Contents

1. Misc

1.1. Contest

1.1.1. Makefile

```
.PRECIOUS: ./p%

%: p%
  ulimit -s unlimited && ./$<
p%: p%.cpp
  g++ -o $@ $< -std=c++17 -Wall -Wextra -Wshadow \
    -fsanitize=address,undefined
```

1.2. How Did We Get Here?

1.2.1. Fast I/O

```
// use Scanner in Simple tasks and if requuired then
// use fast io



import java.io.*;
import java.util.*;

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;
    }

  }

}
```

1.3. Tools

1.3.1. <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());

// inline int rnd(int l, int r) {
//     return uniform_int_distribution<int>(l, r)(rng);
// }

// inline long long rndll(long long l, long long r) {
//     return uniform_int_distribution<long long>(l, r)(rng);
// }
```

1.4. Algorithms

1.4.1. Manacher Algo

```
public class Manacher {
  public static void main(String[] args) {
    String s = "aabaac";
    manacher m = new manacher(s);
    System.out.println(m.getLongestString());
  }
}
class manacher {
  String s, t;
  int[] p;
  public manacher(String s) {
    this.s = s; build();
  }
  public void build() {
    StringBuilder sb = new StringBuilder("#");
    for (char ch : s.toCharArray())
      sb.append(ch).append('#');
    t = sb.toString();
    int n = t.length();
    p = new int[n];
    int l = 0, r = 0;
    for (int i = 0; i < n; i++) {
      int mirror = l + r - i;
      if (i < r)
        p[i] = Math.min(r - i, p[mirror]);
      while (i + p[i] + 1 < n && i - p[i] - 1 >= 0
        && t.charAt(i + p[i] + 1) == t.charAt(i - p[i] - 1))
        p[i]++;
      if (i + p[i] > r) {
        l = i - p[i];
        r = i + p[i];
      }
    }
  }
  public boolean isPal(int l, int r) {
    int center = l + r + 1;
    int length = r - l + 1;
    return p[center] >= length;
  }
  // Returns the length of the longest palindrome
  public int getLongest() {
    int maxLen = 0;
    for (int x : p)
      if (x > maxLen)
        maxLen = x;
    return maxLen;
  }
  // Returns the actual longest palindromic substring
  public String getLongestString() {
    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);
  }
}
```

1.4.2. TreeDiameter

```
import java.io.*;
import java.util.*;
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, in
        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++) {
```

```java
            if (dist[i] > dist[farthest]) {
                farthest = i;
            }
        }
        return farthest;
    }
}
```

1.4.3. GCD On Path (max, min etc)

```java
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);
  }
}
```

1.4.4. SLiding Window

```java
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;
  }
}
```

1.4.5. Cycle IN Directed Graph

```java
import java.io.*;
import java.util.*;
public class CycleinDGPrint {
  static List<Integer>[] adj;
```

```java
  static boolean[] visited, onStack;
  static int[] parent;
  static int start = -1, end = -1;
  public static void main(String[] args) {
    int n = in.nextInt(), m = in.nextInt();
    adj = new ArrayList[n + 1];
    visited = new boolean[n + 1];
    onStack = new boolean[n + 1];
    parent = new int[n + 1];
    for (int i = 1; i <= n; i++)
      adj[i] = new ArrayList<>();
    for (int i = 0; i < m; i++) {
      int a = in.nextInt(), b = in.nextInt();
      adj[a].add(b);
    }
    boolean found = false;
    for (int i = 1; i <= n; i++) {
      if (!visited[i]) {
        if (dfs(i)) {
          found = true;
          break;
        }
      }
    }
    if (!found) {
      out.println("IMPOSSIBLE");
    } else {
      List<Integer> cycle = new ArrayList<>();
      cycle.add(end);
      for (int v = start; v != end; v = parent[v]) {
        cycle.add(v);
      }
      cycle.add(end);
      Collections.reverse(cycle);
      out.println(cycle.size());
      for (int city : cycle)
        out.print(city + " ");
      out.println();
    }
    out.flush();
  }
  static boolean dfs(int u) {
    visited[u] = true;
    onStack[u] = true;
    for (int v : adj[u]) {
      if (!visited[v]) {
        parent[v] = u;
        if (dfs(v))
          return true;
      } else if (onStack[v]) {
        start = u;
        end = v;
        return true;
      }
    }
    onStack[u] = false;
    return false;
  }
}
```

1.4.6. NegCycle IN Directed Graph

```java
import java.io.*;
import java.util.*;
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();
  }
}
```

1.4.7. UpDown Algo IN Tree

```java
import java.io.*;
import java.util.*;
// 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 com
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);
    }
  }
}
```

1.4.8. KMP

```java
import java.util.*;
public class KMP {
  private int n, m;
  private String text, pattern;
  private int[] LPS;
```

```java
  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;
  }
}
```

1.4.9. Palindrome Subsequence

```java

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]);
  }
}
```

1.4.10. 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)

```
import java.util.*;
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)

```
import java.io.*;
import java.util.*;
/*
 * parent[node][i] = parent[parent[node][i - 1]][i - 1];
 * This means that the 2^i th parent of the node is
 *  2^i - 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

```
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);
    int 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

```
import java.io.*;
import java.util.*;
public class euler_tour {
  static List<Integer>[] adj;
  static int time = 0;

  public static void main(String[] args) throws IOException {
    int t = 1;
    while (t-- > 0) {
      solve();
    }
    // out.close();
  }

  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); // Euler array size
    seg.build(0, 2 * n - 1, 0, euler);

    while (q-- > 0) {
      int type = in.nextInt();
      if (type == 1) {
        int s = in.nextInt();
        long 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++;
  }
}
```

2.8. Heavy-Light Decomposition

```


template <bool VALS_EDGES> struct HLD {
  int N, tim = 0;
  vector<vi> adj;
  vi par, siz, depth, rt, pos;
  Node *tree;
  HLD(vector<vi> adj_)
      : N(sz(adj_)), adj(adj_), par(N, -1), siz(N, 1),
        depth(N), rt(N), pos(N), tree(new Node(0, N)) {
    dfsSz(0);
    dfsHld(0);
  }
  void dfsSz(int v) {
    if (par[v] != -1)
      adj[v].erase(find(all(adj[v]), par[v]));
    for (int &u : adj[v]) {
      par[u] = v, depth[u] = depth[v] + 1;
      dfsSz(u);
      siz[v] += siz[u];
      if (siz[u] > siz[adj[v][0]]) swap(u, adj[v][0]);
    }
  }
  void dfsHld(int v) {
    pos[v] = tim++;
    for (int u : adj[v]) {
      rt[u] = (u == adj[v][0] ? rt[v] : u);
      dfsHld(u);
    }
  }
  template <class B> void process(int u, int v, B op) {
    for (; rt[u] != rt[v]; v = par[rt[v]]) {
      if (depth[rt[u]] > depth[rt[v]]) swap(u, v);
      op(pos[rt[v]], pos[v] + 1);
    }
    if (depth[u] > depth[v]) swap(u, v);
    op(pos[u] + VALS_EDGES, pos[v] + 1);
  }
  void modifyPath(int u, int v, int val) {
    process(u, v,
            [&](int l, int r) { tree->add(l, r, val); });
  }
  int queryPath(int u,
                int v) { // Modify depending on problem
    int res = -1e9;
    process(u, v, [&](int l, int r) {
      res = max(res, tree->query(l, r));
    });
    return res;
  }
  int querySubtree(int v) { // modifySubtree is similar
    return tree->query(pos[v] + VALS_EDGES,
                       pos[v] + siz[v]);
  }
};
```

3. Graph

3.1. Modeling

- Maximum/Minimum flow with lower bound / Circulation problem
  1. Construct super source S and sink T.
  2. For each edge (x, y, l, u), connect $x \to y$ with capacity $u - l$.
  3. For each vertex v, denote by $in(v)$ the difference between the sum of incoming lower bounds and the sum of outgoing lower bounds.
  4. If $in(v) > 0$, connect $S \to v$ with capacity $in(v)$, otherwise, connect $v \to T$ with capacity $-in(v)$.
     - To maximize, connect $t \to s$ with capacity ∞ (skip this in circulation problem), and let f be the maximum flow from S to T. If $f \neq \sum_{v \in V, in(v)>0} in(v)$, there's no solution. Otherwise, the maximum flow from s to t is the answer.
     - To minimize, let f be the maximum flow from S to T. Connect $t \to s$ with capacity ∞ and let the flow from S to T be f'. If $f + f' \neq \sum_{v \in V, in(v)>0} in(v)$, there's no solution. Otherwise, f' is the answer.
  5. The solution of each edge e is $l_e + f_e$, where f_e corresponds to the flow of edge e on the graph.
- Construct minimum vertex cover from maximum matching M on bipartite graph (X, Y)
  1. Redirect every edge: $y \to x$ if $(x, y) \in M$, $x \to y$ otherwise.
  2. DFS from unmatched vertices in X.
  3. $x \in X$ is chosen iff x is unvisited.
  4. $y \in Y$ is chosen iff y is visited.
- Minimum cost cyclic flow
  1. Consruct super source S and sink T
  2. For each edge (x, y, c), connect $x \to y$ with $(cost, cap) = (c, 1)$ if $c > 0$, otherwise connect $y \to x$ with $(cost, cap) = (-c, 1)$
  3. For each edge with $c < 0$, sum these cost as K, then increase $d(y)$ by 1, decrease $d(x)$ by 1
  4. For each vertex v with $d(v) > 0$, connect $S \to v$ with $(cost, cap) = (0, d(v))$
  5. For each vertex v with $d(v) < 0$, connect $v \to T$ with $(cost, cap) = (0, -d(v))$
  6. Flow from S to T, the answer is the cost of the flow $C + K$
- Maximum density induced subgraph
  1. Binary search on answer, suppose we're checking answer T
  2. Construct a max flow model, let K be the sum of all weights
  3. Connect source $s \to v$, $v \in G$ with capacity K
  4. For each edge (u, v, w) in G, connect $u \to v$ and $v \to u$ with capacity w
  5. For $v \in G$, connect it with sink $v \to t$ with capacity $K + 2T - (\sum_{e \in E(v)} w(e)) - 2w(v)$
  6. T is a valid answer if the maximum flow $f < K|V|$
- Minimum weight edge cover
  1. For each $v \in V$ create a copy v', and connect $u' \to v'$ with weight $w(u, v)$.

2. Connect $v \to v'$ with weight $2\mu(v)$, where $\mu(v)$ is the cost of the cheapest edge incident to v.
3. Find the minimum weight perfect matching on G'.

- Project selection problem
  1. If $p_v > 0$, create edge (s, v) with capacity p_v; otherwise, create edge (v, t) with capacity $-p_v$.
  2. Create edge (u, v) with capacity w with w being the cost of choosing u without choosing v.
  3. The mincut is equivalent to the maximum profit of a subset of projects.
- 0/1 quadratic programming

$$\sum_x c_x x + \sum_y c_y \bar{y} + \sum_{xy} c_{xy} x\bar{y} + \sum_{xyx'y'} c_{xyx'y'}(x\bar{y} + x'\bar{y'})$$

can be minimized by the mincut of the following graph:
1. Create edge (x, t) with capacity c_x and create edge (s, y) with capacity c_y.
2. Create edge (x, y) with capacity c_{xy}.
3. Create edge (x, y) and edge (x', y') with capacity $c_{xyx'y'}$.

3.2. Matching/Flows

3.2.1. Dinic's Algorithm

```cpp
struct Dinic {
  struct edge {
    int to, cap, flow, rev;
  };
  static constexpr int MAXN = 1000, MAXF = 1e9;
  vector<edge> v[MAXN];
  int top[MAXN], deep[MAXN], side[MAXN], s, t;
  void make_edge(int s, int t, int cap) {
    v[s].push_back({t, cap, 0, (int)v[t].size()});
    v[t].push_back({s, 0, 0, (int)v[s].size() - 1});
  }
  int dfs(int a, int flow) {
    if (a == t || !flow) return flow;
    for (int &i = top[a]; i < v[a].size(); i++) {
      edge &e = v[a][i];
      if (deep[a] + 1 == deep[e.to] && e.cap - e.flow) {
        int x = dfs(e.to, min(e.cap - e.flow, flow));
        if (x) {
          e.flow += x, v[e.to][e.rev].flow -= x;
          return x;
        }
      }
    }
    deep[a] = -1;
    return 0;
  }
  bool bfs() {
    queue<int> q;
    fill_n(deep, MAXN, 0);
    q.push(s), deep[s] = 1;
    int tmp;
    while (!q.empty()) {
      tmp = q.front(), q.pop();
      for (edge e : v[tmp])
        if (!deep[e.to] && e.cap != e.flow)
          deep[e.to] = deep[tmp] + 1, q.push(e.to);
    }
    return deep[t];
  }
  int max_flow(int _s, int _t) {
    s = _s, t = _t;
    int flow = 0, tflow;
    while (bfs()) {
      fill_n(top, MAXN, 0);
      while ((tflow = dfs(s, MAXF))) flow += tflow;
    }
    return flow;
  }
  void reset() {
    fill_n(side, MAXN, 0);
    for (auto &i : v) i.clear();
  }
};
```

3.2.2. Minimum Cost Flow

```cpp
struct MCF {
  struct edge {
    ll to, from, cap, flow, cost, rev;
  } *fromE[MAXN];
  vector<edge> v[MAXN];
  ll n, s, t, flows[MAXN], dis[MAXN], pi[MAXN], flowlim;
  void make_edge(int s, int t, ll cap, ll cost) {
    if (!cap) return;
    v[s].pb(edge{t, s, cap, 0LL, cost, v[t].size()});
    v[t].pb(edge{s, t, 0LL, 0LL, -cost, v[s].size() - 1});
  }
  bitset<MAXN> vis;
  void dijkstra() {
    vis.reset();
    __gnu_pbds::priority_queue<pair<ll, int>> q;
    vector<decltype(q)::point_iterator> its(n);
    q.push({0LL, s});
    while (!q.empty()) {
      int now = q.top().second;
      q.pop();
      if (vis[now]) continue;
      vis[now] = 1;
      ll ndis = dis[now] + pi[now];
      for (edge &e : v[now]) {
        if (e.flow == e.cap || vis[e.to]) continue;
        if (dis[e.to] > ndis + e.cost - pi[e.to]) {
          dis[e.to] = ndis + e.cost - pi[e.to];
          flows[e.to] = min(flows[now], e.cap - e.flow);
          fromE[e.to] = &e;
          if (its[e.to] == q.end())
            its[e.to] = q.push({-dis[e.to], e.to});
          else q.modify(its[e.to], {-dis[e.to], e.to});
        }
      }
    }
  }
  bool AP(ll &flow) {
    fill_n(dis, n, INF);
    fromE[s] = 0;
    dis[s] = 0;
    flows[s] = flowlim - flow;
    dijkstra();
    if (dis[t] == INF) return false;
    flow += flows[t];
    for (edge *e = fromE[t]; e; e = fromE[e->from]) {
      e->flow += flows[t];
      v[e->to][e->rev].flow -= flows[t];
    }
    for (int i = 0; i < n; i++)
      pi[i] = min(pi[i] + dis[i], INF);
    return true;
  }
  pll solve(int _s, int _t, ll _flowlim = INF) {
    s = _s, t = _t, flowlim = _flowlim;
    pll re;
    while (re.F != flowlim && AP(re.F));
    for (int i = 0; i < n; i++)
      for (edge &e : v[i])
        if (e.flow != 0) re.S += e.flow * e.cost;
    re.S /= 2;
    return re;
  }
  void init(int _n) {
    n = _n;
    fill_n(pi, n, 0);
    for (int i = 0; i < n; i++) v[i].clear();
  }
  void setpi(int s) {
    fill_n(pi, n, INF);
    pi[s] = 0;
    for (ll it = 0, flag = 1, tdis; flag && it < n; it++) {
      flag = 0;
      for (int i = 0; i < n; i++)
        if (pi[i] != INF)
          for (edge &e : v[i])
            if (e.cap && (tdis = pi[i] + e.cost) < pi[e.to])
              pi[e.to] = tdis, flag = 1;
    }
  }
};
```

3.2.3. Bipartite Minimum Cover

Requires: Dinic's Algorithm

```cpp

// maximum independent set = all vertices not covered
// x : [0, n), y : [0, m]
struct Bipartite_vertex_cover {
  Dinic D;
  int n, m, s, t, x[maxn], y[maxn];
  void make_edge(int x, int y) { D.make_edge(x, y + n, 1); }
  int matching() {
    int re = D.max_flow(s, t);
    for (int i = 0; i < n; i++)
      for (Dinic::edge &e : D.v[i])
        if (e.to != s && e.flow == 1) {
          x[i] = e.to - n, y[e.to - n] = i;
          break;
        }
    return re;
  }
  // init() and matching() before use
  void solve(vector<int> &vx, vector<int> &vy) {
    bitset<maxn * 2 + 10> vis;
```

```cpp
    queue<int> q;
    for (int i = 0; i < n; i++)
      if (x[i] == -1) q.push(i), vis[i] = 1;
    while (!q.empty()) {
      int now = q.front();
      q.pop();
      if (now < n) {
        for (Dinic::edge &e : D.v[now])
          if (e.to != s && e.to - n != x[now] && !vis[e.to])
            vis[e.to] = 1, q.push(e.to);
      } else {
        if (!vis[y[now - n]])
          vis[y[now - n]] = 1, q.push(y[now - n]);
      }
    }
    for (int i = 0; i < n; i++)
      if (!vis[i]) vx.pb(i);
    for (int i = 0; i < m; i++)
      if (vis[i + n]) vy.pb(i);
  }
  void init(int _n, int _m) {
    n = _n, m = _m, s = n + m, t = s + 1;
    for (int i = 0; i < n; i++)
      x[i] = -1, D.make_edge(s, i, 1);
    for (int i = 0; i < m; i++)
      y[i] = -1, D.make_edge(i + n, t, 1);
  }
};
```

3.2.4. Edmonds' Algorithm

```cpp

struct Edmonds {
  int n, T;
  vector<vector<int>> g;
  vector<int> pa, p, used, base;
  Edmonds(int n)
      : n(n), T(0), g(n), pa(n, -1), p(n), used(n),
        base(n) {}
  void add(int a, int b) {
    g[a].push_back(b);
    g[b].push_back(a);
  }
  int getBase(int i) {
    while (i != base[i])
      base[i] = base[base[i]], i = base[i];
    return i;
  }
  vector<int> toJoin;
  void mark_path(int v, int x, int b, vector<int> &path) {
    for (; getBase(v) != b; v = p[x]) {
      p[v] = x, x = pa[v];
      toJoin.push_back(v);
      toJoin.push_back(x);
      if (!used[x]) used[x] = ++T, path.push_back(x);
    }
  }
  bool go(int v) {
    for (int x : g[v]) {
      int b, bv = getBase(v), bx = getBase(x);
      if (bv == bx) {
        continue;
      } else if (used[x]) {
        vector<int> path;
        toJoin.clear();
        if (used[bx] < used[bv])
          mark_path(v, x, b = bx, path);
        else mark_path(x, v, b = bv, path);
        for (int z : toJoin) base[getBase(z)] = b;
        for (int z : path)
          if (go(z)) return 1;
      } else if (p[x] == -1) {
        p[x] = v;
        if (pa[x] == -1) {
          for (int y; x != -1; x = v)
            y = p[x], v = pa[y], pa[x] = y, pa[y] = x;
          return 1;
        }
        if (!used[pa[x]]) {
          used[pa[x]] = ++T;
          if (go(pa[x])) return 1;
        }
      }
    }
    return 0;
  }
  void init_dfs() {
    for (int i = 0; i < n; i++)
      used[i] = 0, p[i] = -1, base[i] = i;
  }
  bool dfs(int root) {
    used[root] = ++T;
    return go(root);
  }
  void match() {
    int ans = 0;
    for (int v = 0; v < n; v++)
      for (int x : g[v])
        if (pa[v] == -1 && pa[x] == -1) {
          pa[v] = x, pa[x] = v, ans++;
          break;
        }
    init_dfs();
    for (int i = 0; i < n; i++)
      if (pa[i] == -1 && dfs(i)) ans++, init_dfs();
    cout << ans * 2 << "\n";
    for (int i = 0; i < n; i++)
      if (pa[i] > i)
        cout << i + 1 << " " << pa[i] + 1 << "\n";
  }
};
```

3.2.5. Minimum Weight Matching

```cpp
struct Graph {
  static const int MAXN = 105;
  int n, e[MAXN][MAXN];
  int match[MAXN], d[MAXN], onstk[MAXN];
  vector<int> stk;
  void init(int _n) {
    n = _n;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        // change to appropriate infinity
        // if not complete graph
        e[i][j] = 0;
  }
  void add_edge(int u, int v, int w) {
    e[u][v] = e[v][u] = w;
  }
  bool SPFA(int u) {
    if (onstk[u]) return true;
    stk.push_back(u);
    onstk[u] = 1;
    for (int v = 0; v < n; v++) {
      if (u != v && match[u] != v && !onstk[v]) {
        int m = match[v];
        if (d[m] > d[u] - e[v][m] + e[u][v]) {
          d[m] = d[u] - e[v][m] + e[u][v];
          onstk[v] = 1;
          stk.push_back(v);
          if (SPFA(m)) return true;
          stk.pop_back();
          onstk[v] = 0;
        }
      }
    }
    onstk[u] = 0;
    stk.pop_back();
    return false;
  }
  int solve() {
    for (int i = 0; i < n; i += 2) {
      match[i] = i + 1;
      match[i + 1] = i;
    }
    while (true) {
      int found = 0;
      for (int i = 0; i < n; i++) onstk[i] = d[i] = 0;
      for (int i = 0; i < n; i++) {
        stk.clear();
        if (!onstk[i] && SPFA(i)) {
          found = 1;
          while (stk.size() >= 2) {
            int u = stk.back();
            stk.pop_back();
            int v = stk.back();
            stk.pop_back();
            match[u] = v;
            match[v] = u;
          }
        }
      }
      if (!found) break;
    }
    int ret = 0;
    for (int i = 0; i < n; i++) ret += e[i][match[i]];
    ret /= 2;
    return ret;
  }
} graph;
```

3.2.6. Stable Marriage

```cpp
// normal stable marriage problem
/* input:
```

```
3
Albert Laura Nancy Marcy
Brad Marcy Nancy Laura
Chuck Laura Marcy Nancy
Laura Chuck Albert Brad
Marcy Albert Chuck Brad
Nancy Brad Albert Chuck
*/

using namespace std;
const int MAXN = 505;

int n;
int favor[MAXN][MAXN]; // favor[boy_id][rank] = girl_id;
int order[MAXN][MAXN]; // order[girl_id][boy_id] = rank;
int current[MAXN];     // current[boy_id] = rank;
// boy_id will pursue current[boy_id] girl.
int girl_current[MAXN]; // girl[girl_id] = boy_id;

void initialize() {
  for (int i = 0; i < n; i++) {
    current[i] = 0;
    girl_current[i] = n;
    order[i][n] = n;
  }
}

map<string, int> male, female;
string bname[MAXN], gname[MAXN];
int fit = 0;

void stable_marriage() {

  queue<int> que;
  for (int i = 0; i < n; i++) que.push(i);
  while (!que.empty()) {
    int boy_id = que.front();
    que.pop();

    int girl_id = favor[boy_id][current[boy_id]];
    current[boy_id]++;

    if (order[girl_id][boy_id] <
        order[girl_id][girl_current[girl_id]]) {
      if (girl_current[girl_id] < n)
        que.push(girl_current[girl_id]);
      girl_current[girl_id] = boy_id;
    } else {
      que.push(boy_id);
    }
  }
}

int main() {
  cin >> n;

  for (int i = 0; i < n; i++) {
    string p, t;
    cin >> p;
    male[p] = i;
    bname[i] = p;
    for (int j = 0; j < n; j++) {
      cin >> t;
      if (!female.count(t)) {
        gname[fit] = t;
        female[t] = fit++;
      }
      favor[i][j] = female[t];
    }
  }

  for (int i = 0; i < n; i++) {
    string p, t;
    cin >> p;
    for (int j = 0; j < n; j++) {
      cin >> t;
      order[female[p]][male[t]] = j;
    }
  }

  initialize();
  stable_marriage();

  for (int i = 0; i < n; i++) {
    cout << bname[i] << " "
         << gname[favor[i][current[i] - 1]] << endl;
  }
}
```

3.2.7. Kuhn-Munkres algorithm

```
// Maximum Weight Perfect Bipartite Matching
// Detect non-perfect-matching:
// 1. set all edge[i][j] as INF
// 2. if solve() >= INF, it is not perfect matching.

typedef long long ll;
struct KM {
  static const int MAXN = 1050;
  static const ll INF = 1LL << 60;
  int n, match[MAXN], vx[MAXN], vy[MAXN];
  ll edge[MAXN][MAXN], lx[MAXN], ly[MAXN], slack[MAXN];
  void init(int _n) {
    n = _n;
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++) edge[i][j] = 0;
  }
  void add_edge(int x, int y, ll w) { edge[x][y] = w; }
  bool DFS(int x) {
    vx[x] = 1;
    for (int y = 0; y < n; y++) {
      if (vy[y]) continue;
      if (lx[x] + ly[y] > edge[x][y]) {
        slack[y] =
        min(slack[y], lx[x] + ly[y] - edge[x][y]);
      } else {
        vy[y] = 1;
        if (match[y] == -1 || DFS(match[y])) {
          match[y] = x;
          return true;
        }
      }
    }
    return false;
  }
  ll solve() {
    fill(match, match + n, -1);
    fill(lx, lx + n, -INF);
    fill(ly, ly + n, 0);
    for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
        lx[i] = max(lx[i], edge[i][j]);
    for (int i = 0; i < n; i++) {
      fill(slack, slack + n, INF);
      while (true) {
        fill(vx, vx + n, 0);
        fill(vy, vy + n, 0);
        if (DFS(i)) break;
        ll d = INF;
        for (int j = 0; j < n; j++)
          if (!vy[j]) d = min(d, slack[j]);
        for (int j = 0; j < n; j++) {
          if (vx[j]) lx[j] -= d;
          if (vy[j]) ly[j] += d;
          else slack[j] -= d;
        }
      }
    }
    ll res = 0;
    for (int i = 0; i < n; i++) {
      res += edge[match[i]][i];
    }
    return res;
  }
} graph;
```

3.3. Shortest Path Faster Algorithm

```
struct SPFA {
  static const int maxn = 1010, INF = 1e9;
  int dis[maxn];
  bitset<maxn> inq, inneg;
  queue<int> q, tq;
  vector<pii> v[maxn];
  void make_edge(int s, int t, int w) {
    v[s].emplace_back(t, w);
  }
  void dfs(int a) {
    inneg[a] = 1;
    for (pii i : v[a])
      if (!inneg[i.F]) dfs(i.F);
  }
  bool solve(int n, int s) { // true if have neg-cycle
    for (int i = 0; i <= n; i++) dis[i] = INF;
    dis[s] = 0, q.push(s);
    for (int i = 0; i < n; i++) {
      inq.reset();
      int now;
      while (!q.empty()) {
        now = q.front(), q.pop();
        for (pii &i : v[now]) {
          if (dis[i.F] > dis[now] + i.S) {
            dis[i.F] = dis[now] + i.S;
            if (!inq[i.F]) tq.push(i.F), inq[i.F] = 1;
          }
        }
      }
```

```
      q.swap(tq);
    }
    bool re = !q.empty();
    inneg.reset();
    while (!q.empty()) {
      if (!inneg[q.front()]) dfs(q.front());
      q.pop();
    }
    return re;
  }
  void reset(int n) {
    for (int i = 0; i <= n; i++) v[i].clear();
  }
};
```

3.4. Strongly Connected Components

```
struct TarjanScc {
  int n, step;
  vector<int> time, low, instk, stk;
  vector<vector<int>> e, scc;
  TarjanScc(int n_)
      : n(n_), step(0), time(n), low(n), instk(n), e(n) {}
  void add_edge(int u, int v) { e[u].push_back(v); }
  void dfs(int x) {
    time[x] = low[x] = ++step;
    stk.push_back(x);
    instk[x] = 1;
    for (int y : e[x])
      if (!time[y]) {
        dfs(y);
        low[x] = min(low[x], low[y]);
      } else if (instk[y]) {
        low[x] = min(low[x], time[y]);
      }
    if (time[x] == low[x]) {
      scc.emplace_back();
      for (int y = -1; y != x;) {
        y = stk.back();
        stk.pop_back();
        instk[y] = 0;
        scc.back().push_back(y);
      }
    }
  }
  void solve() {
    for (int i = 0; i < n; i++)
      if (!time[i]) dfs(i);
    reverse(scc.begin(), scc.end());
    // scc in topological order
  }
};
```

3.4.1. 2-Satisfiability

Requires: Strongly Connected Components

```


// 1 based, vertex in SCC = MAXN * 2
// (not i) is i + n
struct two_SAT {
  int n, ans[MAXN];
  SCC S;
  void imply(int a, int b) { S.make_edge(a, b); }
  bool solve(int _n) {
    n = _n;
    S.solve(n * 2);
    for (int i = 1; i <= n; i++) {
      if (S.scc[i] == S.scc[i + n]) return false;
      ans[i] = (S.scc[i] < S.scc[i + n]);
    }
    return true;
  }
  void init(int _n) {
    n = _n;
    fill_n(ans, n + 1, 0);
    S.init(n * 2);
  }
} SAT;
```

3.5. Biconnected Components

3.5.1. Articulation Points

```
void dfs(int x, int p) {
  tin[x] = low[x] = ++t;
  int ch = 0;
  for (auto u : g[x])
    if (u.first != p) {
      if (!ins[u.second])
        st.push(u.second), ins[u.second] = true;
      if (tin[u.first]) {
        low[x] = min(low[x], tin[u.first]);
        continue;
      }
      ++ch;
      dfs(u.first, x);
      low[x] = min(low[x], low[u.first]);
      if (low[u.first] >= tin[x]) {
        cut[x] = true;
        ++sz;
        while (true) {
          int e = st.top();
          st.pop();
          bcc[e] = sz;
          if (e == u.second) break;
        }
      }
    }
  if (ch == 1 && p == -1) cut[x] = false;
}
```

3.5.2. Bridges

```
// if there are multi-edges, then they are not bridges
void dfs(int x, int p) {
  tin[x] = low[x] = ++t;
  st.push(x);
  for (auto u : g[x])
    if (u.first != p) {
      if (tin[u.first]) {
        low[x] = min(low[x], tin[u.first]);
        continue;
      }
      dfs(u.first, x);
      low[x] = min(low[x], low[u.first]);
      if (low[u.first] == tin[u.first]) br[u.second] = true;
    }
  if (tin[x] == low[x]) {
    ++sz;
    while (st.size()) {
      int u = st.top();
      st.pop();
      bcc[u] = sz;
      if (u == x) break;
    }
  }
}
```

3.6. Triconnected Components

```


// requires a union-find data structure
struct ThreeEdgeCC {
  int V, ind;
  vector<int> id, pre, post, low, deg, path;
  vector<vector<int>> components;
  UnionFind uf;
  template <class Graph>
  void dfs(const Graph &G, int v, int prev) {
    pre[v] = ++ind;
    for (int w : G[v])
      if (w != v) {
        if (w == prev) {
          prev = -1;
          continue;
        }
        if (pre[w] != -1) {
          if (pre[w] < pre[v]) {
            deg[v]++;
            low[v] = min(low[v], pre[w]);
          } else {
            deg[v]--;
            int &u = path[v];
            for (; u != -1 && pre[u] <= pre[w] &&
                   pre[w] <= post[u];) {
              uf.join(v, u);
              deg[v] += deg[u];
              u = path[u];
            }
          }
          continue;
        }
        dfs(G, w, v);
        if (path[w] == -1 && deg[w] <= 1) {
          deg[v] += deg[w];
          low[v] = min(low[v], low[w]);
          continue;
        }
        if (deg[w] == 0) w = path[w];
        if (low[v] > low[w]) {
          low[v] = min(low[v], low[w]);
          swap(w, path[v]);
        }
```

```
        for (; w != -1; w = path[w]) {
          uf.join(v, w);
          deg[v] += deg[w];
        }
      }
    post[v] = ind;
  }
  template <class Graph>
  ThreeEdgeCC(const Graph &G)
      : V(G.size()), ind(-1), id(V, -1), pre(V, -1),
        post(V), low(V, INT_MAX), deg(V, 0), path(V, -1),
        uf(V) {
    for (int v = 0; v < V; v++)
      if (pre[v] == -1) dfs(G, v, -1);
    components.reserve(uf.cnt);
    for (int v = 0; v < V; v++)
      if (uf.find(v) == v) {
        id[v] = components.size();
        components.emplace_back(1, v);
        components.back().reserve(uf.getSize(v));
      }
    for (int v = 0; v < V; v++)
      if (id[v] == -1)
        components[id[v] = id[uf.find(v)]].push_back(v);
  }
};
```

3.7. Centroid Decomposition

```
public class centroid_decomposition {
  // Find the size of the subtree under this node.
  public static int subtreeSize(int node, int par) {
    int res = 1;
    for (int next : adj[node]) {
      if (next == par) {
        continue;
      }
      res += subtreeSize(next, node);
    }
    return (subSize[node] = res);
  }

  // Find the centroid of the tree (the subtree with <= N/2 nodes)
  public static int getCentroid(int node, int par) {
    for (int next : adj[node]) {
      if (next == par) {
        continue;
      }
      // Keep searching for the centroid if there are subtrees with more
      // than N/2 nodes.
      if (subSize[next] * 2 > N) {
        return getCentroid(next, node);
      }
    }
    return node;
  }
}
```

3.8. Minimum Mean Cycle

```


// d[i][j] == 0 if {i,j} !in E
long long d[1003][1003], dp[1003][1003];

pair<long long, long long> MMWC() {
  memset(dp, 0x3f, sizeof(dp));
  for (int i = 1; i <= n; ++i) dp[0][i] = 0;
  for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= n; ++j) {
      for (int k = 1; k <= n; ++k) {
        dp[i][k] = min(dp[i - 1][j] + d[j][k], dp[i][k]);
      }
    }
  }
  long long au = 1ll << 31, ad = 1;
  for (int i = 1; i <= n; ++i) {
    if (dp[n][i] == 0x3f3f3f3f3f3f3f3f) continue;
    long long u = 0, d = 1;
    for (int j = n - 1; j >= 0; --j) {
      if ((dp[n][i] - dp[j][i]) * d > u * (n - j)) {
        u = dp[n][i] - dp[j][i];
        d = n - j;
      }
    }
    if (u * ad < au * d) au = u, ad = d;
  }
  long long g = __gcd(au, ad);
  return make_pair(au / g, ad / g);
}
```

3.9. Dominator Tree

```
// idom[n] is the unique node that strictly dominates n but
// does not strictly dominate any other node that strictly
// dominates n. idom[n] = 0 if n is entry or the entry
// cannot reach n.
struct DominatorTree {
  static const int MAXN = 200010;
  int n, s;
  vector<int> g[MAXN], pred[MAXN];
  vector<int> cov[MAXN];
  int dfn[MAXN], nfd[MAXN], ts;
  int par[MAXN];
  int sdom[MAXN], idom[MAXN];
  int mom[MAXN], mn[MAXN];

  inline bool cmp(int u, int v) { return dfn[u] < dfn[v]; }

  int eval(int u) {
    if (mom[u] == u) return u;
    int res = eval(mom[u]);
    if (cmp(sdom[mn[mom[u]]], sdom[mn[u]]))
      mn[u] = mn[mom[u]];
    return mom[u] = res;
  }

  void init(int _n, int _s) {
    n = _n;
    s = _s;
    REP1(i, 1, n) {
      g[i].clear();
      pred[i].clear();
      idom[i] = 0;
    }
  }
  void add_edge(int u, int v) {
    g[u].push_back(v);
    pred[v].push_back(u);
  }
  void DFS(int u) {
    ts++;
    dfn[u] = ts;
    nfd[ts] = u;
    for (int v : g[u])
      if (dfn[v] == 0) {
        par[v] = u;
        DFS(v);
      }
  }
  void build() {
    ts = 0;
    REP1(i, 1, n) {
      dfn[i] = nfd[i] = 0;
      cov[i].clear();
      mom[i] = mn[i] = sdom[i] = i;
    }
    DFS(s);
    for (int i = ts; i >= 2; i--) {
      int u = nfd[i];
      if (u == 0) continue;
      for (int v : pred[u])
        if (dfn[v]) {
          eval(v);
          if (cmp(sdom[mn[v]], sdom[u]))
            sdom[u] = sdom[mn[v]];
        }
      cov[sdom[u]].push_back(u);
      mom[u] = par[u];
      for (int w : cov[par[u]]) {
        eval(w);
        if (cmp(sdom[mn[w]], par[u])) idom[w] = mn[w];
        else idom[w] = par[u];
      }
      cov[par[u]].clear();
    }
    REP1(i, 2, ts) {
      int u = nfd[i];
      if (u == 0) continue;
      if (idom[u] != sdom[u]) idom[u] = idom[idom[u]];
    }
  }
} dom;
```

3.10. Manhattan Distance MST

```


// returns [(dist, from, to), ...]
// then do normal mst afterwards
typedef Point<int> P;
vector<array<int, 3>> manhattanMST(vector<P> ps) {
  vi id(sz(ps));
  iota(all(id), 0);
  vector<array<int, 3>> edges;
```

```
  rep(k, 0, 4) {
    sort(all(id), [&](int i, int j) {
      return (ps[i] - ps[j]).x < (ps[j] - ps[i]).y;
    });
    map<int, int> sweep;
    for (int i : id) {
      for (auto it = sweep.lower_bound(-ps[i].y);
           it != sweep.end(); sweep.erase(it++)) {
        int j = it->second;
        P d = ps[i] - ps[j];
        if (d.y > d.x) break;
        edges.push_back({d.y + d.x, i, j});
      }
      sweep[-ps[i].y] = i;
    }
    for (P &p : ps)
      if (k & 1) p.x = -p.x;
      else swap(p.x, p.y);
  }
  return edges;
}
```

4. Math

4.1. Number Theory

4.1.1. Miller-Rabin

```

// checks if Mod::MOD is prime
bool is_prime() {
  if (MOD < 2 || MOD % 2 == 0) return MOD == 2;
  Mod A[] = {2, 7, 61}; // for int values (< 2^31)
  // ll: 2, 325, 9375, 28178, 450775, 9780504, 1795265022
  int s = __builtin_ctzll(MOD - 1), i;
  for (Mod a : A) {
    Mod x = a ^ (MOD >> s);
    for (i = 0; i < s && (x + 1).v > 2; i++) x *= x;
    if (i && x != -1) return 0;
  }
  return 1;
}
```

4.1.2. 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;
                }
            }
        }
    }
}
```

4.1.3. 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;
    }
}
```

4.1.4. Extended GCD

```
// returns (p, q, g): p * a + q * b == g == gcd(a, b)
// g is not guaranteed to be positive when a < 0 or b < 0
tuple<ll, ll, ll> extgcd(ll a, ll b) {
  ll s = 1, t = 0, u = 0, v = 1;
  while (b) {
    ll q = a / b;
    swap(a -= q * b, b);
    swap(s -= q * t, t);
    swap(u -= q * v, v);
  }
  return {s, u, a};
}
```

4.1.5. Chinese Remainder Theorem

```
// for 0 <= a < m, 0 <= b < n, returns the smallest x >= 0
// such that x % m == a and x % n == b
ll crt(ll a, ll m, ll b, ll n) {
  if (n > m) swap(a, b), swap(m, n);
  auto [x, y, g] = extgcd(m, n);
  assert((a - b) % g == 0); // no solution
  x = ((b - a) / g * x) % (n / g) * m + a;
  return x < 0 ? x + m / g * n : x;
}
```

4.1.6. Chinese Sieve

```
const ll N = 1000000;
// f, g, h multiplicative, h = f (dirichlet convolution) g
ll pre_g(ll n);
ll pre_h(ll n);
// preprocessed prefix sum of f
ll pre_f[N];
// prefix sum of multiplicative function f
ll solve_f(ll n) {
  static unordered_map<ll, ll> m;
  if (n < N) return pre_f[n];
  if (m.count(n)) return m[n];
  ll ans = pre_h(n);
  for (ll l = 2, r; l <= n; l = r + 1) {
    r = n / (n / l);
    ans -= (pre_g(r) - pre_g(l - 1)) * djs_f(n / l);
  }
  return m[n] = ans;
}
```

4.2. Combinatorics

4.2.1. 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 litt
      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) {
```

```cpp
      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
    // 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;
    }
  }
}
```

4.3. Algebra

4.3.1. Formal Power Series

```cpp



template <typename mint>
struct FormalPowerSeries : vector<mint> {
  using vector<mint>::vector;
  using FPS = FormalPowerSeries;

  FPS &operator+=(const FPS &r) {
    if (r.size() > this->size()) this->resize(r.size());
    for (int i = 0; i < (int)r.size(); i++)
      (*this)[i] += r[i];
    return *this;
  }

  FPS &operator+=(const mint &r) {
    if (this->empty()) this->resize(1);
    (*this)[0] += r;
    return *this;
  }

  FPS &operator-=(const FPS &r) {
    if (r.size() > this->size()) this->resize(r.size());
    for (int i = 0; i < (int)r.size(); i++)
      (*this)[i] -= r[i];
    return *this;
  }

  FPS &operator-=(const mint &r) {
    if (this->empty()) this->resize(1);
    (*this)[0] -= r;
    return *this;
  }

  FPS &operator*=(const mint &v) {
    for (int k = 0; k < (int)this->size(); k++)
      (*this)[k] *= v;
    return *this;
  }

  FPS &operator/=(const FPS &r) {
    if (this->size() < r.size()) {
      this->clear();
      return *this;
    }
    int n = this->size() - r.size() + 1;
    if ((int)r.size() <= 64) {
      FPS f(*this), g(r);
      g.shrink();
      mint coeff = g.back().inverse();
      for (auto &x : g) x *= coeff;
      int deg = (int)f.size() - (int)g.size() + 1;
      int gs = g.size();
      FPS quo(deg);
      for (int i = deg - 1; i >= 0; i--) {
        quo[i] = f[i + gs - 1];
        for (int j = 0; j < gs; j++)
          f[i + j] -= quo[i] * g[j];
      }
      *this = quo * coeff;
      this->resize(n, mint(0));
      return *this;
    }
    return *this = ((*this).rev().pre(n) * r.rev().inv(n))
                       .pre(n)
                       .rev();
  }

  FPS &operator%=(const FPS &r) {
    *this -= *this / r * r;
    shrink();
    return *this;
  }

  FPS operator+(const FPS &r) const {
    return FPS(*this) += r;
  }
  FPS operator+(const mint &v) const {
    return FPS(*this) += v;
  }
  FPS operator-(const FPS &r) const {
    return FPS(*this) -= r;
  }
  FPS operator-(const mint &v) const {
    return FPS(*this) -= v;
  }
  FPS operator*(const FPS &r) const {
    return FPS(*this) *= r;
  }
  FPS operator*(const mint &v) const {
    return FPS(*this) *= v;
  }
  FPS operator/(const FPS &r) const {
    return FPS(*this) /= r;
  }
  FPS operator%(const FPS &r) const {
    return FPS(*this) %= r;
  }
  FPS operator-() const {
    FPS ret(this->size());
    for (int i = 0; i < (int)this->size(); i++)
      ret[i] = -(*this)[i];
    return ret;
  }

  void shrink() {
    while (this->size() && this->back() == mint(0))
      this->pop_back();
  }

  FPS rev() const {
    FPS ret(*this);
    reverse(begin(ret), end(ret));
    return ret;
  }

  FPS dot(FPS r) const {
    FPS ret(min(this->size(), r.size()));
    for (int i = 0; i < (int)ret.size(); i++)
      ret[i] = (*this)[i] * r[i];
    return ret;
  }

  FPS pre(int sz) const {
    return FPS(begin(*this),
               begin(*this) + min((int)this->size(), sz));
  }

  FPS operator>>(int sz) const {
    if ((int)this->size() <= sz) return {};
    FPS ret(*this);
    ret.erase(ret.begin(), ret.begin() + sz);
    return ret;
  }

  FPS operator<<(int sz) const {
    FPS ret(*this);
    ret.insert(ret.begin(), sz, mint(0));
    return ret;
  }

  FPS diff() const {
    const int n = (int)this->size();
    FPS ret(max(0, n - 1));
    mint one(1), coeff(1);
    for (int i = 1; i < n; i++) {
      ret[i - 1] = (*this)[i] * coeff;
```

```
      coeff += one;
    }
    return ret;
  }

  FPS integral() const {
    const int n = (int)this->size();
    FPS ret(n + 1);
    ret[0] = mint(0);
    if (n > 0) ret[1] = mint(1);
    auto mod = mint::get_mod();
    for (int i = 2; i <= n; i++)
      ret[i] = (-ret[mod % i]) * (mod / i);
    for (int i = 0; i < n; i++) ret[i + 1] *= (*this)[i];
    return ret;
  }

  mint eval(mint x) const {
    mint r = 0, w = 1;
    for (auto &v : *this) r += w * v, w *= x;
    return r;
  }

  FPS log(int deg = -1) const {
    assert((*this)[0] == mint(1));
    if (deg == -1) deg = (int)this->size();
    return (this->diff() * this->inv(deg))
    .pre(deg - 1)
    .integral();
  }

  FPS pow(int64_t k, int deg = -1) const {
    const int n = (int)this->size();
    if (deg == -1) deg = n;
    for (int i = 0; i < n; i++) {
      if ((*this)[i] != mint(0)) {
        if (i * k > deg) return FPS(deg, mint(0));
        mint rev = mint(1) / (*this)[i];
        FPS ret =
        (((*this * rev) >> i).log(deg) * k).exp(deg) *
        ((*this)[i].pow(k));
        ret = (ret << (i * k)).pre(deg);
        if ((int)ret.size() < deg) ret.resize(deg, mint(0));
        return ret;
      }
    }
    return FPS(deg, mint(0));
  }

  static void *ntt_ptr;
  static void set_fft();
  FPS &operator*=(const FPS &r);
  void ntt();
  void intt();
  void ntt_doubling();
  static int ntt_pr();
  FPS inv(int deg = -1) const;
  FPS exp(int deg = -1) const;
};
template <typename mint>
void *FormalPowerSeries<mint>::ntt_ptr = nullptr;
```

4.4. Theorems

4.4.1. Kirchhoff's Theorem

Denote L be a $n \times n$ matrix as the Laplacian matrix of graph G, where $L_{ii} = d(i)$, $L_{ij} = -c$ where c is the number of edge (i, j) in G.

- The number of undirected spanning in G is $|\det(\tilde{L}_{11})|$.
- The number of directed spanning tree rooted at r in G is $|\det(\tilde{L}_{rr})|$.

4.4.2. Tutte's Matrix

Let D be a $n \times n$ matrix, where $d_{ij} = x_{ij}$ (x_{ij} is chosen uniformly at random) if $i < j$ and $(i, j) \in E$, otherwise $d_{ij} = -d_{ji}$. $\frac{rank(D)}{2}$ is the maximum matching on G.

4.4.3. Cayley's Formula

- Given a degree sequence $d_1, d_2, \ldots, d_n$ for each *labeled* vertices, there are
$$\frac{(n-2)!}{(d_1-1)!(d_2-1)!\cdots(d_n-1)!}$$
spanning trees.
- Let $T_{n,k}$ be the number of *labeled* forests on n vertices with k components, such that vertex $1, 2, \ldots, k$ belong to different components. Then $T_{n,k} = kn^{n-k-1}$.

4.4.4. Erdős–Gallai Theorem

A sequence of non-negative integers $d_1 \geq d_2 \geq \ldots \geq d_n$ can be represented as the degree sequence of a finite simple graph on n vertices if and only if $d_1 + d_2 + \ldots + d_n$ is even and

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k)$$

holds for all $1 \leq k \leq n$.

4.4.5. Burnside's Lemma

Let X be a set and G be a group that acts on X. For $g \in G$, denote by X^g the elements fixed by g:

$$X^g = \{x \in X \mid gx \in X\}$$

Then
$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|.$$

5. Numeric

5.1. Barrett Reduction

```
using ull = unsigned long long;
using uL = __uint128_t;
// very fast calculation of a % m
struct reduction {
  const ull m, d;
  explicit reduction(ull m) : m(m), d(((uL)1 << 64) / m) {}
  inline ull operator()(ull a) const {
    ull q = (ull)(((uL)d * a) >> 64);
    return (a -= q * m) >= m ? a - m : a;
  }
};
```

5.2. Long Long Multiplication

```
using ull = unsigned long long;
using ll = long long;
using ld = long double;
// returns a * b % M where a, b < M < 2**63
ull mult(ull a, ull b, ull M) {
  ll ret = a * b - M * ull(ld(a) * ld(b) / ld(M));
  return ret + M * (ret < 0) - M * (ret >= (ll)M);
}
```

5.3. Fast Fourier Transform

```
template <typename T>
void fft_(int n, vector<T> &a, vector<T> &rt, bool inv) {
  vector<int> br(n);
  for (int i = 1; i < n; i++) {
    br[i] = (i & 1) ? br[i - 1] + n / 2 : br[i / 2] / 2;
    if (br[i] > i) swap(a[i], a[br[i]]);
  }
  for (int len = 2; len <= n; len *= 2)
    for (int i = 0; i < n; i += len)
      for (int j = 0; j < len / 2; j++) {
        int pos = n / len * (inv ? len - j : j);
        T u = a[i + j], v = a[i + j + len / 2] * rt[pos];
        a[i + j] = u + v, a[i + j + len / 2] = u - v;
      }
  if (T minv = T(1) / T(n); inv)
    for (T &x : a) x *= minv;
}
```

```

void ntt(vector<Mod> &a, bool inv, Mod primitive_root) {
  int n = a.size();
  Mod root = primitive_root ^ (MOD - 1) / n;
  vector<Mod> rt(n + 1, 1);
  for (int i = 0; i < n; i++) rt[i + 1] = rt[i] * root;
  fft_(n, a, rt, inv);
}
void fft(vector<complex<double>> &a, bool inv) {
  int n = a.size();
  vector<complex<double>> rt(n + 1);
  double arg = acos(-1) * 2 / n;
  for (int i = 0; i <= n; i++)
    rt[i] = {cos(arg * i), sin(arg * i)};
  fft_(n, a, rt, inv);
}
```

5.4. Fast Walsh-Hadamard Transform

```

void fwht(vector<Mod> &a, bool inv) {
  int n = a.size();
  for (int d = 1; d < n; d <<= 1)
    for (int m = 0; m < n; m++)
      if (!(m & d)) {
        inv ? a[m] -= a[m | d] : a[m] += a[m | d]; // AND
        inv ? a[m | d] -= a[m] : a[m | d] += a[m]; // OR
        Mod x = a[m], y = a[m | d];                // XOR
        a[m] = x + y, a[m | d] = x - y;            // XOR
      }
  if (Mod iv = Mod(1) / n; inv) // XOR
    for (Mod &i : a) i *= iv;   // XOR
}
```

5.5. Subset Convolution

```

#pragma GCC target("popcnt")
#include <immintrin.h>

void fwht(int n, vector<vector<Mod>> &a, bool inv) {
  for (int h = 0; h < n; h++)
    for (int i = 0; i < (1 << n); i++)
      if (!(i & (1 << h)))
        for (int k = 0; k <= n; k++)
          inv ? a[i | (1 << h)][k] -= a[i][k]
              : a[i | (1 << h)][k] += a[i][k];
}
// c[k] = sum(popcnt(i & j) == sz && i | j == k) a[i] * b[j]
vector<Mod> subset_convolution(int n, int sz,
                               const vector<Mod> &a_,
                               const vector<Mod> &b_) {
  int len = n + sz + 1, N = 1 << n;
  vector<vector<Mod>> a(1 << n, vector<Mod>(len, 0)), b = a;
  for (int i = 0; i < N; i++)
    a[i][_mm_popcnt_u64(i)] = a_[i],
    b[i][_mm_popcnt_u64(i)] = b_[i];
  fwht(n, a, 0), fwht(n, b, 0);
  for (int i = 0; i < N; i++) {
    vector<Mod> tmp(len);
    for (int j = 0; j < len; j++)
      for (int k = 0; k <= j; k++)
        tmp[j] += a[i][k] * b[i][j - k];
    a[i] = tmp;
  }
  fwht(n, a, 1);
  vector<Mod> c(N);
  for (int i = 0; i < N; i++)
    c[i] = a[i][_mm_popcnt_u64(i) + sz];
  return c;
}
```

5.6. Linear Recurrences

5.6.1. Berlekamp-Massey Algorithm

```
template <typename T>
vector<T> berlekamp_massey(const vector<T> &s) {
  int n = s.size(), l = 0, m = 1;
  vector<T> r(n), p(n);
  r[0] = p[0] = 1;
  T b = 1, d = 0;
  for (int i = 0; i < n; i++, m++, d = 0) {
    for (int j = 0; j <= l; j++) d += r[j] * s[i - j];
    if ((d /= b) == 0) continue; // change if T is float
    auto t = r;
    for (int j = m; j < n; j++) r[j] -= d * p[j - m];
    if (l * 2 <= i) l = i + 1 - l, b *= d, m = 0, p = t;
  }
  return r.resize(l + 1), reverse(r.begin(), r.end()), r;
}
```

5.6.2. Linear Recurrence Calculation

```
template <typename T> struct lin_rec {
  using poly = vector<T>;
  poly mul(poly a, poly b, poly m) {
    int n = m.size();
    poly r(n);
    for (int i = n - 1; i >= 0; i--) {
      r.insert(r.begin(), 0), r.pop_back();
      T c = r[n - 1] + a[n - 1] * b[i];
      // c /= m[n - 1];  if m is not monic
      for (int j = 0; j < n; j++)
        r[j] += a[j] * b[i] - c * m[j];
    }
    return r;
  }
  poly pow(poly p, ll k, poly m) {
    poly r(m.size());
    r[0] = 1;
    for (; k; k >>= 1, p = mul(p, p, m))
      if (k & 1) r = mul(r, p, m);
    return r;
  }
  T calc(poly t, poly r, ll k) {
    int n = r.size();
    poly p(n);
    p[1] = 1;
    poly q = pow(p, k, r);
    T ans = 0;
    for (int i = 0; i < n; i++) ans += t[i] * q[i];
    return ans;
  }
};
```

5.7. Polynomial Interpolation

```

// returns a, such that a[0]x^0 + a[1]x^1 + a[2]x^2 + ...
// passes through the given points
typedef vector<double> vd;
vd interpolate(vd x, vd y, int n) {
  vd res(n), temp(n);
  rep(k, 0, n - 1) rep(i, k + 1, n) y[i] =
  (y[i] - y[k]) / (x[i] - x[k]);
  double last = 0;
  temp[0] = 1;
  rep(k, 0, n) rep(i, 0, n) {
    res[i] += y[k] * temp[i];
    swap(last, temp[i]);
    temp[i] -= last * x[k];
  }
  return res;
}
```

5.8. Simplex Algorithm

```
// Two-phase simplex algorithm for solving linear programs
// of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be
//        stored
//
// OUTPUT: value of the optimal solution (infinity if
// unbounded
//         above, nan if infeasible)
//
// To use this code, create an LPSolver object with A, b,
// and c as arguments.  Then, call Solve(x).

typedef long double ld;
typedef vector<ld> vd;
typedef vector<vd> vvd;
typedef vector<int> vi;

const ld EPS = 1e-9;

struct LPSolver {
  int m, n;
  vi B, N;
  vvd D;

  LPSolver(const vvd &A, const vd &b, const vd &c)
      : m(b.size()), n(c.size()), N(n + 1), B(m),
        D(m + 2, vd(n + 2)) {
    for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++) D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) {
      B[i] = n + i;
      D[i][n] = -1;
      D[i][n + 1] = b[i];
    }
    for (int j = 0; j < n; j++) {
      N[j] = j;
      D[m][j] = -c[j];
    }
    N[n] = -1;
    D[m + 1][n] = 1;
  }

  void Pivot(int r, int s) {
    double inv = 1.0 / D[r][s];
    for (int i = 0; i < m + 2; i++)
      if (i != r)
```

```
      for (int j = 0; j < n + 2; j++)
        if (j != s) D[r][j] -= D[r][j] * D[i][s] * inv;
    for (int j = 0; j < n + 2; j++)
      if (j != s) D[r][j] *= inv;
    for (int i = 0; i < m + 2; i++)
      if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m + 1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) continue;
        if (s == -1 || D[x][j] < D[x][s] ||
            D[x][j] == D[x][s] && N[j] < N[s])
          s = j;
      }
      if (D[x][s] > -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] < EPS) continue;
        if (r == -1 ||
            D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
            (D[i][n + 1] / D[i][s]) ==
            (D[r][n + 1] / D[r][s]) &&
            B[i] < B[r])
          r = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  ld Solve(vd &x) {
    int r = 0;
    for (int i = 1; i < m; i++)
      if (D[i][n + 1] < D[r][n + 1]) r = i;
    if (D[r][n + 1] < -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m + 1][n + 1] < -EPS)
        return -numeric_limits<ld>::infinity();
      for (int i = 0; i < m; i++)
        if (B[i] == -1) {
          int s = -1;
          for (int j = 0; j <= n; j++)
            if (s == -1 || D[i][j] < D[i][s] ||
                D[i][j] == D[i][s] && N[j] < N[s])
              s = j;
          Pivot(i, s);
        }
    }
    if (!Simplex(2)) return numeric_limits<ld>::infinity();
    x = vd(n);
    for (int i = 0; i < m; i++)
      if (B[i] < n) x[B[i]] = D[i][n + 1];
    return D[m][n + 1];
  }
};

int main() {

  const int m = 4;
  const int n = 3;
  ld _A[m][n] = {
  {6, -1, 0}, {-1, -5, 0}, {1, 5, 1}, {-1, -5, -1}};
  ld _b[m] = {10, -4, 5, -5};
  ld _c[n] = {1, -1, 0};

  vvd A(m);
  vd b(_b, _b + m);
  vd c(_c, _c + n);
  for (int i = 0; i < m; i++) A[i] = vd(_A[i], _A[i] + n);

  LPSolver solver(A, b, c);
  vd x;
  ld value = solver.Solve(x);

  cerr << "VALUE: " << value << endl; // VALUE: 1.29032
  cerr << "SOLUTION:"; // SOLUTION: 1.74194 0.451613 1
  for (size_t i = 0; i < x.size(); i++) cerr << " " << x[i];
  cerr << endl;
  return 0;
}
```

6. Geometry

6.1. Point

```
template <typename T> struct P {
  T x, y;
  P(T x = 0, T y = 0) : x(x), y(y) {}
  bool operator<(const P &p) const {
    return tie(x, y) < tie(p.x, p.y);
  }
  bool operator==(const P &p) const {
    return tie(x, y) == tie(p.x, p.y);
  }
  P operator-() const { return {-x, -y}; }
  P operator+(P p) const { return {x + p.x, y + p.y}; }
  P operator-(P p) const { return {x - p.x, y - p.y}; }
  P operator*(T d) const { return {x * d, y * d}; }
  P operator/(T d) const { return {x / d, y / d}; }
  T dist2() const { return x * x + y * y; }
  double len() const { return sqrt(dist2()); }
  P unit() const { return *this / len(); }
  friend T dot(P a, P b) { return a.x * b.x + a.y * b.y; }
  friend T cross(P a, P b) { return a.x * b.y - a.y * b.x; }
  friend T cross(P a, P b, P o) {
    return cross(a - o, b - o);
  }
};
using pt = P<ll>;
```

6.1.1. Quarternion

```
constexpr double PI = 3.141592653589793;
constexpr double EPS = 1e-7;
struct Q {
  using T = double;
  T x, y, z, r;
  Q(T r = 0) : x(0), y(0), z(0), r(r) {}
  Q(T x, T y, T z, T r = 0) : x(x), y(y), z(z), r(r) {}
  friend bool operator==(const Q &a, const Q &b) {
    return (a - b).abs2() <= EPS;
  }
  friend bool operator!=(const Q &a, const Q &b) {
    return !(a == b);
  }
  Q operator-() { return Q(-x, -y, -z, -r); }
  Q operator+(const Q &b) const {
    return Q(x + b.x, y + b.y, z + b.z, r + b.r);
  }
  Q operator-(const Q &b) const {
    return Q(x - b.x, y - b.y, z - b.z, r - b.r);
  }
  Q operator*(const T &t) const {
    return Q(x * t, y * t, z * t, r * t);
  }
  Q operator*(const Q &b) const {
    return Q(r * b.x + x * b.r + y * b.z - z * b.y,
             r * b.y - x * b.z + y * b.r + z * b.x,
             r * b.z + x * b.y - y * b.x + z * b.r,
             r * b.r - x * b.x - y * b.y - z * b.z);
  }
  Q operator/(const Q &b) const { return *this * b.inv(); }
  T abs2() const { return r * r + x * x + y * y + z * z; }
  T len() const { return sqrt(abs2()); }
  Q conj() const { return Q(-x, -y, -z, r); }
  Q unit() const { return *this * (1.0 / len()); }
  Q inv() const { return conj() * (1.0 / abs2()); }
  friend T dot(Q a, Q b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
  }
  friend Q cross(Q a, Q b) {
    return Q(a.y * b.z - a.z * b.y, a.z * b.x - a.x * b.z,
             a.x * b.y - a.y * b.x);
  }
  friend Q rotation_around(Q axis, T angle) {
    return axis.unit() * sin(angle / 2) + cos(angle / 2);
  }
  Q rotated_around(Q axis, T angle) {
    Q u = rotation_around(axis, angle);
    return u * *this / u;
  }
  friend Q rotation_between(Q a, Q b) {
    a = a.unit(), b = b.unit();
    if (a == -b) {
      // degenerate case
      Q ortho = abs(a.y) > EPS ? cross(a, Q(1, 0, 0))
                               : cross(a, Q(0, 1, 0));
      return rotation_around(ortho, PI);
    }
    return (a * (a + b)).conj();
  }
};
```

6.1.2. Spherical Coordinates

```
struct car_p {
  double x, y, z;
};
struct sph_p {
  double r, theta, phi;
```

```
};

sph_p conv(car_p p) {
  double r = sqrt(p.x * p.x + p.y * p.y + p.z * p.z);
  double theta = asin(p.y / r);
  double phi = atan2(p.y, p.x);
  return {r, theta, phi};
}
car_p conv(sph_p p) {
  double x = p.r * cos(p.theta) * sin(p.phi);
  double y = p.r * cos(p.theta) * cos(p.phi);
  double z = p.r * sin(p.theta);
  return {x, y, z};
}
```

6.2. Segments

```
// for non-collinear ABCD, if segments AB and CD intersect
bool intersects(pt a, pt b, pt c, pt d) {
  if (cross(b, c, a) * cross(b, d, a) > 0) return false;
  if (cross(d, a, c) * cross(d, b, c) > 0) return false;
  return true;
}
// the intersection point of lines AB and CD
pt intersect(pt a, pt b, pt c, pt d) {
  auto x = cross(b, c, a), y = cross(b, d, a);
  if (x == y) {
    // if(abs(x, y) < 1e-8) {
    // is parallel
  } else {
    return d * (x / (x - y)) - c * (y / (x - y));
  }
}
```

6.3. Angular Sort

```
auto angle_cmp = [](const pt &a, const pt &b) {
  auto btm = [](const pt &a) {
    return a.y < 0 || (a.y == 0 && a.x < 0);
  };
  return make_tuple(btm(a), a.y * b.x, abs2(a)) <
         make_tuple(btm(b), a.x * b.y, abs2(b));
};
void angular_sort(vector<pt> &p) {
  sort(p.begin(), p.end(), angle_cmp);
}
```

6.4. Convex Polygon Minkowski Sum

```
// O(n) convex polygon minkowski sum
// must be sorted and counterclockwise
vector<pt> minkowski_sum(vector<pt> p, vector<pt> q) {
  auto diff = [](vector<pt> &c) {
    auto rcmp = [](pt a, pt b) {
      return pt{a.y, a.x} < pt{b.y, b.x};
    };
    rotate(c.begin(), min_element(ALL(c), rcmp), c.end());
    c.push_back(c[0]);
    vector<pt> ret;
    for (int i = 1; i < c.size(); i++)
      ret.push_back(c[i] - c[i - 1]);
    return ret;
  };
  auto dp = diff(p), dq = diff(q);
  pt cur = p[0] + q[0];
  vector<pt> d(dp.size() + dq.size()), ret = {cur};
  // include angle_cmp from angular-sort.cpp
  merge(ALL(dp), ALL(dq), d.begin(), angle_cmp);
  // optional: make ret strictly convex (UB if degenerate)
  int now = 0;
  for (int i = 1; i < d.size(); i++) {
    if (cross(d[i], d[now]) == 0) d[now] = d[now] + d[i];
    else d[++now] = d[i];
  }
  d.resize(now + 1);
  // end optional part
  for (pt v : d) ret.push_back(cur = cur + v);
  return ret.pop_back(), ret;
}
```

6.5. Point In Polygon

```
bool on_segment(pt a, pt b, pt p) {
  return cross(a, b, p) == 0 && dot((p - a), (p - b)) <= 0;
}
// p can be any polygon, but this is O(n)
bool inside(const vector<pt> &p, pt a) {
  int cnt = 0, n = p.size();
  for (int i = 0; i < n; i++) {
    pt l = p[i], r = p[(i + 1) % n];
    // change to return 0; for strict version
    if (on_segment(l, r, a)) return 1;
    cnt ^= ((a.y < l.y) - (a.y < r.y)) * cross(l, r, a) > 0;
  }
  return cnt;
}
```

6.5.1. Convex Version

```
// no preprocessing version
// p must be a strict convex hull, counterclockwise
// if point is inside or on border
bool is_inside(const vector<pt> &c, pt p) {
  int n = c.size(), l = 1, r = n - 1;
  if (cross(c[0], c[1], p) < 0) return false;
  if (cross(c[n - 1], c[0], p) < 0) return false;
  while (l < r - 1) {
    int m = (l + r) / 2;
    T a = cross(c[0], c[m], p);
    if (a > 0) l = m;
    else if (a < 0) r = m;
    else return dot(c[0] - p, c[m] - p) <= 0;
  }
  if (l == r) return dot(c[0] - p, c[l] - p) <= 0;
  else return cross(c[l], c[r], p) >= 0;
}

// with preprocessing version
vector<pt> vecs;
pt center;
// p must be a strict convex hull, counterclockwise
// BEWARE OF OVERFLOWS!!
void preprocess(vector<pt> p) {
  for (auto &v : p) v = v * 3;
  center = p[0] + p[1] + p[2];
  center.x /= 3, center.y /= 3;
  for (auto &v : p) v = v - center;
  vecs = (angular_sort(p), p);
}
bool intersect_strict(pt a, pt b, pt c, pt d) {
  if (cross(b, c, a) * cross(b, d, a) > 0) return false;
  if (cross(d, a, c) * cross(d, b, c) >= 0) return false;
  return true;
}
// if point is inside or on border
bool query(pt p) {
  p = p * 3 - center;
  auto pr = upper_bound(ALL(vecs), p, angle_cmp);
  if (pr == vecs.end()) pr = vecs.begin();
  auto pl = (pr == vecs.begin()) ? vecs.back() : *(pr - 1);
  return !intersect_strict({0, 0}, p, pl, *pr);
}
```

6.6. Closest Pair

```
vector<pll> p; // sort by x first!
bool cmpy(const pll &a, const pll &b) const {
  return a.y < b.y;
}
ll sq(ll x) { return x * x; }
// returns (minimum dist)^2 in [l, r)
ll solve(int l, int r) {
  if (r - l <= 1) return 1e18;
  int m = (l + r) / 2;
  ll mid = p[m].x, d = min(solve(l, m), solve(m, r));
  auto pb = p.begin();
  inplace_merge(pb + l, pb + m, pb + r, cmpy);
  vector<pll> s;
  for (int i = l; i < r; i++)
    if (sq(p[i].x - mid) < d) s.push_back(p[i]);
  for (int i = 0; i < s.size(); i++)
    for (int j = i + 1;
         j < s.size() && sq(s[j].y - s[i].y) < d; j++)
      d = min(d, dis(s[i], s[j]));
  return d;
}
```

6.7. Minimum Enclosing Circle

```


typedef Point<double> P;
double ccRadius(const P &A, const P &B, const P &C) {
  return (B - A).dist() * (C - B).dist() * (A - C).dist() /
         abs((B - A).cross(C - A)) / 2;
}
P ccCenter(const P &A, const P &B, const P &C) {
  P b = C - A, c = B - A;
  return A + (b * c.dist2() - c * b.dist2()).perp() /
             b.cross(c) / 2;
}
pair<P, double> mec(vector<P> ps) {
  shuffle(all(ps), mt19937(time(0)));
  P o = ps[0];
  double r = 0, EPS = 1 + 1e-8;
  rep(i, 0, sz(ps)) if ((o - ps[i]).dist() > r * EPS) {
    o = ps[i], r = 0;
    rep(j, 0, i) if ((o - ps[j]).dist() > r * EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      rep(k, 0, j) if ((o - ps[k]).dist() > r * EPS) {
```

```
      o = ccCenter(ps[i], ps[j], ps[k]);
      r = (o - ps[i]).dist();
    }
  }
}
return {o, r};
}
```

6.8. Half Plane Intersection

```
struct Line {
  Point P;
  Vector v;
  bool operator<(const Line &b) const {
    return atan2(v.y, v.x) < atan2(b.v.y, b.v.x);
  }
};
bool OnLeft(const Line &L, const Point &p) {
  return Cross(L.v, p - L.P) > 0;
}
Point GetIntersection(Line a, Line b) {
  Vector u = a.P - b.P;
  Double t = Cross(b.v, u) / Cross(a.v, b.v);
  return a.P + a.v * t;
}
int HalfplaneIntersection(Line *L, int n, Point *poly) {
  sort(L, L + n);

  int first, last;
  Point *p = new Point[n];
  Line *q = new Line[n];
  q[first = last = 0] = L[0];
  for (int i = 1; i < n; i++) {
    while (first < last && !OnLeft(L[i], p[last - 1]))
      last--;
    while (first < last && !OnLeft(L[i], p[first])) first++;
    q[++last] = L[i];
    if (fabs(Cross(q[last].v, q[last - 1].v)) < EPS) {
      last--;
      if (OnLeft(q[last], L[i].P)) q[last] = L[i];
    }
    if (first < last)
      p[last - 1] = GetIntersection(q[last - 1], q[last]);
  }
  while (first < last && !OnLeft(q[first], p[last - 1]))
    last--;
  if (last - first <= 1) return 0;
  p[last] = GetIntersection(q[last], q[first]);

  int m = 0;
  for (int i = first; i <= last; i++) poly[m++] = p[i];
  return m;
}
```

7. Strings

7.1. Knuth-Morris-Pratt Algorithm

```


vector<int> pi(const string &s) {
  vector<int> p(s.size());
  for (int i = 1; i < s.size(); i++) {
    int g = p[i - 1];
    while (g && s[i] != s[g]) g = p[g - 1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}
vector<int> match(const string &s, const string &pat) {
  vector<int> p = pi(pat + '\0' + s), res;
  for (int i = p.size() - s.size(); i < p.size(); i++)
    if (p[i] == pat.size())
      res.push_back(i - 2 * pat.size());
  return res;
}
```

7.2. Suffix Array

```


// sa[i]: starting index of suffix at rank i
//        0-indexed, sa[0] = n (empty string)
// lcp[i]: lcp of sa[i] and sa[i - 1], lcp[0] = 0
struct SuffixArray {
  vector<int> sa, lcp;
  SuffixArray(string &s,
              int lim = 256) { // or basic_string<int>
    int n = sz(s) + 1, k = 0, a, b;
    vector<int> x(all(s) + 1), y(n), ws(max(n, lim)),
    rank(n);
    sa = lcp = y, iota(all(sa), 0);
    for (int j = 0, p = 0; p < n;
         j = max(1, j * 2), lim = p) {
      p = j, iota(all(y), n - j);
      for (int i = 0; i < n; i++)
        if (sa[i] >= j) y[p++] = sa[i] - j;
      fill(all(ws), 0);
      for (int i = 0; i < n; i++) ws[x[i]]++;
      for (int i = 1; i < lim; i++) ws[i] += ws[i - 1];
      for (int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
      swap(x, y), p = 1, x[sa[0]] = 0;
      for (int i = 1; i < n; i++)
        a = sa[i - 1], b = sa[i],

        x[b] = (y[a] == y[b] && y[a + j] == y[b + j])
                 ? p - 1 : p++;

    }
    for (int i = 1; i < n; i++) rank[sa[i]] = i;
    for (int i = 0, j; i < n - 1; lcp[rank[i++]] = k)
      for (k &&k--, j = sa[rank[i] - 1];
           s[i + k] == s[j + k]; k++);
  }
};
```

7.3. Suffix Tree

```
struct SAM {
  static const int maxc = 26;    // char range
  static const int maxn = 10010; // string len
  struct Node {
    Node *green, *edge[maxc];
    int max_len, in, times;
  } *root, *last, reg[maxn * 2];
  int top;
  Node *get_node(int _max) {
    Node *re = &reg[top++];
    re->in = 0, re->times = 1;
    re->max_len = _max, re->green = 0;
    for (int i = 0; i < maxc; i++) re->edge[i] = 0;
    return re;
  }
  void insert(const char c) { // c in range [0, maxc)
    Node *p = last;
    last = get_node(p->max_len + 1);
    while (p && !p->edge[c])
      p->edge[c] = last, p = p->green;
    if (!p) last->green = root;
    else {
      Node *pot_green = p->edge[c];
      if ((pot_green->max_len) == (p->max_len + 1))
        last->green = pot_green;
      else {
        Node *wish = get_node(p->max_len + 1);
        wish->times = 0;
        while (p && p->edge[c] == pot_green)
          p->edge[c] = wish, p = p->green;
        for (int i = 0; i < maxc; i++)
          wish->edge[i] = pot_green->edge[i];
        wish->green = pot_green->green;
        pot_green->green = wish;
        last->green = wish;
      }
    }
  }
  Node *q[maxn * 2];
  int ql, qr;
  void get_times(Node *p) {
    ql = 0, qr = -1, reg[0].in = 1;
    for (int i = 1; i < top; i++) reg[i].green->in++;
    for (int i = 0; i < top; i++)
      if (!reg[i].in) q[++qr] = &reg[i];
    while (ql <= qr) {
      q[ql]->green->times += q[ql]->times;
      if (!(--q[ql]->green->in)) q[++qr] = q[ql]->green;
      ql++;
    }
  }
  void build(const string &s) {
    top = 0;
    root = last = get_node(0);
    for (char c : s) insert(c - 'a'); // change char id
    get_times(root);
  }
  // call build before solve
  int solve(const string &s) {
    Node *p = root;
    for (char c : s)
      if (!(p = p->edge[c - 'a'])) // change char id
        return 0;
    return p->times;
  }
};
```

7.4. Cocke-Younger-Kasami Algorithm

```


struct rule {
  // s -> xy
  // if y == -1, then s -> x (unit rule)
  int s, x, y, cost;
};
int state;
// state (id) for each letter (variable)
// lowercase letters are terminal symbols
map<char, int> rules;
vector<rule> cnf;
void init() {
  state = 0;
  rules.clear();
  cnf.clear();
}
// convert a cfg rule to cnf (but with unit rules) and add
// it
void add_to_cnf(char s, const string &p, int cost) {
  if (!rules.count(s)) rules[s] = state++;
  for (char c : p)
    if (!rules.count(c)) rules[c] = state++;
  if (p.size() == 1) {
    cnf.push_back({rules[s], rules[p[0]], -1, cost});
  } else {
    // length >= 3 -> split
    int left = rules[s];
    int sz = p.size();
    for (int i = 0; i < sz - 2; i++) {
      cnf.push_back({left, rules[p[i]], state, 0});
      left = state++;
    }
    cnf.push_back(
    {left, rules[p[sz - 2]], rules[p[sz - 1]], cost});
  }
}

constexpr int MAXN = 55;
vector<long long> dp[MAXN][MAXN];
// unit rules with negative costs can cause negative cycles
vector<bool> neg_INF[MAXN][MAXN];

void relax(int l, int r, rule c, long long cost,
           bool neg_c = 0) {
  if (!neg_INF[l][r][c.s] &&
      (neg_INF[l][r][c.x] || cost < dp[l][r][c.s])) {
    if (neg_c || neg_INF[l][r][c.x]) {
      dp[l][r][c.s] = 0;
      neg_INF[l][r][c.s] = true;
    } else {
      dp[l][r][c.s] = cost;
    }
  }
}
void bellman(int l, int r, int n) {
  for (int k = 1; k <= state; k++)
    for (rule c : cnf)
      if (c.y == -1)
        relax(l, r, c, dp[l][r][c.x] + c.cost, k == n);
}
void cyk(const string &s) {
  vector<int> tok;
  for (char c : s) tok.push_back(rules[c]);
  for (int i = 0; i < tok.size(); i++) {
    for (int j = 0; j < tok.size(); j++) {
      dp[i][j] = vector<long long>(state + 1, INT_MAX);
      neg_INF[i][j] = vector<bool>(state + 1, false);
    }
    dp[i][i][tok[i]] = 0;
    bellman(i, i, tok.size());
  }
  for (int r = 1; r < tok.size(); r++) {
    for (int l = r - 1; l >= 0; l--) {
      for (int k = l; k < r; k++)
        for (rule c : cnf)
          if (c.y != -1)
            relax(l, r, c,
                  dp[l][k][c.x] + dp[k + 1][r][c.y] +
                  c.cost);
      bellman(l, r, tok.size());
    }
  }
}

// usage example
int main() {
  init();
  add_to_cnf('S', "aSc", 1);
  add_to_cnf('S', "BBB", 1);
  add_to_cnf('S', "SB", 1);
  add_to_cnf('B', "b", 1);
  cyk("abbbbc");
  // dp[0][s.size() - 1][rules[start]] = min cost to
  // generate s
  cout << dp[0][5][rules['S']] << '\n'; // 7
  cyk("acbc");
  cout << dp[0][3][rules['S']] << '\n'; // INT_MAX
  add_to_cnf('S', "S", -1);
  cyk("abbbbc");
  cout << neg_INF[0][5][rules['S']] << '\n'; // 1
}
```

7.5. Z Value

```
int z[n];
void zval(string s) {
  // z[i] => longest common prefix of s and s[i:], i > 0
  int n = s.size();
  z[0] = 0;
  for (int b = 0, i = 1; i < n; i++) {
    if (z[b] + b <= i) z[i] = 0;
    else z[i] = min(z[i - b], z[b] + b - i);
    while (s[i + z[i]] == s[z[i]]) z[i]++;
    if (i + z[i] > b + z[b]) b = i;
  }
}
```

7.6. Minimum Rotation

```
int min_rotation(string s) {
  int a = 0, n = s.size();
  s += s;
  for (int b = 0; b < n; b++) {
    for (int k = 0; k < n; k++) {
      if (a + k == b || s[a + k] < s[b + k]) {
        b += max(0, k - 1);
        break;
      }
      if (s[a + k] > s[b + k]) {
        a = b;
        break;
      }
    }
  }
  return a;
}
```

7.7. Palindromic Tree

```


struct palindromic_tree {
  struct node {
    int next[26], fail, len;
    int cnt,
    num; // cnt: appear times, num: number of pal. suf.
    node(int l = 0) : fail(0), len(l), cnt(0), num(0) {
      for (int i = 0; i < 26; ++i) next[i] = 0;
    }
  };
  vector<node> St;
  vector<char> s;
  int last, n;
  palindromic_tree() : St(2), last(1), n(0) {
    St[0].fail = 1, St[1].len = -1, s.pb(-1);
  }
  inline void clear() {
    St.clear(), s.clear(), last = 1, n = 0;
    St.pb(0), St.pb(-1);
    St[0].fail = 1, s.pb(-1);
  }
  inline int get_fail(int x) {
    while (s[n - St[x].len - 1] != s[n]) x = St[x].fail;
    return x;
  }
  inline void add(int c) {
    s.push_back(c -= 'a'), ++n;
    int cur = get_fail(last);
    if (!St[cur].next[c]) {
      int now = SZ(St);
      St.pb(St[cur].len + 2);
      St[now].fail = St[get_fail(St[cur].fail)].next[c];
      St[cur].next[c] = now;
      St[now].num = St[St[now].fail].num + 1;
    }
    last = St[cur].next[c], ++St[last].cnt;
  }
  inline void count() { // counting cnt
    auto i = St.rbegin();
    for (; i != St.rend(); ++i) {
      St[i->fail].cnt += i->cnt;
    }
  }
  inline int size() { // The number of diff. pal.
```

```
    return SZ(St) - 2;
  }
};
```

8. UTILITY

```
import java.util.*;

public class Utility {
  public static void main(String[] args) {   }
  // If we want to case in which we want small l value and large r value such that
  // we do -L and +R the sort the arr on the basis of li + ri

  // --> We are asked to count the number of non-decreasing sequences of length
  // 2□ where each element is between 1 and n it is same as
  // stars and bars where there are 2m identical object and n boxes so the formula
  // for this is 2m + n - 1 C (n - 1 or 2m)

  // Swapping adjacent elements in a distinct array is basically trying to equate
  // two permutations using adjacent swaps. When is it possible? --> if the parity
  // of inversion in both arrays are same.

  // GCD contains the minimum powers of primes
  // LCM contains the maximum powers of primes

  /*
   * The formula (x+k)+(y+k)=(x+k)□(y+k) is equivalent to (x+k)&(y+k)=0, where &
   * denotes the bitwise AND operation.
   * It can be shown that such an non-negative integer k does not exist when x=y.
   * When x≠y, one can show that k=2n−max(x,y) is a possible answer, where 2n is a
   * power of 2 that is sufficiently large.
   *
   * Important tip : if we do the Xor and we allso take the Xor of the twp numebr
   * then the bit parity never changes
   * means : 1 ^ 1 -> 0 and 1 & 1 = 0. the bit remains the same at that bit.
   */

  // if ax + by = c then Let g = gcd(a, b) then there exists integers x, y such
  // that ax + by = g. Therefore c % g == 0, for the above conditions.

  // we need to find the value of x and y then the formula is (g = gcd(a, b))
  // (only one solution)
  // x => (c / g) * (a / g) ^ -1 * (mod b / g)
  // y => (c - ax) / g.

  /*
   * we have greed and we need to calcullate the sum of some x * y grid, 0 and 1
   * are there in the grid
   *
   */

  public static int lowerBound(List<Integer> list, int val) {
    int pos = Collections.binarySearch(list, val);
    return (pos >= 0) ? pos : -pos - 1; // First index >= val
  }

  public static int upperBound(List<Integer> list, int val) {
    int pos = Collections.binarySearch(list, val);
    return (pos >= 0) ? pos + 1 : -pos - 1; // First index > val
  }

  public static int floorIndex(List<Integer> list, int val) {
    int pos = Collections.binarySearch(list, val);
    return (pos >= 0) ? pos : -pos - 2; // Last index <= val
  }

  public static int lowerThanIndex(List<Integer> list, int val) {
    int pos = Collections.binarySearch(list, val);
    return (pos >= 0) ? pos - 1 : -pos - 2; // Last index < val
  }

  {
    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
      }
    }
  }

  // for each possible value x in the array, the minimum prefix len
  // in every prefix of length ≥ k, the value x appears at least on
  {
    int n = 100000;
    int[] a = new int[n + 1];
    int[] gap = new int[n + 1], last = new int[n + 1], ans = new in
    Arrays.fill(ans, -1);

    for (int i = 1; i <= n; i++) {
      int x = a[i];
      gap[x] = Math.max(gap[x], i - last[x]);
      last[x] = i;
    }
    // now we will calculate for each number from 1 to n, what will
    // of prefix. If lets say for num = 1 min prefix is 3 then for
    // have ans is 1 because we need to deal with the minimum value
    // subarray of the lenghr k from 1 to n.

    for (int x = 1; x <= n; x++) {
      gap[x] = Math.max(gap[x], n - last[x] + 1);
      // so for x the max gap is gap[x].
      for (int j = gap[x]; j <= n && ans[j] == -1; j++) {
        ans[j] = x;
      }
    }
  }

  // hashing function for the string.
  long computeHash(String s) {
    final int p = 31;
    final int m = (int) 1e9 + 9;
    long hashValue = 0;
    long pPower = 1;
    for (int i = 0; i < s.length(); i++) {
      char c = s.charAt(i);
      hashValue = (hashValue + (c - 'a' + 1) * pPower) % m;
      pPower = (pPower * p) % m;
    }
    return hashValue;
  }

  public static int countUniqueSubstrings(String s) {
    int n = s.length();
    final int p = 31;
    final int m = (int) 1e9 + 9;

    long[] pPow = new long[n];
    pPow[0] = 1;
    for (int i = 1; i < n; i++) {
      pPow[i] = (pPow[i - 1] * p) % m;
    }

    // Compute prefix hashes
    long[] h = new long[n + 1];
    for (int i = 0; i < n; i++) {
      h[i + 1] = (h[i] + (s.charAt(i) - 'a' + 1) * pPow[i]) % m;
    }

    int count = 0;
    for (int len = 1; len <= n; len++) {
      Set<Long> hashSet = new HashSet<>();
      for (int i = 0; i <= n - len; i++) {
        long curHash = (h[i + len] - h[i] + m) % m;
        curHash = (curHash * pPow[n - i - 1]) % m;
        hashSet.add(curHash);
      }
      count += hashSet.size();
    }

    return count;
  }

  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)
```

```java
        return true;
      if (!(obj instanceof Pair))
        return false;
      Pair pair = (Pair) obj;
      return pair.first == this.first && pair.second == this.second;
    }

    @Override
    public int hashCode() {
      return Objects.hash(first, second);
    }
  }

  // Funciton that Returns minimum swaps required to sort
  // arr[] in ascending 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;
      int 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;
  }
  private static int lowerBound(long[] a, int start, int end, long val) {
    int lo = start, hi = end, res = end + 1;
    while (lo <= hi) {
      int mid = lo + (hi - lo) / 2;
      if (a[mid] >= val) {
        res = mid;
        hi = mid - 1;
      } else {
        lo = mid + 1;
      }
    }
    return res;
  }
  static long nCr_(int n, int k) {
    if (k > n)
      return 0;
    long numerator = fact[n];
    long denominator = (fact[k] * fact[n - k]) % MOD;
    return (numerator * modInverse(denominator, MOD)) % MOD;
  }
  public static long nCr(int n, int r) {
    if (r > n)
      return 0;
    if (r == 0 || r == n)
      return 1;
    r = Math.min(r, n - r);
    long result = 1;
    for (int i = 0; i < r; i++) {
      result = (result * (n - i)) % MOD;
      result = (modDiv(result, (i + 1), MOD));
    }
    return result;
  }
  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 long binpow(long a, long b) {
    long res = 1;
    while (b > 0) {
      if ((b & 1) == 1)
        res = res * a;
      a = a * a;
      b >>= 1;
    }
    return res;
  }
  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] + derangemen
    }
  }
  private static void SPF() {
    int N = 100;
    int[] spf = new int[N + 1];
    for (int i = 1; i <= N; i++) {
      spf[i] = i;
    }
    for (int i = 2; i * i <= N; i++) {
      if (spf[i] == i) {
        // this is the prime?
        for (int j = i * i; j <= N; j += i) {
          if (spf[j] == j) {
            // this number is not touched ever.
            spf[j] = i;
          }
        }
      }
    }
  }
  private static void addAllPrimFact(int x, HashMap<Integer, Intege
    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);
    }
  }

  static boolean[] isPrime;
  static ArrayList<Integer> primes;
  public static void sieve(int n) {
    isPrime = new boolean[n + 1];
    primes = new ArrayList<>();
    Arrays.fill(isPrime, true);
    isPrime[0] = false;
    isPrime[1] = false;
    for (int i = 2; i * i <= n; i++) {
```

```java
    if (isPrime[i]) {
      for (int j = i * i; j <= n; j += i) {
        isPrime[j] = false;
      }
    }
  }
  for (int i = 2; i <= n; i++) {
    if (isPrime[i]) {
      primes.add(i);
    }
  }
}

// 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;
}
public static int countPrimes(int n) {
  final int S = 10000;
  int nsqrt = (int) Math.sqrt(n);
  List<Integer> primes = new ArrayList<>();
  boolean[] isPrime = new boolean[nsqrt + 1];
  Arrays.fill(isPrime, true);
  for (int i = 2; i <= nsqrt; i++) {
    if (isPrime[i]) {
      primes.add(i);
      for (int j = i * i; j <= nsqrt; j += i) {
        isPrime[j] = false;
      }
    }
  }
  int result = 0;
  boolean[] block = new boolean[S];
  for (int k = 0; k * S <= n; k++) {
    Arrays.fill(block, true);
    int start = k * S;
    for (int p : primes) {
      int startIdx = Math.max((start + p - 1) / p, p);
      int j = startIdx * p - start;
      for (; j < S; j += p) {
        block[j] = false;
      }
    }
    if (k == 0) {
      block[0] = block[1] = false;
    }
    for (int i = 0; i < S && start + i <= n; i++) {
      if (block[i]) {
        result++;
      }
    }
  }
  return result;
}
// to check in arr[i] the j- th bit set or not.
// if((arr[i]&(1<<j))!=0) {
// count++; this means the jth bit is set.increase count
// }
// int bit = (num >> i) & 1;

int flipBit(int n, int j) {
  return n ^ (1 << j);
} // note: if we add 2^(x-1) to num then num will not divisibe by that x again.

// mex calculate for the arr of permutation
// long mex = (n * (n + 1) / 2) - sum;

private static int computeXOR(int n) {
  if (n % 4 == 0)
    return n;
  if (n % 4 == 1)
    return 1;
  if (n % 4 == 2)
    return n + 1;
  return 0;
}

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++;
      }
    }
  }
}
private static int getPrime(int n) {
  while (n % 2 == 0)
    return 2;
  for (int i = 3; i <= Math.sqrt(n); i += 2) {
    while (n % i == 0)
      return i;
  }
  if (n > 2)
    return n;
  return n;
}
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);
}
static long nCk(int n, int k) {
  if (k > n || n < 0 || k < 0)
    return 0;
  return (((fact[n] * factInverse[k]) % mod)
```

```java
    * factInverse[n - k]) % mod;
  }
  static long combination(long n, long r, long[] fact, long[] ifact) {
    if (r > n || r < 0)
      return 0;
    return ((fact[(int) n] * ifact[(int) r])
     % MOD * ifact[(int) (n - r)] % MOD) % MOD;
  }

  // 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);
    }
  }

  // Method to generate the next lexicographical permutation
  public static boolean nextPermutation(char[] array) {
    int n = array.length;
    int i = n - 2;
    while (i >= 0 && array[i] >= array[i + 1]) {
      i--;
    }
    if (i < 0) {
      return false;
    }
    int j = n - 1;
    while (array[j] <= array[i]) {
      j--;
    }
    swap2(array, i, j);
    reverse2(array, i + 1, n - 1);
    return true;
  }
  private static long calculateDigitSum(int n) {
    // to calculate the sum (1 + 2 + .... )
    // (each digit it replaced by there sum of the digit).

    long sum = 0;
    int factor = 1;
    int leftOver = 0;

    // Process each digit position
    while (n > 0) {
      int digit = n % 10;
      int higher = n / 10;
      sum += higher * factor * 45; // Sum of all digits from 0 to 9 is 45
      sum += digit * (digit - 1) / 2 * factor; // Sum of digits within the current group
      sum += digit * leftOver; // Adjust for digits already processed
      leftOver += digit * factor; // Update leftover for next digit position
      factor *= 10;
      n /= 10;
    }
    return sum;
  }
  /*----------------------------------------------------------------------

  // TREES
  private static void dfs(int node, List<List<Integer>> edges, int parent, int[] subtreeSize) {
    // subtreeSize[x] = 1 + sum(subtreeSize[child])
    subtreeSize[node] = 1;
    for (int neighbour : edges.get(node)) {
      if (neighbour != parent) {
        dfs(neighbour, edges, node, subtreeSize);
        // subtreeSize of neighbour child is added.
        subtreeSize[node] += subtreeSize[neighbour];
      }
    }
    // once we move out of the dfs call, the subtreeSize of node is correctly
    // populated
  }

  private static void dfs2(int node, List<List<Integer>> edges, int parent, int[] level) {
```

```java
    if (parent == -1) {
      level[node] = 1;
    } else {
      level[node] = level[parent] + 1;
    }
    for (int neighbour : edges.get(parent))
      if (neighbour != parent)
        dfs(neighbour, edges, node, level);
  }

  /*-----------------------------------------------------------------
  // GRAPH
  static class Pair implements Comparable<Pair> {
    int node, weight;

    Pair(int node, int weight) {
      this.node = node;
      this.weight = weight;
    }

    public int compareTo(Pair other) {
      return this.weight - other.weight;
    }
  }

  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) {
    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];
        int v = edge[1];
        int 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][
        }
      }
    }

    for (int i = 0; i < n; i++) {
      if (dist[i][i] < 0) {
        // negative cycle
      }
    }
  }

  // TOPOSORT and all that stuff
```

```java
  // 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<>();
    // Return empty if cycle exists
  }
}

// MST using DSU (Krushkal ALgorythm)

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