Team notebook

MaxHeap

May 3, 2019

Contents

1 Datastructures **1**
- 1.1 SegmentTree . . . 1
- 1.2 SegmentTreeFast . . . 1
- 1.3 SegmentTreeIntervalMax . . . 3
- 1.4 Sparse . . . 4

2 DynamicProgramming **4**
- 2.1 Lis . . . 4

3 Geometry **5**
- 3.1 LineGeometry . . . 5

4 Graph **7**
- 4.1 Lca . . . 7
- 4.2 MaxFlowDinic . . . 8
- 4.3 MaximumMatching . . . 9
- 4.4 SccTarjan . . . 9
- 4.5 TopologicalSort . . . 10

5 Math **10**
- 5.1 Matrix . . . 10

6 Misc **11**
- 6.1 Mo . . . 11
- 6.2 TernarySearch . . . 11

7 Strings **12**
- 7.1 Hash . . . 12
- 7.2 Kmp . . . 13
- 7.3 SuffixArray . . . 13
- 7.4 ZAlgorithm . . . 14

8 Tree **14**
- 8.1 DsuTemplate . . . 14

1 Datastructures

1.1 SegmentTree

```
class SegmentTree {
  int[] st, delta;

  void build(int root, int l, int r) {
    if (l == r) {
      st[root] = arr[l];
      return;
    }
    int mid = (l + r) >> 1;
    build(root * 2, l, mid);
    build(root * 2 + 1, mid + 1, r);
    st[root] = st[root * 2] + st[root * 2 + 1];
  }

  void propagete(int root, int l, int r) {
    if (delta[root] != 0) {
      st[root] += (r - l + 1) * delta[root];
      delta[root * 2] += delta[root];
      delta[root * 2 + 1] += delta[root];
      delta[root] = 0;
    }
  }
```

```
  int query(int root, int l, int r, int f, int t) {
    if (f > r || t < l)
      return 0;
    if (f <= l && r <= t) {
      return st[root] + delta[root];
    }
    propagete(root, l, r);
    int mid = (l + r) >> 1;
    int left = query(root * 2, l, mid, f, t);
    int right = query(root * 2 + 1, mid + 1, r, f, t);
    return left + right;
  }

  void update(int root, int l, int r, int f, int t, int val) {
    if (f > r || t < l)
      return;
    if (f <= l && r <= t) {
      delta[root] += val;
      return;
    }
    propagete(root, l, r);
    int mid = (l + r) >> 1;
    update(root * 2, l, mid, f, t, val);
    update(root * 2 + 1, mid + 1, r, f, t, val);
    st[root] = st[root * 2] + st[root * 2 + 1];
  }
}
```

1.2 SegmentTreeFast

```
class SegmentTreeFast {

  // Modify the following 5 methods to implement your custom operations
      on the tree.
  // This example implements Add/Max operations. Operations like Add/Sum,
      Set/Max can also be implemented.
  int modifyOperation(int x, int y) {
    return x + y;
  }
  // query (or combine) operation
  int queryOperation(int leftValue, int rightValue) {
    return Math.max(leftValue, rightValue);
  }
  int deltaEffectOnSegment(int delta, int segmentLength) {
    if (delta == getNeutralDelta()) return getNeutralDelta();
    // Here you must write a fast equivalent of following slow code:
    // int result = delta;
    // for (int i = 1; i < segmentLength; i++) result =
        queryOperation(result, delta);
    // return result;
    return delta;
  }

  int getNeutralDelta() {
    return 0;
  }

  int getInitValue() {
    return 0;
  }

  // generic code
  int[] value;
  int[] delta; // delta[i] affects value[2*i+1], value[2*i+2],
      delta[2*i+1] and delta[2*i+2]
  int[] len;

  int joinValueWithDelta(int value, int delta) {
    if (delta == getNeutralDelta()) return value;
    return modifyOperation(value, delta);
  }

  int joinDeltas(int delta1, int delta2) {
    if (delta1 == getNeutralDelta()) return delta2;
    if (delta2 == getNeutralDelta()) return delta1;
    return modifyOperation(delta1, delta2);
  }

  void pushDelta(int i) {
    int d = 0;
    for (; (i >> d) > 0; d++)
      ;
    for (d -= 2; d >= 0; d--) {
      int x = i >> d;
      value[x] = joinValueWithDelta(value[x],
          deltaEffectOnSegment(delta[x >> 1], len[x]));
```

```
    value[(x ^ 1)] = joinValueWithDelta(value[(x ^ 1)],
        deltaEffectOnSegment(delta[x >> 1], len[(x ^ 1)]));
    delta[x] = joinDeltas(this.delta[x], delta[x >> 1]);
    delta[(x ^ 1)] = joinDeltas(this.delta[(x ^ 1)], delta[x >> 1]);
    delta[x >> 1] = getNeutralDelta();
  }
}

public SegmentTreeFast2(int n) {
  value = new int[2 * n];
  for (int i = 0; i < n; i++)
    value[i + n] = getInitValue();
  for (int i = 2 * n - 1; i > 1; i -= 2)
    value[i >> 1] = queryOperation(value[i], value[i ^ 1]);

  delta = new int[2 * n];
  Arrays.fill(delta, getNeutralDelta());

  len = new int[2 * n];
  Arrays.fill(len, n, 2 * n, 1);
  for (int i = 2 * n - 1; i > 1; i -= 2)
    len[i >> 1] = len[i] + len[i ^ 1];
}

public int query(int from, int to) {
  from += value.length >> 1;
  to += value.length >> 1;
  pushDelta(from);
  pushDelta(to);
  int res = 0;
  boolean found = false;
  for (; from <= to; from = (from + 1) >> 1, to = (to - 1) >> 1) {
    if ((from & 1) != 0) {
      res = found ? queryOperation(res, value[from]) : value[from];
      found = true;
    }
    if ((to & 1) == 0) {
      res = found ? queryOperation(res, value[to]) : value[to];
      found = true;
    }
  }
  if (!found)
    throw new RuntimeException();
  return res;
}
```

```
  public void modify(int from, int to, int delta) {
    from += value.length >> 1;
    to += value.length >> 1;
    pushDelta(from);
    pushDelta(to);
    int ta = -1;
    int tb = -1;
    for (; from <= to; from = (from + 1) >> 1, to = (to - 1) >> 1) {
      if ((from & 1) != 0) {
        value[from] = joinValueWithDelta(value[from],
            deltaEffectOnSegment(delta, len[from]));
        this.delta[from] = joinDeltas(this.delta[from], delta);
        if (ta == -1)
          ta = from;
      }
      if ((to & 1) == 0) {
        value[to] = joinValueWithDelta(value[to],
            deltaEffectOnSegment(delta, len[to]));
        this.delta[to] = joinDeltas(this.delta[to], delta);
        if (tb == -1)
          tb = to;
      }
    }
    for (int i = ta; i > 1; i >>= 1)
      value[i >> 1] = queryOperation(value[i], value[i ^ 1]);
    for (int i = tb; i > 1; i >>= 1)
      value[i >> 1] = queryOperation(value[i], value[i ^ 1]);
  }
}
```

1.3 SegmentTreeIntervalMax

```
class SegmentTreeIntervalAddMax {
  int n;
  int[] tmax;
  int[] tadd; // tadd[i] affects tmax[i], tadd[2*i+1] and tadd[2*i+2]

  void push(int root) {
    tmax[root] += tadd[root];
    tadd[2 * root + 1] += tadd[root];
    tadd[2 * root + 2] += tadd[root];
    tadd[root] = 0;
```

```
}

public SegmentTreeIntervalAddMax(int n) {
  this.n = n;
  tmax = new int[4 * n];
  tadd = new int[4 * n];
}

public int max(int from, int to) {
  return max(from, to, 0, 0, n - 1);
}

int max(int from, int to, int root, int left, int right) {
  if (from == left && to == right) {
    return tmax[root] + tadd[root];
  }
  push(root);
  int mid = (left + right) >> 1;
  int res = Integer.MIN_VALUE;
  if (from <= mid)
    res = Math.max(res, max(from, Math.min(to, mid), 2 * root + 1,
        left, mid));
  else if (to > mid)
    res = Math.max(res, max(Math.max(from, mid + 1), to, 2 * root + 2,
        mid + 1, right));
  return res;
}

public void add(int from, int to, int delta) {
  add(from, to, delta, 0, 0, n - 1);
}

void add(int from, int to, int delta, int root, int left, int right) {
  if (from == left && to == right) {
    tadd[root] += delta;
    return;
  }
  // push can be skipped for add, but is necessary for other operations
      such as set
  push(root);
  int mid = (left + right) >> 1;
  if (from <= mid)
    add(from, Math.min(to, mid), delta, 2 * root + 1, left, mid);
  if (to > mid)
    add(Math.max(from, mid + 1), to, delta, 2 * root + 2, mid + 1,
        right);
  tmax[root] = Math.max(tmax[2 * root + 1] + tadd[2 * root + 1], tmax[2
      * root + 2] + tadd[2 * root + 2]);
 }
}
```

1.4 Sparse

```
class Sparse {
  int[] log;
  int[][] sparse;
  int[] arr;
  int k = 22;

  public Sparse(int[] arr, int n) {
    log = new int[n + 1];
    sparse = new int[n + 1][k];
    this.arr = arr;
    build();
  }

  void build() {
    for (int i = 2; i <= n; ++i)
      log[i] = log[i >> 1] + 1;

    for (int i = 0; i < N; i++)
      sparse[i][0] = arr[i];

    for (int j = 1; j <= K; j++)
      for (int i = 0; i + (1 << j) <= N; i++)
        sparse[i][j] = Math.min(sparse[i][j - 1], sparse[i + (1 << (j -
            1))][j - 1]);
  }

  int query(int l, int r) {
    int j = log[r - l + 1];
    return Math.min(sparse[l][j], sparse[r - (1 << j) + 1][j]);
  }
}
```

2 DynamicProgramming

2.1 Lis

```
class Lis {
  public class Lis2 {

    public static int[] lis(int[] a) {
      int n = a.length;
      int[] tail = new int[n];
      int[] prev = new int[n];

      int len = 0;
      for (int i = 0; i < n; i++) {
        int pos = lower_bound(a, tail, len, a[i]);
        len = Math.max(len, pos + 1);
        prev[i] = pos > 0 ? tail[pos - 1] : -1;
        tail[pos] = i;
      }

      int[] res = new int[len];
      for (int i = tail[len - 1]; i >= 0; i = prev[i]) {
        res[--len] = a[i];
      }
      return res;
    }

    static int lower_bound(int[] a, int[] tail, int len, int key) {
      int lo = -1;
      int hi = len;
      while (hi - lo > 1) {
        int mid = (lo + hi) >>> 1;
        if (a[tail[mid]] <= key) {
          lo = mid;
        } else {
          hi = mid;
        }
      }
      return hi;
    }
    public static int lisSize(int[] a) {
      NavigableSet<Integer> s = new TreeSet<>();
      for (int v : a)
        if (s.add(v)) {
          Integer higher = s.higher(v);
          if (higher != null)
            s.remove(higher);
        }
      return s.size();
    }
    static int lisSize2(int[] a) {
      int[] last = new int[a.length];
      Arrays.fill(last, Integer.MAX_VALUE);
      int len = 0;
      for (int v : a) {
        int pos = lower_bound(last, v);
        last[pos] = v;
        len = Math.max(len, pos + 1);
      }
      return len;
    }
    static int lower_bound(int[] a, int key) {
      int lo = -1;
      int hi = a.length;
      while (hi - lo > 1) {
        int mid = (lo + hi) >>> 1;
        if (a[mid] < key) {
          lo = mid;
        } else {
          hi = mid;
        }
      }
      return hi;
    }
}
```

3 Geometry

3.1 LineGeometry

```
class LineGeometry {
  static final double EPS = 1e-10;
  public static int sign(double a) {
    return a < -EPS ? -1 : a > EPS ? 1 : 0;
  }
```

```
public static class Point implements Comparable<Point> {
  public double x, y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public Point minus(Point b) {
    return new Point(x - b.x, y - b.y);
  }

  public double cross(Point b) {
    return x * b.y - y * b.x;
  }

  public double dot(Point b) {
    return x * b.x + y * b.y;
  }

  public Point rotateCCW(double angle) {
    return new Point(x * Math.cos(angle) - y * Math.sin(angle), x *
        Math.sin(angle) + y * Math.cos(angle));
  }

  @Override
  public int compareTo(Point o) {
    // return Double.compare(Math.atan2(y, x), Math.atan2(o.y, o.x));
    return Double.compare(x, o.x) != 0 ? Double.compare(x, o.x) :
        Double.compare(y, o.y);
  }
}

public static class Line {
  public double a, b, c;

  public Line(double a, double b, double c) {
    this.a = a;
    this.b = b;
    this.c = c;
  }

  public Line(Point p1, Point p2) {
    a = +(p1.y - p2.y);
    b = -(p1.x - p2.x);
    c = p1.x * p2.y - p2.x * p1.y;
  }

  public Point intersect(Line line) {
    double d = a * line.b - line.a * b;
    if (sign(d) == 0) {
      return null;
    }
    double x = -(c * line.b - line.c * b) / d;
    double y = -(a * line.c - line.a * c) / d;
    return new Point(x, y);
  }
}

// Returns -1 for clockwise, 0 for straight line, 1 for
    counterclockwise order
public static int orientation(Point a, Point b, Point c) {
  Point AB = b.minus(a);
  Point AC = c.minus(a);
  return sign(AB.cross(AC));
}

public static boolean cw(Point a, Point b, Point c) {
  return orientation(a, b, c) < 0;
}

public static boolean ccw(Point a, Point b, Point c) {
  return orientation(a, b, c) > 0;
}

public static boolean isCrossIntersect(Point a, Point b, Point c, Point
    d) {
  return orientation(a, b, c) * orientation(a, b, d) < 0 &&
      orientation(c, d, a) * orientation(c, d, b) < 0;
}

public static boolean isCrossOrTouchIntersect(Point a, Point b, Point
    c, Point d) {
  if (Math.max(a.x, b.x) < Math.min(c.x, d.x) - EPS || Math.max(c.x,
      d.x) < Math.min(a.x, b.x) - EPS
      || Math.max(a.y, b.y) < Math.min(c.y, d.y) - EPS || Math.max(c.y,
          d.y) < Math.min(a.y, b.y) - EPS) {
    return false;
  }
```

```
  return orientation(a, b, c) * orientation(a, b, d) <= 0 &&
      orientation(c, d, a) * orientation(c, d, b) <= 0;
}

public static double pointToLineDistance(Point p, Line line) {
  return Math.abs(line.a * p.x + line.b * p.y + line.c) /
      fastHypot(line.a, line.b);
}

public static double fastHypot(double x, double y) {
  return Math.sqrt(x * x + y * y);
}

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

public static double angleBetween(Point a, Point b) {
  return Math.atan2(a.cross(b), a.dot(b));
}

public static double angle(Line line) {
  return Math.atan2(-line.a, line.b);
}

public static double signedArea(Point[] points) {
  int n = points.length;
  double area = 0;
  for (int i = 0, j = n - 1; i < n; j = i++) {
    area += (points[i].x - points[j].x) * (points[i].y + points[j].y);
    // area += points[i].x * points[j].y - points[j].x * points[i].y;
  }
  return area / 2;
}

public static enum Position {
  LEFT, RIGHT, BEHIND, BEYOND, ORIGIN, DESTINATION, BETWEEN
}

// Classifies position of point p against vector a
public static Position classify(Point p, Point a) {
  int s = sign(a.cross(p));
  if (s > 0) {
    return Position.LEFT;
  }
  if (s < 0) {
    return Position.RIGHT;
  }
  if (sign(p.x) == 0 && sign(p.y) == 0) {
    return Position.ORIGIN;
  }
  if (sign(p.x - a.x) == 0 && sign(p.y - a.y) == 0) {
    return Position.DESTINATION;
  }
  if (a.x * p.x < 0 || a.y * p.y < 0) {
    return Position.BEYOND;
  }
  if (a.x * a.x + a.y * a.y < p.x * p.x + p.y * p.y) {
    return Position.BEHIND;
  }
  return Position.BETWEEN;
}
// cuts right part of poly (returns left part)
public static Point[] convexCut(Point[] poly, Point p1, Point p2) {
  int n = poly.length;
  List<Point> res = new ArrayList<>();
  for (int i = 0, j = n - 1; i < n; j = i++) {
    int d1 = orientation(p1, p2, poly[j]);
    int d2 = orientation(p1, p2, poly[i]);
    if (d1 >= 0)
      res.add(poly[j]);
    if (d1 * d2 < 0)
      res.add(new Line(p1, p2).intersect(new Line(poly[j], poly[i])));
  }
  return res.toArray(new Point[res.size()]);
}
}
```

4 Graph

4.1 Lca

```
public class Lca {
  int[] depth;
  int[] dfs_order;
  int cnt;
  int[] first;
```

```
int[] minPos;
int n;
void dfs(List<Integer>[] tree, int u, int d) {
  depth[u] = d;
  dfs_order[cnt++] = u;
  for (int v : tree[u])
    if (depth[v] == -1) {
      dfs(tree, v, d + 1);
      dfs_order[cnt++] = u;
    }
}
void buildTree(int node, int left, int right) {
  if (left == right) {
    minPos[node] = dfs_order[left];
    return;
  }
  int mid = (left + right) >> 1;
  buildTree(2 * node + 1, left, mid);
  buildTree(2 * node + 2, mid + 1, right);
  minPos[node] = depth[minPos[(node << 1) + 1]] < depth[minPos[(node <<
      1) + 2]] ?
      minPos[(node << 1) + 1] : minPos[(node << 1) + 2];
}
public Lca(List<Integer>[] tree, int root) {
  int nodes = tree.length;
  depth = new int[nodes];
  Arrays.fill(depth, -1);

  n = 2 * nodes - 1;
  dfs_order = new int[n];
  cnt = 0;
  dfs(tree, root, 0);

  minPos = new int[4 * n];
  buildTree(0, 0, n - 1);

  first = new int[nodes];
  Arrays.fill(first, -1);
  for (int i = 0; i < dfs_order.length; i++)
    if (first[dfs_order[i]] == -1)
      first[dfs_order[i]] = i;
}

public int lca(int a, int b) {
    return minPos(Math.min(first[a], first[b]), Math.max(first[a],
        first[b]), 0, 0, n - 1);
  }

  int minPos(int a, int b, int node, int left, int right) {
    if (a == left && right == b)
      return minPos[node];
    int mid = (left + right) >> 1;
    if (a <= mid && b > mid) {
      int p1 = minPos(a, Math.min(b, mid), 2 * node + 1, left, mid);
      int p2 = minPos(Math.max(a, mid + 1), b, 2 * node + 2, mid + 1,
          right);
      return depth[p1] < depth[p2] ? p1 : p2;
    } else if (a <= mid) {
      return minPos(a, Math.min(b, mid), 2 * node + 1, left, mid);
    } else if (b > mid) {
      return minPos(Math.max(a, mid + 1), b, 2 * node + 2, mid + 1,
          right);
    } else {
      throw new RuntimeException();
    }
  }
}
```

4.2 MaxFlowDinic

```
public class MaxFlowDinic {
  static class Edge {
    int t, rev, cap, f;

    public Edge(int t, int rev, int cap) {
      this.t = t;
      this.rev = rev;
      this.cap = cap;
    }
  }

  public static void addEdge(List<Edge>[] graph, int s, int t, int cap) {
    graph[s].add(new Edge(t, graph[t].size(), cap));
    graph[t].add(new Edge(s, graph[s].size() - 1, 0));
  }
```

```
static boolean dinicBfs(List<Edge>[] graph, int src, int dest, int[]
    dist) {
  Arrays.fill(dist, -1);
  dist[src] = 0;
  int[] Q = new int[graph.length];
  int sizeQ = 0;
  Q[sizeQ++] = src;
  for (int i = 0; i < sizeQ; i++) {
    int u = Q[i];
    for (Edge e : graph[u]) {
      if (dist[e.t] < 0 && e.f < e.cap) {
        dist[e.t] = dist[u] + 1;
        Q[sizeQ++] = e.t;
      }
    }
  }
  return dist[dest] >= 0;
}

static int dinicDfs(List<Edge>[] graph, int[] ptr, int[] dist, int
    dest, int u, int f) {
  if (u == dest)
    return f;
  for (; ptr[u] < graph[u].size(); ++ptr[u]) {
    Edge e = graph[u].get(ptr[u]);
    if (dist[e.t] == dist[u] + 1 && e.f < e.cap) {
      int df = dinicDfs(graph, ptr, dist, dest, e.t, Math.min(f, e.cap -
          e.f));
      if (df > 0) {
        e.f += df;
        graph[e.t].get(e.rev).f -= df;
        return df;
      }
    }
  }
  return 0;
}

public static int maxFlow(List<Edge>[] graph, int src, int dest) {
  int flow = 0;
  int[] dist = new int[graph.length];
  while (dinicBfs(graph, src, dest, dist)) {
    int[] ptr = new int[graph.length];
    while (true) {
      int df = dinicDfs(graph, ptr, dist, dest, src, Integer.MAX_VALUE);
      if (df == 0)
        break;
      flow += df;
    }
  }
  return flow;
}
}
```

4.3 MaximumMatching

```
class MaxMatching {
  public static int maxMatching(List<Integer>[] graph, int n2) {
    int n1 = graph.length;
    int[] matching = new int[n2];
    Arrays.fill(matching, -1);
    int matches = 0;
    for (int u = 0; u < n1; u++) {
      if (findPath(graph, u, matching, new boolean[n1])) {
        ++matches;
      }
    }
    return matches;
  }

  public static int[] getMaxMatches(List<Integer>[] graph, int n2) {
    int n1 = graph.length;
    int[] matching = new int[n2];
    Arrays.fill(matching, -1);
    int matches = 0;
    for (int u = 0; u < n1; u++) {
      if (findPath(graph, u, matching, new boolean[n1])) {
        ++matches;
      }
    }
    return matching;
  }

  static boolean findPath(List<Integer>[] graph, int u1, int[] matching,
      boolean[] vis) {
    vis[u1] = true;
    for (int v : graph[u1]) {
      int u2 = matching[v];
```

```
    if (u2 == -1 || !vis[u2] && findPath(graph, u2, matching, vis)) {
      matching[v] = u1;
      return true;
    }
  }
  return false;
 }
}
```

4.4 SccTarjan

```
public class SCCTarjan {
  List<Integer>[] graph;
  boolean[] visited;
  Stack<Integer> stack;
  int time;
  int[] lowlink;
  List<List<Integer>> components;

  public List<List<Integer>> scc(List<Integer>[] graph) {
    int n = graph.length;
    this.graph = graph;
    visited = new boolean[n];
    stack = new Stack<>();
    time = 0;
    lowlink = new int[n];
    components = new ArrayList<>();

    for (int u = 0; u < n; u++)
      if (!visited[u])
        dfs(u);

    return components;
  }

  void dfs(int u) {
    lowlink[u] = time++;
    visited[u] = true;
    stack.add(u);
    boolean isComponentRoot = true;

    for (int v : graph[u]) {
      if (!visited[v])
        dfs(v);
      if (lowlink[u] > lowlink[v]) {
        lowlink[u] = lowlink[v];
        isComponentRoot = false;
      }
    }

    if (isComponentRoot) {
      List<Integer> component = new ArrayList<>();
      while (true) {
        int x = stack.pop();
        component.add(x);
        lowlink[x] = Integer.MAX_VALUE;
        if (x == u)
          break;
      }
      components.add(component);
    }
  }
}
```

4.5 TopologicalSort

```
class TopologicalSort {
  static void dfs(List<Integer>[] graph, boolean[] used, List<Integer>
      order, int u) {
    used[u] = true;
    for (int v : graph[u])
      if (!used[v])
        dfs(graph, used, order, v);
    order.add(u);
  }
  public static List<Integer> topologicalSort(List<Integer>[] graph) {
    int n = graph.length;
    boolean[] used = new boolean[n];
    List<Integer> order = new ArrayList<>();
    for (int i = 0; i < n; i++)
      if (!used[i])
        dfs(graph, used, order, i);
    Collections.reverse(order);
    return order;
  }
```

```
}
```

5 Math

5.1 Matrix

```
public class Matrix {
  public static int[][] matrixAdd(int[][] a, int[][] b) {
    int n = a.length;
    int m = a[0].length;
    int[][] res = new int[n][m];
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
        res[i][j] = a[i][j] + b[i][j];
      }
    }
    return res;
  }
  public static long[][] matrixMul(long[][] a, long[][] b, long mod) {
    int n = a.length;
    int m = a[0].length;
    int k = b[0].length;
    long[][] res = new long[n][k];
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < k; j++) {
        for (int p = 0; p < m; p++) {
          res[i][j] = (res[i][j] + a[i][p] * b[p][j] % mod + mod) % mod;
        }
      }
    }
    return res;
  }
  public static long[][] matrixPow(long[][] a, long p, long mod) {
    if (p == 0) {
      return matrixUnitLong(a.length);
    } else if (p % 2 == 0) {
      return matrixPow(matrixMul(a, a, mod), p / 2, mod);
    } else {
      return matrixMul(a, matrixPow(a, p - 1, mod), mod);
    }
  }
  public static int[][] matrixPowSum(int[][] a, int p) {
    int n = a.length;
    if (p == 0) {
      return new int[n][n];
    }
    if (p % 2 == 0) {
      return matrixMul(matrixPowSum(a, p / 2), matrixAdd(matrixUnit(n),
          matrixPow(a, p / 2)));
    } else {
      return matrixAdd(a, matrixMul(matrixPowSum(a, p - 1), a));
    }
  }
  public static int[][] matrixUnit(int n) {
    int[][] res = new int[n][n];
    for (int i = 0; i < n; ++i) {
      res[i][i] = 1;
    }
    return res;
  }
}
```

6 Misc

6.1 Mo

```
class Mo {
  public static class Query {
    int index;
    int a;
    int b;

    public Query(int a, int b) {
      this.a = a;
      this.b = b;
    }
  }

  static int add(int[] a, int[] cnt, int i) {
    return ++cnt[a[i]] == 1 ? 1 : 0;
  }

  static int remove(int[] a, int[] cnt, int i) {
```

```
    return --cnt[a[i]] == 0 ? -1 : 0;
  }

  public static int[] processQueries(int[] a, Query[] queries) {
    for (int i = 0; i < queries.length; i++) queries[i].index = i;
    int sqrtn = (int) Math.sqrt(a.length);
    Arrays.sort(queries,
        Comparator.comparingInt((Query q) -> q.a /
            sqrtn).thenComparingInt(q -> q.b));
    int[] cnt = new int[1000_002];
    int[] res = new int[queries.length];
    int L = 1;
    int R = 0;
    int cur = 0;
    for (Query query : queries) {
      while (L < query.a) cur += remove(a, cnt, L++);
      while (L > query.a) cur += add(a, cnt, --L);
      while (R < query.b) cur += add(a, cnt, ++R);
      while (R > query.b) cur += remove(a, cnt, R--);
      res[query.index] = cur;
    }
    return res;
  }
}
```

6.2 TernarySearch

```
public class TernarySearch {
  public static double ternarySearch(DoubleUnaryOperator f, double lo,
      double hi) {
    for (int step = 0; step < 1000; step++) {
      double m1 = lo + (hi - lo) / 3;
      double m2 = hi - (hi - lo) / 3;
      if (f.applyAsDouble(m1) < f.applyAsDouble(m2))
        lo = m1;
      else
        hi = m2;
    }
    return (lo + hi) / 2;
  }
  public static int ternarySearch(IntUnaryOperator f, int fromInclusive,
      int toInclusive) {
    int lo = fromInclusive;
    int hi = toInclusive;
    while (hi > lo + 2) {
      int m1 = lo + (hi - lo) / 3;
      int m2 = hi - (hi - lo) / 3;
      if (f.applyAsInt(m1) < f.applyAsInt(m2))
        lo = m1;
      else
        hi = m2;
    }
    int res = lo;
    for (int i = lo + 1; i <= hi; i++)
      if (f.applyAsInt(res) < f.applyAsInt(i))
        res = i;
    return res;
  }

  public static int ternarySearch2(IntUnaryOperator f, int fromInclusive,
      int toInclusive) {
    int lo = fromInclusive - 1;
    int hi = toInclusive;
    while (hi - lo > 1) {
      int mid = (lo + hi) >>> 1;
      if (f.applyAsInt(mid) < f.applyAsInt(mid + 1)) {
        lo = mid;
      } else {
        hi = mid;
      }
    }
    return hi;
  }
}
```

7 Strings

7.1 Hash

```
class Hash {
  static int mul = 131;
  static final Random random = new Random();
  static final long firstMod = 1297425359;
  static final long secondMod = 1859599523;
  static final long firstInvMul = BigInteger.valueOf(mul)
```

```
    .modInverse(BigInteger.valueOf(firstMod))
    .longValue();
static final long secondInvMul = BigInteger.valueOf(mul)
    .modInverse(BigInteger.valueOf(secondMod))
    .longValue();

long[] firstHash, secondHash;
long[] firstInv, secondInv;
int n;

public Hash(String s) {
  initialize(s);
}

public Hash(String s, int mul) {
  this.mul = mul;
  initialize(s);
}

private void initialize(String s) {
  n = s.length();
  firstHash = new long[n + 1];
  secondHash = new long[n + 1];
  firstInv = new long[n + 1];
  secondInv = new long[n + 1];
  firstInv[0] = 1;
  secondInv[0] = 1;

  long powerFirstMod = 1;
  long powerSecondMod = 1;
  for (int i = 0; i < n; i++) {
    firstHash[i + 1] = (firstHash[i] + s.charAt(i) * powerFirstMod) %
        firstMod;
    powerFirstMod = powerFirstMod * mul % firstMod;
    firstInv[i + 1] = firstInv[i] * firstInvMul % firstMod;
    secondHash[i + 1] = (secondHash[i] + s.charAt(i) * powerSecondMod)
        % secondMod;
    powerSecondMod = powerSecondMod * mul % secondMod;
    secondInv[i + 1] = secondInv[i] * secondInvMul % secondMod;
  }
}

public long getHash(int i, int len) {
  return (((firstHash[i + len] - firstHash[i] + firstMod) * firstInv[i]
      % firstMod) << 32)
      + (secondHash[i + len] - secondHash[i] + secondMod) * secondInv[i]
          % secondMod;
  }
}
```

7.2 Kmp

```
class Kmp {
  public static int[] prefixFunction(String s) {
    int[] p = new int[s.length()];
    int k = 0;
    for (int i = 1; i < s.length(); i++) {
      while (k > 0 && s.charAt(k) != s.charAt(i)) {
        k = p[k - 1];
      }
      if (s.charAt(k) == s.charAt(i)) {
        ++k;
      }
      p[i] = k;
    }
    return p;
  }

  public static void kmpMatcher(String s, String pattern) {
    int m = pattern.length();
    int[] p = prefixFunction(pattern);
    for (int i = 0, k = 0; i < s.length(); i++) {
      System.out.println(k);
      while (k > 0 && pattern.charAt(k) != s.charAt(i)) {
        k = p[k - 1];
      }
      if (pattern.charAt(k) == s.charAt(i)) {
        ++k;
      }
      if (k == m) {
        System.out.println("found " + (i + 1 - m));
        k = p[k - 1];
      }
    }
  }
}
```

7.3 SuffixArray

```
public class SuffixArray {

  // sort suffixes of S in O(n*log(n))
  public static int[] suffixArray(CharSequence S) {
    int n = S.length();

    // stable sort of characters
    int[] sa = IntStream.range(0, n).mapToObj(i -> n - 1 - i).
        sorted(Comparator.comparingInt(S::charAt))
        .mapToInt(Integer::intValue).toArray();

    int[] classes = S.chars().toArray();
    // sa[i] - suffix on i'th position after sorting by first len
        characters
    // classes[i] - equivalence class of the i'th suffix after sorting by
        first len characters
    for (int  len = 1; len < n; len *= 2) {
      int[] c = classes.clone();
      for (int i = 0; i < n; i++) {
        // condition sa[i - 1] + len < n simulates 0-symbol at the point
            of the string
        // a separate class is created for each suffix followed by
            simulated 0-symbol
        classes[sa[i]] = i > 0 && c[sa[i - 1]] == c[sa[i]] && sa[i - 1] +
            len < n
           && c[sa[i - 1] + len / 2] == c[sa[i] + len / 2] ? classes[sa[i
               - 1]] : i;
      }
      // Suffixes are already sorted by first len characters
      // Now sort suffixes by first len * 2 characters
      int[] cnt = IntStream.range(0, n).toArray();
      int[] s = sa.clone();
      for (int i = 0; i < n; i++) {
        // s[i] - order of suffixes sorted by first len characters
        // (s[i] - len) - orde
        // r of suffixes sorted only by second len characters
        int s1 = s[i] - len;
        // sort only suffixes of length > len, others are already sorted
        if (s1 >= 0)
          sa[cnt[classes[s1]]++] = s1;
      }
    }
    return sa;
  }

  public static int[] rotationArray(CharSequence S) {
    int n = S.length();
    int[] sa = IntStream.range(0, n).mapToObj(Integer::valueOf).
        sorted((a, b) -> Character.compare(S.charAt(a), S.charAt(b)))
        .mapToInt(Integer::intValue).toArray();
    int[] classes = S.chars().toArray();
    for (int len = 1; len < n; len *= 2) {
      int[] c = classes.clone();
      for (int i = 0; i < n; i++)
        classes[sa[i]] = i > 0 && c[sa[i - 1]] == c[sa[i]] &&
        c[(sa[i - 1] + len / 2) % n] == c[(sa[i] + len / 2) % n] ?
            classes[sa[i - 1]] : i;
      int[] cnt = IntStream.range(0, n).toArray();
      int[] s = sa.clone();
      for (int i = 0; i < n; i++) {
        int s1 = (s[i] - len + n) % n;
        sa[cnt[classes[s1]]++] = s1;
      }
    }
    return sa;
  }
  public static int[] lcp(int[] sa, CharSequence s) {
    int n = sa.length;
    int[] rank = new int[n];
    for (int i = 0; i < n; i++)
      rank[sa[i]] = i;
    int[] lcp = new int[n - 1];
    for (int i = 0, h = 0; i < n; i++) {
      if (rank[i] < n - 1) {
        for (int j = sa[rank[i] + 1]; Math.max(i, j) + h < s.length()
            && s.charAt(i + h) == s.charAt(j + h); ++h)
          ;
        lcp[rank[i]] = h;
        if (h > 0)
          --h;
      }
    }
    return lcp;
  }
}
```

7.4 ZAlgorithm

```
class ZAlgorithm {
  public static int[] zFunction(String s) {
    int[] z = new int[s.length()];
    for (int i = 1, l = 0, r = 0; i < z.length; ++i) {
      if (i <= r)
        z[i] = Math.min(r - i + 1, z[i - l]);
      while (i + z[i] < z.length && s.charAt(z[i]) == s.charAt(i + z[i]))
        ++z[i];
      if (r < i + z[i] - 1) {
        l = i;
        r = i + z[i] - 1;
      }
    }
    return z;
  }

  static int find(String s, String pattern) {
    int[] z = zFunction(pattern + "\0" + s);
    for (int i = pattern.length() + 1; i < z.length; i++)
      if (z[i] == pattern.length())
        return i - pattern.length() - 1;
    return -1;
  }
}
```

8 Tree

8.1 DsuTemplate

```
abstract class DsuTemplate {

  final List<Integer>[] g;
  int n;
  int[] size;

  public DsuTemplate(List<Integer>[] g) {
    this.g = g;
    n = g.length;
    size = new int[n];
    findSize(1, 0);
    dfs(1, 0, true);
  }

  void dfs(int u, int p, boolean keep) {
    int mx = -1, big = -1;
    for (int v : g[u]) {
      if (v != p && size[v] > mx) {
        mx = size[v];
        big = v;
      }
    }
    for (int v : g[u]) {
      if (v != p && v != big) {
        dfs(v, u, false);
      }
    }
    if (big != -1) {
      dfs(big, u, true);
    }
    add(u, p, big);
    // do answer query for node here
    if (!keep) {
      erase(u, p);
    }
  }

  public abstract void add(int v, int p, int big);

  public abstract void erase(int v, int p);

  void findSize(int u, int p) {
    size[u] = 1;
    for (int v : g[u]) {
      if (v != p) {
        findSize(v, u);
        size[u] += size[v];
      }
    }
  }
}
```