, int d, int prev) {
    depth[u] = d;
    pa[u][0] = prev;
    for (int v : adj[u])
      if (v != prev)
        dfs(v, d + 1, u);
  }

  void precompute () {
    for (int i = 1; i < LN; i++)
      for (int j = 0; j < N; j++)
        if (pa[j][i - 1] != -1)
          pa[j][i] = pa[pa[j][i - 1]][i - 1];

  }

  int getLca (int u, int v) {
    if (depth[u] < depth[v])
      swap(u, v);
    for (int k = LN - 1; k >= 0; k--)
      if (pa[u][k] != -1 && depth[pa[u][k]] >= depth[v])
        u = pa[u][k];
    if (u == v)
      return u;
    for (int k = LN - 1; k >= 0; k--)
      if (pa[u][k] != -1 && pa[v][k] != -1 && pa[u][k] != pa[v][k])
        u = pa[u][k], v = pa[v][k];
    return pa[u][0];
  }
};
```

5.9 HLD.cc

```
/* Time: O(log N) query; O(N) construction
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

struct HLD {
  int N, chainIndex;
  vector<vector<int>> adj;
  vector<int> sz, depth, chain, par, head;

  HLD (int N): N(N), adj(N), sz(N), depth(N), chain(N), par(N), head(N) {
    fill(head.begin(), head.end(), -1);
  }

  void addEdge (int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
  }

  void dfs (int u, int p, int d) {
    par[u] = p;
    depth[u] = d;
    sz[u] = 1;
    for (int v : adj[u]) {
      if (v != p) {
        dfs(v, u, d + 1);
        sz[u] += sz[v];
      }
    }
  }

  void build (int u, int p) {
    if (head[chainIndex] == -1)
      head[chainIndex] = u;
    chain[u] = chainIndex;

    int maxIndex = -1;
    for (int v : adj[u])
      if (v != p && (maxIndex == -1 || sz[v] > sz[maxIndex]))
        maxIndex = v;
    if (maxIndex != -1)
      build(maxIndex, u);
    for (int v : adj[u])
      if (v != p && v != maxIndex) {
        chainIndex++;
        build(v, u);
      }
  }

  void precompute () {
    dfs(0, -1, 0);
    build(0, -1);
  }

  int getLca (int u, int v) {
    while (chain[u] != chain[v]) {
      if (depth[head[chain[u]]] < depth[head[chain[v]]])
        v = par[head[chain[v]]];
      else
        u = par[head[chain[u]]];
    }
    return depth[u] < depth[v] ? u : v;
  }
};
```

6 Mathematics

6.1 General.cc

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

using namespace std;

typedef long long ll;

vector<int> getPrimesEratosthenes (int N) {
  vector<bool> prime (N + 1);
  vector<int> ret;

  fill(prime.begin(), prime.end(), true);

  for (int i = 2; i * i <= N; i++)
    if (prime[i])
      for (int j = i * i; j <= N; j += i)
        prime[j] = false;

  for (int i = 2; i <= N; i++)
    if (prime[i])
      ret.push_back(i);

  return ret;
}

vector<int> eulerTotient (int N) {
  vector<int> ret (N + 1);
  for (int i = 1; i <= N; i++)
    ret[i] = i;
  for (int i = 2; i <= N; i++)
    if (ret[i] == i)
      for (int j = i; j <= N; j += i)
        ret[j] -= ret[j] / i;
  return ret;
}

ll gcd (ll a, ll b) {
  return b == 0 ? a : gcd(b, a % b);
}

ll multmod (ll a, ll b, ll m) {
  ll x = 0, y = a % m;
  for (; b > 0; b >>= 1) {
    if ((b & 1) == 1)
      x = (x + y) % m;
    y = (y << 1) % m;
  }
  return x % m;
}

ll randLong () {
  return ((rand() * 1LL) << 47) | ((rand() * 1LL) << 32) | ((rand() * 1LL) << 16)
      | rand();
}

ll brent (ll n) {
  if (n % 2 == 0)
    return 2;
  ll y = randLong() % (n - 1) + 1;
  ll c = randLong() % (n - 1) + 1;
  ll m = randLong() % (n - 1) + 1;
  ll g = 1, r = 1, q = 1, ys = 0, hi = 0, x = 0;
  while (g == 1) {
    x = y;

    for (int i = 0; i < r; i++)
      y = (multmod(y, y, n) + c) % n;

    for (ll k = 0; k < r && g == 1; k += m) {
      ys = y;
      hi = min(m, r - k);
      for (int j = 0; j < hi; j++) {
        y = (multmod(y, y, n) + c) % n;
        q = multmod(q, x > y ? x - y : y - x, n);
      }
      g = gcd(q, n);
    }

    r *= 2;
  }

  if (g == n)
    do {
      ys = (multmod(ys, ys, n) + c) % n;
      g = gcd(x > ys ? x - ys : ys - x, n);
    } while (g <= 1);

  return g;
}
```

6.2 Miller_Rabin.cc

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

using namespace std;

typedef unsigned long long ULL;
ULL mulmod (ULL a, ULL b, ULL c) {
  ULL x = 0, y = a % c;
  for (; b > 0; b >>= 1) {
    if (b & 1) x = (x + y) % c;
    y = (y << 1) % c;
  }
  return x % c;
}

ULL powmod (ULL a, ULL b, ULL c) {
  ULL x = 1, y = a;
  for (; b > 0; b >>= 1) {
    if (b & 1) x = mulmod(x, y, c);
    y = mulmod(y, y, c);
  }
  return x % c;
}

inline ULL rand64U () {
  return ((ULL)rand() << 48) | ((ULL)rand() << 32) | ((ULL)rand() << 16) | ((ULL)
       rand());
}

bool isPrime (long long N, int k = 5) {
  if (N < 2 || (N != 2 && !(N & 1)))
    return 0;

  ULL s = N - 1, p = N - 1, x, R;

  while (!(s & 1))
    s >>= 1;

  for (int i = 0; i <= k-1; i++) {
    R = powmod(rand64U() % p + 1, s, N);
    for (x = s; x != p && R != 1 && R != p; x <<= 1)
      R = mulmod(R, R, N);
    if (R != p && !(x & 1))
      return 0;
  }
  return 1;
}
```

6.3 Euclid.cc

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

using namespace std;

int mod (int a, int b) {
  return ((a % b) + b) % b;
}

int gcd (int a, int b) {
  return b == 0 ? a : (gcd(b, a % b));
```

```
}

int lcm (int a, int b) {
  return a / gcd(a, b) * b;
}

// returns (d, x, y) such that d = gcd(a, b) and d = ax * by
vector<int> euclid (int a, int b) {
  int x = 1, y = 0, x1 = 0, y1 = 1, t;
  while (b != 0) {
    int q = a / b;
    t = x;
    x = x1;
    x1 = t - q * x1;
    t = y;
    y = y1;
    y1 = t - q * y1;
    t = b;
    b = a - q * b;
    a = t;
  }
  vector<int> ret = {a, x, y};
  if (a <= 0) ret = {-a, -x, -y};
  return ret;
}

// finds all solutions to ax = b mod n
vector<int> linearEquationSolver (int a, int b, int n) {
  vector<int> ret;
  vector<int> res = euclid(a, b);
  int d = res[0], x = res[1];

  if (b % d == 0) {
    x = mod(x * (b / d), n);
    for (int i = 0; i < d; i++)
      ret.push_back(mod(x + i * (n / d), n));
  }

  return ret;
}

// computes x and y such that ax + by = c; on failure, x = y = -1 << 30
void linearDiophantine (int a, int b, int c, int &x, int &y) {
  int d = gcd(a, b);

  if (c % d != 0) {
    x = y = -(1 << 30);
  } else {
    a /= d;
    b /= d;
    c /= d;
    vector<int> ret = euclid(a, b);
    x = ret[1] * c;
    y = ret[2] * c;
  }
}

// precondition: m > 0 && gcd(a, m) = 1
int modInverse (int a, int m) {
  a = mod(a, m);
  return a == 0 ? 0 : mod((1 - modInverse(m % a, a) * m) / a, m);
}

// precondition: p is prime
vector<int> generateInverse (int p) {
  vector<int> res(p);
  res[1] = 1;
  for (int i = 2; i < p; ++i)
    res[i] = (p - (p / i) * res[p % i] % p) % p;
  return res;
}

// solve x = a[i] (mod p[i]), where gcd(p[i], p[j]) == 1
int simpleRestore (vector<int> a, vector<int> p) {
  int res = a[0];
  int m = 1;
  for (int i = 1; i < (int)a.size(); i++) {
    m *= p[i - 1];
    while (res % p[i] != a[i])
      res += m;
  }
  return res;
}

int garnerRestore (vector<int> a, vector<int> p) {
  vector<int> x(a.size());
  for (int i = 0; i < (int)x.size(); ++i) {
    x[i] = a[i];
    for (int j = 0; j < i; ++j) {
      x[i] = (int) modInverse(p[j], p[i]) * (x[i] - x[j]);
      x[i] = (x[i] % p[i] + p[i]) % p[i];
    }
  }
  int res = x[0];
  int m = 1;
  for (int i = 1; i < (int)a.size(); i++) {
    m *= p[i - 1];
    res += x[i] * m;
  }
  return res;
}
```

6.4 Combinatorics.cc

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

typedef long long ll;

ll modpow (ll base, ll pow, ll mod) {
  if (pow == 0)
    return 1L;
  if (pow == 1)
    return base;
  if (pow % 2)
    return base * modpow(base * base % mod, pow / 2, mod) % mod;
  return modpow(base * base % mod, pow / 2, mod);
}

ll factorial (ll n, ll m) {
  ll ret = 1;
  for (int i = 2; i <= n; i++)
    ret = (ret * i) % m;
  return ret;
}

// precondition: p is prime
ll divMod (ll i, ll j, ll p) {
  return i * modpow(j, p - 2, p) % p;
}

// precondition: p is prime; O(log P) if you precompute factorials
ll fastChoose (ll n, ll k, ll p) {
  return divMod(divMod(factorial(n, p), factorial(k, p), p), factorial(n - k, p),
      p);
}

// number of partitions of n
ll partitions (ll n, ll m) {
  ll dp[n + 1];
  memset(dp, 0, sizeof dp);
  dp[0] = 1;
  for (int i = 1; i <= n; i++)
    for (int j = i; j <= n; j++)
      dp[j] = (dp[j] + dp[j - 1]) % m;
  return dp[n] % m;
}

ll stirling1 (int n, int k, long m) {
  ll dp[n + 1][k + 1];
  memset(dp, 0, sizeof dp);
  dp[0][0] = 1;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= k; j++) {
      dp[i][j] = ((i - 1) * dp[i - 1][j]) % m;
      dp[i][j] = (dp[i][j] + dp[i - 1][j - 1]) % m;
```

```
    }
  return dp[n][k];
}

ll stirling2 (int n, int k, ll m) {
  ll dp[n + 1][k + 1];
  memset(dp, 0, sizeof dp);
  dp[0][0] = 1;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= k; j++) {
      dp[i][j] = (j * dp[i - 1][j]) % m;
      dp[i][j] = (dp[i][j] + dp[i - 1][j - 1]) % m;
    }
  return dp[n][k];
}

ll eulerian1 (int n, int k, ll m) {
  if (k > n - 1 - k)
    k = n - 1 - k;
  ll dp[n + 1][k + 1];
  memset(dp, 0, sizeof dp);
  for (int j = 1; j <= k; j++)
    dp[0][j] = 0;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= k; j++) {
      dp[i][j] = ((i - j) * dp[i - 1][j - 1]) % m;
      dp[i][j] = (dp[i][j] + ((j + 1) * dp[i - 1][j]) % m) % m;
    }
  return dp[n][k] % m;
}

ll eulerian2 (int n, int k, ll m) {
  ll dp[n + 1][k + 1];
  memset(dp, 0, sizeof dp);
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= k; j++) {
      if (i == j) {
        dp[i][j] = 0;
      } else {
        dp[i][j] = ((j + 1) % dp[i - 1][j]) % m;
        dp[i][j] = (((2 * i - 1 - j) * dp[i - 1][j - 1]) % m + dp[i][j]) % m;
      }
    }
  return dp[n][k] % m;
}

// precondition: p is prime
ll catalan (int n, ll p) {
  return fastChoose(2 * n, n, p) * modpow(n + 1, p - 2, p) % p;
}
```

6.5 Gauss_Jordon.cc

```
/*
 * 1) Solving system of linear equations (AX=B), stored in B
 * 2) Inverting matrices (AX=I), stored in A
 * 3) Computing determinants of square matrices, returned as T
 *
 * Time: O(N^3)
 * Memory: O(N^2)
 */

#include <bits/stdc++.h>

#define EPS 1e-10

using namespace std;

typedef vector<int> VI;
typedef double T;
typedef vector<T> VT;
typedef vector<VT> VVT;

T GaussJordan(VVT &a, VVT &b) {
  const int n = a.size();
  const int m = b[0].size();
  VI irow(n), icol(n), ipiv(n);
  T det = 1;

  for (int i = 0; i < n; i++) {
    int pj = -1, pk = -1;
    for (int j = 0; j < n; j++) if (!ipiv[j])
      for (int k = 0; k < n; k++) if (!ipiv[k])
        if (pj == -1 || fabs(a[j][k]) > fabs(a[pj][pk])) { pj = j; pk = k; }
    if (fabs(a[pj][pk]) < EPS)
      return 0;
    ipiv[pk]++;
    swap(a[pj], a[pk]);
    swap(b[pj], b[pk]);
    if (pj != pk) det *= -1;
    irow[i] = pj;
    icol[i] = pk;

    T c = 1.0 / a[pk][pk];
    det *= a[pk][pk];
    a[pk][pk] = 1.0;
    for (int p = 0; p < n; p++) a[pk][p] *= c;
    for (int p = 0; p < m; p++) b[pk][p] *= c;
    for (int p = 0; p < n; p++) if (p != pk) {
      c = a[p][pk];
      a[p][pk] = 0;
      for (int q = 0; q < n; q++) a[p][q] -= a[pk][q] * c;
      for (int q = 0; q < m; q++) b[p][q] -= b[pk][q] * c;
    }
  }

  for (int p = n-1; p >= 0; p--) if (irow[p] != icol[p]) {
    for (int k = 0; k < n; k++) swap(a[k][irow[p]], a[k][icol[p]]);
  }
  return det;
}
```

6.6 Matrix.cc

```
/*
 * From Alex Li
 * Basic matrix class with support for arithmetic operations
 * as well as matrix multiplication and exponentiation. You
 * can access/modify indices using m(r, c) or m[r][c]. You
 * can also treat it as a 2d vector, since the cast operator
 * to a reference to its internal 2d vector is defined. This
 * makes it compatible with the 2d vector functions such as
 * det() and lu_decompose() in later sections.
 */

#include <ostream>
#include <vector>
#define cmr const matrix &
#define fbo friend bool operator
#define fmo friend matrix operator

using namespace std;

template<class T> struct matrix {
  int r, c;
  vector<vector<T>> mat;

  matrix(int rows, int cols, T init = T()) {
    r = rows;
    c = cols;
    mat.resize(r, vector<T>(c, init));
  }

  matrix(const vector<vector<T>> & m) {
    r = m.size();
    c = m[0].size();
    mat = m;
    mat.resize(r, vector<T>(c));
  }

  template<size_t rows, size_t cols>
    matrix(T (&init)[rows][cols]) {
      r = rows;
```

```
    c = cols;
    mat.resize(r, vector<T>(c));
    for (int i = 0; i < r; i++)
      for (int j = 0; j < c; j++)
        mat[i][j] = init[i][j];
  }

operator vector<vector<T>>&() { return mat; }
T & operator() (int r, int c) { return mat[r][c]; }
vector<T> & operator[] (int r) { return mat[r]; }

fbo <  (cmr a, cmr b) { return a.mat < b.mat; }
fbo >  (cmr a, cmr b) { return a.mat > b.mat; }
fbo <= (cmr a, cmr b) { return a.mat <= b.mat; }
fbo >= (cmr a, cmr b) { return a.mat >= b.mat; }
fbo == (cmr a, cmr b) { return a.mat == b.mat; }
fbo != (cmr a, cmr b) { return a.mat != b.mat; }

fmo + (cmr a, cmr b) {
  matrix res(a);
  for (int i = 0; i < res.r; i++)
    for (int j = 0; j < res.c; j++)
      res.mat[i][j] += b.mat[i][j];
  return res;
}

fmo - (cmr a, cmr b) {
  matrix res(a);
  for (int i = 0; i < a.r; i++)
    for (int j = 0; j < a.c; j++)
      res.mat[i][j] -= b.mat[i][j];
  return res;
}

fmo * (cmr a, cmr b) {
  matrix res(a.r, b.c, 0);
  for (int i = 0; i < a.r; i++)
    for (int j = 0; j < b.c; j++)
      for (int k = 0; k < a.c; k++)
        res.mat[i][j] += a.mat[i][k] * b.mat[k][j];
  return res;
}

fmo + (cmr a, const T & v) {
  matrix res(a);
  for (int i = 0; i < a.r; i++)
    for (int j = 0; j < a.c; j++) res.mat[i][j] += v;
  return res;
}

fmo - (cmr a, const T & v) {
  matrix res(a);
  for (int i = 0; i < a.r; i++)
    for (int j = 0; j < a.c; j++) res.mat[i][j] -= v;
  return res;
}

fmo * (cmr a, const T & v) {
  matrix res(a);
  for (int i = 0; i < a.r; i++)
    for (int j = 0; j < a.c; j++) res.mat[i][j] *= v;
  return res;
}

fmo / (cmr a, const T & v) {
  matrix res(a);
  for (int i = 0; i < a.r; i++)
    for (int j = 0; j < a.c; j++)
      res.mat[i][j] /= v;
  return res;
}

friend ostream & operator << (ostream & out, cmr m) {
  out << "[";
  for (int i = 0; i < m.r; i++) {
    out << (i > 0 ? ",[" : "[");
    for (int j = 0; j < m.c; j++)
      out << (j > 0 ? "," : "") << m.mat[i][j];
    out << "]";
  }
  out << "]";
  return out;
 }
};

template <class T>
matrix<T> eye(int n) {
  matrix<T> res(n, n);
  for (int i = 0; i < n; i++) res[i][i] = 1;
  return res;
}

template <class T>
matrix<T> operator ^ (const matrix<T>& a, unsigned int n) {
  if (n == 0) return eye<T>(a.r);
  if (n % 2 == 0) return (a * a) ^ (n / 2);
  return a * (a ^ (n - 1));
}

//returns a^1 + a^2 + ... + a^n
template <class T>
matrix<T> powsum(const matrix<T>& a, unsigned int n) {
  if (n == 0) return matrix<T>(a.r, a.r);
  if (n % 2 == 0)
    return powsum(a, n / 2) * (eye<T>(a.r) + (a ^ (n / 2)));
  return a + a * powsum(a, n - 1);
}

/*** Example Usage ***/

#include <cassert>
#include <iostream>
using namespace std;

int main() {
  int a[2][2] = {{1,8}, {5,9}};
  matrix<int> m(5, 5, 10), m2(a);
  for (int i=0;i<m.r;++i)
    for(int j=0;j<m.c;++j)
      m[i][j] += 10;
  m[0][0] += 10;
  assert(m[0][0] == 30 && m[1][1] == 20);
  assert(powsum(m2, 3) == m2 + m2*m2 + (m2^3));
  return 0;
}
```

6.7 FFT.cc

```
/* Time: O(N log N)
 * Memory: O(N)
 */

#include <bits/stdc++.h>

#define M_PI 3.141592653589

using namespace std;

typedef complex<double> C;
typedef long long LL;
vector<C> roots;

vector<C> getRoots (int N) {
  vector<C> ret;
  for (int i = 0; i < N; i++)
    ret.push_back(polar(1.0, 2 * i * M_PI / N));
  return ret;
}

template<class T> void FFT (T *in, C *out, int sz, int step = 1) {
  if (sz == 1) {
    *out = *in;
  } else {
    FFT(in, out, sz >> 1, step << 1);
    FFT(in + step, out + (sz >> 1), sz >> 1, step << 1);
    for (int i = 0, j = 0; i < (sz >> 1); i++, j += step) {
```

```
      auto temp = out[i + (sz >> 1)] * roots[j];
      out[i + (sz >> 1)] = out[i] - temp;
      out[i] = out[i] + temp;
    }
  }
}

vector<double> multiplyPolynomial (vector<double> a, vector<double> b) {
  int N = (int)(a.size() + b.size() - 1);
  while (N & (N - 1))
    N++;

  a.resize(N);
  b.resize(N);
  vector<double> c(N);
  roots = getRoots(N);
  vector<C> x(N), y(N);
  FFT(a.data(), x.data(), N);
  FFT(b.data(), y.data(), N);

  for (int i = 0; i < N; i++) {
    x[i] *= y[i];
    roots[i] = conj(roots[i]);
  }
  FFT(x.data(), y.data(), N);
  vector<double> ret(N);
  for (int i = 0; i < N; i++) {
    ret[i] = real(y[i]) / N;
  }
  return ret;
}

vector<int> multiply (vector<int> a, vector<int> b) {
  int N = (int)(a.size() + b.size());
  while (N & (N - 1))
    N++;

  a.resize(N);
  b.resize(N);
  roots = getRoots(N);
  vector<C> x(N), y(N);
  FFT(a.data(), x.data(), N);
  FFT(b.data(), y.data(), N);

  for (int i = 0; i < N; i++) {
    x[i] *= y[i];
    roots[i] = conj(roots[i]);
  }
  FFT(x.data(), y.data(), N);
  vector<int> ret(N);
  for (int i = 0; i < N; i++) {
    ret[i] = (int)((real(y[i]) + 0.5) / N);
  }

  for (int i = 0; i < (int)ret.size(); i++) {
    if (ret[i] >= 10) {
      if (i == (int)ret.size() - 1)
        ret.push_back(ret[i] / 10);
      else
        ret[i + 1] += ret[i] / 10;
      ret[i] %= 10;
    }
  }
  while (ret.size() > 1 && ret.back() == 0)
    ret.pop_back();
  return ret;
}
```

7 String

7.1 Manacher's.cc

```
/* Time: O(N)
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

string getLongestPalindrome (string s) {
  int len = (int)s.size() * 2 + 1;
  char text[len];
  for (int i = 0; i < len; i++)
    text[i] = '#';
  for (int i = 1; i < len; i += 2)
    text[i] = s[i / 2];
  int maxLen[len];
  memset(maxLen, 0, sizeof maxLen);

  int c = 0, r = 0;
  for (int i = 1; i < len; i++) {
    int j = (c - (i - c));
    maxLen[i] = r > i ? min(r - i, maxLen[j]) : 0;
    while (i + 1 + maxLen[i] < len && i - 1 - maxLen[i] >= 0 && text[i + 1 +
        maxLen[i]] == text[i - 1 - maxLen[i]])
      maxLen[i]++;

    if (i + maxLen[i] > r) {
      r = i + maxLen[i];
      c = i;
    }
  }

  int maxLength = 0;
  int index = 0;
  for (int i = 1; i < len - 1; i++) {
    int currLen = maxLen[i];
    if (currLen > maxLength) {
      maxLength = currLen;
      index = i;
    }
  }
  maxLength = maxLength + (index - maxLength) % 2;
  return s.substr((index - maxLength + 1) / 2, maxLength);
}
```

7.2 KMP.cc

```
/* Time: O(N) query; O(N) construction
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

struct KMP {
  string pattern;
  vector<int> lcp;
  KMP (string pattern): pattern(pattern), lcp(pattern.size()) {
    buildLcp();
  }

  void buildLcp () {
    for (int i = 1; i < (int)pattern.size(); i++) {
      int j = lcp[i - 1];
      while (j > 0 && pattern[j] != pattern[i])
        j = lcp[j - 1];
      if (pattern[j] == pattern[i])
        j++;
      lcp[i] = j;
    }
    for (int i = 0; i < (int)pattern.size(); i++)
      printf("%d\n", lcp[i]);
  }

  int search (string text) {
    int j = 0;
    for (int i = 0; i < (int)text.size(); i++) {
      while (j > 0 && text[i] != pattern[j])
        j = lcp[j - 1];
```

```
      if (text[i] == pattern[j])
        j++;
      if (j == (int)pattern.size())
        return i - j + 1;
    }
    return -1;
  }
};
```

7.3 Rabin_Karp.cc

```
/* Time: O(N) query, O(N) construction
 * Memory: O(N) time
 */

#include <bits/stdc++.h>

#define MOD 1000000007L
#define R 256L

using namespace std;

typedef long long ll;

struct RabinKarp {
  ll pow, patternHash;
  string pattern;
  RabinKarp (string pattern): pattern(pattern) {
    initialize();
  }

  ll getHash (string s, int len) {
    ll ret = 0;
    for (int i = 0; i < len; i++)
      ret = (R * ret + s[i]) % MOD;
    return ret;
  }

  void initialize () {
    patternHash = getHash(pattern, pattern.size());
    pow = 1;
    for (int i = 0; i < (int)pattern.size() - 1; i++)
      pow = (pow * R) % MOD;
  }

  int search (string text) {
    if (pattern.size() > text.size())
      return -1;
    ll currHash = getHash(text, pattern.size());
    if (currHash == patternHash)
      return 0;
    for (int i = (int)pattern.size(); i < (int)text.size(); i++) {
      currHash = ((currHash - pow * text[i - (int)pattern.size()]) % MOD + MOD) %
          MOD;
      currHash = (currHash * R + text[i]) % MOD;
      if (currHash == patternHash)
        return i - (int)pattern.size() + 1;
    }
    return -1;
  }
};
```

7.4 Z_Algorithm.cc

```
/*
 * Produces an array Z where Z[i] is the length of the longest substring
 * starting from S[i] which is also a prefix of S.
 *
 * Time: O(N) construction
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

vector<int> compute (string s) {
  vector<int> z(s.size());
  int l = 0, r = 0;
  for (int i = 1; i < (int)s.size(); i++) {
    if (i > r) {
      l = r = i;
      while (r < (int)s.size() && s[r] == s[r - l])
        r++;
      r--;
      z[i] = r - l + 1;
    } else {
      int j = i - l;
      if (z[j] < r - i + 1)
        z[i] = z[j];
      else {
        l = i;
        while (r < (int)s.size() && s[r] == s[r - l])
          r++;
        r--;
        z[i] = r - l + 1;
      }
    }
  }
  return z;
}
```

7.5 Suffix_Array.cc

```
/* Time: O(N log^2 N)
 * Memory: O(N)
 */

#include <bits/stdc++.h>

using namespace std;

struct Suffix {
  int index;
  pair<int, int> rank;
  Suffix () {}
  Suffix (int index, int rank1, int rank2): index(index), rank{rank1, rank2} {}
  bool operator < (const Suffix& s) const {
    return rank < s.rank;
  }
  bool operator == (const Suffix& s) const {
    return rank == s.rank;
  }
};

vector<int> buildSuffixArray (string s) {
  int N = (int)s.size();
  vector<Suffix> suff(N);
  vector<int> ind(N), ret(N);

  for (int i = 0; i < N; i++)
    suff[i] = Suffix(i, s[i], i + 1 < N ? s[i + 1] : -1);

  for (int i = 2;; i <<= 1) {
    sort(suff.begin(), suff.end());
    ind[suff[0].index] = 0;
    for (int j = 1; j < N; j++)
      ind[suff[j].index] = (suff[j] == suff[j - 1] ? 0 : 1) + ind[suff[j - 1].
          index];
    for (int j = 0; j < N; j++) {
      suff[j].rank.second = suff[j].index + i < N ? ind[suff[j].index + i] : -1;
      suff[j].rank.first = ind[suff[j].index];
    }
    if ((*--suff.end()).rank.first == N - 1)
      break;
  }
  for (int i = 0; i < N; i++)
    ret[ind[i]] = i;
  return ret;
}
```

7.6 Suffix_Tree.cc

```
/* Time: O(N) construction
 * Memory: O(N)
 */

#include <bits/stdc++.h>

#define END 1 << 30
#define RADIX 256

using namespace std;

struct Node {
  // represents the string [s, e)
  int s, e;
  Node *child[RADIX];
  Node *suffix;

  Node (int s, int e): s(s), e(e) {
    for (int i = 0; i < RADIX; i++)
      child[i] = nullptr;
    suffix = nullptr;
  }

  int getLength (int currentPos) {
    return min(currentPos + 1, e) - s;
  }
};
struct SuffixTree {
  string input;
  int len, currentPos, activeEdge, activeLength, remainder;
  bool firstNodeCreated;
  Node *root, *activeNode, *lastNodeCreated;

  SuffixTree (string input): input(input) {
    initialize();
  }

  void initialize () {
    len = input.size();
    root = new Node(0, 0);
    activeEdge = 0;
    activeLength = 0;
    remainder = 0;
    activeNode = root;
    currentPos = 0;
    lastNodeCreated = nullptr;
    firstNodeCreated = false;
  }

  void compute () {
    for (currentPos = 0; currentPos < len; currentPos++)
      addSuffix();
  }

  void addSuffixLink (Node* curr) {
    if (!firstNodeCreated)
      lastNodeCreated->suffix = curr;
    firstNodeCreated = false;
    lastNodeCreated = curr;
  }

  void addSuffix () {
    remainder++;
    firstNodeCreated = true;
    while (remainder > 0) {
      if (activeLength == 0)
        activeEdge = currentPos;
      if (activeNode->child[(int)input[activeEdge]] == nullptr) {
        activeNode->child[(int)input[activeEdge]] = new Node(currentPos, END);
        addSuffixLink(activeNode);
      } else {
        int nextLen = activeNode->child[(int)input[activeEdge]]->getLength(
            currentPos);
        if (activeLength >= nextLen) {
          activeNode = activeNode->child[(int)input[activeEdge]];
          activeEdge += nextLen;
          activeLength -= nextLen;
          continue;
        }

        if (input[activeNode->child[(int)input[activeEdge]]->s + activeLength] ==
            input[currentPos]) {
          activeLength++;
          addSuffixLink(activeNode);
          break;
        } else {
          Node* old = activeNode->child[(int)input[activeEdge]];
          Node* split = new Node(old->s, old->s + activeLength);
          activeNode->child[(int)input[activeEdge]] = split;
          Node* leaf = new Node(currentPos, END);
          split->child[(int)input[currentPos]] = leaf;
          old->s += activeLength;
          split->child[(int)input[old->s]] = old;
          addSuffixLink(split);
        }
      }
      remainder--;
      if (activeNode == root && activeLength > 0) {
        activeLength--;
        activeEdge = currentPos - remainder + 1;
      } else {
        if (activeNode->suffix != nullptr) {
          activeNode = activeNode->suffix;
        } else {
          activeNode = root;
        }
      }
    }
  }
};
```

8 Misc

8.1 .vimrc

```
syntax on
set nocp
set rnu nu
set ts=2 sts=2 sw=2
set ai si et sta
set cul is ls=2
set mouse=a

set makeprg=g++\ -std=c++14\ -g\ -Wall\ -Wextra\ -fsanitize=undefined,address\ %\
    -o\ %:r
command -nargs=* Make make <args> | cwindow
nnoremap <buffer> <F4> :Make<CR>
nnoremap <buffer> <F5> :!./%:r<CR>

nnoremap <buffer> ]q :cn<CR>
nnoremap <buffer> [q :cp<CR>
```