Contents

1. Misc

1.1. Contest

1.1.1. Fast I/O

```
public class fast_io {
  public static PrintWriter out =
  new PrintWriter(new
   ↪  BufferedOutputStream(System.out));
  static FASTIO in = new FASTIO();
  public static void main(String[] args) throws
   ↪  IOException {
    int cp = in.nextInt();
    while (cp-- > 0) {
      solve();
    }
    out.close();
```

```
  }
  static void solve() {}
  static class FASTIO {
    BufferedReader br;
    StringTokenizer st;
    public FASTIO() {
      br = new BufferedReader(
        new InputStreamReader(System.in)
      );
    }
    String next() {
      while (st == null || !st.hasMoreElements()) {
        try {
          st = new StringTokenizer(br.readLine());
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
      return st.nextToken();
    }
    int nextInt() {
      return Integer.parseInt(next());
    }
    long nextLong() {
      return Long.parseLong(next());
    }
    double nextDouble() {
      return Double.parseDouble(next());
    }
    String nextLine() {
      String str = "";
      try {
        st = null;
        str = br.readLine();
      } catch (IOException e) {
        e.printStackTrace();
      }
      return str;
    }
  }
}
```

1.2. Tools

1.2.1. <random>

```
import java.util.Random;

class random {
    static final Random rng = new Random();

    static int randInt(int l, int r) {
        return l + rng.nextInt(r - l + 1);
    }

    static long randLong(long l, long r) {
        return l + (Math.abs(rng.nextLong()) % (r - l
        ↪  + 1));
    }
    // use inside the main
    // int a = randInt(1, 10);
    // long b = randLong(100, 1000);
}

// ---------- RANDOM (CP TEMPLATE) ----------
// mt19937_64
↪  rng(chrono::steady_clock::now().time_since_epoch().count());
```

```
20
21 // inline int rnd(int l, int r) {
22 //     return uniform_int_distribution<int>(l,
    ↪  r)(rng);
23 // }
24
25 // inline long long rndll(long long l, long long r) {
26 //     return uniform_int_distribution<long long>(l,
    ↪  r)(rng);
27 // }
```

1.3. Algorithms

1.3.1. Manacher Algo

```
1  public class Manacher {
2    public static void main(String[] args) {
3      String s = "aabaac";
4      manacher m = new manacher(s);
5      System.out.println(m.getLongestString());
6    }
7  }
8  class manacher {
9    String s, t;
10   int[] p;
11   public manacher(String s) {
12     this.s = s; build();
13   }
14   public void build() {
15     StringBuilder sb = new StringBuilder("#");
16     for (char ch : s.toCharArray())
17       sb.append(ch).append('#');
18     t = sb.toString();
19     int n = t.length();
20     p = new int[n];
21     int l = 0, r = 0;
22     for (int i = 0; i < n; i++) {
23       int mirror = l + r - i;
24       if (i < r)
25         p[i] = Math.min(r - i, p[mirror]);
26       while (i + p[i] + 1 < n && i - p[i] - 1 >= 0
27         && t.charAt(i + p[i] + 1) == t.charAt(i - p[i]
    ↪  - 1))
28         p[i]++;
29       if (i + p[i] > r) {
30         l = i - p[i];
31         r = i + p[i];
32       }
33     }
34   }
35   public boolean isPal(int l, int r) {
36     int center = l + r + 1;
37     int length = r - l + 1;
38     return p[center] >= length;
39   }
40   // Returns the length of the longest palindrome
41   public int getLongest() {
42     int maxLen = 0;
43     for (int x : p)
44       if (x > maxLen)
45         maxLen = x;
46     return maxLen;
47   }
48   // Returns the actual longest palindromic substring
49   public String getLongestString() {
50     int maxLen = 0, center = 0;
```

```
    for (int i = 0; i < p.length; i++) {
      if (p[i] > maxLen) {
        maxLen = p[i];
        center = i;
      }
    }
    // Map back to original string
    int start = (center - maxLen) / 2;
    return s.substring(start, start + maxLen);
  }
}
```

1.3.2. TreeDiameter

```
public class TreeDiameter {
    public static void main(String[] args) {
        solve();// out.close();
    }
    private static void solve() {
        int n = in.nextInt();
        List<List<Integer>> edges = new ArrayList<>();
        for (int i = 0; i <= n; i++) {
            edges.add(new ArrayList<>());
        }
        for (int i = 0; i < n - 1; i++) {
            int u = in.nextInt();
            int v = in.nextInt();
            edges.get(u).add(v);
            edges.get(v).add(u);
        }
        int[] distX = new int[n + 1];
        int[] distY = new int[n + 1];
        Arrays.fill(distX, -1);
        Arrays.fill(distY, -1);
        int x = 1;
        // First DFS from a random node to find a
        // farthest node
        dfs(x, edges, -1, distX);
        int y = farthestNode(n, distX);
        // Second DFS from farthest node to
        // find the farthest node from it
        dfs(y, edges, -1, distY);
        int z = farthestNode(n, distY);
        // Print the diameter of the tree
        System.out.println(distY[z]);
    }
    private static void dfs(int curr,
     ↪  List<List<Integer>> edges, int parent, int[]
     ↪  level) {
        if (parent == -1)
            level[curr] = 0;
        else
            level[curr] = level[parent] + 1;
        for (int neighbor : edges.get(curr)) {
            if (neighbor != parent)
                dfs(neighbor, edges, curr, level);
        }
    }
    // Find the farthest node from a given node
    private static int farthestNode(int n, int[] dist)
     ↪  {
        int farthest = 0;
        for (int i = 0; i <= n; i++) {
            if (dist[i] > dist[farthest])
                farthest = i;
```

```
        }
        return farthest;
    }
}
```

1.3.3. GCD On Path (max, min etc)

```
import java.util.*;
public class GCDOnPath {
  static final int MAX_LOG = 20;
  static final int N = (int) 2e5 + 1;
  static int[][] parent = new int[N][MAX_LOG];
  static int[][] gcdVal = new int[N][MAX_LOG];
  static int[] depth = new int[N];
  static int[] arr = new int[N];
  static List<List<Integer>> adj;

  public static void main(String[] args) { }

  private static void solve() {
    dfs(1, 0);
  }
  private static void dfs(int node, int par) {
    depth[node] = depth[par] + 1;
    parent[node][0] = par;
    gcdVal[node][0] = gcd(arr[node], arr[par]);
    for (int j = 1; j < MAX_LOG; j++) {
      int midParent = parent[node][j - 1];
      parent[node][j] = parent[midParent][j - 1];
      gcdVal[node][j] = gcd(gcdVal[node][j - 1],
        gcdVal[midParent][j - 1]);
    }
    for (int child : adj.get(node)) {
      if (child != par) {
        dfs(child, node);
      }
    }
  }
  private static int getGCDOnPath(int a, int b) {
    int g = gcd(arr[a], arr[b]);
    if (depth[a] < depth[b]) {
      int temp = a;
      a = b; b = temp;
    }
    int diff = depth[a] - depth[b];
    for (int j = MAX_LOG - 1; j >= 0; j--) {
      if (((1 << j) & diff) != 0) {
        g = gcd(g, gcdVal[a][j]);
        a = parent[a][j];
      }
    }

    if (a == b)
      return g;
    for (int j = MAX_LOG - 1; j >= 0; j--) {
      if (parent[a][j] != parent[b][j]) {
        g = gcd(g, gcd(gcdVal[a][j], gcdVal[b][j]));
        a = parent[a][j];
        b = parent[b][j];
      }
    }
    g = gcd(g, gcd(gcdVal[a][0], gcdVal[b][0]));
    return g;
  }
  private static int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
```

```
  }
}
```

1.3.4. SLiding Window

```
import java.util.*;
// Dequeue Optimization-->

public class SlidingWindow {
  public static void main(String[] args) { }
  // Function to find the minimum in each subarray of size k
  private static List<Integer> sliding_wind_min(int[] arr, int k) {
    List<Integer> ans = new ArrayList<>();
    int n = arr.length;
    Deque<Integer> deque = new LinkedList<>();
    for (int i = 0; i < n; i++) {
      // Remove elements out of the current window
      if (!deque.isEmpty() && deque.getFirst() < i - k + 1) {
        deque.pollFirst();
      }
      // Remove elements from the deque that are
      // greater than or equal to the current
      // element
      while (!deque.isEmpty() && arr[deque.getLast()] >= arr[i]) {
        deque.pollLast();
      }
      // Add the current element index to the deque
      deque.offerLast(i);
      // Once the first window is fully traversed,
      //  start adding results
      if (i >= k - 1)
        ans.add(arr[deque.getFirst()]);
    }
    return ans;
  }
  // code to find the sliding window maximum of size k.
  public int[] maxSlidingWindow(int[] nums, int k) {
    int n = nums.length;
    int[] ans = new int[n + 1 - k];
    TreeMap<Integer, Integer> map = new TreeMap<>();
    int l = 0;
    for (int r = 0; r < n; r++) {
      map.put(nums[r], map.getOrDefault(nums[r], 0) + 1);
      if (r - l + 1 == k) {
        ans[l] = map.lastKey();
        int val = nums[l];
        if (map.get(val) == 1) {
          map.remove(val);
        } else {
          map.put(val, map.get(val) - 1);
        }
        l++;
      }
    }
    return ans;
  }  // max num is sliding window of size k.
  public int[] maxSlidingWindow2(int[] nums, int k) {
    int n = nums.length;
    int[] ans = new int[n - k + 1];
```

```
    int idx = 0;
    Deque<Integer> deque = new LinkedList<>();
    for (int i = 0; i < n; i++) {
      if (!deque.isEmpty() && deque.getFirst() < i - k
      ↪  + 1) {
        deque.pollFirst();
      }
      while (!deque.isEmpty() && nums[deque.getLast()]
      ↪  <= nums[i]) {
        deque.pollLast();
      }
      deque.offerLast(i);
      if (i >= k - 1)
        ans[idx++] = nums[deque.getFirst()];
    }
    return ans;
  } // Function to find the sliding window Meadian.
  public double[] medianSlidingWindow(int[] nums, int
  ↪  k) {
    TreeSet<Integer> minSet = new TreeSet<>(
        (a, b) -> nums[a] == nums[b] ? a - b
            : Integer.compare(nums[a], nums[b]));
    TreeSet<Integer> maxSet = new TreeSet<>(
        (a, b) -> nums[a] == nums[b] ? a - b
            : Integer.compare(nums[a], nums[b]));

    double[] ans = new double[nums.length - k + 1];
    for (int i = 0; i < nums.length; i++) {
      minSet.add(i); // add the index in the low
      maxSet.add(minSet.pollLast());
      // add the last of minSet to max.
      if (minSet.size() < maxSet.size())
        // if low < high add the first from the high
        ↪  to the low set.
        minSet.add(maxSet.pollFirst());
      if (i >= k - 1) {
        if (k % 2 == 0)
          ans[i - k + 1] = ((double)
          ↪  nums[minSet.last()]
              + nums[maxSet.first()]) / 2;
        else
          ans[i - k + 1] = (double)
          ↪  nums[minSet.last()];
        if (!minSet.remove(i - k + 1))
          maxSet.remove(i - k + 1);
      }
    }
    return ans;
  }
}
```

1.3.5. NegCycle IN Directed Graph

```
public class negCycleDetectDG {
  public static void main(String[] args) throws
  ↪  IOException { }
  static void solve() {
    int[] parent = new int[n + 1];
    long[] dist = new long[n + 1];
    Arrays.fill(dist, INF);
    Arrays.fill(parent, -1);
    dist[1] = 0;
    int startNode = -1;
    // Run Bellman-Ford n times
    for (int i = 0; i < n; i++) {
```

```
      startNode = -1;
      for (long[] e : edges) {
        int u = (int) e[0], v = (int) e[1];
        long w = e[2];
        if (dist[u] + w < dist[v]) {
          dist[v] = dist[u] + w;
          parent[v] = u;
          startNode = v;
        }
      }
    }
    if (startNode == -1) {
      out.println("NO");
      return;
    } // To ensure we are inside the cycle
    for (int i = 0; i < n; i++) {
      startNode = parent[startNode];
    }
    List<Integer> cycle = new ArrayList<>();
    int v = startNode;
    do {
      cycle.add(v);
      v = parent[v];
    } while (v != startNode);
    cycle.add(startNode);
    Collections.reverse(cycle);
    out.println("YES");
    for (int node : cycle) {
      out.print(node + " ");
    }
    out.println();
  }
}
```

1.3.6. UpDown Algo IN Tree

```
// here we are finding the max dist in subTree in down1[node]
// and second max leaf dist in down2[node], also the up[node]
//= max dist not in the subTree. and the heavy[node] give in
//subTree frim which node the max distance means down1[node] is comming .
public class UpDownDist {
  public static void main(String[] args) {  }
  static List<List<Integer>> adj;
  static int[] depth, parent, down1, down2, heavy, up;
  static void solve() {
    depth = new int[n + 1];
    down1 = new int[n + 1];
    down2 = new int[n + 1];
    heavy = new int[n + 1];
    up = new int[n + 1];
    parent = new int[n + 1];
    dfsDepth(1, -1);
    dfsDown(1, -1);
    up[1] = 0;
    dfsUp(1, -1);
    long ans = -INF;
    for (int node = 1; node <= n; node++) {
      long curr = k * (long) Math.max(down1[node],
          up[node]) - c * (long) depth[node];
      ans = Math.max(ans, curr);
    }
```

```
    out.println(ans);
  }
  static void dfsUp(int node, int par) {
    for (int adjNode : adj.get(node)) {
      if (adjNode == par)
        continue;
      int curr = (heavy[node] == adjNode ?
        down2[node] : down1[node]);
      up[adjNode] = 1 + Math.max(up[node], curr);
      dfsUp(adjNode, node);
    }
  }
  static void dfsDown(int node, int p) {
    down1[node] = down2[node] = 0;
    heavy[node] = -1;
    for (int adjNode : adj.get(node)) {
      if (adjNode == p)
        continue;
      dfsDown(adjNode, node);
      int curr = 1 + down1[adjNode];
      if (curr > down1[node]) {
        down2[node] = down1[node];
        down1[node] = curr;
        heavy[node] = adjNode;
      } else if (curr > down2[node]) {
        down2[node] = curr;
      }
    }
  }
  static void dfsDepth(int node, int p) {
    parent[node] = p;
    for (int adjNode : adj.get(node)) {
      if (adjNode == p)
        continue;
      depth[adjNode] = 1 + depth[node];
      dfsDepth(adjNode, node);
    }
  }
}
```

1.3.7. KMP

```
import java.util.*;
public class KMP {
  private int n, m;
  private String text, pattern;
  private int[] LPS;
  public KMP(String text, String pattern) {
    this.text = text;
    this.pattern = pattern;
    this.n = text.length();
    this.m = pattern.length();
    this.LPS = new int[m];
    generateLPS();
  }
  private void generateLPS() {
    int len = 0;
    int i = 1;
    while (i < m) {
      if (pattern.charAt(i) == pattern.charAt(len)) {
        LPS[i++] = ++len;
      } else {
        if (len != 0) {
          len = LPS[len - 1];
        } else {
          LPS[i++] = 0;
```

```
        }
      }
    }
  }
  public List<int[]> countOccurrences() {
    List<int[]> result = new ArrayList<>();
    int i = 0, j = 0;
    while (i < n) {
      if (text.charAt(i) == pattern.charAt(j)) {
        i++;
        j++;
      }
      if (j == m) {
        int start = i - m;
        int end = i - 1;
        result.add(new int[] { start, end });
        j = LPS[j - 1];
      } else if (i < n && text.charAt(i) !=
      ↪  pattern.charAt(j)) {
        if (j != 0) {
          j = LPS[j - 1];
        } else {
          i++;
        }
      }
    }
    return result;
  }
}
```

1.3.8. Palindrome Subsequence

```

public class palSubsequence {
  public static void main(String[] args) {
    solve();
  }
  public static void solve() {
    for (int gap = 0; gap < n; gap++) {
      for (int i = 0, j = gap; j < n; i++, j++) {
        if (gap == 0) {
          // single char is a palindrome
          dp[i][j] = 1;
        } else if (gap == 1) {
          // if both char are same then 3 else 2
          if (s.charAt(i) == s.charAt(j)) {
            dp[i][j] = 3;
          } else {
            dp[i][j] = 2;
          }
        } else {
          // the we have two cases
          if (s.charAt(i) == s.charAt(j)) {
            dp[i][j] = dp[i][j - 1] + dp[i + 1][j] +
            ↪  1;
          } else {
            dp[i][j] = dp[i][j - 1] + dp[i + 1][j] -
            ↪  dp[i + 1][j - 1];
          }
        }
      }
    }
    // println(dp[0][n - 1]);
  }
}
```

1.3.9. Longest Increasing Subsequence

```java
import java.util.*;
public class lis {
    public static void main(String[] args) {
        // int[] arr = new int[n];
        List<Long> dp = new ArrayList<>();
        for (long x : arr) {
        // Find the position to replace or extend
          int pos = Collections.binarySearch(dp, x);
          if (pos < 0)
            pos = -(pos + 1); // If not found, get insertion point
            // If pos is within dp, replace the element
            if (pos < dp.size()) {
                dp.set(pos, x);
            } else {
                // Else, extend the subsequence
                dp.add(x);
            }
        }
        // out.println(dp.size()); length of LIS
    }
}
```

2. Data Structures

2.1. Fenwick Tree

```java
public class FT {
  static int[] fTree;
  public static void main(String[] args) {
    // int[] arr = new int[n + 1]; // 1-based
    // // preProcess(arr);
  } // 1-based indexing
  static void preProcess(int[] arr) {
    int n = arr.length - 1;
    fTree = new int[n + 1];
    for (int i = 1; i <= n; i++) {
      update(i, arr[i]);
    }
  }
  static int query(int l, int r) { return prefixSum(r) - prefixSum(l - 1);}
  static int prefixSum(int idx) {
    int sum = 0;
    while (idx > 0) {
      sum += fTree[idx]; idx -= (idx & -idx);
    }
    return sum;
  }
  static void update(int idx, int delta) {
    while (idx < fTree.length) {
      fTree[idx] += delta; idx += (idx & -idx);
    }
  }
}
```

2.2. Segment Tree (SIMPLE)

```java

public class SegTreeSimple { }
class SegmentTree {
```

```
  private int[] tree; private int n;
  public SegmentTree(int[] arr) {
    this.n = arr.length; this.tree = new int[4 * n];
    build(arr, 0, 0, n - 1);
  }
  private void build(int[] arr,int node,int start,int
   ↪  end) {
    if (start == end) {
      tree[node] = arr[start]; return;
    }
    int mid = (start + end) / 2;
    build(arr, 2 * node + 1, start, mid);
    build(arr, 2 * node + 2, mid + 1, end);
    tree[node] = tree[2 * node + 1] + tree[2 * node +
     ↪  2];
  }
  public void update(int index, int value) {
    update(0, 0, n - 1, index, value);
  }
  private void update(int node,int st,int en,int
   ↪  id,int val) {
    if (st == en) {
      tree[node] = val; return;
    } int mid = (st + en) / 2;
    if (id <= mid) {
      update(2 * node + 1, st, mid, id, val);
    } else {
      update(2 * node + 2, mid + 1, en, id, val);
    }
    tree[node] = tree[2 * node + 1] + tree[2 * node +
     ↪  2];
  }
  public int query(int left, int right) {
    return query(0, 0, n - 1, left, right);
  }
  private int KthOne(int node,int start,int end,int k)
   ↪  {
    if (start == end) return start;
    int leftCount = tree[2 * node + 1];
    if (k < leftCount) {
      return KthOne(2*node+1,start,(start+end)/2,k);
    } else {
      return
       ↪  KthOne(2*node+2,(start+end)/2+1,end,k-leftCount);
    }
  }
  public int findKthOne(int k) {
    return KthOne(0, 0, n - 1, k);
  }
  private int query(int node,int start,int end,int
   ↪  l,int r) {
    if (r < start || l > end) return 0;// outside
    if (l <= start && end <= r) return tree[node];//
     ↪  inside
    int mid = (start + end) / 2;
    int leftSum = query(2 * node + 1, start, mid, l,
     ↪  r);
    int rightSum = query(2 * node + 2, mid + 1, end,
     ↪  l, r);
    return leftSum + rightSum;
  }
}
```

2.3. Lazy Segment Tree (SIMPLE)

```
public class LazySimple {
```

```
  private int n;
  private long[] st;
  private long[] lazy;
  public void init(int _n) {
    this.n = _n; st = new long[4 * n];
    lazy = new long[4 * n];
  }
  private long combine(long a, long b) { return a + b; }
  private void push(int start, int end, int node) {
    if (lazy[node] != 0) {
      st[node] += (end - start + 1) * lazy[node];
      if (start != end) {
        lazy[2 * node + 1] += lazy[node];
        lazy[2 * node + 2] += lazy[node];
      } lazy[node] = 0;
    }
  }
  private void build(int start,int end,int node,long[] v) {
    if (start == end)
      st[node] = v[start]; return;
    int mid = (start + end) / 2;
    build(start, mid, 2 * node + 1, v);
    build(mid + 1, end, 2 * node + 2, v);
    st[node] = combine(st[2 * node + 1], st[2 * node + 2]);
  }
  private long query(int start,int end,int l,int r,int node) {
    push(start, end, node);
    if (start > r || end < l) return 0;
    if (start >= l && end <= r) return st[node];
    int mid = (start + end) / 2;
    long q1 = query(start, mid, l, r, 2 * node + 1);
    long q2 = query(mid + 1, end, l, r, 2 * node + 2);
    return combine(q1, q2);
  }
  private void update(int sta,int en,int node,int l,
    int r,long val) {
    push(sta, en, node);
    if (sta > r || en < l) return;
    if (sta >= l && en <= r) {
      lazy[node] = val; push(sta, en, node); return;
    }
    int mid = (sta + en) / 2;
    update(sta, mid, 2 * node + 1, l, r, val);
    update(mid + 1, en, 2 * node + 2, l, r, val);
    st[node] = combine(st[2 * node + 1], st[2 * node + 2]);
  }
  public void build(long[] v) { build(0, n - 1, 0, v); }
  public long query(int l, int r) { return query(0, n - 1, l, r, 0); }
  public void update(int l, int r, long x) { update(0, n - 1, 0, l, r, x);}
}
```

2.4. Binary Lifting (1 based)

```
// parent[node][i] = parent[parent[node][i - 1]][i - 1];
// This means that the 2^i th parent of the node is
```

```java
// 2i - 1 th parent of the node ka 2^i-1 th parent
public class BinaryLiftting {
  private static final int MAX_LOG = 20;
  private static void solve() {
    int[][] par = new int[n + 1][MAX_LOG];
    dfs(1, 0, adj, par);
  }
  private static void dfs(int node, int parent,
     List<List<Integer>> adj, int[][] par) {
    par[node][0] = parent;
    for (int j = 1; j < MAX_LOG; j++) {
      par[node][j] = par[par[node][j - 1]][j - 1];
    }
    for (int adjNode : adj.get(node)) {
      if (adjNode != parent)
        dfs(adjNode, node, adj, par);
    }
  }
  static int Kthparent(int node, int k, int[][] par) {
    for (int i = MAX_LOG - 1; i >= 0; i--) {
      if (((1 << i) & k) != 0) {
        node = par[node][i];
        if (node == 0) return 0;
      }
    }
    return node;
  }
}
```

2.5. DSU

```java
public class DSU {
  private int[] parent, rank, size;
  int component;
  public DSU(int n) {
    parent = new int[n]; rank = new int[n];
    size = new int[n]; //
    for (int i = 0; i < n; i++) {
      parent[i] = i; size[i] = 1;//
    } component = n;
  }
  public int find(int x) {
    if (parent[x] != x) parent[x] = find(parent[x]);
    return parent[x];
  }
  public boolean union(int u, int v) {
    int rootU = find(u), rootV = find(v);
    if (rootU == rootV)
      return false;
    component--;
    if (rank[rootU] > rank[rootV]) {
      parent[rootV] = rootU;
      size[rootU] += size[rootV];//
    } else if (rank[rootU] < rank[rootV]) {
      parent[rootU] = rootV;
      size[rootV] += size[rootU];//
    } else {
      parent[rootV] = rootU; rank[rootU]++;
      size[rootU] += size[rootV];//
    }
    return true;
  }
  public int getComp() { return component; }
  public int getSize(int x) { return size[find(x)];}
}
```

2.6. SparseTable

```
public class SparseTable {
  int[][] st;
  int[] log;
  public SparseTable(int[] arr) {
    int n = arr.length;
    int K = 32 - Integer.numberOfLeadingZeros(n);
    st = new int[n][K];
    log = new int[n + 1];
    log[1] = 0;
    for (int i = 2; i <= n; i++) {
      log[i] = log[i / 2] + 1;
    }
    for (int i = 0; i < n; i++) {
      st[i][0] = arr[i];
    }
    for (int j = 1; j < K; j++) {
      for (int i = 0; i + (1 << j) <= n; i++) {
        st[i][j] = Math.min(st[i][j - 1],
           st[i + (1 << (j - 1))][j - 1]);
      }
    }
  }
  public int query(int l, int r) {
    int len = r - l + 1;
    int j = log[len];
    return Math.min(st[l][j], st[r - (1 << j) +
     ↪  1][j]);
  }
}
```

2.7. EulerTour

```
public class euler_tour {
  static List<Integer>[] adj;
  static int time = 0;
  public static void main(String[] args)
  throws IOException {solve();}
  static void solve() {
    long[] euler = new long[2 * n];
    int[] inTime = new int[n + 1];
    int[] outTime = new int[n + 1];
    dfs(1, -1, inTime, outTime);
    for (int i = 1; i <= n; i++) {
      euler[inTime[i]] = v[i - 1];
      euler[outTime[i]] = -v[i - 1];
    }
    SegTree seg = new SegTree(); seg.init(2 * n);
    seg.build(0, 2 * n - 1, 0, euler);
    while (q-- > 0) {
      int type = in.nextInt();
      if (type == 1) {
        int s = in.nextInt(), x = in.nextLong();
        seg.update(0, 2 * n - 1, inTime[s], 0, x);
        seg.update(0, 2 * n - 1, outTime[s], 0, -x);
      } else {
        int s = in.nextInt();
        out.println(seg.query(0, 2 * n - 1, 0,
         ↪  inTime[s], 0));
      }
    }
  }
  private static void dfs(int node, int parent, int[]
   ↪  inTime, int[] outTime) {
```

```
    inTime[node] = time++;
    for (int adjNode : adj[node]) {
      if (adjNode != parent)
        dfs(adjNode, node, inTime, outTime);
    }
    outTime[node] = time++;
  }
}
```

3. Math

3.1. Number Theory

3.1.1. Linear Sieve

```
public  class prime_sieve {
    static final int MAXN = 1_000_000;
    static boolean[] isPrime = new boolean[MAXN];
    public static void main(String[] args) { }
    static void sieve() {
        Arrays.fill(isPrime, true);
        isPrime[0] = false;
        isPrime[1] = false;
        for (int i = 2; (long) i * i < MAXN; i++) {
            if (isPrime[i]) {
                for (int j = i * i; j < MAXN; j += i)
                ↪ {
                    isPrime[j] = false;
                }
            }
        }
    }
}
```

3.1.2. Get Factors and SPF Fucn

```
import java.util.*;

public class allfactor {
    public static void main(String[] args) { }
    static int N = 100000;
    static int[] spf = new int[N + 1];
    // store the smallest prime factor of i in
    ↪  spf[i].
    static void spf() {
        for (int i = 2; i <= N; i++) {
            spf[i] = i;
        }
        // Sieve of Eratosthenes modified to find
        ↪  smallest prime factor
        for (int i = 2; i * i <= N; i++) {
            if (spf[i] == i) { // If i is prime
                for (int j = i * i; j <= N; j += i) {
                    if (spf[j] == j)
                        // Mark spf[j] with the
                        ↪  smallest prime factor
                        spf[j] = i;
                }
            }
        }
    }
    static List<Integer> allFactors(int n) {
        List<Integer> fac = new ArrayList<>();
        fac.add(1);
        while (n > 1) {
            int p = spf[n];
            List<Integer> cur = new ArrayList<>();
```

```
            cur.add(1);
            while (n % p == 0) {
                n /= p;
                cur.add(cur.get(cur.size() - 1) * p);
            }
            List<Integer> next = new ArrayList<>();
            for (int x : fac)
                for (int y : cur)
                    next.add(x * y);
            fac = next;
        }
        return fac;
    }
}
```

3.2. Combinatorics

3.2.1. Comb

```
public class Main {
  static final long MOD = 998244353L;
  static final long INF = (long) 1e18;
  static class Combinatorics {
    final int MOD;
    long[] fact, invFact;
    public Combinatorics(int maxN, int mod) {
      this.MOD = mod;
      fact = new long[maxN + 1];
      invFact = new long[maxN + 1];
      precompute(maxN);
    }
    void precompute(int maxN) {
      fact[0] = 1;
      for (int i = 1; i <= maxN; i++) {
        fact[i] = (i * fact[i - 1]) % MOD;
      }
      invFact[maxN] = modPow(fact[maxN], MOD - 2); // Fermats little theorem
      for (int i = maxN - 1; i >= 0; i--) {
        invFact[i] = (invFact[i + 1] * (i + 1)) % MOD;
      }
    }
    // NCK : no of ways to choose the k elements
    // from n distinct element wihout caring order.
    long nCk(int n, int k) {
      if (k > n || k < 0)
        return 0;
      return (((fact[n] * invFact[k]) % MOD) * invFact[n - k]) % MOD;
    }
    // NPK : no. of ways to arrange k elements out of n,
    // where order matters
    long nPk(int n, int k) {
      if (k > n || k < 0)
        return 0;
      return (fact[n] * invFact[n - k]) % MOD;
    }
    long factorial(int n) {
      return fact[n];
    }
    // stars and bars fomula C (n + k - 1, n) --> no. of ways to distribute n
    // identical stars into k bins
    long starsAndBars(int n_stars, int k_bins) {
      if (n_stars == 0)
```

```
        return 1;
      if (k_bins == 0)
        return 0;
      return nCk(n_stars + k_bins - 1, n_stars);
    }
    long modPow(long a, long b) {
      long res = 1;
      while (b > 0) {
        if ((b & 1) == 1)
          res = (res * a) % MOD;
        b >>= 1;
        a = (a * a) % MOD;
      }
      return res;
    }
  }
}
```

4. UTILITY

```
public class Utility {
  public static void main(String[] args) {   }
  { int[][] prefix = new int[n + 2][m + 2];
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= m; j++) {
        int g = (s[i - 1][j - 1] == 1) ? 1 : 0;
        prefix[i][j] = prefix[i - 1][j] +
        prefix[i][j - 1] - prefix[i - 1][j - 1] + g;
      }
    }
    int totalG = 0;
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
        totalG += (s[i][j] == 1) ? 1 : 0;   }
    }
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
        // checking for the sum of 2k * 2k grid.
        int r1 = Math.max(0, i - k + 1); // top row
        int r2 = Math.min(n, i + k); // bottom row
        ↪ (exclusive)
        int c1 = Math.max(0, j - k + 1); // left col
        int c2 = Math.min(m, j + k); // right col
        ↪ (exclusive)
        // Number of 1s in the rec. (r1, c1) to
        ↪ (r2-1, c2-1)
        int count = prefix[r2][c2] -
        prefix[r2][c1] - prefix[r1][c2] +
        ↪ prefix[r1][c1];
      }
    }
  }
  static class Pair {
    int first, second;
    Pair(int first, int second) {
      this.first = first;
      this.second = second;
    }
    @Override
    public boolean equals(Object obj) {
      if (obj == this)
        return true;
      if (!(obj instanceof Pair))
        return false;
      Pair pair = (Pair) obj;
```

```java
    return pair.first == this.first && pair.second
    ↪  == this.second;
  }
  @Override
  public int hashCode() {
    return Objects.hash(first, second);
  }
}
// Returns min. swaps required to sort arr[] in asc
↪  order
static int minSwaps(int[] arr) {
  int n = arr.length;
  int[][] paired = new int[n][2];
  for (int i = 0; i < n; i++) {
    paired[i][0] = arr[i]; paired[i][1] = i;
  }
  Arrays.sort(paired, (a, b) ->
  ↪  Integer.compare(a[0], b[0]));
  boolean[] visited = new boolean[n];
  int swaps = 0;
  for (int i = 0; i < n; i++) {
    if (visited[i] || paired[i][1] == i) continue;
    int cycleSize = 0,j = i;
    while (!visited[j]) {
      visited[j] = true;
      j = paired[j][1]; cycleSize++;
    }
    if (cycleSize > 1) swaps += (cycleSize - 1);
  }
  return swaps;
}
private static long maxSubarraySum(long[] a, int
↪  left, int right) {
  long curr = 0, maxSum = 0;
  for (int i = left; i <= right; i++) {
    curr += a[i];
    maxSum = Math.max(maxSum, curr);
    if (curr < 0) curr = 0;
  }
  return maxSum;
}
private static long minSubarraySum(long[] a, int
↪  left, int right) {
  long curr = 0, maxSum = 0;
  for (int i = left; i <= right; i++) {
    curr -= a[i];
    maxSum = Math.max(maxSum, curr);
    if (curr < 0) curr = 0;
  }
  return -maxSum;
}
static long modInverse(long a, long mod) {
  return modPow(a, mod - 2, mod);
}
static long modDiv(long x, long y, long mod) {
  // x * y^(MOD-2) % MOD
  return (x * modPow(y, mod - 2, mod)) % mod;
}
static long modPow(long base, long exp, long mod) {
  long result = 1;
  base = base % mod;
  while (exp > 0) {
    if ((exp & 1) == 1)
      result = (result * base) % mod;
    base = (base * base) % mod;
```

```
      exp >>= 1;
    }
    return result;
  }
  static long modMul(long a, long b, long mod) {
    long result = 0;
    a %= mod;
    b %= mod;
    while (b > 0) {
      if ((b & 1) == 1) {
        result = (result + a) % mod;
      }
      a = (a << 1) % mod; // a = (a * 2) % mod
      b >>= 1; // b = b / 2
    }
    return result;
  }
  static void derangement() {
    int k = 4;
    int[] derangements = new int[k + 1];
    derangements[0] = 1; // D(0) =
    if (k > 0)
      derangements[1] = 0; // D(1) =
    for (int i = 2; i <= k; i++) {
      derangements[i] = (i - 1) *
      (derangements[i - 1] + derangements[i - 2]);
    }
  }
  private static void addAllPrimFact(int x,
   ↪  HashMap<Integer, Integer> map) {
    int i = 2;
    while (i * i <= x) {
      while (x % i == 0) {
        map.put(i, map.getOrDefault(i, 0) + 1);
        x /= i;
      }
      i++;
    }
    if (x > 1) map.put(x, map.getOrDefault(x, 0) + 1);
  } // Find primes in range
  public static List<Boolean> segmentedSieve(long L,
   ↪  long R) {
    long lim = (long) Math.sqrt(R);
    boolean[] mark = new boolean[(int) (lim + 1)];
    List<Long> primes = new ArrayList<>();
    for (long i = 2; i <= lim; i++) {
      if (!mark[(int) i]) {
        primes.add(i);
        for (long j = i * i; j <= lim; j += i) {
          mark[(int) j] = true;
        }
      }
    }
    List<Boolean> isPrime = new ArrayList<>();
    for (int i = 0; i <= R - L; i++) {
      isPrime.add(true);
    }
    for (long prime : primes) {
      long start = Math.max(prime * prime, (L + prime
       ↪  - 1) / prime * prime);
      for (long j = start; j <= R; j += prime) {
        isPrime.set((int) (j - L), false);
      }
    }
```

```
    if (L == 1)
      isPrime.set(0, false);
    return isPrime;
  }
  // int bit = (num >> i) & 1;
  int flipBit(int n, int j) {
    return n ^ (1 << j);
  }
  // mex calculate for the arr of permutation
  // long mex = (n * (n + 1) / 2) - sum;
  public static int findMSB(long n) {
    int msb = 0;
    while (n > 1) {
      n >>= 1;
      msb++;
    }
    return 1 << msb;
  }
  public static long gcd(long a, long b) {
    if (a == 0)
      return b;
    return gcd(b % a, a);
  }
  public static void factor(long n) {
    long count = 0;
    for (int i = 1; i * i <= n; i++) {
      if (n % i == 0) {
        // i -> is the one factor
        count++;
        if (i != n / i) {
          // n / i -> is the other factor
          count++;
        }
      }
    }
  }
  public static long MahantaDist(long x1, long y1,
  ↪  long x2, long y2) {
    return Math.abs(x1 - x2) + Math.abs(y1 - y2);
  }
  public static long numberOfDivisors(long num) {
    long total = 1;
    for (long i = 2; i * i <= num; i++) {
      if (num % i == 0) {
        int e = 0;
        while (num % i == 0) {
          e++;
          num /= i;
        }
        total *= (e + 1);
      }
    }
    if (num > 1) {
      total *= 2;
    }
    return total;
  }
  public static long sumOfDivisors(long num) {
    long total = 1;
    for (long i = 2; i * i <= num; i++) {
      if (num % i == 0) {
        int e = 0;
        while (num % i == 0) {
          e++;
          num /= i;
```

```
        }
        long sum = 0, pow = 1;
        while (e-- >= 0) {
          sum += pow;
          pow *= i;
        }
        total *= sum;
      }
    }
    if (num > 1) {
      total *= (1 + num);
    }
    return total;
  }
  public static long lcm(long a, long b) {
    return Math.abs(a * b) / gcd(a, b);
  }
  // This is used when we use Pair inside the map
  Map<Pair, Integer> map = new HashMap<>();
  static class Pair {
    long first, second;
    Pair(long first, long second) {
      this.first = first;
      this.second = second;
    }
    @Override
    public boolean equals(Object o) {
      if (this == o)
        return true;
      if (o == null || getClass() != o.getClass())
        return false;
      Pair pair = (Pair) o;
      return first == pair.first && second ==
      ↪  pair.second;
    }
    @Override
    public int hashCode() {
      return (int) (31 * first + second);
    }
  }
  static int[] dijkstra(List<List<Pair>> graph, int
  ↪  src, int n) {
    PriorityQueue<Pair> pq = new PriorityQueue<>();
    int[] dist = new int[n];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[src] = 0;
    pq.add(new Pair(src, 0));
    while (!pq.isEmpty()) {
      Pair p = pq.poll();
      int u = p.node;
      if (p.weight > dist[u]) continue;
      for (Pair neighbor : graph.get(u)) {
        int v = neighbor.node;
        int weight = neighbor.weight;
        if (dist[u] + weight < dist[v]) {
          dist[v] = dist[u] + weight;
          pq.add(new Pair(v, dist[v]));
        }
      }
    }
    return dist;
  }
  public static int[] bellmanFord(int n, int[][]
  ↪  edges, int src) {
```

```java
    int[] dist = new int[n + 1];
    Arrays.fill(dist, (int) 1e9);
    dist[src] = 0;
    // Relax all edges (n - 1) times
    for (int i = 1; i <= n - 1; i++) {
      boolean any = false;
      for (int[] edge : edges) {
        int u = edge[0],v = edge[1], wt = edge[2];
        if (dist[u] != (int) 1e9 && dist[u] + wt <
        ↪  dist[v]) {
          dist[v] = dist[u] + wt;
          any = true;
        }
      }
      if (!any) break;
      if (i == n - 1) return new int[] {};
    }
    return dist;
  }
  // static final int INF = 1_000_000_000;
  static void floydWarshall(int[][] dist, int n) {
    for (int k = 0; k < n; k++) {
      for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
          if (dist[i][k] < INF && dist[k][j] < INF)
            dist[i][j] = Math.min(dist[i][j],
            ↪  dist[i][k] + dist[k][j]);
        }
      }
    }
    for (int i = 0; i < n; i++) {
      if (dist[i][i] < 0) {
        // negative cycle
      }
    }
  }
  // TOPOSORT and all that stuff toposort + cycle
  ↪  detection
  public static boolean dfs(int node, int[] used,
  ↪  List<List<Integer>> adj, List<Integer> ans) {
    used[node] = 1; // in recurtion stack
    for (int adjNode : adj.get(node)) {
      if (used[adjNode] == 1) {
        return false; // detected a cycle
      } else if (used[adjNode] == 0) {
        // not visited
        if (!dfs(adjNode, used, adj, ans))
          return false;
      }
    }
    used[node] = 2; // visited but out of stack
    ans.add(node);
    return true;
  }
  // DFS cycle detection (Recomended)
  public static boolean dfsCycleDG(int node,
  ↪  List<List<Integer>> adj,
     boolean[] visited, boolean[] onStack) {
    visited[node] = true;
    onStack[node] = true;
    for (int neighbor : adj.get(node)) {
      if (!visited[neighbor]) {
        if (dfsCycleDG(neighbor, adj, visited,
        ↪  onStack))
```

```
            return true;
        } else if (onStack[neighbor]) {
          return true; // Cycle detected
        }
      }
      onStack[node] = false;
      return false;
    }
    // BFS Cycle Detection (Kahn's Algorithm)
    public static boolean hasCycle(int n,
     ↪  List<List<Integer>> adj) {
      int[] inDegree = new int[n];
      for (int u = 0; u < n; u++) {
        for (int v : adj.get(u))
          inDegree[v]++;
      }
      Queue<Integer> q = new LinkedList<>();
      for (int i = 0; i < n; i++) {
        if (inDegree[i] == 0)
          q.add(i);
      }
      int count = 0;
      while (!q.isEmpty()) {
        int u = q.poll();
        count++;
        for (int v : adj.get(u)) {
          if (--inDegree[v] == 0)
            q.add(v);
        }
      } return count != n; // If count < n, there is a
       ↪  cycle
    }
```

```
    // DFS-Based Topological Sort
    public static List<Integer> topoSortDfs(int n,
     ↪  List<List<Integer>> adj) {
      boolean[] visited = new boolean[n];
      List<Integer> topo = new ArrayList<>();
      for (int i = 0; i < n; i++) {
        if (!visited[i])
          dfsTopo(i, adj, visited, topo);
      }
      Collections.reverse(topo);
      return topo;
    }
    public static void dfsTopo(int node,
     ↪  List<List<Integer>> adj,
       boolean[] visited, List<Integer> topo) {
      visited[node] = true;
      for (int neighbor : adj.get(node)) {
        if (!visited[neighbor])
          dfsTopo(neighbor, adj, visited, topo);
      }
      topo.add(node);
    }
    public static List<Integer> topoSortBFS(int n,
     ↪  List<List<Integer>> adj) {
      int[] inDegree = new int[n];
      for (int u = 0; u < n; u++) {
        for (int v : adj.get(u)) {
          inDegree[v]++;
        }
      }
      Queue<Integer> q = new LinkedList<>();
      for (int i = 0; i < n; i++) {
```

```
      if (inDegree[i] == 0) q.add(i);
    }
    List<Integer> topo = new ArrayList<>();
    while (!q.isEmpty()) {
      int u = q.poll();
      topo.add(u);
      for (int v : adj.get(u)) {
        if (--inDegree[v] == 0) q.add(v);
      }
    }
    return topo.size() == n ? topo : new
     ↪  ArrayList<>();
  }
} // MST using DSU (Krushkal ALgorythm)
  int n;// Nodes int m; // Edges
  Edge[] edges = new Edge[m];
  for (int i = 0; i < m; i++) {
    int u = in.nextInt();
    int v = in.nextInt();
    int w = in.nextInt();
    edges[i] = new Edge(u, v, w);
  }
  Arrays.sort(edges); // Sort edges by weight
  DSU dsu = new DSU(n);
  long mstWeight = 0;
  ArrayList<Edge> mstEdges = new ArrayList<>();
  for (Edge e : edges) {
    if (dsu.union(e.u, e.v)) { // If u and v are in
     ↪  different sets
      mstWeight += e.w;
      mstEdges.add(e);
    }
  }
}
static class Edge implements Comparable<Edge> {
  int u, v, w;
  Edge(int u, int v, int w) {
    this.u = u; this.v = v; this.w = w;
  }
  public int compareTo(Edge o) {
    return Integer.compare(this.w, o.w);
  }
} // MST using PriorityQueue Prims Algorythm
static long primsMST(int n, List<List<int[]>> adj) {
  boolean[] visited = new boolean[n + 1];
  PriorityQueue<int[]> pq = new PriorityQueue<>((x, y)
   ↪  -> (x[1] - y[1]));
  pq.add(new int[] { 1, 0 }); // Start from node 1
  long mstWeight = 0;
  while (!pq.isEmpty()) {
    int[] curr = pq.poll();
    int u = curr[0], w = curr[1];
    if (visited[u])
      continue;
    visited[u] = true;
    mstWeight += w;
    for (int[] v : adj.get(u)) {
      if (!visited[v[0]]) {
        pq.add(new int[] { v[0], v[1] });
      }
    }
  }
  return mstWeight;
}
```