ICPC - UCSD Team 2 JAVA Reference Sheet

Composed and implemented by Kevin Chan (t-sekai)

Content:

Segment Tree

Initiation

```
int x = (int) (Math.ceil(Math.log(N) / Math.log(2)));
int size = 2 * (int) Math.pow(2, x) - 1;
// buildST(0,N-1,0,input);
// updateST(pos,0,N-1,0,val);
// queryST(qLow, qHigh, 0, N-1, 0);
```

Lazy Propagation

```
private static int segTree[];
private static int lazy[];
private static void buildST(int low, int high, int pos, int val[]) {
      if(low == high) {
            segTree[pos] = input[low];
            return;
      }
      int mid = (low+high)/2;
      buildST(low,mid,2*pos+1);
      buildST(mid+1,high,2*pos+2);
      segTree[pos] = segTree[2*pos+1]+segTree[2*pos+2];
}
private static void updateST(int qLow, int qHigh, int low, int high,
int pos, int val) {
      //lazy propagation
      if(lazy[pos] != 0) {
            segTree[pos] += (high-low+1)*lazy[pos];
            if(low != high) {
                  lazy[pos*2 + 1]  += lazy[pos];
                      lazy[pos*2 + 2]  += lazy[pos];
            }
            lazy[pos] = 0;
      }
      //no coverage
      if(low > high || low > qHigh || high < qLow) {
            return;
      }
      //full coverage
      if(high <= qHigh && low >= qLow) {
            segTree[pos] += (high-low+1)*val;
            if (high != low) {
                  lazy[pos*2 + 1] += val;
                  lazy[pos*2 + 2] += val;
            }
            return;
      }
      int mid = (low+high)/2;
      updateST(qLow,qHigh, low, mid, 2*pos+1, val);
      updateST(qLow,qHigh, mid+1, high, 2*pos+2, val);
      segTree[pos] = segTree[2*pos+1] + segTree[2*pos+2];
}
```

```
private static int queryST(int qLow, int qHigh, int low, int high, int
pos) {
      //lazy propagation
      if (lazy[pos] != 0)  {
            segTree[pos] = (high-low+1)*lazy[pos];
            if (high != low) {
                  lazy[pos*2 + 1] += lazy[pos];
                  lazy[pos*2 + 2] += lazy[pos];
            }
            lazy[pos] = 0;
      }
      if(qLow <= low && qHigh >= high) { //total overlap
            return segTree[pos];
      }
      if(qLow > high || qHigh < low) { //no overlap
            return 0;
      }
      //partial overlap
      int mid = (low+high)/2;
      return queryST(qLow,qHigh,low,mid,2*pos+1)+
                        queryST(qLow,qHigh,mid+1,high,2*pos+2);
}
```

Point Update

```
private static int segTree[];
private static void buildST(int low, int high, int pos, int input[]) {
      if(low == high) {
            segTree[pos] = input[low];
            return;
      }
      int mid = (low+high)/2;
      buildST(low,mid,2*pos+1,input);
      buildST(mid+1,high,2*pos+2,input);
      segTree[pos] = segTree[2*pos+1] + segTree[2*pos+2]);
}
private static void updateST(int point, int low, int high, int pos,
int val) {
      if(low > high || point > high || point < low) return; //no
overlap
      if(point == low && point == high) { //total overlap
            segTree[pos] = val;
            return;
      }
      int mid = (low+high)/2; //partial overlap
```

```
      updateST(point,low,mid,2*pos+1,val);
      updateST(point,mid+1,high,2*pos+2,val);
      segTree[pos] = segTree[2*pos+1]+segTree[2*pos+2];
}
private static int queryST(int qLow, int qHigh, int low, int high, int
pos) {
      if(qLow > high || qHigh < low) return 0; //no coverage
      if(qLow <= low && qHigh >= high) return segTree[pos]; //total
overlap
      int mid = (low+high)/2;//partial overlap
      return queryST(qLow,qHigh,low,mid,2*pos+1) +
                        queryST(qLow,qHigh,mid+1,high,2*pos+2);
}
```

Sparse Table

```
private static int [][]lookup;
private static void buildSparseTable(int arr[], int n){
      for (int i = 0; i < n; i++)
            lookup[i][0] = arr[i];
      for (int j = 1; (1 << j) <= n; j++) {
            for (int i = 0; (i + (1 << j) - 1) < n; i++) {
                  if (lookup[i][j - 1] <
                                    lookup[i + (1 << (j - 1))][j - 1])
                        lookup[i][j] = lookup[i][j - 1];
                  else
                      lookup[i][j] = lookup[i + (1 << (j - 1))][j - 1];
             }
      }
}
// Returns minimum of arr[L..R]
private static int query(int L, int R){
    int j = (int)Math.log(R - L + 1);
    if (lookup[L][j] <= lookup[R - (1 << j) + 1][j])
        return lookup[L][j];

    else
        return lookup[R - (1 << j) + 1][j];
}
```

Lowest Common Ancestor (LCA)

```
private static int lca[][];
private static int depth[];
```

```
// need to initialize the first layer of parent nodes before running
the initLCA() function
private static void initLCA(){
      for(int i = 1; i < 17; ++i) {
            for(int j = 0; j < N; ++j) {
                  lca[i][j] = lca[i-1][lca[i-1][j]];
            }
      }
}
private static int LCA(int u, int v) {
      if(depth[u]<depth[v]) { //swap
            int t = u; u = v; v = t;
      }
      for(int i = 16; i >= 0; --i) {
            if(depth[u] - (1<<i) >= depth[v]) {
                  u = lca[i][u];
            }
      }
      for(int i = 16; i >= 0; --i) {
            if(lca[i][u] != lca[i][v]) {
                  u = lca[i][u]; v = lca[i][v];
            }
      }
      if(u!=v) {
            u = lca[0][u]; v = lca[0][v];
      }
      return u;
}
```

Heavy Light Decomposition (HLD)

```
import java.io.*;
import java.util.*;
public class HLDalgorithm {
      private static StreamTokenizer st;
      private static int nextInt() throws IOException{
            st.nextToken();
            return (int)st.nval;
      }
      public static void main(String[] args) throws IOException{
            st = new StreamTokenizer(new BufferedReader(new
InputStreamReader(System.in)));
            int N = nextInt();
            HLD hld = new HLD(N);
```

```
            for(int i = 0; i < N-1; ++i) hld.addEdge(nextInt()-1,
nextInt()-1);
            hld.initTree();
            hld.update(1, 5);
            System.out.println(hld.query(0,2));
            hld.update(2, 4);
            System.out.println(hld.query(2,2));
      }
      private static class HLD{
            private static int N; //number of node on the tree
            private static ArrayList<Integer> edge[]; //edge list
            private static int[] depth, size, val; //tree values
            @SuppressWarnings("unchecked")
            public HLD(int numNode) { //initial lists
                  N = numNode;
                  depth = new int[N]; size = new int[N];
                  chainHead = new int[N];
                  nodeToSegTree = new int[N];
                  segTreeToNode = new int[N];
                  Arrays.fill(chainHead, -1);
                  edge = new ArrayList[N];
                  for(int i = 0; i < N; ++i)
                        edge[i] = new ArrayList<>();
                  lca = new int[17][N];
                  val = new int[N];
                  int x = (int) (Math.ceil(Math.log(N) / Math.log(2)));
                  int size = 2 * (int) Math.pow(2, x) - 1;
                  segTree = new int[size];
            }
            public void addEdge(int u, int v) { //add bidirectional edge
                  edge[u].add(v); edge[v].add(u);
            }
            public void initTree() { //no previous value
                  DFS(0,-1);
                  initLCA();
                  hld(0,0);
            }
            /*
            public void initTree(int[] value) { //have previous value
                  DFS(0,-1);
                  initLCA();
                  hld(0,0);
                  val = value;
                  buildST(0, N-1, 0);
            }
```

```
            */
            public int query(int u, int v) {
                  int anc = LCA(u,v);
                  return Math.max(val[anc],Math.max(pathQuery(u,anc),
pathQuery(v,anc)));
            }
            public void update(int idx, int value) {
                  val[idx] = value;
                  updateST(nodeToSegTree[idx],0,N-1,0,value);
            }
            private static int pathQuery(int child, int par) {
                  int ret = 0;
                  while(child != par) {
                        if(chainHead[child] == child) {
                              //light edge
                              ret = Math.max(ret,val[child]);
                              child = lca[0][child];
                        }else if(depth[chainHead[child]] > depth[par]){
                              ret = Math.max(ret,
queryST(nodeToSegTree[chainHead[child]],nodeToSegTree[child],0,N-1,0))
;
                              child = lca[0][chainHead[child]];
                        }else {
                              ret =
Math.max(ret,queryST(nodeToSegTree[par]+1,nodeToSegTree[child],0,N-1,0
));
                              break;
                        }
                  }
                  return ret;
            }
            private static int lca[][];
            private static void initLCA(){
                  for(int i = 1; i < 17; ++i)
                      for(int j = 0; j < N; ++j)
                        lca[i][j] = lca[i-1][lca[i-1][j]];
            }
            private static int LCA(int u, int v) {
                  if(depth[u]<depth[v]) //swap
                        int t = u; u = v; v = t;
                  for(int i = 16; i >= 0; --i)
                        if(depth[u] - (1<<i) >= depth[v])
                              u = lca[i][u];
                  for(int i = 16; i >= 0; --i)
                        if(lca[i][u] != lca[i][v]){
```

```
                        u = lca[i][u]; v = lca[i][v];
                    }
                if(u!=v) {
                    u = lca[0][u]; v = lca[0][v];
                }
                return u;
            }
            private static void DFS(int curr, int prev){
                ++size[curr];
                for(int i : edge[curr]) {
                    if(i == prev)continue;
                    lca[0][i] = curr;
                    depth[i] = depth[curr]+1;
                    DFS(i,curr);
                    size[curr]+=size[i];
                }
            }
            private static int segTreeIdxCount = 0;
            private static int[] chainHead,nodeToSegTree,segTreeToNode;
            private static void hld(int curr, int top) {
                segTreeToNode[segTreeIdxCount] = curr;
                nodeToSegTree[curr] = segTreeIdxCount;
                ++segTreeIdxCount;
                chainHead[curr]=top;
                int schild = -1, maxSize = -1;
                for(int i : edge[curr])
                    if(i != lca[0][curr] && size[i] > maxSize) {
                        maxSize = size[i];
                        schild = i;
                    }
               if(schild >= 0) hld(schild,top);
               for(int i : edge[curr])
                   if(i != lca[0][curr] && i != schild)
                       hld(i,i);
            }
            private static int segTree[];
            /*
            private static void buildST(int low, int high, int pos) {
//segment tree (max query)
                if(low == high) {
                    segTree[pos] = val[segTreeToNode[low]];
                    return;
                }
                int mid = (low+high)/2;
                buildST(low,mid,2*pos+1); buildST(mid+1,high,2*pos+2);
```

```
            segTree[pos] = Math.max(segTree[2*pos+1],
                                    segTree[2*pos+2]);
        }
        */
        private static void updateST(int point, int low, int high,
                            int pos, int val) { //point update
            if(low > high || point > high || point < low)
                return; //no overlap
            if(point == low && point == high) { //total overlap
                segTree[pos] = val;
                return;
            }
            int mid = (low+high)/2; //partial overlap
            updateST(point,low,mid,2*pos+1,val);
            updateST(point,mid+1,high,2*pos+2,val);
            segTree[pos] = Math.max(segTree[2*pos+1],
                                     segTree[2*pos+2]);
        }
        private static int queryST(int qLow, int qHigh, int low,
                            int high, int pos) { //range query
            if(qLow > high || qHigh < low) //no coverage
                return 0;
            if(qLow <= low && qHigh >= high) //total overlap
                        return segTree[pos];
            int mid = (low+high)/2;//partial overlap
            return Math.max(queryST(qLow,qHigh,low,mid,2*pos+1),
                        queryST(qLow,qHigh,mid+1,high,2*pos+2));
        }
    }
}
```

Fenwick Tree (Binary Indexed Tree)

```
private static int BIT[];
private static void update(int i, int x) {
    while(i <= BIT.length) {
        BIT[i-1] += x;
        i += i & -i;
    }
}
private static int query(int i) {
    int ret = 0;
    while(i > 0) {
        ret+=BIT[i-1];
```

```
            i -= i & -i;
      }
      return ret;
}
```

LCM & GCD

```
private static int[] buildLCM(int W) {
      int LCM[] = new int[1<<W]; //bitmask for 0 to W-1
      for(int i = 1; i <= W; ++i)LCM[1<<(i-1)] = i;
      for(int i = 1; i < 1<<W; ++i) {
            for(int j = 1; j <= W; ++j) {
                  if((i&(1<<(j-1)))!=0)continue;
                  LCM[i+(1<<(j-1))] =
                              (int)((long)LCM[i]*j/gcd(LCM[i],j));
            }
      }
      return LCM;
}
private static int gcd(int a, int b) { //logn
      if(b == 0)return a;
      return gcd(b,a%b);
}
```

Union Find Set (Disjoin Set Union)

```
private static class UnionFindSet{
      int[]parent;
      int[]size;
      public UnionFindSet(int n) {
            parent = new int[n];
            size = new int[n];
            for(int i = 0; i < n; i ++) {
                  parent[i] = i;
                  size[i] = 1;
            }
      }
      public int find(int x) {
            if(parent[x] == x)return x;
            return parent[x] = find(parent[x]);
      }
      public void merge(int x, int y) {
            int px = find(x), py = find(y);
            if(size[px] < size[py]) {
```

```
                parent[px] = py;
                size[py] += size[px];
            }else{
                parent[py] = px;
                size[px] += size[py];
            }
        }
}
```

Convex Hull

```
private static class Point implements Comparable<Point>{
    int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public int compareTo(Point o) {
        return x - o.x;
    }
}
private static boolean CCW(Point a, Point b, Point c) {
    return ((b.x-a.x)*(c.y-a.y)-(b.y-a.y)*(c.x-a.x)) >= 0;
}
private static ArrayList<Point> ConvexHull(Point[] pt,boolean up){
    ArrayList<Point> conHull = new ArrayList<>();
    if(up) {
        for(int i = 0; i < pt.length; ++i) {
            conHull.add(pt[i]);
            while(conHull.size()>2 &&
CCW(conHull.get(conHull.size()-3),conHull.get(conHull.size()-2),conHul
l.get(conHull.size()-1))) {
                conHull.remove(conHull.size()-2);
            }
        }
    }else {
        for(int i = 0; i < pt.length; ++i) {
            conHull.add(pt[i]);
            while(conHull.size()>2 &&
!CCW(conHull.get(conHull.size()-3),conHull.get(conHull.size()-2),conHu
ll.get(conHull.size()-1))) {
                conHull.remove(conHull.size()-2);
            }
```

```
            }
      }
      return conHull;
}

int N = nextInt();
Point pt[] = new Point[N];
for(int i = 0; i < N; i++) pt[i] = new Point(nextInt(),nextInt());
Arrays.sort(pt);
ArrayList<Point> up = ConvexHull(pt,true);
ArrayList<Point> down = ConvexHull(pt,false);
```

Knuth-Morris-Pratt (KMP) Algorithm

```
private static void KMPSearch(String txt, String pat) {
      int M = pat.length();
      int N = txt.length();
      int lps[] = new int[M];
      int j = 0;
      computeLPSArray(pat, M, lps);
      int i = 0;
      while (i < N) {
            if (pat.charAt(j) == txt.charAt(i)) {
                  ++j;
                  ++i;
            }
            if (j == M) {
                  pw.println("Found pattern "
                                    + "at index " + (i - j));
                  j = lps[j - 1];
            } else if (i < N && pat.charAt(j) != txt.charAt(i)) {
                  if (j != 0)
                        j = lps[j - 1];
                  else ++i;
            }
      }
}
private static void computeLPSArray(String pat, int M, int lps[]) {
      int len = 0;
      int i = 1;
      lps[0] = 0;
      while (i < M) {
            if (pat.charAt(i) == pat.charAt(len)) {
                  ++len;
```

```
                lps[i] = len;
                ++i;
            } else {
                if (len != 0) {
                    len = lps[len - 1];
                } else {
                    lps[i] = len;
                    ++i;
                }
            }
        }
}
```

Dijkstra's Algorithm

```
int dist[] = new int[P+2];
for(int i = 1; i < P+2; ++i)dist[i] = Integer.MAX_VALUE;
PriorityQueue<Edge> pq = new PriorityQueue<>();
pq.add(new Edge(0,0));
while(!pq.isEmpty()) {
    Edge e = pq.poll();
    if(dist[e.v]!=e.w)continue;
    for(Edge i : edge[e.v]) {
        if(i.w+e.w < dist[i.v]) {
            dist[i.v] = i.w+e.w;
            pq.add(new Edge(i.v,i.w+e.w));
        }
    }
}
private static class Edge implements Comparable<Edge>{
    int v, w;
    public Edge(int v, int w) {
        this.v = v; this.w = w;
    }
    @Override
    public int compareTo(Edge o) {
        return w-o.w;
    }
}
```

Prim's Algorithm

```
List<Edge> primMST(int src) {
        PriorityQueue<Edge> pq = new PriorityQueue<>();
```

```
        List<Edge> mst = new ArrayList<>();
        boolean[] visited = new boolean[V];
        visited[src] = true;
        for (Edge e : adj.get(src)) pq.add(e);
        while (!pq.isEmpty()) {
            Edge e = pq.poll();
            if (visited[e.to]) continue;
            visited[e.to] = true;
            mst.add(e);
            for (Edge f : adj.get(e.to))
                if (!visited[f.to])
                    pq.add(f);
        }
        return mst;
    }
```

Bellman Ford's Algorithm

```
private static class Edge{
      int u, v, w;
      public Edge(int u, int v, int w) {
            this.u = u; this.v = v; this.w = w;
      }
}
private static boolean BellmanFord(int N, ArrayList<Edge>edges) {
      int dist[] = new int[N];
      for(int i = 1; i < N; ++i) dist[i] = Integer.MAX_VALUE;
      for(int i = 0; i < N-1; ++i)
            for(Edge e : edges)
                  if(dist[e.u]!=Integer.MAX_VALUE && dist[e.u] + e.w
                                                      < dist[e.v])
                        dist[e.v] = dist[e.u] + e.w;
      for(Edge e : edges)
            if(dist[e.u] + e.w < dist[e.v])
                  return false;
      return true;
}
```

Floyd Washall

```
int dist[][] = new int[M][M];
//remember to set up dist from input edges first
for(int k = 0; k < M; ++k)
      for(int i = 0; i < M; ++i)
```

```
            for(int j = 0; j < M; ++j)
                  if (dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j];
```

2-SAT

```
private static ArrayList<Integer> edge[];
public static void main(String[] args) throws IOException{
      N = nextInt(); int M = nextInt();
      edge = new ArrayList[N*2]; //i = true i+N = false
      HashSet<Integer> hs[] = new HashSet[N*2]; //remove duplicates
      for(int i = 0; i < N*2; ++i)edge[i] = new ArrayList<>();
      for(int i = 0; i < N*2; ++i)hs[i] = new HashSet<>();
      for(int i = 0; i < M; ++i) {
            int B = nextInt()-1; boolean vB = read().equals("Y");
            int C = nextInt()-1; boolean vC = read().equals("Y");
            //set up graph
            if(!hs[B+(vB?N:0)].contains(C+(vC?0:N))) {
                  edge[B+(vB?N:0)].add(C+(vC?0:N));
                  hs[B+(vB?N:0)].add(C+(vC?0:N));
            }
            if(!hs[C+(vC?N:0)].contains(B+(vB?0:N))) {
                  edge[C+(vC?N:0)].add(B+(vB?0:N));
                  hs[C+(vC?N:0)].add(B+(vB?0:N));
            }
      }
      //2sat
      char ans[] = new char[N];
      for(int i = 0; i < N; ++i) {
            boolean a = check(i), b = check(i+N);
            if(a&&b)ans[i] = '?';
            else if(a)ans[i] = 'Y';
            else if(b)ans[i] = 'N';
            else {
                  pw.println("IMPOSSIBLE");
                  return;
            }
      }
      for(int i = 0; i < N; ++i)pw.println(ans[i]);
}
private static int N;
private static boolean visited[];
private static boolean check(int u) {
      visited = new boolean[N*2];
```

```
	DFS(u);
	for(int i = 0; i < N; ++i)
		if(visited[i]&&visited[i+N])
			return false;
	return true;
}
private static void DFS(int u) {
	if(visited[u])return;
	visited[u] = true;
	for(int i : edge[u]) if(!visited[i])DFS(i);
}
```

Strongly Connected Components

```
// it works, don't question :(
public class StronglyConnectedComponents {
	public static void main(String[] args) throws IOException{
		int N = nextInt();
		to = new int[N]; color = new int[N]; in = new ArrayList[N];
		for(int i = 0; i < N; ++i) in[i] = new ArrayList<>();
		for(int i = 0; i < N; ++i) {
			to[i] = nextInt()-1;
			in[to[i]].add(i);
		}
		visited = new boolean[N]; stk = new Stack<>();
		for(int i = 0; i < N; ++i) if(!visited[i]) DFS1(i);
		while(!stk.isEmpty()) {
			int e = stk.pop();
			if(color[e]==0) {
				++count;
				DFS2(e);
			}
		}
		System.out.println(count);
	}
	private static ArrayList<Integer> in[];
	private static int to[], color[], count;
	private static boolean[] visited;
	private static Stack<Integer> stk;
	private static void DFS1(int u) {
		visited[u] = true;
		if(!visited[to[u]])DFS1(to[u]);
		stk.add(u);
	}
```

```
	private static void DFS2(int u) {
		color[u] = count;
		for(int i : in[u]) if(color[i]==0)DFS2(i);
	}
}
```

Articulation Point

```
//it works, don't question :(
public class ArticulationPoints {
	private static ArrayList<Integer> edge[];
	public static void main(String[] args) throws IOException{
		int N = nextInt(), M = nextInt();
		edge = new ArrayList[N];
		for(int i = 0; i < N; ++i) edge[i] = new ArrayList<>();
		for(int i = 0; i < M; ++i) {
			int u = nextInt()-1, v = nextInt()-1;
			edge[u].add(v); edge[v].add(u);
		}
		isAP = new boolean[N]; visited = new boolean[N];
		num = new int[N]; low = new int[N];
		for(int i = 0; i < N; ++i) if(!visited[i])DFS(i,-1);
		int ans = 0;
		for(int i = 0; i < N; ++i) ans += isAP[i]?1:0;
		pw.println(ans);
	}
	private static boolean[] isAP, visited;
	private static int timer,num[], low[];
	private static void DFS(int v, int p) {
		visited[v] = true;
		num[v] = low[v] = timer++;
		int children = 0;
		for(int i : edge[v]) {
			if(i == p)continue;
			if(visited[i])low[v] = Math.min(low[v], num[i]);
			else {
				DFS(i,v);
				low[v] = Math.min(low[v], low[i]);
				if(low[i]>=num[v]&&p!=-1) isAP[v] = true;
				++children;
			}
		}
		if(p==-1&&children>1) isAP[v] = true;
	}
```

```
}
```

Bridges

```
public class Bridge {
     private static int time;
     private static ArrayList <Integer> edge[];
     private static ArrayList <Edge> bridges;
     private static int[] disc, low, parent;
     public static void findBridges(int n) {
        time = 0;
        low = new int[n];
        disc = new int[n];
        parent = new int[n];
        Arrays.fill(parent, -1);
        bridges = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            if (disc[i] == 0) {
                dfs(i);
            }
        }
    }
    private static void dfs(int u) {
        disc[u] = low[u] = ++time;
        for (int v : edge[u]) {
            if (disc[v] == 0) {
                parent[v] = u;
                dfs(v);
                low[u] = Math.min(low[u], low[v]);
                if (low[v] > disc[u]) {
                    bridges.add(new Edge(u,v));
                }
            } else if (v != parent[u]) {
                low[u] = Math.min(low[u], disc[v]);
            }
        }
    }
     public static void main(String[] args){
           int n = nextInt(), m = nextInt();
           edge = new ArrayList[n];
           for(int i = 0; i < n; ++i) edge[i] = new ArrayList<>();
           for(int i = 0; i < m; ++i) {
                int u = nextInt(), v = nextInt();
                edge[u].add(v);
```

```
                edge[v].add(u);
            }
            findBridges(n);
            for(Edge e : bridges) {
                System.out.println(e.u + " " + e.v);
            }
        }
        private static class Edge{
            int u, v;
            public Edge(int u, int v) {
                this.u = u; this.v = v;
            }
        }
}
```

Topological Sorting

```
public class TopologicalSort {
    int n; // number of vertices
    List<List<Integer>> adj; // adjacency list of graph
    boolean[] visited;
    List<Integer> ans;
    public void dfs(int v) {
        visited[v] = true;
        for (int u : adj.get(v)) {
            if (!visited[u])
                dfs(u);
        }
        ans.add(v);
    }
    public void topologicalSort() {
        visited = new boolean[n];
        Arrays.fill(visited, false);
        ans = new ArrayList<>();
        for (int i = 0; i < n; ++i) {
            if (!visited[i])
                dfs(i);
        }
        Collections.reverse(ans);
    }
}
```

Manacher's Algorithm

```
public static String longestPalindrome(String s) {
      int[][] palindrome = manacher(s.toCharArray());
      int max = 0, odd = 0, idx = 0;
      for (int i = 0; i < 2; ++i)
            for (int j = 0; j < s.length(); ++j)
                  if (palindrome[i][j] >= max) {
                        max = Math.max(max, palindrome[i][j]);
                        odd = i;
                        idx = j;
                  }
      return s.substring(idx - max, idx + max + odd);
}
private static int[][] manacher(char s[]) {
      int n = s.length;
      int[][] p = new int[2][n];
      for (int z = 0; z < 2; ++z) {
            for (int i = 0, l = 0, r = 0; i < n; ++i) {
                  int t = r - i + (z ^ 1);
                  if (i < r)
                        p[z][i] = Math.min(t, p[z][l + t]);
                  int L = i - p[z][i], R = i + p[z][i] - (z ^ 1);
                  while (L >= 1 && R + 1 < n && s[L - 1] == s[R + 1]) {
                        ++p[z][i];
                        --L;
                        ++R;
                  }
                  if (R > r) {
                        l = L;
                        r = R;
                  }
            }
      }
      return p;
}
```

Fast Exponential Mod

```
public static long fastExpMod(long base, long exp, long mod) {
    if (mod == 1) return 0;
    long result = 1;
    base %= mod;
    while (exp > 0) {
```

```
        if (exp % 2 == 1)
            result = (result * base) % mod;
        exp >>= 1;
        base = (base * base) % mod;
    }
    return result;
}
```

Mod Inverse

```
public static long modInverse(long a, long mod) {
    long m0 = mod;
    long y = 0, x = 1;
    if (mod == 1)
        return 0;
    while (a > 1) {
        long q = a / mod;
        long t = mod;
        mod = a % mod;
        a = t;
        t = y;
        y = x - q * y;
        x = t;
    }
    if (x < 0)
        x += m0;
    return x;
}
//Example:
long a = 7;
long b = 5;
long mod = 11;
long inverse = modInverse(b, mod);
long result = (a * inverse) % mod;
if (result < 0) result += mod; // Ensure that the result is positive
System.out.println(result); // Output: 3
```

Java Comparator and HashCode

```
//Example for Comparator (sometimes you need to compare both x and y,
but you already compared x using compareTo in the class):
private static class compareY implements Comparator<Point>{
     @Override
     public int compare(Point o1, Point o2) {
```

```
            if(o1.y == o2.y) return o1.x-o2.x;
            return o1.y-o2.y;
      }
}
//TreeSet<Point> ts = new TreeSet<>(new compareY());
//PriorityQueue<Point> pq = new PriorityQueue<>(new compareY());
//Example for HashCode:
private static int N;
private static class Combo{
      int x, y, z;
      public Combo(int x, int y, int z) {
            this.x = x; this.y = y; this.z = z;
      }
      public int hashCode() {
           return x*N*N+y*N+z; //can mod 1e9+7 if necessary
      }
      @Override
      public boolean equals(Object obj) {
            if(this == obj) return true;
            if(obj == null || obj.getClass()!= this.getClass())
                  return false;
            Combo p = (Combo) obj;
            return (p.x == this.x  && p.y == this.y && p.z == this.z);
      }
}
```

Java Arrays/Collections Binary Search

```
int ans = Arrays.binarySearch(arr, val);
if(ans < 0)ans = -(ans+2); //largest element index that is >= val
if(ans == -1)System.out.println("not found");

if (ans < 0) ans = -(ans + 1); //smallest element index that is <= val
if (ans == arr.length) System.out.println("not found");
```

Dinic's Algorithm (Max Flow Min Cut)

```
class Dinic {
    static class Edge {
        int to, rev;
        long flow, cap;
        public Edge(int to, int rev, long cap) {
            this.to = to; this.rev = rev; this.cap = cap;
        }
```

```
    }
    int n;
    List<Edge>[] graph;
    int[] dist;
    int[] q;
    public Dinic(int n) {
        this.n = n;
        graph = new List[n];
        for (int i = 0; i < n; i++) graph[i] = new ArrayList<>();
        dist = new int[n];
        q = new int[n];
    }
    public void addEdge(int from, int to, long cap) {
        graph[from].add(new Edge(to, graph[to].size(), cap));
        graph[to].add(new Edge(from, graph[from].size() - 1, 0));
    }
    boolean bfs(int source, int sink) {
        Arrays.fill(dist, -1);
        int head = 0, tail = 0;
        q[tail++] = source;
        dist[source] = 0;
        while (head < tail) {
            int u = q[head++];
            for (Edge e : graph[u]) {
                if (dist[e.to] == -1 && e.flow < e.cap) {
                    dist[e.to] = dist[u] + 1;
                    q[tail++] = e.to;
                }
            }
        }
        return dist[sink] != -1;
    }
    long dfs(int u, int sink, long flow) {
        if (u == sink || flow == 0) return flow;
        for (Edge e : graph[u]) {
            if (e.flow < e.cap && dist[e.to] == dist[u] + 1) {
                long df = dfs(e.to, sink, Math.min(flow, e.cap -
e.flow));
                if (df > 0) {
                    e.flow += df;
                    graph[e.to].get(e.rev).flow -= df;
                    return df;
                }
            }
        }
```

```
        return 0;
    }
    public long maxFlow(int source, int sink) {
        long flow = 0;
        while (bfs(source, sink)) {
            while (true) {
                long df = dfs(source, sink, Long.MAX_VALUE);
                if (df == 0) break;
                flow += df;
            }
        }
        return flow;
    }
    public List<Edge> minCut(int source, int sink) {
              maxFlow(source, sink);
              bfs(source, sink);
              List<Edge> cut = new ArrayList<>();
              for (int u = 0; u < n; u++)
                  for (Edge e : graph[u])
                      if (dist[u] != -1 && dist[e.to] == -1
                                                        && e.cap > 0)
                          cut.add(e);
              return cut;
          }
}
//Example:
Dinic dinic = new Dinic(4);
dinic.addEdge(0, 1, 3); dinic.addEdge(0, 2, 2); dinic.addEdge(1, 2,
1); dinic.addEdge(1, 3, 2);dinic.addEdge(2, 3, 3);
System.out.println(dinic.maxFlow(0, 3));  // Output: 5
```