Contents



# 1. Code Templates

**1.1. Java Template.** A Java template.

```
import java.util.*;
import java.io.*;
public class A {
    void solve(BufferedReader in) throws Exception {

    }
    int toInt(String s) {return Integer.parseInt(s);}
    int[] toInts(String s) {
        String[] a = s.split(" ");
        int[] o = new int[a.length];
        for(int i = 0; i<a.length; i++)
            o[i] = toInt(a[i]);
        return o;
    }
    void e(Object o) {
        System.err.println(o);
    }
    public static void main(String[] args)
    throws Exception {
        BufferedReader in = new BufferedReader
            (new InputStreamReader(System.in));
        (new A()).solve(in);
    }
}
```

**1.2. Python Template.** A Python template

```
import sys
line = sys.stdin.read()
```

**1.3. C++ Template.** A C++ template

```
#include<iostream>
#include<vector>
#include<queue>
#include<unordered_set>
using namespace std;
int main() {
  vector<string> in;
  string line;
  while(getline(cin, line)) in.push_back(line);
}
```

**1.4. Fast IO Java.** Kattio with easier names

```
import java.util.StringTokenizer;
import java.io.*;
class Sc {
  public Sc(InputStream i) {
    r = new BufferedReader(new InputStreamReader(i));
  }
  public boolean hasM() {
    return peekToken() != null;
  }
  public int nI() {
    return Integer.parseInt(nextToken());
  }
  public double nD() {
    return Double.parseDouble(nextToken());
  }
  public long nL() {
    return Long.parseLong(nextToken());
  }
  public String n() {
    return nextToken();
  }
  private BufferedReader r;
  private String line;
  private StringTokenizer st;
  private String token;
  private String peekToken() {
    if (token == null)
      try {
        while (st == null || !st.hasMoreTokens()) {
          line = r.readLine();
          if (line == null) return null;
          st = new StringTokenizer(line);
        }
        token = st.nextToken();
      } catch (IOException e) { }
    return token;
  }
  private String nextToken() {
    String ans = peekToken();
    token = null;
    return ans;
  }
}
```

# 2. Data Structures

**2.1. Binary Indexed Tree.** Also called a fenwick tree. Builds in $\mathcal{O}(n \log n)$ from an array. Querry sum from 0 to i in $\mathcal{O}(\log n)$ and updates an element in $\mathcal{O}(\log n)$.

```
private static class BIT {
  long[] data;
  public BIT(int size) {
    data = new long[size+1];
  }
  public void update(int i, int delta) {
    while(i< data.length) {
      data[i] += delta;
      i += i&-i; // Integer.lowestOneBit(i);
    }
  }
```

```
  public long sum(int i) {
    long sum = 0;
    while(i>0) {
      sum += data[i];
      i -= i&-i;
    }
    return sum;
  }
}
```

2.2. **Segment Tree.** More general than a fenwick tree. Can adapt other operations than sum, e.g. min and max.

```
private static class ST {
  int li, ri;
  int sum; //change to max/min
  ST lN;
  ST rN;
}
static ST makeSgmTree(int[] A, int l, int r) {
  if(l == r) {
    ST node = new ST();
    node.li = l;
    node.ri = r;
    node.sum = A[l]; //max/min
    return node;
  }
  int mid = (l+r)/2;
  ST lN = makeSgmTree(A,l,mid);
  ST rN = makeSgmTree(A,mid+1,r);
  ST root = new ST();
  root.li = lN.li;
  root.ri = rN.ri;
  root.sum = lN.sum + rN.sum; //max/min
  root.lN = lN;
  root.rN = rN;
  return root;
}
static int getSum(ST root, int l, int r) {//max/min
  if(root.li>=l && root.ri<=r)
    return root.sum; //max/min
  if(root.ri<l || root.li > r)
    return 0; //minInt/maxInt
  else //max/min
    return getSum(root.lN,l,r) + getSum(root.rN,l,r);
}
static int update(ST root, int i, int val) {
  int diff = 0;
  if(root.li==root.ri && i == root.li) {
    diff = val-root.sum; //max/min
    root.sum=val; //max/min
    return diff; //root.max
  }
  int mid = (root.li + root.ri) / 2;
  if (i <= mid) diff = update(root.lN, i, val);
  else diff = update(root.rN, i, val);
  root.sum+=diff; //ask other child
  return diff; //and compute max/min
}
```

2.3. **Lazy Segment Tree.** More general implementation of a segment tree where its possible to increase whole segments by some diff, with lazy propagation. Implemented with arrays instead of nodes, which probably has less overhead to write during a competition.

```
class LazySegmentTree {
  private int n;
  private int[] lo, hi, sum, delta;
  public LazySegmentTree(int n) {
    this.n = n;
    lo = new int[4*n + 1];
    hi = new int[4*n + 1];
    sum = new int[4*n + 1];
    delta = new int[4*n + 1];
    init();
  }
  public int sum(int a, int b) {
    return sum(1, a, b);
  }
  private int sum(int i, int a, int b) {
    if(b < lo[i] || a > hi[i]) return 0;
    if(a <= lo[i] && hi[i] <= b) return sum(i);
    prop(i);
    int l = sum(2*i, a, b);
    int r = sum(2*i+1, a, b);
    update(i);
    return l + r;

  }
  public void inc(int a, int b, int v) {
    inc(1, a, b, v);
  }
  private void inc(int i, int a, int b, int v) {
    if(b < lo[i] || a > hi[i]) return;
    if(a <= lo[i] && hi[i] <= b) {
      delta[i] += v;
      return;
    }
    prop(i);
    inc(2*i, a, b, v);
    inc(2*i+1, a, b, v);
    update(i);
  }

  private void init() {
    init(1, 0, n-1, new int[n]);
  }
  private void init(int i, int a, int b, int[] v) {
    lo[i] = a;
    hi[i] = b;
    if(a == b) {
      sum[i] = v[a];
      return;
    }
    int m = (a+b)/2;
    init(2*i, a, m, v);
    init(2*i+1, m+1, b, v);
    update(i);
  }
  private void update(int i) {
    sum[i] = sum(2*i) + sum(2*i+1);
  }
  private int range(int i) {
    return hi[i] - lo[i] + 1;
  }
  private int sum(int i) {
    return sum[i] + range(i)*delta[i];
  }
  private void prop(int i) {
    delta[2*i] += delta[i];
    delta[2*i+1] += delta[i];
    delta[i] = 0;
```

```
  }
}
```

2.4. **Union Find.** This data structure is used in varoius algorithms, for example Kruskals algorithm for finding a Minimal Spanning Tree in a weighted graph. Also it can be used for backward simulation of dividing a set.

```
private class Node {
  Node parent;
  int h;
  public Node() {
    parent = this;
    h = 0;
  }
  public Node find() {
    if(parent != this) parent = parent.find();
    return parent;
  }
}
static void union(Node x, Node y) {
  Node xR = x.find(), yR = y.find();
  if(xR == yR) return;
  if(xR.h > yR.h)
    yR.parent = xR;
  else {
    if(yR.h == xR.h) yR.h++;
    xR.parent = yR;
  }
}
```

2.5. **Monotone Queue.** Used in sliding window algorithms where one would like to find the minimum in each interval of a given length. Amortized $\mathcal{O}(n)$ to find min in each of these intervals in an array of length $n$. Can easily be used to find the maximum as well.

```
private static class MinMonQue {
    LinkedList<Integer> que = new LinkedList<>();
    public void add(int i) {
        while(!que.isEmpty() && que.getFirst() > i)
            que.removeFirst();
        que.addFirst(i);
    }
    public int last() {
        return que.getLast();
    }
    public void remove(int i) {
        if(que.getLast() == i) que.removeLast();
    }
}
```

## 3. Graph Algorithms

3.1. **Djikstras algorithm.** Finds the shortest distance between two Nodes in a weighted graph in $\mathcal{O}(|E|\log|V|)$ time.

```
//Requires java.util.LinkedList and java.util.TreeSet
private static class Node implements Comparable<Node>{
  LinkedList<Edge> edges = new LinkedList<>();
  int w;
  int id;
  public Node(int id) {
    w = Integer.MAX_VALUE;
    this.id = id;
  }
  public int compareTo(Node n) {
    if(w != n.w) return w - n.w;
    return id - n.id;
  }
  //Asumes all nodes have weight MAXINT.
  public int djikstra(Node x) {
    this.w = 0;
    TreeSet<Node> set = new TreeSet<>();
    set.add(this);
    while(!set.isEmpty()) {
      Node curr = set.pollFirst();
      if(x == curr) return x.w;
      for(Edge e: curr.edges) {
        Node other = e.u == curr? e.v : e.u;
        if(other.w > e.cost + curr.w) {
          set.remove(other);
          other.w = e.cost + curr.w;
          set.add(other);
        }
      }
    }
    return -1;
  }
}
private static class Edge {
  Node u,v;
  int cost;
  public Edge(Node u, Node v, int c) {
    this.u = u; this.v = v;
    cost = c;
  }
}
```

3.2. **Bipartite Graphs.** The Hopcroft-Karp algorithm finds the maximal matching in a bipartite graph. Also, this matching can together with Könings theorem be used to construct a minimal vertex-cover, which as we all know is the complement of a maximum independent set. Runs in $\mathcal{O}(|E|\sqrt{|V|})$.

```
import java.util.*;
class Node {
  int id;
  LinkedList<Node> ch = new LinkedList<>();
  public Node(int id) {
    this.id = id;
  }
}
public class BiGraph {
  private static int INF = Integer.MAX_VALUE;
  LinkedList<Node> L, R;
  int N, M;
  Node[] U;
  int[] Pair, Dist;
  int nild;
  public BiGraph(LinkedList<Node> L, LinkedList<Node> R){
    N = L.size(); M = R.size();
    this.L = L; this.R = R;
    U = new Node[N+M];
    for(Node n: L) U[n.id] = n;
    for(Node n: R) U[n.id] = n;
  }
  private boolean bfs() {
    LinkedList<Node> Q = new LinkedList<>();
    for(Node n: L)
      if(Pair[n.id] == -1) {
        Dist[n.id] = 0;
        Q.add(n);
      }else
        Dist[n.id] = INF;
```

```
    nild = INF;
    while(!Q.isEmpty()) {
      Node u = Q.removeFirst();
      if(Dist[u.id] < nild)
        for(Node v: u.ch) if(distp(v) == INF){
          if(Pair[v.id] == -1)
            nild = Dist[u.id] + 1;
          else {
            Dist[Pair[v.id]] = Dist[u.id] + 1;
            Q.addLast(U[Pair[v.id]]);
          }
        }
    }
    return nild != INF;
  }
  private int distp(Node v) {
    if(Pair[v.id] == -1) return nild;
    return Dist[Pair[v.id]];
  }
  private boolean dfs(Node u) {
    for(Node v: u.ch) if(distp(v) == Dist[u.id] + 1) {
      if(Pair[v.id] == -1 || dfs(U[Pair[v.id]])) {
        Pair[v.id] = u.id;
        Pair[u.id] = v.id;
        return true;
      }
    }
    Dist[u.id] = INF;
    return false;
  }
  public HashMap<Integer, Integer> maxMatch() {
    Pair = new int[M+N];
    Dist = new int[M+N];
    for(int i = 0; i<M+N; i++) {
      Pair[i] = -1;
      Dist[i] = INF;
    }
    HashMap<Integer, Integer> out = new HashMap<>();
    while(bfs()) {
      for(Node n: L) if(Pair[n.id] == -1)
        dfs(n);
    }
    for(Node n: L) if(Pair[n.id] != -1)
      out.put(n.id, Pair[n.id]);
    return out;
  }
  public HashSet<Integer> minVTC() {
    HashMap<Integer, Integer> Lm = maxMatch();
    HashMap<Integer, Integer> Rm = new HashMap<>();
    for(int x: Lm.keySet()) Rm.put(Lm.get(x), x);
    boolean[] Z = new boolean[M+N];
    LinkedList<Node> bfs = new LinkedList<>();
    for(Node n: L) {
      if(!Lm.containsKey(n.id)) {
        Z[n.id] = true;
        bfs.add(n);
      }
    }
    while(!bfs.isEmpty()) {
      Node x = bfs.removeFirst();
      int nono = -1;
      if(Lm.containsKey(x.id))
        nono = Lm.get(x.id);
      for(Node y: x.ch) {
        if(y.id == nono || Z[y.id]) continue;
        Z[y.id] = true;
        if(Rm.containsKey(y.id)){
          int xx = Rm.get(y.id);
          if(!Z[xx]) {
            Z[xx] = true;
            bfs.addLast(U[xx]);
          }
        }
      }
    }
    HashSet<Integer> K = new HashSet<>();
    for(Node n: L) if(!Z[n.id]) K.add(n.id);
    for(Node n: R) if(Z[n.id]) K.add(n.id);
    return K;
  }
}
```

3.3. **Network Flow.** The Floyd Warshall algorithm for determining the maximum flow through a graph can be used for a lot of unexpected problems. Given a problem that can be formulated as a graph, where no ideas are found trying, it might help trying to apply network flow. The running time is $\mathcal{O}(C \cdot m)$ where $C$ is the maximum flow and $m$ is the amount of edges in the graph. If $C$ is very large we can change the running time to $\mathcal{O}(\log Cm^2)$ by only studying edges with a large enough capacity in the beginning.

```
import java.util.*;
class Node {
  LinkedList<Edge> edges = new LinkedList<>();
  int id;
  boolean visited = false;
  Edge last = null;
  public Node(int id) {
    this.id = id;
  }
  public void append(Edge e) {
    edges.add(e);
  }
}
class Edge {
  Node source, sink;
  int cap;
  int id;
  Edge redge;
  public Edge(Node u, Node v, int w, int id){
    source = u; sink = v;
    cap = w;
    this.id = id;
  }
}
class FlowNetwork {
  Node[] adj;
  int edgec = 0;
  HashMap<Integer,Integer> flow = new HashMap<>();
  ArrayList<Edge> real = new ArrayList<Edge>();
  public FlowNetwork(int size) {
    adj = new Node[size];
    for(int i = 0; i<size; i++) {
      adj[i] = new Node(i);
    }
  }
  void add_edge(int u, int v, int w, int id){
    Node Nu = adj[u], Nv = adj[v];
    Edge edge = new Edge(Nu, Nv, w, edgec++);
    Edge redge = new Edge(Nv, Nu, 0, edgec++);
    edge.redge = redge;
```

```
    redge.redge = edge;
    real.add(edge);
    adj[u].append(edge);
    adj[v].append(redge);
    flow.put(edge.id, 0);
    flow.put(redge.id, 0);
  }

  void reset() {
    for(int i = 0; i<adj.length; i++) {
      adj[i].visited = false; adj[i].last = null;
    }
  }

  LinkedList<Edge> find_path(Node s, Node t,
          List<Edge> path){
    reset();
    LinkedList<Node> active = new LinkedList<>();
    active.add(s);
    while(!active.isEmpty() && !t.visited) {
      Node now = active.pollFirst();
      for(Edge e: now.edges) {
        int residual = e.cap - flow.get(e.id);
        if(residual>0 && !e.sink.visited) {
          e.sink.visited = true;
          e.sink.last = e;
          active.addLast(e.sink);
        }
      }
    }
    if(t.visited) {
      LinkedList<Edge> res = new LinkedList<>();
      Node curr = t;
      while(curr != s) {
        res.addFirst(curr.last);
        curr = curr.last.sink;
      }
      return res;
    } else return null;
  }

  int max_flow(int s, int t) {
    Node source = adj[s];
    Node sink = adj[t];
    LinkedList<Edge> path = find_path(source, sink,
            new LinkedList<Edge>());
    while (path != null) {
      int min = Integer.MAX_VALUE;
      for(Edge e : path) {
        min = Math.min(min, e.cap - flow.get(e.id));
      }
      for (Edge e : path) {
        flow.put(e.id, flow.get(e.id) + min);
        Edge r = e.redge;
        flow.put(r.id, flow.get(r.id) - min);
      }
      path = find_path(source, sink,
              new LinkedList<Edge>());
    }
    int sum = 0;
    for(Edge e: source.edges) {
      sum += flow.get(e.id);
    }
    return sum;
  }

  LinkedList<Edge> min_cut(int s, int t) {
    HashSet<Node> A = new HashSet<>();
    LinkedList<Node> bfs = new LinkedList<>();
    bfs.add(adj[s]);
    A.add(adj[s]);
    while(!bfs.isEmpty()) {
      Node i = bfs.removeFirst();
      for(Edge e: i.edges) {
        int c = e.cap - flow.get(e.id);
        if(c > 0 && !A.contains(e.sink)) {
          bfs.add(e.sink);
          A.add(e.sink);
          if(e.sink.id == t) return null;
        }
      }
    }
    LinkedList<Edge> out = new LinkedList<>();
    for(Node n: A) for(Edge e: n.edges)
      if(!A.contains(e.sink) && e.cap != 0)
        out.add(e);
    return out;
  }
}
```

## 4. Dynamic Programming

**4.1. Longest Increasing Subsequence.** Finds the longest increasing subsequence in an array in $\mathcal{O}(n \log n)$ time. Can easility be transformed to longest decreasing/nondecreasing/nonincreasing subsequence.

```
public static int lis(int[] X) {
  int n = X.length;
  int P[] = new int[n];
  int M[] = new int[n+1];
  int L = 0;
  for(int i = 0; i<n; i++) {
    int lo = 1;
    int hi = L;
    while(lo<=hi) {
      int mid = lo + (hi - lo + 1)/2;
      if(X[M[mid]]<X[i])
        lo = mid+1;
      else
        hi = mid-1;
    }
    int newL = lo;
    P[i] = M[newL-1];
    M[newL] = i;
    if (newL > L)
      L = newL;

  }
  int[] S = new int[L];
  int k = M[L];
  for (int i = L-1; i>=0; i--) {
    S[i] = k; //or X[k]
    k = P[k];
  }
  return L; // or S
}
```

**4.2. Knuuth Morris Pratt substring.** Finds if $w$ is a subarray to $s$ in linear time.

```
//assumes s.length>=w.length
public static boolean kmp(int [] w, int [] s) {
  int T[] = new int[w.length];
```

```
  T[0] = -1; T[1] = 0;
  int m = 0, i = 2;
  while(i<w.length) {
    if(w[i-1] == w[m]) {
      T[i] = ++m;
      i++;
    } else if (m>0) {
      m = T[m];
    } else {
      T[i] = 0;
      i++;
    }
  }

  m = 0; i = 0;
  while(m+i<s.length){
    if(w[i] == s[m+i]) {
      if(i == w.length - 1)
        return true; //m
      i++;
    } else {
      if(T[i] > -1) {
        m = m + i - T[i];
        i = T[i];
      } else {
        i = 0;
        m = m+1;
      }}}
  return false;
}
```

## 5. Etc

**5.1. System of Equations.** Solves the system of equations $\boldsymbol{Ax} = \boldsymbol{b}$ by Gaussian elimination. This can for example be used to determine the expected value of each node in a markov chain. Runns in $\mathcal{O}(N^3)$.

```
//Computes A^-1 * b
static double[] solve(double[][] A, double[] b) {
  int N = b.length;
  // Gaussian elimination with partial pivoting
  for (int i = 0; i < N; i++) {
    // find pivot row and swap
    int max = i;
    for (int j = i + 1; j < N; j++)
      if (Math.abs(A[j][i]) > Math.abs(A[max][i]))
        max = j;
    double[] tmp = A[i];
    A[i] = A[max];
    A[max] = tmp;
    double tmp2 = b[i];
    b[i] = b[max];
    b[max] = tmp2;
    // A doesn't have full rank
    if (Math.abs(A[i][i])<0.00001) return null;
    // pivot within b
    for (int j = i + 1; j < N; j++)
      b[j] -= b[i] * A[j][i] / A[i][i];
    // pivot within A
    for (int j = i + 1; j < N; j++) {
      double m = A[j][i] / A[i][i];
      for (int k = i+1; k < N; k++)
        A[j][k] -= A[i][k] * m;
      A[j][i] = 0.0;
    }
  }
  // back substitution
  double[] x = new double[N];
  for (int j = N - 1; j >= 0; j--) {
    double t = 0.0;
    for (int k = j + 1; k < N; k++)
      t += A[j][k] * x[k];
    x[j] = (b[j] - t) / A[j][j];
  }
  return x;
}
```

**5.2. Convex Hull.** From a collection of points in the plane the convex hull is often used to compute the largest distance or the area covered, or the length of a rope that encloses the points. It can be found in $\mathcal{O}(N \log N)$ time by sorting the points on angle and the sweeping over all of them.

```
import java.util.*;
public class ConvexHull {
  static class Point implements Comparable<Point> {
    static Point xmin;
    int x, y;
    public Point(int x, int y) {
      this.x = x;this.y = y;
    }
    public int compareTo(Point p) {
      int c = cross(this, xmin, p);
      if(c!=0) return c;
      double d = dist(this,xmin) - dist(p,xmin);
      return (int) Math.signum(d);
    }
  }
  static double dist(Point p1, Point p2) {
    return Math.hypot(p1.x - p2.x, p1.y - p2.y);
  }
  static int cross(Point a, Point b, Point c) {
    int dx1 = b.x - a.x;
    int dy1 = b.y - a.y;
    int dx2 = c.x - b.x;
    int dy2 = c.y - b.y;
    return dx1*dy2 - dx2*dy1;
  }
  Point[] convexHull(Point[] S) {
    int N = S.length;
    // find a point on the convex hull.
    Point xmin = S[0];
    int id = 0;
    for(int i = 0; i<N; i++) {
      Point p = S[i];
      if(xmin.x > p.x ||
         xmin.x == p.x && xmin.y > p.y) {
        xmin = p;
        id = i;
      }
    }
    S[id] = S[N-1];
    S[N-1] = xmin;
    Point.xmin = xmin;
    // Sort on angle to xmin.
    Arrays.sort(S, 0, N-1);
    Point[] H = new Point[N+1];
    H[0] = S[N-2];
    H[1] = xmin;
    for(int i = 0; i<N-1; i++)
      H[i+2] = S[i];
    int M = 1;
    // swipe over the points
```

```
    for(int i = 2; i<=N; i++) {
      while(cross(H[M-1],H[M],H[i]) <= 0) {
        if(M>1)
          M--;
        else if (i == N)
          break;
        else
          i += 1;
      }
      M+=1;
      Point tmp = H[M];
      H[M] = H[i];
      H[i] = tmp;
    }
    Point[] Hull = new Point[M];
    for(int i = 0; i<M; i++)
      Hull[i] = H[i];
    return Hull;
  }
}
```

## 6. NP tricks

6.1. **MaxClique.** The max clique problem is one of Karp's 21 NP-complete problems. The problem is to find the lagest subset of an undirected graph that forms a clique - a complete graph. There is an obvious algorithm that just inspects every subset of the graph and determines if this subset is a clique. This algorithm runns in $\mathcal{O}(n^2 2^n)$. However one can use the meet in the middle trick (one step divide and conquerer) and reduce the complexity to $\mathcal{O}(n^2 2^{\frac{n}{2}})$.

```
static int max_clique(int n, int[][] adj) {
  int fst = n/2;
  int snd = n - fst;
  int[] maxc = new int[1<<fst];
  int max = 1;
  for(int i = 0; i<(1<<fst); i++) {
    for(int a = 0; a<fst; a++) {
      if((i&1<<a) != 0)
        maxc[i] = Math.max(maxc[i], maxc[i^(1<<a)]);
    }
    boolean ok = true;
    for(int a = 0; a<fst; a++) if((i&1<<a) != 0) {
      for(int b = a+1; b<fst; b++) {
          if((i&1<<b) != 0 && adj[a][b] == 0)
            ok = false;
      }
    }
    if(ok) {
      maxc[i] = Integer.bitCount(i);
      max = Math.max(max, maxc[i]);
    }
  }
  for(int i = 0; i<(1<<snd); i++) {
    boolean ok = true;
    for(int a = 0; a<snd; a++) if((i&1<<a) != 0) {
      for(int b = a+1; b<snd; b++) {
        if((i&1<<b) != 0)
          if(adj[a+fst][b+fst] == 0)
            ok = false;
      }
    }
    if(!ok) continue;
    int mask = 0;
    for(int a = 0; a<fst; a++) {
      ok = true;
      for(int b = 0; b<snd; b++) {
        if((i&1<<b) != 0) {
          if(adj[a][b+fst] == 0) ok = false;
        }
      }
      if(ok) mask |= (1<<a);
    }
    max = Math.max(Integer.bitCount(i) + maxc[mask],
            max);
  }
  return max;
}
```