

# BAPC 2015

## Teams UCool and UCoolTwo

Authors: François Aubry, Guillaume Derval, Benoît Legat, Anthony Géo.

## Contents

<b>1</b>	<b>Remarks</b>	<b>2</b>			
1.1	Warning!	2		3.7.1	$O(n \log(n)^2)$ , full matrix, need $n \leq 10K$
1.2	Operations on bits	2		3.7.2	$O(n \log(n))$ , only last line, need $n \leq 100K$
1.3	Complexity table	2			
<b>2</b>	<b>Graphs</b>	<b>2</b>	<b>4</b>	<b>Geometry in 2D</b>	<b>13</b>
2.1	Basics	2	4.1	Vectors	13
2.2	BFS	2	4.1.1	Rotation around (0,0)	13
2.2.1	Connected components	3	4.2	Points	13
2.2.2	Girth	3	4.2.1	Point in box	14
2.3	DFS	3	4.2.2	Polar sort	14
2.3.1	Topological order	3	4.2.3	Closest pair of points	14
2.3.2	Strongly connected components	3	4.2.4	Orientation	14
2.3.3	SCC, Bridges and Articulation Points in C	3	4.2.5	Angle visibility	14
2.3.4	Directed Graph to toposorted DAG	4	4.2.6	Fixed radius neighbors (1D)	14
2.4	Minimum Spanning Tree	4	4.2.7	Fixed radius neighbors (2D)	15
2.4.1	Prim	4	4.3	Lines	15
2.4.2	Kruskal	5	4.3.1	Intersections	15
2.5	Dijkstra	5	4.3.2	Perpendicular line	15
2.6	Bellman-Ford	5	4.3.3	Orthogonal Symmetry	15
2.7	Floyd-Warshall	5	4.4	Segments	15
2.8	Directed Max flow	5	4.4.1	Intersection	15
2.8.1	Edmonds-Karps (BFS)	5	4.4.2	Intersections problem	15
2.8.2	Ford-Fulkerson	6	4.5	Circles	16
2.8.3	Min cut	6	4.5.1	Circles from 3 points	16
2.8.4	Maximum number of disjoint paths	6	4.5.2	Circle-line intersection	16
2.8.5	Maximum weighted bipartite matching	6	4.5.3	Circle-circle or circle-point tangents	16
2.9	Directed Min cost flow	7	4.6	Polygons	16
2.10	DAG path cover	8	4.6.1	Triangulation	16
2.10.1	Cover vertex: disjoint paths	8	4.6.2	Triangles	16
2.10.2	Cover vertex: non-disjoint	8	4.6.3	Check convexity	16
2.10.3	Cover edges: disjoint	8	4.6.4	Winding number	17
2.11	Max-Flow with demands	8	4.6.5	Convex Hull	17
2.11.1	Node demande	8	4.7	Interval Tree	17
2.11.2	Edge lower bounds	8	4.8	Area of union of rectangles	18
2.12	Chinese Postman Problem	8			
2.13	Bipartite graph	8	<b>5</b>	<b>Geometry in 3D</b>	<b>18</b>
2.13.1	Max Cardinality Bipartite Matching (MCBM)	8	5.1	Cross product	18
2.13.2	Independent Set (or Dominating Set)	9	5.2	Equation of a plane	18
2.13.3	Vertex Cover	9	5.2.1	with a normal vector and a point	18
			5.2.2	with a point and two vectors in the plane	18
			5.2.3	with three points	18
<b>3</b>	<b>Dynamic programming</b>	<b>9</b>	5.3	Equation of a line	18
3.1	Bottom-up	9	5.3.1	With a point and a vector	18
3.2	Top-down	10	5.3.2	With two points	19
3.3	Knapsack problem	10	5.4	Distance from a point to a line	19
3.3.1	No repetition	10	5.5	Distance from a point to a plane	19
3.3.2	An object can be repeated	10	5.6	Orthogonal projection of a point on a line	19
3.3.3	Several knapsacks	10	5.7	Orthogonal projection of a point on a plane	19
3.4	Longest common sub-sequence (LCS)	10	5.8	Orthogonal projection of a line on a plane	19
3.5	Matrix Chain Multiplication (MCM)	10	5.9	Finding if a point is in a 3D polygon	19
3.5.1	Generalized MCM	11	5.10	Intersection of a line and a plane	19
3.6	Edit distance	11	<b>6</b>	<b>Math</b>	<b>19</b>
3.7	Suffix array	11	6.1	Permutations, Combinations, Arrangements...	
				<i>untested</i>	19
			6.2	Decomposition in unit fractions	<i>untested</i>
			6.3	Combination	20
			6.3.1	Catalan numbers	20
			6.4	Fibonacci series	20
			6.5	Cycle finding	20
			6.6	Number theory	20
			6.6.1	Misc	20
			6.6.2	Équations diophantiennes	20

6.6.3	Chinese remainder theorem . . . . .	20
6.6.4	Euler phi . . . . .	21
6.6.5	Quadratic residue (QR) . . . . .	21
6.7	Linear equations . . . . .	22
6.8	Ternary Search . . . . .	22
6.9	Integration . . . . .	22
<b>7</b>	<b>Strings <i>untested</i></b>	<b>22</b>
7.1	Longest palindrome . . . . .	22
7.2	Occurrences in a string . . . . .	23
7.3	Multipattern search: Aho-Corasick . . . . .	23
7.4	Match with hash: Rabin-Karp . . . . .	24
<b>8</b>	<b>Miscellaneous</b>	<b>24</b>
8.1	The answer . . . . .	24
8.2	FFT . . . . .	24
8.3	Sort algorithms <i>untested</i> . . . . .	25
8.4	Huffman (compression) . . . . .	26
8.5	Union Find . . . . .	26
8.6	Fenwick Tree (RSQ solver) . . . . .	26
<b>9</b>	<b>Being first?</b>	<b>27</b>

## 1 Remarks

### 1.1 Warning!

1. Read every statement!
2. Do not copy-paste without thinking about it.
3. Be careful of overflows! Use long!
4. **Do not trust this document!**

### 1.2 Operations on bits

1. Check parity of  $n$ :  $(n \& 1) == 0$
2.  $2^n$ :  $1L \ll n$ .
3. Test of the  $i$ th bit of  $n$  is 0:  $(n \& 1L \ll i) != 0$
4. Set the  $i$ th bit of  $n$  at 0:  $n \&= \sim(1L \ll i)$
5. Set the  $i$ th bit of  $n$  at 1:  $n |= (1L \ll i)$
6. Union:  $a | b$
7. Intersection:  $a \& b$
8. Subtraction bits:  $a \& \sim b$
9. Verify if  $n$  is a power of 2:  $(n \& (n-1) == 0)$
10. Least significant bit not null of  $n$ :  $(n \& (-n))$
11. Negate:  $0 \times 7fffffff \wedge n$

## 1.3 Complexity table

$n \leq$	Maximum complexity
[10, 11]	$O(n!), O(n^6)$
[15, 18]	$O(2^n n^2)$
[18, 22]	$O(2^n n)$
100	$O(n^4)$
400	$O(n^3)$
$2K$	$O(n^2 \log(n))$
$5K$	$O(n^2)$
$1M$	$O(n \log(n))$
$10M$	$O(n), O(\log(n)), O(1)$

Not so obvious complexity:

$$\sum_{k=1}^n \frac{1}{k} = O(\log(n))$$

## 2 Graphs

### 2.1 Basics

- Adjacency matrix:  $A[i][j] = 1$  if  $i$  is connected to  $j$  and 0 otherwise
- Undirected graph:  $A[i][j] = A[j][i] \forall i, j$  ( $A = A^T$ )
- Adjacency list: `LinkedList<Integer>[] g`;  $g[i]$  stores all neighbors of  $i$
- Useful alternatives:  

```
HashSet<Integer>[] g; // for edge deletion
HashMap<Integer, Integer>[] g; // for weighted graph
```
- Basic classes  

```
class Edge implements Comparable<Edge> {
    int o, d, w;
    public Edge(int o, int d, int w) {
        this.o = o; this.d = d; this.w = w;
    }
    public int compareTo(Edge o) {
        return w - o.w;
    }
}
```

### 2.2 BFS

Computes  $d$ , an array of distance from start vertex  $v$ .  $d[v] = 0$ ,  $d[u] = \infty$  if  $u$  not connected to  $v$ . If  $(u, w) \in E$  and  $d[u]$  known and  $d[w]$  unknown,  $d[w] = d[u] + 1$ .

```
int[] bfsVisit(LinkedList<Integer>[] g, int v, int c
[]) { //c is for connected components only
    Queue<Integer> Q = new LinkedList<Integer>();
    Q.add(v);
    int[] d = new int[g.length];
    c[v]=v; //for connected components
    Arrays.fill(d, Integer.MAX_VALUE);
    // set distance to origin to 0
    d[v] = 0;
    while(!Q.isEmpty()) {
        int cur = Q.poll();
        // go over all neighbors of cur
        for(int u : g[cur]) {
            // if u is unvisited
            if(d[u] == Integer.MAX_VALUE) { //or c[u] ==
-1 if we calculate connected components
                c[u] = v; //for connected components
                Q.add(u);
            }
        }
    }
}
```

```

        // set the distance from v to u
        d[u] = d[cur] + 1;
    }
}
return d;
}

```

### 2.2.1 Connected components

```

int[] bfs(LinkedList<Integer>[] g)
{
    int[] c = new int[g.length];
    Arrays.fill(c, -1);
    for(int v = 0; v < g.length; v++)
        if(c[v] == -1)
            bfsVisit(g, v, c);
    return c;
}

```

### 2.2.2 Girth

The girth of an undirected graph is the length of its shortest cycle ( $\infty$  if none). Complexity  $O(|V||E|)$ .

```

int girth(LinkedList<Integer>[] g) {
    int girth = Integer.MAX_VALUE;
    for(int v = 0; v < g.length; v++) {
        girth = Math.min(girth, checkFromV(v, g));
    }
    return girth;
}

int checkFromV(int v, LinkedList<Integer>[] g) {
    int[] parent = new int[g.length];
    Arrays.fill(parent, -1);
    int[] d = new int[g.length];
    Arrays.fill(d, Integer.MAX_VALUE);
    Queue<Integer> Q = new LinkedList<Integer>();
    Q.add(v);
    d[v] = 0;
    while(!Q.isEmpty()) {
        int cur = Q.poll();
        for(int u : g[cur]) {
            if(u != parent[cur]) {
                if(d[u] == Integer.MAX_VALUE) {
                    parent[u] = cur;
                    d[u] = d[cur] + 1;
                    Q.add(u);
                } else {
                    return d[cur] + d[u] + 1;
                }
            }
        }
    }
    return Integer.MAX_VALUE;
}

```

## 2.3 DFS

Equals to BFS with *Stack* instead of *Queue* or recursive implementation. Complexity  $O(|V| + |E|)$

```

int UNVISITED = 0, OPEN = 1, CLOSED = 2;
boolean cycle; // true iff there is a cycle

void dfsVisit(LinkedList<Integer>[] g, int v, int[] label) {
    label[v] = OPEN;
    for(int u : g[v]) {
        if(label[u] == UNVISITED)
            dfsVisit(g, u, label);
        if(label[u] == OPEN)
            cycle = true;
    }
    label[v] = CLOSED;
}

```

```

void dfs(LinkedList<Integer>[] g) {
    int[] label = new int[g.length];
    Arrays.fill(label, UNVISITED);
    cycle = false;
    for(int v = 0; v < g.length; v++)
        if(label[v] == UNVISITED)
            dfsVisit(g, v, label);
}

```

### 2.3.1 Topological order

Graph must be acyclic.

```

Stack<Integer> toposort; // add stack to global variables
/* ... */
void dfs(LinkedList<Integer>[] g) {
    /* ... */
    toposort = new Stack<Integer>();
    for(int v = 0; v < g.length; v++) { /* ... */ }
}

void dfsVisit(LinkedList<Integer>[] g, int v, int[] label) {
    /* ... */
    toposort.push(v); // push vertex when closing it
    label[v] = CLOSED;
}

```

### 2.3.2 Strongly connected components

Uses BFS following the topologic order.

```

int[] scc(LinkedList<Integer>[] g) {
    // compute the reverse graph
    LinkedList<Integer>[] gt = transpose(g);
    // compute ordering
    dfs(gt);
    // !! last position will contain the number of scc
    int[] scc = new int[g.length + 1];
    Arrays.fill(scc, -1);
    int nbComponents = 0;
    // simulate bfs loop but in toposort ordering
    while(!toposort.isEmpty()) {
        int v = toposort.pop();
        if(scc[v] == -1) {
            nbComponents++;
            bfsVisit(g, v, scc);
        }
    }
    scc[g.length] = nbComponents;
    return scc;
}

```

### 2.3.3 SCC, Bridges and Articulation Points in C

C version of SCC (shorter).

```

void tarjanSCC(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounder++; //
    dfs_low[u] <= dfs_num[u]
    S.push_back(u); // stores u in a vector based on
    order of visitation
    visited[u] = 1;
    for(int j = 0; j < (int)AdjList[u].size(); j++) {
        int v = AdjList[u][j];
        if(dfs_num[v].first == UNVISITED)
            tarjanSCC(v.first);
        if(visited[v.first]) // condition for update
            dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
    }
    if(dfs_low[u] == dfs_num[u]) { // if this is a
        root (start) of an SCC
        printf("SCC %d:", ++numSCC); // this part is
        done after recursion
        while(1) {

```

```

    int v = S.back(); S.pop_back(); visited[v] =
0;
    printf(" %d", v);
    if(u == v) break;
}
printf("\n");
}
}

int main() {
    dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0)
;
    visited.assign(V, 0); dfsNumberCounter = numSCC =
0;
    for(int i = 0; i < V; i++)
        if(dfs_num[i] == UNVISITED)
            tarjanSCC(i);
}

Bridges are edges that, when removed, increases the number
of connected components. Articulation points are the same,
but for vertices.
void articulationPointAndBridge(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; //
dfs_low[u] <= dfs_num[u]
    for(int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];
        if(dfs_num[v.first] == UNVISITED) { // a tree
edge
            dfs_parent[v.first] = u;
            if(u == dfsRoot) rootChildren++; // special
case if u is a root
            articulationPointAndBridge(v.first);
            if(dfs_low[v.first] >= dfs_num[u]) // for
articulation point
                articulation_vertex[u] = true; // store this
information first
            if(dfs_low[v.first] > dfs_num[u]) // for
bridge
                printf("Edge (%d %d) is a bridge\n", u, v.
first);
            dfs_low[u] = min(dfs_low[u], dfs_low[v.first])
; // update dfs_low[u]
        }
        else if(v.first != dfs_parent[u]) // a back edge
and not direct cycle
            dfs_low[u] = min(dfs_low[u], dfs_num[v.first])
; // update dfs_low[u]
    }
}

int main() {
    dfsNumberCounter = 0; dfs_num.assign(V, UNVISITED)
;
    dfs_low.assign(V, 0); dfs_parent.assign(V, 0);
    articulation_vertex.assign(V, 0);
    printf("Bridges:\n");
    for(int i = 0; i < V; i++) {
        dfsRoot = i; rootChildren = 0;
        articulationPointBridge(i);
        articulation_vertex[dfsRoot] = (rootChildren >
1); // special case
    }
    printf("Articulation Points:\n");
    for(int i = 0; i < V; i++)
        if(articulation_vertex[i])
            printf("Vertex %d\n", i);
}

```

### 2.3.4 Directed Graph to toposorted DAG

In  $O(n+m)$ , with Tarjan SCC algo, we merge the SCCs and take the resulting DAG, (remembering their size in `scc_size`) which is reverse toposorted (i.e. node 0 has no outgoing edge), ready for bottom up DP (starting with node 0 ending with node  $N$ ) !

```
static Integer[] dfs_num;
```

```

static int[] dfs_low, scc_id;
static BitSet visited;
static int dfsNumberCounter;
static Stack<Integer> S;
static void tarjanSCC(LinkedList<Integer>[] g, int u
, LinkedList<LinkedList<Integer>> SCCs) {
    dfs_low[u] = dfsNumberCounter;
    dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <=
dfs_num[u]
    S.add(u); // stores u in a vector based on order
of visitation
    visited.set(u);
    for(int v : g[u]) {
        if(dfs_num[v] == null)
            tarjanSCC(g, v, SCCs);
        if(visited.get(v)) // condition for update
            dfs_low[u] = Math.min(dfs_low[u], dfs_low[v]);
    }
    if(dfs_low[u] == dfs_num[u]) { // if this is a
root (start) of an SCC
        LinkedList<Integer> newSCC = new LinkedList<
Integer>();
        int id = SCCs.size();
        for(;;) {
            int v = S.pop(); visited.clear(v);
            newSCC.add(v);
            scc_id[v] = id;
            if(u == v) break;
        }
        SCCs.add(newSCC);
    }
}

static LinkedList<Integer>[] DirectedGraphToDag (
    LinkedList<Integer>[] g) {
    int n = g.length;
    dfs_num = new Integer[n];
    dfs_low = new int[n];
    scc_id = new int[n];
    visited = new BitSet(n);
    dfsNumberCounter = 0;
    S = new Stack<Integer>();
    LinkedList<LinkedList<Integer>> SCCs = new
LinkedList<LinkedList<Integer>>();
    for(int i = 0; i < n; i++)
        if(dfs_num[i] == null)
            tarjanSCC(g, i, SCCs);
    int N = SCCs.size();
    @SuppressWarnings("unchecked")
    LinkedList<Integer>[] G = new LinkedList[N];
    scc_size = new int[N];
    int i = 0;
    for (LinkedList<Integer> SCC : SCCs) {
        G[i] = new LinkedList<Integer>();
        scc_size[i] = SCC.size();
        BitSet reachable = new BitSet(N);
        reachable.set(i);
        for (int u : SCC) {
            for (int v : g[u])
                if (!reachable.get(scc_id[v])) {
                    G[i].add(scc_id[v]);
                }
        }
        i++;
    }
    return G;
}

static int[] scc_size; // bonus information

```

## 2.4 Minimum Spanning Tree

### 2.4.1 Prim

```

double prim(LinkedList<Edge>[] g) {
    boolean[] inTree = new boolean[g.length];
    PriorityQueue<Edge> PQ = new PriorityQueue<Edge>()
;
    // add 0 to the tree and initialize the priority
queue
    inTree[0] = true;
}

```

```

for (Edge e : g[0]) PQ.add(e);
double weight = 0;
int size = 1;
while (size != g.length) {
    // poll the minimum weight edge in PQ
    Edge minE = PQ.poll();
    // if its endpoint in not in the tree, add it
    if (!inTree[minE.d]) {
        // add edge minE to the MST
        inTree[minE.d] = true;
        weight += minE.w;
        size++;
        // add edge leading to new endpoints to the PQ
        for (Edge e : g[minE.d])
            if (!inTree[e.d]) PQ.add(e);
    }
}
return weight;
}

```

#### 2.4.2 Kruskal

Uses Union-Find (See section 8.5).

```

double kruskal(LinkedList<Edge> g, int n) {
    Collections.sort(g);
    UnionFind uf = new UnionFind(n);
    double w = 0;
    int c = 0;
    for (Edge e : g) {
        if (c == n-1) return w;
        if (uf.find(e.o) != uf.find(e.d)) {
            w += e.w;
            c++;
            uf.union(e.o, e.d);
        }
    }
    return w;
}

```

### 2.5 Dijkstra

Shortest path from a node  $v$  to other nodes. Graph must not have any negative weighted cycle.  $O((|V| + |E|) \log(|V|))$

```

double[] dijkstra(LinkedList<Edge> g, int v) {
    double[] d = new double[g.length];
    Arrays.fill(d, Double.POSITIVE_INFINITY);
    d[v] = 0;
    PriorityQueue<Edge> PQ = new PriorityQueue<Edge>();
    for (Edge e : g[v])
        PQ.add(e);
    while (!PQ.isEmpty()) {
        Edge minE = PQ.poll();
        if (d[minE.d] == Double.POSITIVE_INFINITY) {
            d[minE.d] = minE.w;
            for (Edge e : g[minE.dest])
                if (d[e.d] == Double.POSITIVE_INFINITY)
                    PQ.add(new Edge(e.o, e.d, e.w + d[e.o]));
        }
    }
    return d;
}

```

### 2.6 Bellman-Ford

Shortest path from a node  $v$  to other nodes. Graph can have negative weighted cycles: Bellman-Ford won't give the correct shortest path, but will warn that a negative cycle exists.  $O(|V||E|)$ .

```

static double[] bellmanFord(LinkedList<Edge> gt, int v, int n) {
    double[] dist = new double[n];
    Arrays.fill(dist, Double.POSITIVE_INFINITY);
    dist[v] = 0;
    for (int i=0; i < n-1; i++)
        for (Edge e : gt)

```

```

            if (dist[e.o] + e.w < dist[e.d])
                dist[e.d] = dist[e.o] + e.w;
    for (Edge e : gt)
        if (dist[e.o] + e.w < dist[e.d])
            return null;
    return dist;
}

```

```

static double[] spfa (LinkedList<Edge> g, int s) {
    int n = g.length;
    double[] dist = new double[n];
    Arrays.fill(dist, Double.POSITIVE_INFINITY);
    Queue<Integer> q = new LinkedList<Integer>();
    BitSet inQueue = new BitSet(n);
    int[] timesIn = new int[n];
    dist[s] = 0;
    q.add(s);
    inQueue.set(s);
    timesIn[s]++;
    while (!q.isEmpty()) {
        int cur = q.poll(); inQueue.clear(cur);
        for (Edge next : g[cur]) {
            int v = next.d, w = next.w;
            if (dist[cur] + w < dist[v]) {
                dist[v] = dist[cur] + w;
                if (!inQueue.get(v)) {
                    q.add(v);
                    inQueue.set(v);
                    timesIn[v]++;
                    if (timesIn[v] >= n) {
                        return null; // Infinite loop
                    }
                }
            }
        }
    }
    return dist;
}

```

### 2.7 Floyd-Warshall

Shortest path from a node  $v$  to other nodes. Graph can have negative weighted cycles: Floyd-Warshall won't give the correct shortest path, but will warn that a negative cycle exists. Negative weighted cycles exists iff  $result[v][v] < 0$ .  $O(|V|^3)$  in time and  $O(|V|^2)$  in memory.

```

double[][] floydWarshall(double[][] A)
{
    int n = A.length;
    for (int k = 0; k < n; k++)
        for (int v = 0; v < n; v++)
            for (int u = 0; u < n; u++)
                A[v][u] = Math.min(A[v][u], A[v][k] + A[k][u]);
    //or:
    A[v][u] = min(A[v][u], max(A[v][k], A[k][u]));
    //minimax
    A[v][u] = max(A[v][u], min(A[v][k], A[k][u]));
    //maximin
    A[v][u] = max(A[v][u], A[v][k] * A[k][u]);
    //safest path (A contains probability)
    return A;
}

```

### 2.8 Directed Max flow

#### 2.8.1 Edmonds-Karps (BFS)

Path in residual graph searched via BFS.  $O(|V||E|^2)$ .

```

int maxflowEK(TreeMap<Integer, Integer> g, int source, int sink) {
    int flow = 0;
    int pcap;

```

```

while((pcap = augmentBFS(g, source, sink)) != -1)
{
    flow += pcap;
}
return flow;
}

int augmentBFS(TreeMap<Integer, Integer>[] g, int
source, int sink) {
    // initialize bfs
    Queue<Integer> Q = new LinkedList<Integer>();
    Integer[] p = new Integer[g.length];
    int[] pcap = new int[g.length];
    pcap[source] = Integer.MAX_VALUE;
    p[source] = -1;
    Q.add(source);
    // compute path
    while(p[sink] == null && !Q.isEmpty()) {
        int u = Q.poll();
        for(Entry<Integer, Integer> e : g[u].entrySet())
        {
            int v = e.getKey();
            if(e.getValue() > 0 && p[v] == null) {
                p[v] = u;
                pcap[v] = Math.min(pcap[u], e.getValue());
                Q.add(v);
            }
        }
    }
    if(p[sink] == null) return -1;
    // update graph
    int cur = sink;
    while(cur != source) {
        int prev = p[cur];
        int cap = g[prev].get(cur);
        g[prev].put(cur, cap - pcap[sink]);
        Integer backcap = g[cur].get(prev);
        g[cur].put(prev, backcap == null? pcap[sink] :
        backcap + pcap[sink]);
        cur = prev;
    }
    return pcap[sink];
}

```

## 2.8.2 Ford-Fulkerson

Equals to Edmonds-Karps, but with a DFS.  $O(|E|f^*) = O(|V||E|^2)$  where  $f^*$  is the value of the max flow.

```

int pcap;

int maxflowFF(TreeMap<Integer, Integer>[] g, int
source, int sink) {
    int flow = 0;
    pcap = Integer.MAX_VALUE;
    while(augmentDFS(g, source, sink, new boolean[g.
length])) {
        flow += pcap;
        pcap = Integer.MAX_VALUE;
    }
    return flow;
}

boolean augmentDFS(TreeMap<Integer, Integer>[] g,
int cur, int sink, boolean[] done) {
    if(cur == sink) return true;
    if(done[cur]) return false;
    done[cur] = true;
    for(Entry<Integer, Integer> e : g[cur].entrySet())
    {
        if(e.getValue() > 0) {
            int oldcap = pcap;
            pcap = Math.min(pcap, e.getValue());
            if(augmentDFS(g, e.getKey(), sink, done)) {
                g[cur].put(e.getKey(), e.getValue() - pcap);
                Integer backcap = g[e.getKey()].get(cur);
                g[e.getKey()].put(cur, backcap == null? pcap
: backcap + pcap);
                return true;
            }
        }
    }
}

```

```

} else {
    pcap = oldcap;
}
}
}
return false;
}

```

## 2.8.3 Min cut

We search, between two nodes  $s$  and  $t$ , subsets of nodes  $V_1$  and  $V_2$  so as  $s \in V_1$ ,  $t \in V_2$  and  $\sum_{e \in E(V_1, V_2)} w(e)$  minimum. We just have to compute the max-flow between  $s$  and  $t$  and to apply a BFS/DFS on the residual graph. All node which are visited are in  $V_1$ , others in  $V_2$ . The weight from the cut is the max-flow.

## 2.8.4 Maximum number of disjoint paths

For edge disjoint paths just compute the max flow with unit capacities. For vertex disjoint paths split vertices into two with unit capacity edge between them.

## 2.8.5 Maximum weighted bipartite matching

Assignment problem: Given a set of  $n$  persons and  $n$  jobs, and a cost matrix  $M$ , assign a job to each person such that the sum of the costs is minimized. It also works for  $n$  persons and  $m$  jobs with  $n \neq m$ . Just fill make a square matrix using dummy values. Can also be solve with min cost max flow but it is slower.

$O(n^3)$  solution:

```

static int[][] cost;
static int n;
static int[] lx, ly;
static int maxMatch;
static boolean[] S, T;
static int[] slack, slackx, prev, xy, yx;

static int minHungarian(int[][] M) {
    for(int i = 0; i < M.length; i++)
        for(int j = 0; j < M.length; j++)
            M[i][j] = -M[i][j];
    return maxHungarian(M);
}

static int maxHungarian(int[][] M) {
    cost = M;
    n = cost.length;
    slack = new int[n];
    slackx = new int[n];
    prev = new int[n];
    xy = new int[n];
    yx = new int[n];
    maxMatch = 0;
    for(int i = 0; i < n; i++) {
        xy[i] = -1;
        yx[i] = -1;
    }
    initLabels();
    augment();
    int ret = 0;
    int[] assignment = new int[n];
    for(int x = 0; x < n; x++) {
        ret += cost[x][xy[x]];
        assignment[x] = xy[x];
    }
    return assignment;
}

static void initLabels() {
    lx = new int[n];
    ly = new int[n];
}

```

```

    for(int x = 0; x < n; x++)
        for(int y = 0; y < n; y++)
            lx[x] = Math.max(lx[x], cost[x][y]);
}

static void augment() {
    if(maxMatch == n) {return;}
    int x, y, root = 0;
    int[] q = new int[n];
    int wr = 0, rd = 0;
    S = new boolean[n];
    T = new boolean[n];
    for(x = 0; x < n; x++) {
        prev[x] = -1;
        for(x = 0; x < n; x++) {
            if(xy[x] == -1) {
                q[wr++] = root = x;
                prev[x] = -2;
                S[x] = true;
                break;
            }
        }
        for(y = 0; y < n; y++) {
            slack[y] = lx[root] + ly[y] - cost[root][y];
            slackx[y] = root;
        }
        while(true) {
            while(rd < wr) {
                x = q[rd++];
                for(y = 0; y < n; y++) {
                    if(cost[x][y] == lx[x] + ly[y] && !T[y]) {
                        if(yx[y] == -1) {break;}
                        T[y] = true;
                        q[wr++] = yx[y];
                        addToTree(yx[y], x);
                    }
                }
            }
            if (y < n) {break;}
        }
        if (y < n) {break;}
        updateLabels();
        wr = rd = 0;
        for (y = 0; y < n; y++) {
            if (!T[y] && slack[y] == 0) {
                if(yx[y] == -1) {
                    x = slackx[y];
                    break;
                } else {
                    T[y] = true;
                    if(!S[yx[y]]) {
                        q[wr++] = yx[y];
                        addToTree(yx[y], slackx[y]);
                    }
                }
            }
        }
        if(y < n) {break;}
    }
    if(y < n) {
        maxMatch++;
        for(int cx=x, cy=y, ty; cx!=-2; cx=prev[cx], cy=ty){
            ty = xy[cx];
            yx[cy] = cx;
            xy[cx] = cy;
        }
        augment();
    }
}

static void updateLabels() {
    int delta = Integer.MAX_VALUE;
    for(int y = 0; y < n; y++)
        if(!T[y])
            delta = Math.min(delta, slack[y]);
    for(int i = 0; i < n; i++) {
        if(S[i]) {lx[i] -= delta;}
        if(T[i]) {ly[i] += delta;}
        if(!T[i]) {slack[i] -= delta;}
    }
}

```

```

    }
}

static void addToTree(int x, int prevx) {
    S[x] = true;
    prev[x] = prevx;
    for(int y = 0; y < n; y++) {
        if(lx[x] + ly[y] - cost[x][y] < slack[y]) {
            slack[y] = lx[x] + ly[y] - cost[x][y];
            slackx[y] = x;
        }
    }
}

O(n2n) solution using DP (very simple to code):
int n;
double[][] w;
Double[] memo;

double minCostMatching(int paired) {
    if(memo[paired] != null) return memo[paired];
    if(paired == (1 << n) - 1) return 0.0;
    double min = Double.POSITIVE_INFINITY;
    int i = 0;
    while(((paired >> i) & 1) == 1) i++;
    for(int j = i + 1; j < n; j++) {
        if(((paired >> j) & 1) == 0) {
            min = Math.min(min, w[i][j] + minCostMatching(
                paired | (1 << i) | (1 << j)));
        }
    }
    memo[paired] = min;
    return min;
}

```

## 2.9 Directed Min cost flow

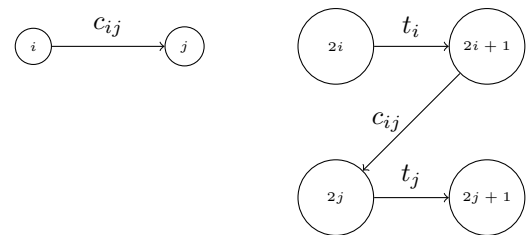
Avoiding parallel edges:

1. Split nodes



where  $t_i$  is the number of times node  $i$  can be used (usually  $\infty$ ).

2. Link nodes



```

    TreeMap<Integer, Edge>[] preprocess(TreeMap<Integer,
        Edge>[] g) {
        TreeMap<Integer, Edge>[] h =
            new TreeMap<Integer, Edge>[2*g.length];
        for(int v = 0; v < h.length; v++)
            h[v] = new TreeMap<Integer, Edge>();
        for(int v = 0; v < g.length; v++) {
            for(Entry<Integer, Edge> entry : g[v].entrySet()) {
                int u = entry.getKey();
                Edge e = entry.getValue();
                h[2*v+1].put(2*u, e);
            }
            h[2*v].put(2*v+1, new Edge(Integer.MAX_VALUE, 0));
        }
        return h;
    }
}

```



Min cost flow analogous to max flow but using Bellman-Ford to find paths (can be made faster using Dijkstra by chaining costs).

```
int[] p;

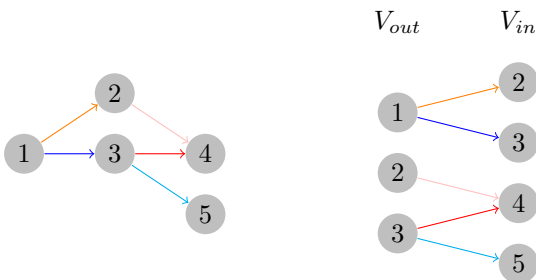
int minCostFlow(TreeMap<Integer, Edge>[] g, int s,
    int t) {
    int mincost = 0;
    while(spfa(g, s) != null && p[t] != -1) {
        // compute path capacity
        int cur = t;
        int pcap = Integer.MAX_VALUE;
        while(cur != s) {
            int prev = p[cur];
            pcap = Math.min(pcap, g[prev].get(cur).cap);
            cur = prev;
        }
        // update graph
        cur = t;
        int pcost = 0;
        while(cur != s) {
            int prev = p[cur];
            Edge epath = g[prev].get(cur);
            pcost += epath.cost * pcap;
            // update current edge
            if(epath.cap == pcap) g[prev].remove(cur);
            else epath.cap -= pcap;
            // update reverse edge
            Edge eback = g[cur].get(prev);
            if(eback != null) eback.cap += pcap;
            else g[cur].put(prev, new Edge(pcap, -epath.cost));
            cur = prev;
        }
        mincost += pcost;
    }
    return mincost;
}
```

Some changes to SPFA may be necessary. Computation of global variable  $p$  containing parents is required.

## 2.10 DAG path cover

### 2.10.1 Cover vertex: disjoint paths

Build a bipartite graph as in the picture:



And compute the maximum bipartite matching. If the number of vertices is  $n$  and the matching is  $m$  then the answer is  $n - m$ .

### 2.10.2 Cover vertex: non-disjoint

Same algorithm but on the transitive closure. Transitive closure is the graph same graph with  $(v, u)$  connected if there is a path from  $v$  to  $u$ .

### 2.10.3 Cover edges: disjoint

No flow. This formula gives the number of paths:

$$\sum_{v \in V} \max(\text{out-degree}(v) - \text{in-degree}(v), 0)$$

## 2.11 Max-Flow with demands

### 2.11.1 Node demande

Intead of conservation constraints we have for all  $v \in V$ :

$$\text{flow-in}(v) - \text{flow-out}(v) = d_v$$

Add a node  $s^*$  connected to each node  $v$  with  $d_v < 0$  with an edge of capacity  $-d_v$ . Add a node  $t^*$  and connect each node with  $d_v > 0$  to it with an edge of capacity  $d_v$ . Solution exists iff

$$\text{max-flow}(s^*, t^*) = \text{in-capacity}(t^*)$$

### 2.11.2 Edge lower bounds

Add lower bound  $l_e$  to each edge. Constraint becomes

$$l_e \leq f(e) \leq c_e$$

To change into max-flow: (1) define

$$L_v = \sum_{e \text{ into } v} l_e - \sum_{e \text{ out of } v} l_e;$$

(2) set demands  $d'_v = d_v - L_v$  where  $d_v$  are the input demands (usually 0); (3) set  $c'_e = c_e - l_e$ ; (4) solve max flow with node demands  $d'_v$  and capacities  $c'_e$ .

## 2.12 Chinese Postman Problem

Given an undirected weighted graph, compute the minimum length tour that visits every edge (edges may be visited several times, unavoidable if odd degree vertices exist). The number of odd degree vertices is even. Hence we can compute the minimum weight bipartite matching between them where  $w_{ij}$  is the length of the shortest path between  $i$  and  $j$ . Then the length of the tour is given by the sum of the lengths of all edges plus the weight of the matching.

## 2.13 Bipartite graph

Check if bipartite

```
boolean isBipartite(LinkedList<Integer>[] g)
{
    int[] d = bfs(g);
    for(int u = 0; u < g.length; u++)
        for(Integer v: g[u])
            if((d[u]%2) != (d[v]%2)) return false;
    return true;
}
```

### 2.13.1 Max Cardinality Bipartite Matching (MCBM)

Pairing of adjacent nodes. No node in two different pairs.

- Max Flow.
- Augmenting Path: path starting at non matched, ending at non-matched, even edges are matching. MCBM ssi no augmenting path. Start from non-matched, if augmenting path, augment (do not have to take all matching in the augmenting path).

MCBM : Number of matching.

Hungarian algorithm  $O(|V||E|)$ :



```

static int n; // V
static int m; // vertex on the left subset of V
static LinkedList<Integer>[] g;
static int[] match;
static BitSet visited;

private static int Aug(int left) {
    if (visited.get(left)) return 0;
    visited.set(left);

    for (int right : g[left]) {
        if (match[right] == -1 || Aug(match[right]) ==
            1) {
            match[right] = left;
            return 1; // we found one matching
        }
    }

    return 0; // no matching
}

static int hungarian () {
    int MCBM = 0;
    match = new int[n];
    for (int i = 0; i < n; i++) {
        match[i] = -1;
    }
    for (int l = 0; l < m; l++) {
        visited = new BitSet(n);
        MCBM += Aug(l);
    }
    return MCBM;
}

```

Hopcroft-Karp algorithm  $O(\sqrt{|V|}|E|)$ :

```

static int n;
static LinkedList<Integer>[] g;
static Integer[] match;
static int INF;
static int[] dist;
static BitSet left;

static boolean BFS () {
    Queue<Integer> q = new LinkedList<Integer>();
    dist = new int[n];
    for (int u = 0; u < n; u++) {
        if (left.get(u)) {
            if (match[u] == null) {
                dist[u] = 0;
                q.add(u);
            } else
                dist[u] = INF;
        }
    }
    int found = INF;
    while (!q.isEmpty()) {
        int u = q.poll();
        if (dist[u] < found) {
            for (int v : g[u]) {
                if (match[v] == null) {
                    if (found == INF)
                        found = dist[u] + 1;
                } else if (dist[match[v]] == INF) {
                    dist[match[v]] = dist[u] + 1;
                    q.add(match[v]);
                }
            }
        }
    }
    return found != INF;
}

static boolean DFS (Integer u) {
    if (u != null) {
        for (int v : g[u]) {
            if (match[v] == null || dist[match[v]] == dist
                [u] + 1)
                if (DFS(match[v])) {
                    match[v] = u;
                    match[u] = v;
                }
        }
    }
}

```

```

        return true;
    }
    }
    dist[u] = INF;
    return false;
}
return true;
}

static void left_right () {
    BitSet vis = new BitSet(n);
    left = new BitSet(n);
    Queue<Integer> q = new LinkedList<Integer>();
    for (int i = 0; i < n; i++) {
        if (vis.get(i)) continue;
        vis.set(i);
        left.set(i);
        q.add(i);
        while (!q.isEmpty()) {
            int cur = q.poll();
            for (int next : g[cur]) {
                if (!vis.get(next)) {
                    vis.set(next);
                    if (!left.get(cur))
                        left.set(next);
                    q.add(next);
                }
            }
        }
    }
}

static int hopcroftKarp () {
    left_right();
    INF = n+1;
    match = new Integer[n];
    int MCBM = 0;
    while (BFS())
        for (int u = 0; u < n; u++)
            if (left.get(u) && match[u] == null)
                if (DFS(u))
                    MCBM++;
    return MCBM;
}

```

### 2.13.2 Independent Set (or Dominating Set)

Set of vertices with no edges between them. MIS, add a vertex create an edge. In **bipartite** graph, MIS + MCBM = V.

### 2.13.3 Vertex Cover

Vertices such that each edge is adjacent to at least one vertex. Min Vertex Cover (MVC). In **bipartite** graph, MVC = MCBM.

In **general** graph, MIS + MVC = |V| and the MVC is the complementary of MIS.

## 3 Dynamic programming

### 3.1 Bottom-up

Give n objects of value  $v[i]$  to 3 people such that  $\max_i V_i - \min_i V_i$  is minimum ( $V_i$  is total value for person  $i$ ).

$canDo[i][v_1][v_2] = 1$  if we can give the objects  $0, 1, \dots, i$  such that  $v_1$  is going to  $P_1$  and  $v_2$  to  $P_2$ , 0 otherwise.  $v_3$  is determined from the sum.

Base case  $i = 0$ :

- $canDo[0][0][0] = 1$

- $canDo[0][v[0]][0] = 1$

- $canDo[0][0][v[0]] = 1$

Case  $i \geq 1$ :

$$canDo[i][v_1][v_2] =$$

$$canDo[i-1][v_1][v_2] \vee$$

$$canDo[i-1][v_1-v[i]][v_2] \vee$$

$$canDo[i-1][v_1][v_2-v[i]]$$

**Sol. :**  $\min_{v_1, v_2: \text{canDo}[n-1][v_1][v_2]} [\max(v_1, v_2, S - v_1 - v_2) - \min(v_1, v_2, S - v_1 - v_2)]$

```
int solveDP() {
    boolean[][][] canDo = new boolean[v.length][sum + 1][sum + 1];
    // initialize base cases
    canDo[0][0][0] = true;
    canDo[0][v[0]][0] = true;
    canDo[0][0][v[0]] = true;
    // compute solutions using recurrence relation
    for(int i = 1; i < v.length; i++) {
        for(int a = 0; a <= sum; a++) {
            for(int b = 0; b <= sum; b++) {
                boolean giveA = a - v[i] >= 0 && canDo[i - 1][a - v[i]][b];
                boolean giveB = b - v[i] >= 0 && canDo[i - 1][a][b - v[i]];
                boolean giveC = canDo[i - 1][a][b];
                canDo[i][a][b] = giveA || giveB || giveC;
            }
        }
    }
    // compute best solution
    int best = Integer.MAX_VALUE;
    for(int a = 0; a <= sum; a++) {
        for(int b = 0; b <= sum; b++) {
            if(canDo[v.length - 1][a][b]) {
                best = Math.min(best, max(a, b, sum - a - b) - min(a, b, sum - a - b));
            }
        }
    }
    return best;
}
```

### 3.2 Top-down

Same problem as bottom-up. Main idea : memoization (Remember intermediate results).

```
int solve(int i, int a, int b) {
    if(i == n) {
        memo[i][a][b] = max(a, b, sum - a - b) - min(a, b, sum - a - b);
        return memo[i][a][b];
    }
    if(memo[i][a][b] != null) {
        return memo[i][a][b];
    }
    int giveA = solve(i + 1, a + v[i], b);
    int giveB = solve(i + 1, a, b + v[i]);
    int giveC = solve(i + 1, a, b);
    memo[i][a][b] = min(giveA, giveB, giveC);
    return memo[i][a][b];
}
```

### 3.3 Knapsack problem

Given  $n$  objects of value  $v[i]$  and weight  $w[i]$ , an integer  $W$  :

- Maximize  $\sum_i x[i]v[i]$
- Such that  $\sum_i x[i]w[i] \leq W$  where  $x[i] = 0$  (not taken) or 1 (taken)

#### 3.3.1 No repetition

$\text{best}[i][w]$  = best way to take objects  $0, 1, \dots, i$  in a knapsack of capacity  $w$ .

**Base case:**

- $\text{best}[0][w] = v[0]$  si  $w[0] \leq w$
- 0 else

**Other cases:**

$$\text{best}[i][w] = \max\{\text{best}[i-1][w], \text{best}[i-1][w - w[i]] + v[i]\}$$

#### 3.3.2 An object can be repeated

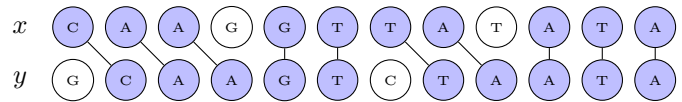
- $\text{best}[0] = 0$
- $\text{best}[w] = \max_{i: w[i] \leq w} \{\text{best}[w - w[i]] + v[i]\}$

#### 3.3.3 Several knapsacks

$\text{best}[i][w_1][w_2]$  = best way to take objects  $0, 1, \dots, i$  in knapsacks of capacity  $w_1$  and  $w_2$ .

### 3.4 Longest common sub-sequence (LCS)

Given two String  $x$  and  $y$ . Find the longest common sub-sequence between  $x$  and  $y$ .



- **Formulation:**  $\text{lcs}[i][j]$  = size of  $\text{LCS}(x[0]x[1] \dots x[i-1], y[0]y[1] \dots y[j-1])$
- **Base case:**  $\text{lcs}[0][j] = 0$        $\text{lcs}[i][0] = 0$
- **Other cases:**

- Si  $x[i-1] = y[j-1]$  alors:  
 $\text{lcs}[i][j] = 1 + \text{lcs}[i-1][j-1]$
- Si  $x[i-1] \neq y[j-1]$  alors:  
 $\text{lcs}[i][j] = \max\{\text{lcs}[i-1][j], \text{lcs}[i][j-1]\}$

### 3.5 Matrix Chain Multiplication (MCM)

Given a list of matrices, find the order minimizing the number of multiplications to compute their product.

- Number to multiply a matrix of size  $n \times m$  by a matrix of size  $m \times r$  :  $n \cdot m \cdot r$ .
- Example:  $A : 10 \times 30$ ,  $B : 30 \times 5$  et  $C : 5 \times 60$ .
  - For  $(AB)C$  :  $10 \cdot 30 \cdot 5 + 10 \cdot 5 \cdot 60 = 4500$  multiplications.
  - For  $A(BC)$  :  $30 \cdot 5 \cdot 60 + 10 \cdot 30 \cdot 60 = 27000$  multiplications.

- **Formulation :**  $\text{best}[i][j]$  = min cost to multiply  $A_i, \dots, A_j$

- **Base case :**  $\text{best}[i][i] = 0$

- **Other cases:**

$$\text{best}[i][j] = \min_{i \leq k < j} \text{best}[i][k] + \text{best}[k+1][j] + A_i.n_1 \times A_k.n_2 \times A_j.n_2$$

### 3.5.1 Generalized MCM

Given a list of objects  $x[0], \dots, x[n-1]$  and an operation  $\odot$  with an associated cost, find the order in which perform the operations to minimize the total cost. The matrix product is replaced by  $\odot$ .

$$best[i][j] = \min_{i \leq k < j} best[i][k] + best[k+1][j] + cost(i, j, k)$$

$cost(i, j, k)$  is the cost of  $(x[i] \odot \dots \odot x[k]) \odot (x[k+1] \odot \dots \odot x[j])$ .

```
int bestParenthesize() {
    int n = x.length; // x is a global variable
    int[][] best = new int[n][n];
    for(int i = 0; i < n; i++) {
        best[i][i] = 0;
    }
    for(int l = 1; l <= n; l++) {
        for(int i = 0; i < n - l; i++) {
            int j = i + l;
            int min = Integer.MAX_VALUE;
            for(int k = i; k < j; k++) {
                min = Math.min(min, best[i][k] + best[k+1][j] + cost(i, j, k)); // cost is problem-independent
            }
            best[i][j] = min;
        }
    }
    return best[0][n-1];
}
```

### 3.6 Edit distance

Given two String  $x$  and  $y$ , by performing operations on  $x$ , compute the minimal cost to transform  $x$  into  $y$ .

We can (operation cost):

1. Remove a character (D=1)
2. Insert a character (I=1)
3. Replace a character (R=2)

• **Formulation:**  $editDist[i][j] = \min$ . cost to transform  $x_0 \dots x_{i-1}$  into  $y_0 \dots y_{j-1}$

• **Base case:**

$$editDist[i][0] = i \cdot D \quad editDist[0][j] = j \cdot I$$

• **Other cases:**

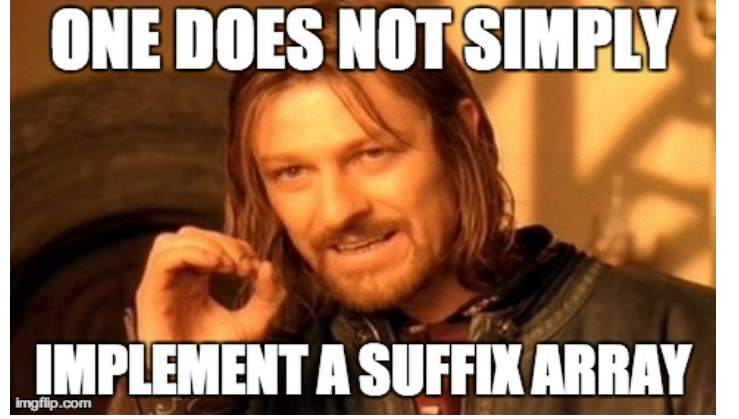
$$editDist[i][j] = \min \begin{cases} editDist[i-1][j] + D, \\ editDist[i][j-1] + I, \\ editDist[i-1][j-1] + R^* \end{cases}$$

where  $R^* = R$  if  $x[i-1] \neq y[j-1]$ , 0 else.

```
int editDistance(String txt1, String txt2, int I,
    int D, int R){
    int[][] d = new int[txt1.length()+1][txt2.length()+1];
    for(int i=0; i <= txt1.length(); i++)
        d[i][0] = i*D;
    for(int j=0; j <= txt2.length(); j++)
        d[0][j] = j*I;
    for(int i=1; i <= txt1.length(); i++){
        for(int j=1; j <= txt2.length(); j++){
            int cost;
            // Non-equality cost
```

```
if(txt1.charAt(i-1) == txt2.charAt(j-1))
    cost = 0;
else
    cost = R;
// Deletion, Insertion, Replacement
d[i][j] = Math.min(Math.min(d[i-1][j] + D, d[i][j-1] + I), d[i-1][j-1] + cost);
}
}
// Last computed element is the edit distance
return d[txt1.length()][txt2.length()];
}
```

### 3.7 Suffix array



#### 3.7.1 $O(n \log(n)^2)$ , full matrix, need $n \leq 10K$

- Suffix array of *algorithm* = algorithm, gorithm, hm, ithm, lgorithm, m, orithm, rithm, thm
- Characterized by its starting index  
Example : Suffix array of *algorithm*:

[0, 2, 7, 5, 1, 8, 3, 4, 6]

Example : Given  $suf_j$  suffix beginning at index  $j$ , and  $C(i, j, k)$  comparison result of  $suf_j$  and  $suf_k$  on the  $2^i$  first characters.

$$C(i, j, k) = \begin{cases} C(i-1, j, k) & \text{si } C(i-1, j, k) \neq 0 \\ C(i-1, j+2^{i-1}, k+2^{i-1}) & \text{else} \end{cases}$$

- Define a matrix *so* such that:

$$\begin{aligned} so[i][j] = so[i][k] &\Leftrightarrow C(i, j, k) = 0 \\ so[i][j] < so[i][k] &\Leftrightarrow C(i, j, k) < 0 \\ so[i][j] > so[i][k] &\Leftrightarrow C(i, j, k) > 0 \end{aligned}$$

$so[i]$  is the order of sorted suffixes on the  $2^i$  first characters.

- **Base case:**  $so[0][j] = (int)s.charAt(j)$

Example: for  $s = ccacab$  we have  
 $s[0] = [97, 97, 95, 97, 95, 96]$

- For every  $j$  we define a triplet  $(l, r, j)$ :

$$\begin{aligned} (s[i-1][j], s[i-1][j+2^{i-1}], j) & \quad \text{si } j+2^{i-1} < n \\ (s[i-1][j], -1, j) & \quad \text{si } j+2^{i-1} \geq n \end{aligned}$$

```

class Triple implements Comparable<Triple> {
    int l, r, index;
    public Triple(int half1, int half2, int index) {
        this.l = half1;
        this.r = half2;
        this.index = index;
    };
    public int compareTo(Triple other) {
        if(l != other.l) {
            return l - other.l;
        }
        return r - other.r;
    }
}

int [][] suffixOrder(String s) { // O(n log^2(n))
    int n = s.length();
    int lg = (int) Math.ceil((Math.log(n) / Math.log(2)) + 1);
    int [][] so = new int[lg][n];
    // initialize so[0] with character order
    for(int i = 0; i < n; i++) {
        so[0][i] = s.charAt(i);
    }
    Triple[] next = new Triple[n];
    for(int i = 1; i < lg; i++) {
        // build the next array
        for(int j = 0; j < n; j++) {
            int k = j + (1 << (i - 1));
            next[j] = new Triple(so[i - 1][j], k < n ? so[i - 1][k] : -1, j);
        }
        // sort next array
        Arrays.sort(next);
        // build so[i]
        for(int j = 0; j < n; j++) {
            if(j == 0) {
                // smallest elements gets value 0
                so[i][next[j].index] = 0;
            } else if(next[j].compareTo(next[j - 1]) == 0) {
                // equal to previous so it gets the same value
                so[i][next[j].index] = so[i][next[j - 1].index];
            } else {
                // largest than previous so get + 1
                so[i][next[j].index] = so[i][next[j - 1].index] + 1;
            }
        }
    }
    return so;
}

//Calcule le Suffix Array pour un so donne:
int [] suffixArray(int [][] so) {
    int [] sa = new int[so[0].length];
    for(int j = 0; j < so[0].length; j++) {
        sa[so[so.length - 1][j]] = j;
    }
    return sa;
}

//Retourne le plus long prefixe commun de suf_j (le
// suffixe de s commençant a j = s.substr(j)) et
// suf_k pour un so donne:
int lcp(int [][] so, int j, int k) { // O(log(n))
    int lcp = 0;
    int n = so[0].length;
    for(int i = so.length - 1; i >= 0; i--) {
        if(j < n && k < n && so[i][j] == so[i][k]) {
            lcp += (1 << i);
            j += (1 << i);
            k += (1 << i);
        }
    }
    return lcp;
}

```

```

//Quelques exemples
String maxStrRepeatedKTimes(String s, int k) {
    int [][] so = suffixOrder(s);
    int [] SA = suffixArray(so);
    int n = s.length();
    int max = Integer.MIN_VALUE;
    int j = 0;
    for(int i = 0; i <= n - k; i++) {
        int lcp = lcp(so, SA[i], SA[i + k - 1]);
        if(lcp > max) {
            max = lcp;
            j = SA[i];
        }
    }
    return s.substring(j, j + max);
}

String minLexicographicRotation(String s) {
    int n = s.length();
    s += s;
    int [] SA = suffixArray(suffixOrder(s));
    int i = 0;
    while(!(0 <= SA[i] && SA[i] < n)) {
        i++;
    }
    return s.substring(SA[i], SA[i] + n);
}

class MaxLexConc implements Comparator<String> {
    public int compare(String x, String y) {
        String xy = x + y;
        String yx = y + x;
        if(xy.compareTo(yx) < 0 ||
            (xy.equals(yx) && x.length() < y.length())) {
            return 1;
        }
        return -1;
    }
}

3.7.2 O(n log(n)), only last line, need n ≤ 100K

static final int MAXN = 100010;
static Integer[] tempSA, sa;
static int[] c, ra;
static int[] lcp, plcp;
static void countingSort(int n, int k) {
    int i, sum, maxi = Math.max(300, n); // up to 255
    // ASCII chars or length of n
    for (i = 0; i < MAXN; i++) c[i] = 0; // clear
    // frequency table
    for (i = 0; i < n; i++) // count the frequency of
    // each rank
        c[i + k < n ? ra[i + k] : 0]++;
    for (i = sum = 0; i < maxi; i++) {
        int t = c[i]; c[i] = sum; sum += t;
    }
    for (i = 0; i < n; i++) // shuffle
    // the suffix array if necessary
        tempSA[c[sa[i] + k < n ? ra[sa[i] + k] : 0]++] =
        sa[i];
    for (i = 0; i < n; i++)
        // update the suffix array SA
        sa[i] = tempSA[i];
}

static void constructSA(char[] s) { // O(n log(n))
    // → n ≤ 100K
    int i, k, r, n = s.length;
    tempSA = new Integer[n]; sa = new Integer[n];
    ra = new int[n]; int[] tempRA = new int[n];
    c = new int[MAXN];
    for (i = 0; i < n; i++) ra[i] = s[i];
    // initial rankings
    for (i = 0; i < n; i++) sa[i] = i; //
    // initial SA: {0, 1, 2, ..., n-1}
    for (k = 1; k < n; k <= 1) { // repeat
        // sorting process log n times
        countingSort(n, k); // actually radix sort
    }
}

```

```

: sort based on the second item
countingSort(n, 0); // then (
stable) sort based on the first item
tempRA[sa[0]] = r = 0; // re-
ranking; start from rank r = 0
for (i = 1; i < n; i++)
// compare adjacent suffixes
tempRA[sa[i]] = // if same pair => same
rank r; otherwise, increase r
(ra[sa[i]] == ra[sa[i-1]] && ra[sa[i]+k] == ra
[sa[i-1]+k]) ? r : ++r;
for (i = 0; i < n; i++)
// update the rank array RA
ra[i] = tempRA[i];
} }

static void computeLCP(char[] s) {
int i, L, n = s.length;
int[] phi = new int[n];
lcp = new int[n]; plcp = new int[n];
phi[sa[0]] = -1; // default value
for (i = 1; i < n; i++) // compute Phi in O(n)
phi[sa[i]] = sa[i-1]; // remember which suffix
is behind this suffix
for (i = L = 0; i < n; i++) { // compute Permuted
LCP in O(n)
if (phi[i] == -1) { plcp[i] = 0; continue; } //
special case
while (i + L < n && phi[i] + L < n && s[i + L]
== s[phi[i] + L]) L++; // L will be increased
max n times
plcp[i] = L;
L = Math.max(L-1, 0); // L will be decreased max
n times
}
for (i = 1; i < n; i++) // compute LCP in O(n)
lcp[i] = plcp[sa[i]]; // put the permuted LCP
back to the correct position
}

static int strncmp(char[] a, int i, char[] b, int j,
int n){
for (int k=0; i+k < a.length && j+k < b.length; k
++){
if (a[i+k] != b[j+k]) return a[i+k] - b[j+k];
}
return 0;
}

static int[] stringMatching(char[] s, char[] p) {
// string matching in O(m log n)
int n = s.length, m = p.length;
constructSA(s);
int lo = 0, hi = n-1, mid = lo; // valid matching
= [0 .. n-1]
while (lo < hi) { // find lower bound
mid = (lo + hi) / 2;
int res = strncmp(s, sa[mid], p, 0, m); // try
to find P in suffix 'mid'
if (res >= 0) hi = mid;
else lo = mid + 1;
}
if (strncmp(s, sa[lo], p, 0, m) != 0) return new int
[]{-1, -1}; // not found
int[] ans = new int[] { lo, 0 } ;

lo = 0; hi = n - 1; mid = lo;
while (lo < hi) { // if lower bound is found, find
upper bound
mid = (lo + hi) / 2;
int res = strncmp(s, sa[mid], p, 0, m);
if (res > 0) hi = mid;
else lo = mid + 1;
}
if (strncmp(s, sa[hi], p, 0, m) != 0) hi--; //
special case
ans[1] = hi;
return ans;
} // return lower/upper bound as the first/second

```

```

item of the pair, respectively

static String LRS(char[] s) { // Longest Repeating
substring
int n = s.length;
constructSA(s);
computeLCP(s);
int i, idx = 0, maxLCP = 0;

for (i = 1; i < n; i++) // O(n)
if (lcp[i] > maxLCP) {
maxLCP = lcp[i];
idx = i;
}
return new String(s).substring(sa[idx], sa[idx]+
maxLCP);
}

static int owner(int idx, int n, int m) { return (idx
< n-m-1) ? 1 : 2; }

static String LCS(String T, String P) { // Longest
common substring
int i, idx = 0;

int m = P.length();
char[] s = (T + "$" + P + "#").toCharArray(); //
append P and '#'
int n = s.length; // update n
constructSA(s); // O(n log n)
computeLCP(s); // O(n)

int maxLCP = -1;
for (i = 1; i < n; i++)
if (lcp[i] > maxLCP && owner(sa[i], n, m) != owner
(sa[i-1], n, m)) { // different owner
maxLCP = lcp[i];
idx = i;
}

return new String(s).substring(sa[idx], sa[idx] +
maxLCP);
}

```

## 4 Geometry in 2D

Be careful of rounding errors. Define  $E$  in function of the problem. `Double.parseDouble` is a lot slower than `Integer.parseInt`

```

boolean eq(double a, double b){return Math.abs(a - b)
<= E;}
boolean le(double a, double b){return a < b - E;}
boolean leq(double a, double b){return a <= b + E;}

```

### 4.1 Vectors

#### 4.1.1 Rotation around (0,0)

$$(x, y) \leftrightarrow x + yi$$

$$\rho e^{i\theta} = \rho \cos(\theta) + i\rho \sin(\theta)$$

$(x, y)$  rotated by  $\alpha$  is  $(\cos(\alpha)x - \sin(\alpha)y, \sin(\alpha)x + \cos(\alpha)y)$

### 4.2 Points

```

class Point implements Comparable<Point>
{
double x, y;
public int compareTo(Point o) { //xcomp
if (a.x < b.x) return -1;
if (a.x > b.x) return 1;
if (a.y < b.y) return -1;
if (a.y > b.y) return 1;
return 0;
}
}

```

```

class yComp implements Comparator<Point> {
    public int compare(Point p, Point q) {
        if (p.y == q.y) {return Double.compare(p.x, q.x);
        }
        return Double.compare(p.y, q.y);
    }
}

```

#### 4.2.1 Point in box

```

boolean inBox(Point p1, Point p2, Point p) {
    return Math.min(p1.x, p2.x) <= p.x && p.x <= Math.
        max(p1.x, p2.x) &&
        Math.min(p1.y, p2.y) <= p.y && p.y <= Math.
        max(p1.y, p2.y);
}

```

#### 4.2.2 Polar sort

```

LinkedList<Point> sortPolar(Point[] P, Point o)
{
    LinkedList<Point> above = new LinkedList<Point>();
    LinkedList<Point> samePos = new LinkedList<Point>();
    LinkedList<Point> sameNeg = new LinkedList<Point>();
    LinkedList<Point> bellow = new LinkedList<Point>();
    for (Point p : P)
    {
        if (p.y > o.y)
            above.add(p);
        else if (p.y < o.y)
            bellow.add(p);
        else
        {
            if (p.x < o.x)
                sameNeg.add(p);
            else
                samePos.add(p);
        }
    }
    PolarComp comp = new PolarComp(o);
    Collections.sort(samePos, comp);
    Collections.sort(sameNeg, comp);
    Collections.sort(above, comp);
    Collections.sort(bellow, comp);
    LinkedList<Point> sorted = new LinkedList<Point>();
    for (Point p : samePos) sorted.add(p);
    for (Point p : above) sorted.add(p);
    for (Point p : sameNeg) sorted.add(p);
    for (Point p : bellow) sorted.add(p);
    return sorted;
}

```

```

class PolarCmp implements Comparator<Point> {
    static Point orig = new Point(0, 0);
    public int compare(Point p, Point q) {
        double o = orient(orig, p, q);
        if (o == 0) {
            if (p.x * p.x + p.y * p.y > q.x * q.x + q.y * q.y)
                return 1;
            return -1;
        }
        return -(int) Math.signum(o);
    }
}

```

#### 4.2.3 Closest pair of points

```

double closestPair(Point[] points) {
    if (points.length == 1) {return Double.
        POSITIVE_INFINITY;}
    Arrays.sort(points, new xComp());
    double min = dist(points[0], points[1]);
    // keep track of the leftmost point
    int leftmost = 0;
    TreeSet<Point> candidates = new TreeSet<Point>(new
        yComp());
    candidates.add(points[0]);
}

```

```

candidates.add(points[1]);
for (int i = 2; i < points.length; i++) {
    Point cur = points[i];
    // eliminate points s.t cur.x - x > min
    while (cur.x - points[leftmost].x > min) {
        candidates.remove(points[leftmost]);
        leftmost++;
    }
    Point low = new Point(0, cur.y - min);
    Point high = new Point(0, cur.y + min);
    // check all points in the rectangle
    for (Point point : candidates.subSet(low, high))
        min = Math.min(min, dist(cur, point));
    candidates.add(cur);
}
return min;
}

```

#### 4.2.4 Orientation

$$orient(p, q, r) = \begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}$$

$$orient(p, q, r) \begin{cases} = 0 & p, q, r \text{ are collinear} \\ < 0 & p \rightarrow q \rightarrow r \text{ is clockwise} \\ > 0 & p \rightarrow q \rightarrow r \text{ is counterclockwise} \end{cases}$$

$$|orient(p, q, r)| = 2 \cdot \text{area} \triangle(p, q, r)$$

```

double orient(Point p, Point q, Point r) {
    return q.x * r.y - r.x * q.y - p.x * (r.y - q.y) +
        p.y * (r.x - q.x);
}

```

#### 4.2.5 Angle visibility

$x$  lies strictly inside the angle formed by  $p, q, r$  iff

$$\begin{aligned} \text{sgn}(orient(p, q, x)) &= \text{sgn}(orient(p, x, r)) \\ \text{sgn}(orient(p, r, x)) &= \text{sgn}(orient(p, x, q)) \end{aligned}$$

To allow it to lie on the border simply check if

$$\text{sgn}(orient(p, q, x)) = 0 \text{ or } \text{sgn}(orient(p, r, x)) = 0$$

#### 4.2.6 Fixed radius neighbors (1D)

```

List<Double[]> findPairs1D(double[] x, double r) {
    HashMap<Integer, List<Double>> H = new HashMap<
        Integer, List<Double>>();
    // fill buckets
    for (int i = 0; i < x.length; i++) {
        int b = (int)(x[i] / r);
        if (H.containsKey(b)) {
            H.get(b).add(x[i]);
        } else {
            List<Double> L = new ArrayList<Double>();
            L.add(x[i]);
            H.put(b, L);
        }
    }
    // find pairs in consecutive buckets
    List<Double[]> pairs = new LinkedList<Double[]>();
    for (int i = 0; i < x.length; i++) {
        int b = (int)(x[i] / r);
        List<Double> bucket = H.get(b + 1);
        if (bucket != null)
            for (double y : bucket)
                if (y - x[i] <= r)
                    pairs.add(new Double[] {x[i], y});
    }
    // add points in buckets
    for (List<Double> bucket : H.values())
        for (int i = 0; i < bucket.size(); i++)
            for (int j = i + 1; j < bucket.size(); j++)

```



```

        pairs.add(new Double[] {bucket.get(i),
        bucket.get(j)});
    return pairs;
}

```

#### 4.2.7 Fixed radius neighbors (2D)

```

List<Point[]> findPairs2D(Point[] points, double r)
{
    HashMap<Integer, List<Point>> H = new HashMap<
    Integer, List<Point>>();
    // fill buckets
    for(int i = 0; i < points.length; i++) {
        int bx = (int)(points[i].x / r);
        int by = (int)(points[i].y / r);
        int key = 33 * bx + by;
        if(H.containsKey(key)) {
            H.get(key).add(points[i]);
        } else {
            List<Point> L = new ArrayList<Point>();
            L.add(points[i]);
            H.put(key, L);
        }
    }
    // find pairs in adjacent buckets
    List<Point[]> pairs = new LinkedList<Point[]>();
    int[][] dir = new int[][] {new int[] {1,0}, new
    int[] {0,1}, new int[] {1,1}};
    for(int i = 0; i < points.length; i++) {
        int bx = (int)(points[i].x / r);
        int by = (int)(points[i].y / r);
        for(int[] d : dir) {
            List<Point> bucket = H.get(33 * (bx + d[0]) +
            (by + d[1]));
            if(bucket != null)
                for(Point y : bucket)
                    if(sqDist(points[i], y) <= r * r)
                        pairs.add(new Point[] {points[i], y});
        }
    }
    // add points in buckets
    for(List<Point> bucket : H.values())
        for(int i = 0; i < bucket.size(); i++)
            for(int j = i + 1; j < bucket.size(); j++)
                if(sqDist(bucket.get(i), bucket.get(j)) <= r
                * r)
                    pairs.add(new Point[] {bucket.get(i),
                    bucket.get(j)});
    return pairs;
}

```

### 4.3 Lines

General equation:  $Ax + By = C$ . The line through  $(x_1, y_1), (x_2, y_2)$  is given by:  $A = y_2 - y_1$ ,  $B = x_1 - x_2$ ,  $C = Ax_1 + By_1$ .

#### 4.3.1 Intersections

Intersection exists there is a solution for  $A_1x + B_1y = C_1$  and  $A_2x + B_2y = C_2$ . This happens if and only if

$$d := \det \begin{pmatrix} A_1 & B_1 \\ A_2 & B_2 \end{pmatrix} \neq 0$$

Intersection is given by

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} A_1 & B_1 \\ A_2 & B_2 \end{pmatrix}^{-1} \begin{pmatrix} C_1 \\ C_2 \end{pmatrix} = \frac{1}{d} \begin{pmatrix} B_2 & -B_1 \\ -A_2 & A_1 \end{pmatrix} \begin{pmatrix} C_1 \\ C_2 \end{pmatrix}$$

#### 4.3.2 Perpendicular line

The lines perpendicular to  $Ax + By = C$  are

$$-Bx + Ay = D \quad \text{for } D \in \mathbb{R}$$

If we want the one that goes through  $(x_0, y_0)$  set

$$D = -Bx_0 + Ay_0$$

#### 4.3.3 Orthogonal Symmetry

For a line, find  $X'$ , the point which is the orthogonal symmetry of  $X$  on line  $a$ .

Computes the perpendicular of the given line that goes through  $X$ . Compute intersection  $Y$ .  $X' = Y - (X - Y)$ .

### 4.4 Segments

#### 4.4.1 Intersection

- Treat segments as lines.
- If  $d \neq 0$ , compute line intersection  $(x, y)$ .
- Segments intersect iff

$$\min(x_1, x_2) \leq x \leq \max(x_1, x_2)$$

$$\min(y_1, y_2) \leq y \leq \max(y_1, y_2)$$

```

boolean intersects(Point p1, Point p2, Point p3,
    Point p4) {
    double o1 = orient(p1, p2, p3);
    double o2 = orient(p1, p2, p4);
    double o3 = orient(p3, p4, p1);
    double o4 = orient(p3, p4, p2);
    // check first condition of the lemma
    if(o1 * o2 < 0 && o3 * o4 < 0) return true;
    // check seconds condition of the lemma
    if(o1 == 0 && inBox(p1, p2, p3)) return true;
    if(o2 == 0 && inBox(p1, p2, p4)) return true;
    if(o3 == 0 && inBox(p3, p4, p1)) return true;
    if(o4 == 0 && inBox(p3, p4, p2)) return true;
    return false;
}

```

#### 4.4.2 Intersections problem

Given a lot of segments, return true if it exists a pair that intersects.

```

boolean segmentIntersection(Segment[] S) {
    Event[] events = new Event[2 * S.length];
    // create event points
    for(int i = 0, j = 0; i < S.length; i++) {
        events[j++] = new Event(S[i].l.x, true, S[i]);
        events[j++] = new Event(S[i].r.x, false, S[i]);
    }
    Arrays.sort(events);
    SegmentCmp cmp = new SegmentCmp();
    TreeSet<Segment> T = new TreeSet<Segment>(cmp);
    // sweep line
    for(Event event : events) {
        Segment s = event.s;
        cmp.x = event.x;
        if(event.isLeft) {
            // new segment found. check if it intersects
            // one of its neighbors
            T.add(s);
            Segment above = T.higher(s);
            Segment below = T.lower(s);
            if((above != null && intersects(above, s)) ||
            (below != null && intersects(below, s)))
                return true;
        } else {
            // end of segment. check if its neighbors
            // intersect
            Segment above = T.higher(s);
            Segment below = T.lower(s);
            if(above != null && below != null &&
            intersects(above, below))
                return true;
            T.remove(s);
        }
    }
}

```

```

    return false;
}

class Event implements Comparable<Event> {
    double x;
    boolean isLeft;
    Segment s;
    public Event(double x, boolean isLeft, Segment s) {
        this.x = x;
        this.isLeft = isLeft;
        this.s = s;
    }
    public int compareTo(Event other) {
        int cmp = Double.compare(x, other.x);
        // ensure that left comes before right
        if(cmp == 0) return isLeft? -1 : 1;
        return cmp;
    }
    public String toString() {
        return x + " " + isLeft;
    }
}

class SegmentCmp implements Comparator<Segment> {
    double x;
    public int compare(Segment s1, Segment s2) {
        // compute A,B,C from eq Ax + by = C for each segment
        double A1 = s1.r.y - s1.l.y;
        double B1 = s1.l.x - s1.r.x;
        double C1 = A1 * s1.l.x + B1 * s1.l.y;

        double A2 = s2.r.y - s2.l.y;
        double B2 = s2.l.x - s2.r.x;
        double C2 = A2 * s2.l.x + B2 * s2.l.y;

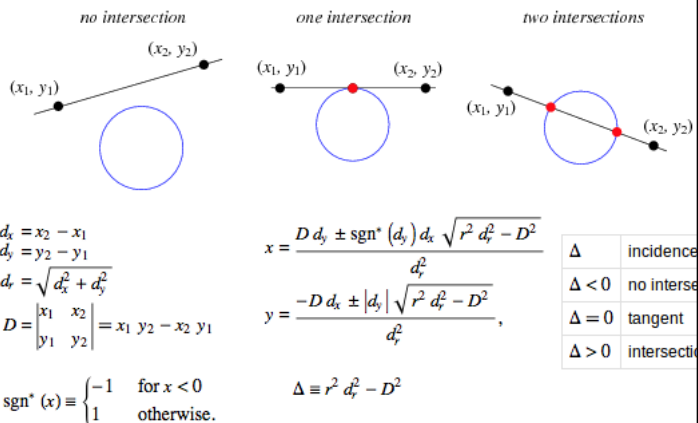
        // no divisions =)
        double t1 = B2 * (C1 - A1 * x);
        double t2 = B1 * (C2 - A2 * x);
        if(t1 == t2) {
            return s1 == s2? 0 : -1;
        } else if(B1 * B2 > 0) {
            return Double.compare(t1, t2);
        } else {
            return Double.compare(t2, t1);
        }
    }
}
    
```

## 4.5 Circles

### 4.5.1 Circles from 3 points

- 3 non collinear points define a unique circle.
- $c$  = intersection of bisectors of  $XY$  and  $YZ$ .

### 4.5.2 Circle-line intersection



### 4.5.3 Circle-circle or circle-point tangents

Find lines tangent to both circles  $(C_1, r_1)$  and  $(C_2, r_2)$ . Let  $d = |C_1 C_2|$ .

- **Inner tangents:** Condition:  $r_1 + r_2 \leq d$  (if equal, only one). Let  $\alpha = \text{acos}(\frac{r_1 + r_2}{d})$ , then the tangency two points  $T$  on either circle are such that  $\widehat{C_2 C_1 T} = \alpha$  and  $\widehat{C_1 C_2 T} = \alpha$  respectively.
- **Outer tangents:** Condition:  $|r_1 - r_2| \leq d$  (if equal, only one). Same, but with  $\widehat{C_2 C_1 T} = \text{acos}(\frac{r_1 - r_2}{d})$  and  $\widehat{C_1 C_2 T} = \text{acos}(\frac{r_2 - r_1}{d})$ .

For circle-point tangents, set  $r_2 = 0$  on inner tangents.

## 4.6 Polygons

### 4.6.1 Triangulation

A vertex  $i$  of a polygon is a ear if the triangle formed by vertices  $i-1, i$  and  $i+1$  is inside the polygon. Every polygon has at least two ears. Therefore to triangulate we can remove the ears until only a triangle remains. Any triangulation has always exactly  $n-2$  triangles. Implemented naively this gives a  $O(n^3)$  algorithm. Can be done in sweep line in  $O(n \log(n))$  but is it harder.

```

// assumes that pol is in counter-clockwise order
private static boolean ear(Point[] pol, int i) {
    int n = pol.length;
    int j = (i - 1 + n) % n;
    int k = (i + 1 + n) % n;
    // if ccw then points must also be ccw
    if(orient(pol[j], pol[i], pol[k]) < eps) return false;
    for(int m = 0; m < n; m++)
        // inTriangle not in the sheets. checks if pol[m]
        // is inside triangle pol[j]pol[i]pol[k]
        if(m != i && m != j && m != k && inTriangle(pol[m], pol[j], pol[i], pol[k]))
            return false;
    return true;
}
    
```

### 4.6.2 Triangles

- côtés  $a, b, c$ , angles  $A, B, C$ , hauteurs  $h_A, h_B, h_C$ ,  $s = \frac{a+b+c}{2}$ , aire  $S$ .
- Aire:  $S = \frac{ah_A}{2}$ ,  $S = ab \sin C / 2$ ,  $S = \sqrt{s(s-a)(s-b)(s-c)}$ .
- Inradius  $r = \frac{S}{s}$ .
- Outradius  $2R = \frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$ .
- $rR = \frac{abc}{4s}$ .

### 4.6.3 Check convexity

```

boolean isConvex(Point[] P) {
    if(P.length < 3) return false;
    double o1 = orient(P[P.length-1], P[0], P[1]);
    for (int i = 0; i < P.length; i++) {
        double o2 = orient(P[i], P[i+1], P[i+2]);
        if(o1 * o2 < 0) {
            return false;
        } else if (o2 != 0) {
            o1 = o2;
        }
    }
    return true;
}
    
```

#### 4.6.4 Winding number

Number of times a path of points "turn around" another point. (can check if a point is inside a polygon: in this case, winding numbe != 0)

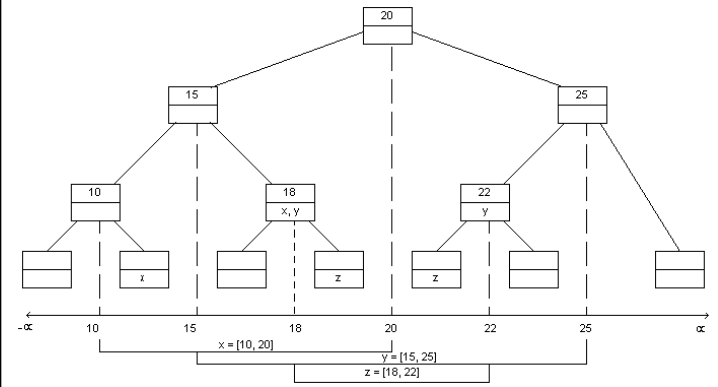
```
// assumes p is not on P
double winding(Point[] P, Point p) {
    //make a translation so p = (0, 0)
    for(Point q : P) {
        q.x -= p.x;
        q.y -= p.y;
    }
    double w = 0;
    for(int i = 0; i < P.length - 1; i++) {
        if(P[i].y * P[i+1].y < 0) {
            // segment crosses the x-axis
            double r = (P[i].y - P[i+1].y) * P[i].x + P[i+1].y * (P[i+1].x - P[i].x);
            //check for intersection with the positive x-axis
            if((P[i].y - P[i+1].y > 0 && r > 0) || (P[i].y - P[i+1].y < 0 && r < 0)) {
                // segment fully crosses the x-axis
                // - to + add 1, + to - subtract 1
                w += P[i].y < 0 ? 1 : -1;
            } else if(P[i].y == 0 && P[i].x > 0) {
                // the segment starts at the x-axis
                // 0 to + add 0.5, 0 to - subtract 0.5
                w += P[i+1].y > 0 ? 0.5 : -0.5;
            } else if(P[i+1].y == 0 && P[i+1].x > 0) {
                // the segment ends at the x-axis
                // - to 0 add 0.5, + to 0 subtract 0.5
                w += P[i].y < 0 ? 0.5 : -0.5;
            }
        }
    }
    return w;
}
```

#### 4.6.5 Convex Hull

```
Point[] convexHull(Point[] points) {
    // sort points by increasing x coordinates
    Arrays.sort(points, new xComp());
    // build upper chain
    Point[] upChain = buildChain(points, 1);
    // build lower chain
    Point[] loChain = buildChain(points, -1);
    Point[] hull = new Point[upChain.length + loChain.length - 2];
    int i;
    // build convex hull from upper and lower chain
    for(i = 0; i < upChain.length; i++) {
        hull[i] = upChain[i];
    }
    for(int j = loChain.length - 2; j >= 1; j--) {
        hull[i] = loChain[j]; i++;
    }
    return hull;
}

Point[] buildChain(Point[] points, int sgn) {
    Point[] S = new Point[points.length];
    int k = 0;
    S[k++] = points[0]; // push points[0]
    S[k++] = points[1]; // push points[1]
    // build chain
    for(int i = 2; i < points.length; i++) {
        //double orient = orient(S[k-2], S[k-1], points[i]);
        while(k >= 2 && sgn * orient(S[k-2], S[k-1], points[i]) >= 0) {
            S[k-1] = null; // pop
            k--;
        }
        S[k++] = points[i]; // push points[i]
    }
    return Arrays.copyOf(S, k);
}
```

#### 4.7 Interval Tree



```
class IntervalTree {
    Node root;
    public IntervalTree(int[] x) {
        root = new Node();
        buildTree(root, 0, x.length - 1, x);
    }
    public int measure() {
        return root.measure;
    }
    public void buildTree(Node node, int i, int j, int[] x) {
        if(j - i == 1) {
            node.l = x[i];
            node.r = x[j];
            node.m = -1;
        } else {
            node.l = x[i];
            node.r = x[j];
            int mid = (i + j) / 2;
            Node left = new Node();
            buildTree(left, i, mid, x);
            Node right = new Node();
            buildTree(right, mid, j, x);
            node.m = x[mid];
            node.left = left;
            left.parent = node;
            node.right = right;
            right.parent = node;
        }
    }
    public void remove(int x1, int x2) {
        remove(root, x1, x2);
    }
    private void remove(Node node, int x1, int x2) {
        if(node.l == x1 && node.r == x2) {
            node.count = Math.max(0, node.count - 1);
            if(node.left == null || node.right == null) {
                node.measure = node.count == 0 ? 0 : node.measure;
            } else {
                node.measure = node.count == 0 ? node.left.measure + node.right.measure : node.measure;
            }
        } else {
            // go down the tree to delete new interval
            int mid = node.m;
            if(x1 < mid && mid < x2) {
                // split
                remove(node.left, x1, mid);
                remove(node.right, mid, x2);
            } else if(node.l <= x1 && x2 <= mid) {
                // contained on left
                remove(node.left, x1, x2);
            } else {
                // contained on right
                remove(node.right, x1, x2);
            }
            // update measures when going up
            if(node.count == 0) {
                node.measure = node.left.measure + node.right.measure;
            }
        }
    }
}
```

```

    }
}
public void add(int x1, int x2) {
    add(root, x1, x2);
}
private void add(Node node, int x1, int x2) {
    if (node.l == x1 && node.r == x2) {
        node.measure = x2 - x1;
        node.count++;
    } else {
        // go down the tree to add new interval
        int mid = node.m;
        if (x1 < mid && mid < x2) {
            // split
            add(node.left, x1, mid);
            add(node.right, mid, x2);
        } else if (node.l <= x1 && x2 <= mid) {
            // contained on left
            add(node.left, x1, x2);
        } else {
            // contained on right
            add(node.right, x1, x2);
        }
        // update measures when going up
        if (node.count == 0) {
            node.measure = node.left.measure + node.
            right.measure;
        }
    }
}
}
public class Node {
    int l, r, m;
    int count, measure;
    Node left, right, parent;
}
}

```

## 4.8 Area of union of rectangles

```

long area(R[] r) {
    // sort y coordinates
    int[] y = new int[2 * r.length];
    int k = 0;
    for (R rect : r) {
        y[k++] = rect.y1;
        y[k++] = rect.y2;
    }
    Arrays.sort(y);
    // build interval tree
    IntervalTree T = new IntervalTree(y);
    // initialize event queue
    PriorityQueue<Event> Q = new PriorityQueue<Event>
    >();
    for (R rectangle : r) {
        Q.add(new Event(rectangle.x1, rectangle));
        Q.add(new Event(rectangle.x2, rectangle));
    }
    long area = 0;
    Event previous = null;
    // loop over all events
    while (!Q.isEmpty()) {
        // poll next event
        Event e = Q.poll();
        if (previous == null) {
            // first vertical line
            T.add(e.r.y1, e.r.y2);
        } else {
            // found a new vertical line
            // update area by dx * tree measure
            int dx = e.x - previous.x;
            area += dx * T.measure();
            if (e.x == e.r.x1) {
                // new rectangle, add segment to T
                T.add(e.r.y1, e.r.y2);
            } else {
                // exiting rectangle, remove segment from T
                T.remove(e.r.y1, e.r.y2);
            }
        }
    }
}

```

```

    // update previous
    previous = e;
}
return area;
}

class Event implements Comparable<Event> {
    int x;
    R r;
    public Event(int x, R r) {
        this.x = x;
        this.r = r;
    }
    public int compareTo(Event other) {
        return x - other.x;
    }
}

class R {
    int x1, y1, x2, y2;
    public R(int x1, int y1, int x2, int y2) {
        this.x1 = x1; this.y1 = y1; this.x2 = x2; this.y2 =
        y2;
    }
}

```

## 5 Geometry in 3D

### 5.1 Cross product

With vectors  $\vec{V}_1 = (a_1, b_1, c_1)$  and  $\vec{V}_2 = (a_2, b_2, c_2)$ :

$$\vec{V}_1 \times \vec{V}_2 = (b_1 c_2 - c_1 b_2, c_1 a_2 - a_1 c_2, a_1 b_2 - b_1 a_2)$$

### 5.2 Equation of a plane

#### 5.2.1 with a normal vector and a point

A plane is defined by a point  $(x_0, y_0, z_0)$  and an normal vector  $(a, b, c)$ .

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0$$

$$ax + by + cz = ax_0 + by_0 + cz_0 = d$$

#### 5.2.2 with a point and two vectors in the plane

A plane is defined by a point  $(x_0, y_0, z_0)$  and two vectors  $(\alpha_1, \beta_1, \gamma_1)$  and  $(\alpha_2, \beta_2, \gamma_2)$ . We obtain the parametric equations:

$$x = x_0 + t_1 \alpha_1 + t_2 \alpha_2$$

$$y = y_0 + t_1 \beta_1 + t_2 \beta_2$$

$$z = z_0 + t_1 \gamma_1 + t_2 \gamma_2$$

Or we can find the normal vector of the plane by doing the vector product of the two vectors

#### 5.2.3 with three points

Make vectors from these three points and use one of the methods above.

## 5.3 Equation of a line

### 5.3.1 With a point and a vector

A line is defined by a point  $(x_0, y_0, z_0)$  and a vector  $(a, b, c)$ .

$$x = x_0 + ta$$

$$y = y_0 + tb$$

$$z = z_0 + tc$$

### 5.3.2 With two points

$$x = x_1 + t(x_2 - x_1)$$

$$y = y_1 + t(y_2 - y_1)$$

$$z = z_1 + t(z_2 - z_1)$$

### 5.4 Distance from a point to a line

Distance from a point  $M_P = (x_p, y_p, z_p)$  to a line defined with a point  $M_L = (x_l, y_l, z_l)$  and a vector  $\vec{V} = (a, b, c)$  equals to

$$\frac{\|M_L \vec{M}_P \times \vec{V}\|}{\|\vec{V}\|}$$

### 5.5 Distance from a point to a plane

The distance to a plane is 0 if a point is in the plane.

$$\frac{|ax_p + by_p + cz_p - d|}{\sqrt{a^2 + b^2 + c^2}}$$

### 5.6 Orthogonal projection of a point on a line

If  $p_p$  is the point,  $s$  the direction vector of the line and  $p_l$  the base point for the vector, the projection is

$$\frac{(p_p - p_l) \cdot s}{s \cdot s} s + p_l$$

### 5.7 Orthogonal projection of a point on a plane

$$P_p = (x + \lambda a, y + \lambda b, z + \lambda c)$$

$$\lambda = -\frac{ax_p + by_p + cz_p - d}{a^2 + b^2 + c^2}$$

### 5.8 Orthogonal projection of a line on a plane

Take two points of the line, project them on the plane, recreate the line from the two new points.

### 5.9 Finding if a point is in a 3D polygon

Take any ray in the plane of the polygon, starting from the point you want to check (simply fix one of the coordinate of the point to find the ray); if it intersects an even number of time with the sides of the polygon, the point is inside it.

### 5.10 Intersection of a line and a plane

Given a plane  $ax + by + cz = d$  and a line with parametric equations:

$$x = x_0 + \alpha t$$

$$y = y_0 + \beta t$$

$$z = z_0 + \gamma t$$

The value of  $t$  associated with the intersection is

$$t = \frac{d - ax_0 - by_0 - cz_0}{a\alpha + b\beta + c\gamma}$$

## 6 Math

### 6.1 Permutations, Combinations, Arrangements... *untested*

```
void nextPerm(int[] p) {
    int n = p.length;
    int k = n - 2;
    while(k >= 0 && p[k] >= p[k + 1]) {k--;}
    int l = n - 1;
    while(p[k] >= p[l]) {l--;}
    swap(p, k, l);
    reverse(p, k + 1, n);
}
```

```
LinkedList<Integer> getIPermutation(int n, int index) {
    LeftRightArray lr = new LeftRightArray(n);
    lr.freeAll();
    LinkedList<Integer> perm = new
    LinkedList<Integer>();
    getPermutation(lr, index, fact(n), perm);
    return perm;
}
```

```
void getPermutation(LeftRightArray lr, int i, long
    fact, LinkedList<Integer> perm) {
    int n = lr.size();
    if(n == 1) {
        perm.add(lr.freeIndex(0, false));
    } else {
        fact /= n;
        int j = (int)(i / fact);
        perm.add(lr.freeIndex(j, true));
        i -= j * fact;
        getPermutation(lr, i, fact, perm);
    }
}
```

```
int[] getICombinadic(int n, int k, long i) {
    int[] comb = new int[k];
    int j = 0;
    for(int z = 1; z <= n; z++) {
        if (k == 0) {
            break;
        }
        long threshold = C(n - z, k - 1);
        if (i < threshold) {
            comb[j] = z - 1;
            j++;
            k = k - 1;
        } else if (i >= threshold) {
            i = i - threshold;
        }
    }
    return comb;
}
```

```
void combinations(int n, int k) {
    combinations(n, 0, new int[k], 0);
}
```

```
void combinations(int n, int j, int[] comb, int k) {
    if(k == comb.length) {
        System.out.println(Arrays.toString(comb));
    } else {
        for(int i = j; i < n; i++) {
            comb[k] = i;
            combinations(n, i + 1, comb, k + 1);
        }
    }
}
```

```
void subsets(int[] set) {
    int n = (1 << set.length);
    for(int i = 0; i < n; i++) {
        int[] sub = new int[Integer.bitCount(i)];
        int k = 0, j = 0;
        while((i << j) <= i) {
            if((i & (1 << j)) == (1 << j)) {
```

```

        sub[k++] = set[j];
    }
    j++;
}
System.out.println(Arrays.toString(sub));
}

```

## 6.2 Decomposition in unit fractions *untested*

Write  $0 < \frac{p}{q} < 1$  as a sum of  $\frac{1}{k}$

```

void expandUnitFrac(long p, long q) {
    if (p != 0) {
        long i = q % p == 0 ? q/p : q/p + 1;
        System.out.println("1/" + i);
        expandUnitFrac(p*i-q, q*i);
    }
}

```

## 6.3 Combination

Number of combinations of  $k$  elements within  $n$  ones ( $C_n^k$ )

Special case :  $C_n^k \bmod 2 = n \oplus m$

```

long C(int n, int k) {
    double r = 1;
    k = Math.min(k, n - k);
    for (int i = 1; i <= k; i++)
        r /= i;
    for (int i = n; i >= n - k + 1; i--)
        r *= i;
    return Math.round(r);
}

```

### 6.3.1 Catalan numbers

$$\text{cat}(n) = \frac{C_n^{2n}}{n+1} \text{cat}(n+1) = \frac{(2n+2)(2n+1)}{(n+2)(n+1)} \text{cat}(n)$$

- distinct binary trees with  $n$  vertices.
- expressions containing  $n$  pairs of parentheses correctly matched (e.g.  $n = 3$   $()()()$ ,  $()(())$ ,  $((())()$ ,  $((()))$ ,  $((()())$ ).
- parenthesize  $n + 1$  factors (e.g.  $n = 3$   $(ab)(cd)$ ,  $a(b(cd))$ ,  $((ab)c)(d)$ ,  $(a(bc))(d)$ ,  $a((bc)d)$ ).
- triangulate a convex polygon of  $n + 2$  sides.
- number of monotonic paths along the edge of a  $n \times n$  grid which do not pass above de diagonal.

Compute all Catalan number  $\leq n$

```

long[] allCatalan(int n) {
    long[] catalanNumbers = new long[n];
    catalanNumbers[0] = 1;
    for (int i = 1; i < n; i++) {
        int j = i - 1;
        long b = j * j;
        long a = 4 * b + 6 * j + 2;
        b += 3 * j + 2;
        catalanNumbers[i] = catalanNumbers[j] * a/b;
    }
    return catalanNumbers;
}

```

## 6.4 Fibonacci series

$f(0) = 0$ ,  $f(1) = 1$  et  $f(n) = f(n-1) + f(n-2)$ .

The following relation enables us to compute every number of the series in  $O(\log(n))$  :

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

## 6.5 Cycle finding

```

int[] floydCycleFinding (int x0) {
    int tortoise = f(x0), hare = f(f(x0));
    while (tortoise != hare) {
        tortoise = f(tortoise);
        hare = f(f(hare));
    }
    int mu = 0; hare = x0; // first
    while (tortoise != hare) {
        tortoise = f(tortoise); hare = f(hare); mu++;
    }
    int lambda = 1; hare = f(tortoise); // length
    while (tortoise != hare) {
        hare = f(hare); lambda++;
    }
    return new int[] {mu, lambda};
}

```

## 6.6 Number theory

### 6.6.1 Misc

$$ax \leq b \Leftrightarrow x \leq \left\lfloor \frac{b}{a} \right\rfloor \quad ax \geq b \Leftrightarrow x \leq \left\lceil \frac{b}{a} \right\rceil \quad \left\lceil \frac{a}{b} \right\rceil = \left\lfloor \frac{a+b-1}{b} \right\rfloor.$$

```

long gcd (long a, long b) {
    return (b == 0) ? a : gcd(b, a % b);
}
long lcm (long a, long b) {
    return a * (b / gcd(a,b));
}
long modInverse (long a, long b) {
    return big(a).modInverse(big(b)).longValue();
}
long modInverse (long a, long b) {
    extendedEuclid(a, b);
    return x;
}

```

In prime factorization of  $n$ , the power of  $p$  is

$$\sum_{i=1}^{\infty} \left\lfloor \frac{n}{p^i} \right\rfloor$$

```

int factopower (int n, int p) {
    int pow = 0;
    while (n > 0) {
        pow += n / p;
        n /= p;
    }
    return pow;
}

```

### 6.6.2 Équations diophantiennes

$ax + by = c$ .  $d = \gcd(a, b)$ , no sol si  $d$  divise pas  $c$  sinon  $(a, b) = (x(n/d) + (b/d)n, y(n/d) + (a/d)n)$  où  $ax + by = d$   $n \in \mathbb{Z}$ .

```

static int x, y;
static int extendedEuclid(int a, int b) {
    if (b == 0) { x = 1; y = 0; return a; }
    int d = extendedEuclid(b, a % b);
    int x1 = y;
    int y1 = x - (a / b) * y;
    x = x1;
    y = y1;
    return d;
}

```

### 6.6.3 Chinese remainder theorem

```

static long[] chinese (long[] b, long[] m) {
    long x = b[0], l = m[0];
    for (int i = 1; i < m.length; i++) {
        long m1 = m[i], b1 = b[i];
        long d = gcd(l, m1);
        if ((x - b1) % d != 0) return null;
        long lcm = l * (m1 / d);
        long t1 = (((x - b1) / d) % lcm) * (modInverse(m1/d, l/d) % lcm) % lcm;
        x = (b1 + (t1 * m1) % lcm) % lcm;
    }
}

```



```

    l = lcm;
}
return new long[] {x, l};
}

```

### 6.6.4 Euler phi

$\phi(N) = N \times \prod_{p|N} (1 - \frac{1}{p}) = \#\{k < N \mid \gcd(k, N) = 1\}$

```

long phi(long n, int primes[]) {
    long ans = n; // Method 1
    for (int i = 0; i < primes.length && primes[i] *
        primes[i] <= n; i++) {
        int p = primes[i];
        if (n % p == 0) ans -= ans / p;
        while (n % p == 0) n /= p;
    }
    if (n != 1) ans -= ans / n;
    return ans;
}
for (int i = 1; i <= 1000000; i++) phi[i] = i;
for (int i = 2; i <= 1000000; i++) // Method 2
    if (phi[i] == i) // i is prime
        for (int j = i; j <= 1000000; j += i)
            phi[j] = (phi[j] / i) * (i - 1);

```

- If  $\phi(1) = 1$ ,  $n = \sum_{d|n} \phi(d)$ .
- $p$  prime iff there exists a number relatively prime with  $p$  of order  $p - 1$  (primitive root of  $p$ ).
- There is  $\phi(d)$  number of orders  $d$  modulo  $p$ .
- If  $g$  is order  $d \bmod p$ ,  $\{g^k \mid k = 1, \dots, d - 1 : (k, d) = 1\}$  are the  $\phi(d)$  numbers of order  $d \bmod p$ .

Let  $\phi_S(n) = \sum_{i=1}^n \phi(i)$ .

$$\phi_S(n) = \frac{n^2 + n}{2} - \sum_{d=2}^n \phi_S\left(\left\lfloor \frac{n}{d} \right\rfloor\right).$$

Discrete log

$$a^x \equiv a^y \pmod{n} \Leftrightarrow x \equiv y \pmod{O_n(a)} \\ \Leftrightarrow x \equiv y \pmod{\phi(n)}$$

and in particular, if  $g$  is a primitive root of  $p$ ,

$$g^x \equiv g^y \pmod{p} \Leftrightarrow x \equiv y \pmod{p-1}$$

so for an equation  $(p \nmid a, b)$

$$a^{k_1} \equiv b^{k_2} \pmod{p}$$

we take  $\ell_1$  and  $\ell_2$  such that  $a = g^{\ell_1}$  and  $b = g^{\ell_2}$  and it becomes

$$k_1 \ell_1 \equiv k_2 \ell_2 \pmod{p-1}$$

### 6.6.5 Quadratic residue (QR)

$p$  odd prime. Let  $g$  primitive root mod  $p$ .  $\forall n$ ,  $g^{2n}$  is QR mod  $p$  and  $g^{2n+1}$  is not. There is  $\frac{p-1}{2}$  QR and  $\frac{p-1}{2}$  not QR.

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p} \\ = \prod_{r=1}^{\frac{p-1}{2}} \varepsilon(ar)$$

where  $\varepsilon(x) = 1$  if  $x \equiv 1, \dots, \frac{p-1}{2} \pmod{p}$  and  $-1$  otherwise.  
 $b$  odd  $((\frac{a}{b}) = 1$  does not mean  $a$  QR mod  $b$  !!!)

$$\left(\frac{a}{b}\right) \triangleq \prod \left(\frac{a}{p_i}\right)^{e_i}$$

$$\bullet \left(\frac{-1}{b}\right) = 1 \text{ iff } b \equiv 1 \pmod{4}.$$

$$\bullet \left(\frac{2}{b}\right) = 1 \text{ iff } b \equiv \pm 1 \pmod{8}.$$

$b$  odd

$$\left(\frac{ac}{b}\right) = \left(\frac{a}{b}\right) \left(\frac{c}{b}\right)$$

$a, b$  odd

$$\left(\frac{a}{b}\right) \left(\frac{b}{a}\right) = (-1)^{\frac{a-1}{2} \frac{b-1}{2}}.$$

```

static long modpow (long a, long n, long m) {
    if (n == 0) {
        return 1 % m;
    }
    if (n % 2 == 0) {
        long demi = modpow(a, n/2, m);
        return (demi * demi) % m;
    } else {
        return (modpow(a, n-1, m) * a) % m;
    }
}

static long modular_sqrt(long a, long p) {
    /*
     * Solve the congruence of the form:
     * x^2 = a (mod p)
     * And returns x. Note that p - x is also a root.
     *
     * 0 is returned is no square root exists for
     * these a and p.
     */
    /*
     * The Tonelli-Shanks algorithm is used (except
     * for some simple cases in which the solution
     * is known from an identity). This algorithm
     * runs in polynomial time (unless the
     * generalized Riemann hypothesis is false).
     */
    // Simple cases
    if (legendre_symbol(a, p) != 1) {
        return 0;
    } else if (a == 0) {
        return 0;
    } else if (p == 2) {
        return a;
    } else if (p % 4 == 3) {
        return modpow(a, (p + 1) / 4, p);
    }

    /* Partition p-1 to s * 2^e for an odd s (i.e.
     * reduce all the powers of 2 from p-1)
     */
    long s = p - 1;
    long e = 0;
    while (s % 2 == 0) {
        s /= 2;
        e += 1;
    }

    /* Find some 'n' with a legendre symbol n|p = -1.
     * Shouldn't take long.*/
    long n = 2;
    while (legendre_symbol(n, p) != -1) {
        n += 1;
    }

    /* x is a guess of the square root that gets
     * better
     * with each iteration.
     * b is the "fudge factor" - by how much we're off
     * with the guess. The invariant x^2 = ab (mod p)
     * is maintained throughout the loop.
     * g is used for successive powers of n to update
     * both a and b
     * r is the exponent - decreases with each update
     */

```

```

long x = modpow(a, (s + 1) / 2, p);
long b = modpow(a, s, p);
long g = modpow(n, s, p);
long r = e;

for (;;) {
    long t = b;
    long m = 0;
    for (m = 0; m < r; m++) {
        if (t == 1) {
            break;
        }
        t = (t * t) % p;
    }

    if (m == 0) {
        return x;
    }

    long pow2 = 1;
    for (int i = 0; i < r - m - 1; i++) { pow2 *= 2; }
    long gs = modpow(g, pow2, p);
    g = (gs * gs) % p;
    x = (x * gs) % p;
    b = (b * g) % p;
    r = m;
}

static long legendre_symbol1(long a, long p) {
    // p is prime and a is rel. prime to b
    long ls = modpow(a, (p - 1) / 2, p);
    return ls == p - 1 ? -1 : ls;
}

static long legendre_symbol(long a, long b) {
    // b is odd and rel. prime to a
    a %= b;
    if (a == 0) {
        return 0;
    }
    int exp2 = 0;
    while (a % 2 == 0) {
        a /= 2;
        exp2++;
    }
    int cur = 1;
    if (exp2 % 2 == 1 && (b % 8 == 3 || b % 8 == 5)) {
        cur *= -1;
    }
    if (a < 0) {
        if (b % 4 == 3) {
            cur *= -1;
        }
        a *= -1;
    }
    if (a == 1) {
        return cur;
    }
    if (a % 4 == 3 && b % 4 == 3) {
        cur *= -1;
    }
    return cur * legendre_symbol(b, a);
}

```

## 6.7 Linear equations

Solve  $Ax = b$ .

```

double[] gaussElim(double[][] A, double[] b) {
    int N = b.length;
    for (int p = 0; p < N; p++) {
        int max = p;
        for (int i = p + 1; i < N; i++) {
            if (Math.abs(A[i][p]) > Math.abs(A[max][p])) {
                max = i;
            }
        }
    }
}

```

```

swap(A, p, max);
swap(b, p, max);
// singular or nearly singular
if (Math.abs(A[p][p]) <= E) {
    return null;
}
// pivot within A and b
for (int i = p + 1; i < N; i++) {
    double alpha = A[i][p] / A[p][p];
    b[i] -= alpha * b[p];
    for (int j = p; j < N; j++) {
        A[i][j] -= alpha * A[p][j];
    }
}
// back substitution
double[] x = new double[N];
for (int i = N - 1; i >= 0; i--) {
    double sum = 0.0;
    for (int j = i + 1; j < N; j++) {
        sum += A[i][j] * x[j];
    }
    x[i] = (b[i] - sum) / A[i][i];
}
return x;
}

```

## 6.8 Ternary Search

Find minimum of unimodal function.

```

double ternarySearch(double left, double right) {
    if (right - left < E) {
        return (right + left) / 2;
    }
    double leftThird = (left * 2 + right) / 3;
    double rightThird = (left + right * 2) / 3;
    // minimize >, maximize <
    if (f(leftThird) > f(rightThird)) {
        return ternarySearch(leftThird, right);
    }
    return ternarySearch(left, rightThird);
}

```

## 6.9 Integration

Compute integral.

```

double integral(double a, double b) {
    double h = b - a;
    double c = (a + b) / 2.0;
    double d = (a + c) / 2.0;
    double e = (b + c) / 2.0;
    double Q1 = h/6 * (f(a) + 4*f(c) + f(b));
    double Q2 = h/12 * (f(a) + 4*f(d) + 2*f(c) + 4*f(e) + f(b));
    if (Math.abs(Q2 - Q1) <= E) {
        return Q2 + (Q2 - Q1) / 15;
    } else {
        return integral(a, c) + integral(c, b);
    }
}

```

## 7 Strings *untested*

Reverse a String

```
new StringBuilder(line).reverse().toString()
```

### 7.1 Longest palindrome

```

int[] calculateAtCenters(String s) {
    int n = s.length();
    int[] L = new int[2 * n + 1];
    int i = 0, palLen = 0, k = 0;
    while (i < n) {

```

```

    if((i > palLen) &&
        (s.charAt(i - palLen - 1) == s.charAt(i))) {
        palLen += 2;
        i += 1;
        continue;
    }
    L[k++] = palLen;
    int e = k - 2 - palLen;
    boolean found = false;
    for(int j = k - 2; j > e; j--) {
        if(L[j] == j - e - 1) {
            palLen = j - e - 1;
            found = true;
            break;
        }
        L[k++] = Math.min(j - e - 1, L[j]);
    }
    if(!found) {
        i += 1;
        palLen = 1;
    }
}
L[k++] = palLen;
int e = 2 * (k - n) - 3;
for(i = k - 2; i > e; i--) {
    int d = i - e - 1;
    L[k++] = Math.min(d, L[i]);
}
return L;
}

String getPalindrome(String s, int[] L) {
    int max = L[0];
    int maxI = 0;
    for(int i = 1; i < L.length; i++) {
        if(L[i] > max) {
            max = L[i];
            maxI = i;
        }
    }
    int b = 0, e = 0;
    b = maxI / 2 - L[maxI] / 2;
    e = maxI / 2 + L[maxI] / 2;
    e += maxI % 2 == 0 ? 0 : 1;
    return s.substring(b, e);
}

```

```

String getPalindrome(String s)
{
    return getPalindrome(s, calculateAtCenters(s));
}

```

## 7.2 Occurences in a string

KMP(s,p) returns occurrences index of p in s.

```

int[] kmpPreprocess(char[] p) {
    int m = p.length;
    int[] b = new int[m+1];
    int i = 0, j = -1; b[0] = -1; // starting values
    while (i < m) { // pre-process the pattern string p
        while (j >= 0 && p[i] != p[j]) j = b[j]; // if
        different, reset j using b
        i++; j++; // if same, advance both pointers
        b[i] = j;
    }
    return b; }

```

```

LinkedList<Integer> kmpSearchAll(char[] s, char[] p)
{ // text, pattern
    int[] b = kmpPreprocess(p); // back table
    int n = s.length, m = p.length;
    LinkedList<Integer> found = new LinkedList<Integer>
    >();
    int i = 0, j = 0; // starting values
    while (i < n) { // search through string s
        while (j >= 0 && s[i] != p[j]) j = b[j]; // if
        different, reset j using b
        i++; j++; // if same, advance both pointers
        if (j == m) { // a match found when j == m

```

```

            found.add(i-j);
            j = b[j]; // prepare j for the next possible
            match
        } }
    return found; }

```

```

int kmpSearchFirst(char[] s, char[] p) { // text,
    pattern
    int[] b = kmpPreprocess(p); // back table
    int n = s.length, m = p.length;
    int i = 0, j = 0; // starting values
    while (i < n) { // search through string s
        while (j >= 0 && s[i] != p[j]) j = b[j]; // if
        different, reset j using b
        i++; j++; // if same, advance both pointers
        if (j == m) { // a match found when j == m
            return i - j;
        }
    }
    return n - j; }

```

## 7.3 Multipattern search: Aho-Corasick

The complexity is the sum of the lengths of the patterns + the length of the text + the sum of the matches of each pattern in other patterns.

```

static class Node {
    Node[] next;
    Node fall_node;
    LinkedList<Integer> pattern_ids;
    public Node(int alphabet_len) {
        next = new Node[alphabet_len];
        fall_node = null;
        pattern_ids = null;
    }
}

static int next_id = 0;
static int TrieInsert(Node node, int p[], int
    alphabet_len) {
    for (int i = 0; i < p.length; i++) {
        if (node.next[p[i]] == null)
            node.next[p[i]] = new Node(alphabet_len);
        node = node.next[p[i]];
    }
    int cur_id;
    if (node.pattern_ids == null) {
        cur_id = next_id++;
        node.pattern_ids = new LinkedList<Integer>();
        node.pattern_ids.add(cur_id);
    } else {
        cur_id = node.pattern_ids.getFirst();
    }
    return cur_id;
    // two identical patterns have the same id
}

static Node BuildTrie(ArrayList<int[]> patterns, int
    [] ids, int alphabet_len) {
    Node trie_root = new Node(alphabet_len);

    // Insert pattern lines in the trie.
    for (int i = 0; i < patterns.size(); i++)
        ids[i] = TrieInsert(trie_root, patterns.get(i),
            alphabet_len);

    // Build fall function.
    LinkedList<Node> q = new LinkedList<Node>();
    for (int i = 0; i < alphabet_len; i++)
        if (trie_root.next[i] == null)
            trie_root.next[i] = trie_root; // Complete
        the next function for the root.
        else {
            q.add(trie_root.next[i]);
            trie_root.next[i].fall_node = trie_root;
        }
    while (!q.isEmpty()) {
        Node cur = q.poll();
        if (cur.fall_node.pattern_ids != null) {
            if (cur.pattern_ids == null)
                cur.pattern_ids = new LinkedList<Integer>();

```

```

        cur.pattern_ids.addAll(cur.fall_node.
pattern_ids);
    }
    for (int i = 0; i < alphabet_len; i++)
        if (cur.next[i] != null) {
            q.add(cur.next[i]);
            Node v = cur.fall_node;
            while (v.next[i] == null)
                v = v.fall_node;
            cur.next[i].fall_node = v.next[i];
        }
    }
    return trie_root;
}
static LinkedList<Integer>[] AhoCorasickSearch(Node
trie_root, int[] text) {
    LinkedList<Integer>[] match = new LinkedList[text.
length];
    Node cur = trie_root;
    for (int i = 0; i < text.length; i++) {
        int ind = text[i];
        while (cur.next[ind] == null) {
            cur = cur.fall_node;
        }
        cur = cur.next[ind];
        match[i] = cur.pattern_ids;
    }
    return match;
}

```

## 7.4 Match with hash: Rabin-Karp

```

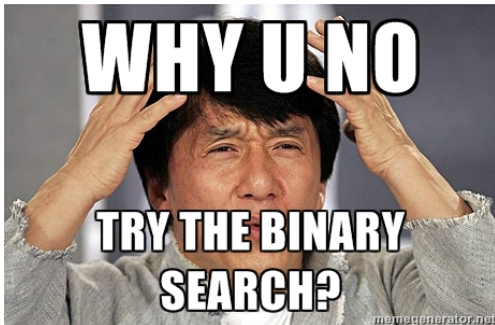
static final long MOD = 2147483647;
static final long BASE = 2;

static int RabinKarp(int[] p, int[] s) {
    if (s.length < p.length) return -1;
    int m = p.length, n = s.length;
    long phash = 0;
    long hash = 0;
    long exp = 1;
    for (int i = m-1; i >= 0; i--) {
        hash = (hash + ((s[i]*exp) % MOD)) % MOD;
        phash = (phash + ((p[i]*exp) % MOD)) % MOD;
        if (i > 0) exp = (exp * BASE) % MOD;
    }
    if (hash == phash) return 0;

    for (int i = m; i < n; i++) {
        // subtract top number
        hash = (hash + MOD - ((s[i-m]*exp) % MOD)) % MOD;
        // shift hash
        hash = (hash * BASE) % MOD;
        // add new number
        hash = (hash + s[i]) % MOD;
        if (hash == phash) return i-m+1;
    }
    return -1;
}

```

## 8 Miscellaneous



### 8.1 The answer

```
int reponse() { return 42; }
```

## 8.2 FFT

Efficiently compute the coefficients of the polynomial

$$\left(\sum_{i=0}^n a_i x^i\right) \left(\sum_{i=0}^n b_i x^i\right)$$

That is, compute the convolution

$$c_k = a \otimes b = \sum_{i=0}^k a_i b_{k-i}.$$

For any two vectors  $a$  and  $b$  of length  $n$  that is a power of two,

$$a \otimes b = \text{DFT}_{2n}^{-1}(\text{DFT}_{2n}(a) \cdot \text{DFT}_{2n}(b)).$$

where  $a$  and  $b$  are padded with 0s to length  $2n$ ,  $\cdot$  denotes the componentwise product and DFT is  $n \log(n)$ !

```

public static void fft(double[] re, double[] im,
boolean invert) {
    int count = re.length;
    for (int i = 1, j = 0; i < count; i++) {
        int bit = count >> 1;
        for (; j >= bit; bit >>= 1)
            j -= bit;
        j += bit;
        if (i < j) {
            double temp = re[i];
            re[i] = re[j];
            re[j] = temp;
            temp = im[i];
            im[i] = im[j];
            im[j] = temp;
        }
    }
    for (int len = 2; len <= count; len <<= 1) {
        int halfLen = len >> 1;
        double angle = 2 * Math.PI / len;
        if (invert)
            angle = -angle;
        double wLenA = Math.cos(angle);
        double wLenB = Math.sin(angle);
        for (int i = 0; i < count; i += len) {
            double wA = 1;
            double wB = 0;
            for (int j = 0; j < halfLen; j++) {
                double uA = re[i + j];
                double uB = im[i + j];
                double vA = re[i + j + halfLen] * wA - im[i
+ j + halfLen] * wB;
                double vB = re[i + j + halfLen] * wB + im[i
+ j + halfLen] * wA;
                re[i + j] = uA + vA;
                im[i + j] = uB + vB;
                re[i + j + halfLen] = uA - vA;
                im[i + j + halfLen] = uB - vB;
                double nextWA = wA * wLenA - wB * wLenB;
                wB = wA * wLenB + wB * wLenA;
                wA = nextWA;
            }
        }
    }
    if (invert) {
        for (int i = 0; i < count; i++) {
            re[i] /= count;
            im[i] /= count;
        }
    }
}

```

```

public static long[] poly_mult(long[] a, long[] b) {
    int resultSize = Integer.highestOneBit(Math.max(a.
length, b.length) - 1) << 2;
    resultSize = Math.max(resultSize, 1);
    double[] aReal = new double[resultSize];
    double[] aImaginary = new double[resultSize];
    double[] bReal = new double[resultSize];

```

```

double[] bImaginary = new double[resultSize];
for (int i = 0; i < a.length; i++)
    aReal[i] = a[i];
for (int i = 0; i < b.length; i++)
    bReal[i] = b[i];
fft(aReal, aImaginary, false);
if (a == b) {
    System.arraycopy(aReal, 0, bReal, 0, aReal.length);
    System.arraycopy(aImaginary, 0, bImaginary, 0, aImaginary.length);
} else
    fft(bReal, bImaginary, false);
for (int i = 0; i < resultSize; i++) {
    double real = aReal[i] * bReal[i] - aImaginary[i] * bImaginary[i];
    aImaginary[i] = aImaginary[i] * bReal[i] + bImaginary[i] * aReal[i];
    aReal[i] = real;
}
fft(aReal, aImaginary, true);
long[] result = new long[resultSize];
for (int i = 0; i < resultSize; i++)
    result[i] = Math.round(aReal[i]);
return result;
}

```

### 8.3 Sort algorithms *untested*

```

int findKth(int[] A, int k, int n) {
    if (n <= 10) {
        Arrays.sort(A, 0, n);
        return A[k];
    }
    int nG = (int) Math.ceil(n / 5.0);
    int[][] group = new int[nG][];
    int[] kth = new int[nG];
    for (int i = 0; i < nG; i++) {
        if (i == nG - 1 && n % 5 != 0) {
            group[i] = Arrays.copyOfRange(A, (n/5)*5, n);
            kth[i] = findKth(group[i], group[i].length / 2, group[i].length);
        } else {
            group[i] = Arrays.copyOfRange(A, i*5, (i+1)*5);
            kth[i] = findKth(group[i], 2, group[i].length);
        }
    }
    int M = findKth(kth, nG / 2, nG);
    int[] S = new int[n];
    int[] E = new int[n];
    int[] B = new int[n];
    int s = 0, e = 0, b = 0;
    for (int i = 0; i < n; i++) {
        if (A[i] < M) {
            S[s++] = A[i];
        } else if (A[i] > M) {
            B[b++] = A[i];
        } else {
            E[e++] = A[i];
        }
    }
    if (k < s) {
        return findKth(S, k, s);
    } else if (k >= s + e) {
        return findKth(B, k - s - e, b);
    }
    return M;
}

int[] countSort(int[] A, int k) { // O(n + k)
    int[] C = new int[k];
    for (int j = 0; j < A.length; j++) {
        C[A[j]]++;
    }
    for (int j = 1; j < k; j++) {
        C[j] += C[j - 1];
    }
    int[] B = new int[A.length];

```

```

    for (int j = A.length - 1; j >= 0; j--) {
        B[C[A[j]] - 1] = A[j];
        C[A[j]]--;
    }
    return B;
}

int[][] radixSort(int[][] nums, int k) { // O(d*(n+k))
    int n = nums.length;
    int m = nums[0].length;
    int[][] B = null;
    for (int i = m - 1; i >= 0; i--) {
        int[] C = new int[k];
        for (int j = 0; j < n; j++) {
            C[nums[j][i]]++;
        }
        for (int j = 1; j < k; j++) {
            C[j] += C[j - 1];
        }
        B = new int[n][];
        for (int j = n - 1; j >= 0; j--) {
            B[C[nums[j][i]] - 1] = nums[j];
            C[nums[j][i]] = C[nums[j][i]] - 1;
        }
        nums = B;
    }
    return nums;
}

int mergeSort(int[] a) {
    int n = a.length;
    if (n == 1) {return 0;}
    int m = n / 2;
    int[] left = Arrays.copyOfRange(a, 0, m);
    int[] right = Arrays.copyOfRange(a, m, n);
    int inv = mergeSort(left);
    inv += mergeSort(right);
    inv += merge(left, right, a);
    return inv;
}

int merge(int[] left, int[] right, int[] a) {
    int i = 0, l = 0, r = 0, inv = 0;
    while (l < left.length && r < right.length) {
        if (left[l] <= right[r]) {
            a[i++] = left[l++];
        } else {
            inv += left.length - l;
            a[i++] = right[r++];
        }
    }
    for (int j = l; j < left.length; j++) {
        a[i++] = left[j];
    }
    for (int j = r; j < right.length; j++) {
        a[i++] = right[j];
    }
    return inv;
}

int countMinSwapsToSort(int[] a) {
    int[] b = a.clone();
    Arrays.sort(b);
    int nSwaps = 0;
    for (int i = 0; i < a.length; i++) {
        // cuidado com elementos repetidos!
        int j = Arrays.binarySearch(b, a[i]);
        if (b[i] == a[j] && i != j) {
            nSwaps++;
            swap(a, i, j);
        }
    }
    for (int i = 0; i < a.length; i++) {
        if (a[i] != b[i]) {
            nSwaps++;
        }
    }
    return nSwaps;
}

```

```

}

//Count (i, j): h[i] <= h[k] <= h[j], k = i+1,...,j
-1.
int countVisiblePairs(int[] h) { // O(n)
    int n = h.length;
    int[] p = new int[n];
    int[] r = new int[n];
    Stack<Integer> S = new Stack<Integer>();
    for(int i = 0; i < n; i++) {
        int c = 0;
        if(S.isEmpty()) {
            S.push(h[i]);
            p[i] = 0;
        } else {
            if(S.peek() == h[i]) {
                p[i] = p[i - 1] + 1 - r[i - 1];
            } else {
                while(!S.isEmpty() && S.peek() < h[i]) {
                    S.pop();
                    c++;
                }
                p[i] = c;
                r[i] = c;
                if(!S.isEmpty()) {
                    p[i]++;
                }
                S.push(h[i]);
            }
        }
    }
    return sum(p);
}

void shuffle(Object[] a)
{
    int N = a.length;
    for (int i = 0; i < N; i++) {
        int r = i + (int) (Math.random() * (N-i));
        swap(a, i, r);
    }
}

```

## 8.4 Huffman (compression)

Usually used for characters, but usable with everything in which we can count occurrences.

Make a prefix tree we use to decode and we unstack to encode.

```

class HuffmanNode implements Comparable<HuffmanNode>
{
    public boolean isLeaf;
    public int occurrences;
    public int charIndex;
    public HuffmanNode left, right;
    public HuffmanNode(HuffmanNode left, HuffmanNode right)
    {
        this.occurrences = left.occurrences+right.occurrences;
        this.left = left;
        this.right = right;
        isLeaf = false;
    }
    public HuffmanNode(int charIndex, int occurrences)
    {
        this.charIndex = charIndex;
        this.occurrences = occurrences;
        isLeaf = true;
    }
    @Override
    public int compareTo(HuffmanNode o) {
        return occurrences-o.occurrences;
    }
}

HuffmanNode getHuffmanTree(int[] occurrences) {
    PriorityQueue<HuffmanNode> q = new PriorityQueue<HuffmanNode>();
    for(int i = 0; i < occurrences.length; i++)
        q.add(new HuffmanNode(i, occurrences[i]));
}

```

```

while(q.size() != 1) {
    HuffmanNode right = q.poll();
    HuffmanNode left = q.poll();
    q.add(new HuffmanNode(left, right));
}
return q.poll();
}

void getHuffmanTable(HuffmanNode tree, BitSet[] result, BitSet current, int pos){
    if(tree.isLeaf) {
        BitSet finalBitSet = new BitSet();
        for(int i = 0; i < pos; i++)
            finalBitSet.set(i, current.get(pos-i-1));
        result[tree.charIndex] = finalBitSet;
    } else {
        BitSet leftBitSet = new BitSet();
        leftBitSet.or(current);
        leftBitSet.set(pos, false);
        getHuffmanTable(tree.left, result, leftBitSet, pos+1);

        BitSet rightBitSet = new BitSet();
        rightBitSet.or(current);
        rightBitSet.set(pos, true);
        getHuffmanTable(tree.right, result, rightBitSet, pos+1);
    }
}

//n=occurrences.length
static BitSet[] getHuffmanTable(int n, HuffmanNode tree) {
    BitSet[] result = new BitSet[n];
    getHuffmanTable(tree, result, new BitSet(), 0);
    return result;
}

```

## 8.5 Union Find

```

static class UnionFind {
    int[] depth; int[] leader; int[] size;
    public UnionFind(int n) {
        depth = new int[n]; leader = new int[n]; size = new int[n];
        Arrays.fill(depth, 1); Arrays.fill(size, 1);
        for(int i = 0; i < n; i++) leader[i] = i;
    }
    public int find(int a) {
        if(a != leader[a])
            leader[a] = find(leader[a]);
        return leader[a];
    }
    public void union(int a, int b) {
        int leaderA = find(a);
        int leaderB = find(b);
        if(leaderA == leaderB) return;
        if(size[leaderA] > size[leaderB]) {
            union(leaderB, leaderA); return;
        }
        leader[leaderA] = leaderB;
        depth[leaderB] = Math.max(depth[leaderA]+1, depth[leaderB]);
        size[leaderB] += size[leaderA];
    }
}

```

## 8.6 Fenwick Tree (RSQ solver)

```

static class FenwickTree {
    private int[] ft;
    private int LSONe(int S) { return (S & (-S)); }
    public FenwickTree(int n) { // ignore index 0
        ft = new int[n+1];
        for (int i = 0; i <= n; i++) ft[i] = 0;
    }
    public int rsq(int b) { // returns RSQ(1, b)
        PRE 1 <= b <= n
        int sum = 0; for (; b > 0; b -= LSONe(b)) sum += ft[b];
        return sum;
    }
}

```



```
}  
public int rsq(int a, int b) { // returns RSQ(a, b)  
    PRE 1 <= a, b <= n  
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1));  
}  
void adjust(int k, int v) { // n = ft.size() - 1  
    PRE 1 <= k <= n  
    for (; k < ft.length; k += LOne(k)) ft[k] += v;  
}
```



## 9 Being first?

