

**ATENEO DE DAVAO UNIVERSITY**

**PROGRAMMING VARSITY**

**Team ATENEO BLUE KNIGHTS II**

**Coach: NISPEROS,** Sheila Dawn M.

**Members:**

**BACALSO,** Kurt S.

**CASTAÑEDA,** Dharyl Daniel

**GARCIA,** Felicia Rose D.

ACM-ICPC Team Reference Document

1. **Data Structures**

e. **Bit Manipulation**

class BitMan {

public int setBit(int S, int j)

{ return S | (1 << j); }

public int isOn(int S, int j)

{ return S & (1 << j); }

public int clearBit(int S, int j)

{ return S & ~(1 << j); }

public int toggleBit(int S, int j)

{ return S ^ (1 << j); }

public int lowBit(int S)

{ return S & (~S); }

public int setAll(int n)

{ return (1 << n) – 1; }

public int module(int x, int N)

{ return ((x) & (N – 1)); }

public int isPowerOfTwo(int x)

{ return (x & (x – 1)}; )

public int nearestPowerOfTwo(int x)

{ return ((int)Math.pow(2.0, (int)((Math.log((double)x) /

Math.log(2.0)) + 0.5))); }

}

h. **PriorityQueue**

PriorityQueue<T> pq = new PriorityQueue<T>(); //T must have Comparable

implemented

pq.offer(T);

t = pq.poll();

i. **Graph**

class Node implements Comparable<Node> {

T node; //must have comparable implemented

Node<T> edges;

public Node(T v) {

vertex = v;

edges = new Node<T>();

}

public int compareTo(Node v) {

if(node.compareTo(v.node) != 0)

return node.compareTo(v.node);

else

// return something

}

}

// usage: Vector<Vertex> graph;

j. **Union-Find Disjoint Sets**

class UFDSet {

Vector<Integer> pset;

public UFDSet() {

pset = new Vector<Integer>();

}

public void initSet(int size) {

pset = new Vector<Integer>();

for(int i=0; i<size; i++)

pset.add(i);

}

public int findSet(int i) {

if(pset.get(i) == i)

return i;

else {

int ret = findSet(pset.get(i));

pset.set(i, ret);

return ret;

}

}

public void unionSet(int i, int j) {

pset.set(findSet(i), findSet(j));

}

public boolean isSameSet(int i, int j) {

return findSet(i) == findSet(j);

}

}

k. **Segment Tree**

class SegmentTree {

int[] stree;

public void build(int[] array, int vertex, int left, int right) {

if(left == right)

stree[vertex] = left;

else {

int nL = 2 \* vertex, nR = 2 \* vertex + 1;

build(array, nL, left, (left+right)/2);

build(array, nR, (left+right)/2+1, right);

int lContent = stree[nL], rContent = stree[nR];

int lValue = array[lContent], rValue = array[rContent];

stree[vertex] = (lValue <= rValue) ? lContent : rContent;

}

}

public void create(int[] array) {

int len = (int)(2 \* Math.pow(2.0,

Math.floor((Math.log((double)A.length) / Math.log(2.0)) + 1)));

stree = new int[len];

for(int i=0; i<len; i++) stree[i] = 0;

build(array, 1, 0, array.length-1);

}

public int rmq(int[] array, int vertex, int left,

int right, int start, int end) {

if(start > right || end < left) return -1;

if(left >= start && right <= end)

return stree[vertex];

int pos1 = rmq(array, 2\*vertex, left, (left+right)/2, start, end);

int pos2 = rmq(array, 2\*vertex+1, (left+right)/2+1, right, start,

end);

if(pos1 == -1) return pos2;

if(pos2 == -1) return pos1;

return (array[pos1] <= array[pos2]) ? pos1 : pos2;

}

}

l. **Fenwick Tree**

class FenwickTree {

public int LSOne(int S)

{ return (S & (-S)); }

public Vector<Integer> create(int n) {

Vector<Integer> v = new Vector<Integer>();

for(int i=0; i<=n; i++)

v.add(0);

return v;

}

public int rsq(Vector<Integer> ft, int b) {

int sum = 0;

for(; b > 0; b -= LSOne(b))

sum += ft.get(b);

return sum;

}

public int rsq(Vector<Integer> ft, int a, int b){

return rsq(ft, b) – (a == 1 ? 0 : rsq(ft, a – 1));

}

public void adjust(Vector fr, int k, int v) {

for(; k < ft.size(); k += LSOne(k))

ft.set(k, ft.get(k) + v);

}

}

1. **Problem Solving Paradigms** **– Problems**

a. **8 Queens Chess Problem (backtracking)**

public class EightQueens {

static int[] row;

static int[][] sols;

static int a, b, count;

public static boolean place(int col, int i) {

for(int j=1; j<col; j++)

if(row[j] == i || (Math.abs(row[j] – i) == Math.abs(j – col)))

return false;

return true;

}

public static void backtrack(int col) {

for(int i=1; i<=8; i++)

if(place(col, i)) {

row[col] = i;

if(col == 8 && row[b] == a) {

for(int j=1; j<=8; j++)

sols[count][j] = row[j];

}

else backtrack(col + 1);

}

}

public static void Main(String[] args) {

row = new int[9];

sols = new int[100][9];

a = b = 1;

count = 0;

backtrack(1);

}

}

b. **Longest Increasing Subsequence (Iterative)**

public static int lis\_length(int[] array) {

int n = a.length;

int[] q = new int[n];

for(int k=0; k<n; k++) {

int max = 0;

for(int j=0; j<k; j++)

if(a[k] > a[j])

if(q[j] > max)

max = q[j];

q[k] = max + 1;

max = 0;

for(int i=0; i<n; i++)

if(q[i] > max) max = q[i];

return max;

}

c. **Maximum 1D Sum (DP)**

public void max1Dsum(int[] arr) {

int maxSum = -1000000000, curMaxSum = 0, curSIndex = 0, curEIndex,

maxSIndex = 0, maxEIndex = 0;

for(curEIndex=0; curEIndex<n; curEIndex++) {

curMaxSum = curMaxSum + array[curEIndex];

if(curMaxSum > maxSum) {

maxSum = curMaxSum;

maxSIndex = curSIndex;

maxEIndex = curEIndex;

}

if(curMaxSum < 0) {

curMaxSum = 0;

curSIndex = curEIndex + 1;

}

}

// maxSum, maxSIndex, maxEIndex

}

d. **Maximum 2D Sum (DP)**

public void max2Dsum(int[][] matrix) {

int possibleMax, currentMax=0, uncounted=1;

int n = matrix[0].length;

for(int i=0; i<n; i++)

for(int j=0; j<n; j++) {

if(j > 0)

matrix[i][j] += matrix[i][j-1];

}

for(int left=0; left<n; left++)

for(int right=left; right<n; right++) {

possibleMax = 0;

for(int row=0; row<n; row++) {

if(left > 0)

possibleMax += matrix[row][right] – matrix[row][left-

1];

else

possibleMax+=matrix[row][right];

if(possibleMax < 0) possibleMax = 0;

if(unCounted == 1 || possibleMax > currentMax) {

currentMax = possibleMax;

unCounted = 0;

}

}

}

}

e. **0-1 Knapsack (DP)**

double[] values, weights;

double maxWeight;

double[][] cost;

public static void knapsack01() {

int n = values.length;

for(int j=0; j<maxWeight; j++)

cost[0][j] = 0;

for(int i=0; i<n; i++) {

cost[i][0] = 0;

for(int j=1; j<maxWeight; j++) {

if(j[i] <= w)

cost[i][j] = cost[i-1][j];

else {

if(cost[i-1][j] > v[i] + cost[i-1][j-w[i]])

cost[i][j] = cost[i-1][j];

else

cost[i][j] = v[i] + cost[i-1][j-w[i]];

}

}

}

}

f. **Longest Common Subsequence (DP)**

public int LCS(String x, String y) {

int[][] cost = new int[x.length()+1][y.length()+1];

for(int i=0; i<x.length()+1; i++)

cost[i][0] = 0;

for(int j=0; j<y.length()+1; j++)

cost[0][j] = 0;

for(int i=1; i<x.length()+1; i++)

for(int j=1; j<x.length()+1; j++) {

if(x.charAt(i) == y.charAt(j))

cost[i][j] = cost[i-1][j-1] + 1;

else

cost[i][j] = Math.max(cost[i][j-1],

cost[i-1][j]);

}

return cost[x.length()][y.length()];

}

// call the first function with i=m, j=n

public Vector<String> backtrack(int[][] cost, String x, String y, int i,

int j) {

if(i == 0 || j == 0)

return new Vector<String>();

else if(x.charAt(i) == y.charAt(j)) {

Vector<String> temp = backtrack(cost, x, y, i-1, j-1);

for(int k=0; k<temp.size(); k++) {

String tmp = temp.elementAt(k);

temp.setElementAt(tmp+x.charAt(i), k);

}

return temp;

}

else {

Vector<String> r = new Vector<String>();

if(cost[i][j-1] >= cost[i-1][j])

r = backtrack(cost, x, y, i, j-1);

if(cost[i-1][j] >= cost[i][j-1]) {

Vector<String> temp = backtrack(cost, x, y, i-1, j);

for(int k=0; k<temp.size(); k++)

r.addElement(temp.elementAt(k));

}

return r;

}

}

g. **Coin Change (DP)**

// 1 <= i <= denom.length-1 (n+1), 0 <= j <= change

public void coinChange(int[] denom, int change, int[][] cost, boolean[][]

used) {

for(int j=0; j<=change; j++) {

cost[denom.length-1][j] = j;

used[denom.length-1][j] = true;

}

for(int i=denom.length-1; i>=1; i--)

for(int j=0; j<=change; j++) {

if(denom[i] > j || cost[i+1][j] < 1 + cost[i][j-denom[i]]) {

cost[i][j] = cost[i+1][j];

used[i][j] = false;

}

else {

cost[i][j] = 1 + cost[i][j-denom[i]];

used[i][j] = true;

}

}

}

public void print(int[] denom, int change,

boolean[][] used) {

int temp = change;

for(int i=denom.length-1; i>=1; i--) {

if(used[i][change])

while(temp >= denom[i]) {

temp -= denom[i];

System.out.println(denom[i]);

}

}

}

h. **Levenshtein Distance (DP)**

public int editDistance(String s, String t) {

int sLength = s.length();

int tLength = t.length();

int cost = 0;

if(s.charAt(0) != t.charAt(0))

cost = 1;

if(sLength == 0)

return tLength;

else if(tLength == 0)

return sLength;

else

return Math.min(Math.min(editDistance(s.substring(1), t)+1,

editDistance(s, t.substring(1))+1), editDistance(s.substring(1),

t.substring(1))+cost);

}

i. **Traveling Salesman Problem (DP)**

int[] x, y;

int[][] dist, memo;

public int tsp(int pos, int bitmask) {

if(bitmask == (1 << (n+1)) – 1)

return dist[pos][0];

if(memo[pos][bitmask] != -1)

return memo[pos][bitask];

int ans = 2000000000;

for(int nxt=0; nxt<=n; nxt++)

if(nxt != pos && (bitmask & (1 << nxt)) == 0)

ans = Math.min(ans, dist[pos][nxt] + tsp(nxt, bitmas | (1 <<

nxt)));

return memo[pos][bitmas] = ans;

}

// 11 = number of nodes

x = new int[11]; // coordinates

y = new int[11];

dist = new int[11][11];

memo = new int[11][1 << 11];

for(int i=0; i<11; i++)

for(int j=0; j<11; j++)

dist[i][j] = Math.abs(x[i]-x[j]) + Math.abs(y[i]-y[j]);

for(int i=0; i<11; i++)

for(int j=0; j<(1<<11); j++)

memo[i][j] = -1;

int tspLength = tsp(0, 1);

j. **Binary Search Tree (Divide & Conquer)**

public class BinarySearchTree {

BinarySearchTree left, right;

String value;

int count;

public BinarySearchTree(String val) {

left = right = null;

count = 0;

value = val;

}

public boolean search(String val) {

if(value == null)

return false;

else if(val.compareTo(value) < 0) {

if(left != null)

return left.search(val);

else

return false;

}

else if(val.compareTo(value) > 0) {

if(right != null)

return right.search(val);

else

return false;

}

else return true;

}

public void insert(String val) {

if(value == null) {

value = val;

count++;

}

else if(val.compareTo(value) < 0) {

if(left == null)

left = new BinarySearchTree();

left.insert(val);

}

else if(val.compareTo(value) > 0) {

if(right == null)

right = new BinarySearchTree();

right.insert(val);

}

else if(val.compareTo(value) == 0)

count++;

}

public void remove(String val) {

if(val.compareTo(value) == 0) {

if(left == null && right == null) {

value = null;

count = 0;

}

else if(left == null) {

left = right.left;

right = right.right;

value = right.value;

count = right.count;

else if(right == null) {

right = left.right;

left = left.left;

value = left.value;

count = left.count;

}

else {

value = right.getMinVal();

right.removeAll(value);

}

}

else if(val.compareTo(value) < 0)

left.removeAll(val);

else if(val.compareTo(value) > 0)

right.removeAll(val);

}

public String getMinVal() {

if(left == null)

return value;

else

return left.getMinVal();

}

public void inOrder(Vector<String> sorted) {

if(temp.value == null)

return;

if(temp.left != null)

temp.left.inOrder(sorted);

sorted.addElement(temp.value);

if(temp.right != null)

temp.right.inOrder(sorted);

}

public void preOrder() {

if(temp.value == null)

return;

System.out.println(temp.value);

if(temp.left != null)

temp.left.preOrder();

if(temp.right != null)

temp.right.preOrder();

}

public void postOrder() {

if(temp.value == null)

return;

if(temp.left != null)

temp.left.preOrder();

if(temp.right != null)

temp.right.preOrder();

System.out.println(temp.value);

}

}

1. **Towers of Hanoi**

public void solve(int count, char source, char destination, char

intermediate) {

if(count == 1)

// move top disc from pole <source> to pole <destination>

else {

solve(count-1, source, intermediate, destination);

solve(1, source, destination, intermediate);

solve(count-1, intermediate, destination, source);

}

}

1. **Josephus Problem**

public int josephus(int n, int k) {

if(n == 1) return 1;

else return ((josephus(n-1, k) + k-1) % n) +1;

}

1. **Graph** // using graph data structure above
   1. **Topological Sort**

public Vector<Node> topoSort(Vector<Node> graph) {

int[] incoming = new int[graph.size()];

for(int i=0; i<graph.size(); i++) {

Vector<Node> edges = graph.elementAt(i).edges;

for(int j=0; j<edges.size(); j++) {

Node temp = edges.elementAt(j);

incoming[temp.index]++;

}

}

Vector<Node> topo = new Vector<Node>();

Vector<Node> edges;

Queue<Node> q = new LinkedList<Node>();

for(int i=0; i<incoming.length; i++)

if(incoming[i] == 0)

q.add(graph.elementAt(i));

while(!q.isEmpty()) {

Node temp = q.remove();

topo.addElement(temp);

edges = temp.edges;

for(int i=0; i<edges.size(); i++) {

temp = edges.elementAt(i);

incoming[temp.index]--;

if(incoming[temp.index] == 0)

q.add(temp);

}

}

int sum = 0;

for(int i=0; i<incoming.length; i++)

sum += incoming[i];

if(sum == 0)

return topo;

else

return null;

}

* 1. **Bipartite Graph**

public boolean isBipartite(Vector<Node> graph) {

Queue<Node> q = new LinkedList<Node>();

boolean[] visited = new boolean[graph.size()];

visited[0] = true;

q.add(graph.elementAt(0));

graph.elementAt(0).color = true;

while(!q.isEmpty()) {

Node temp = q.remove();

Vector<Node> edges = temp.edges;

for(int i=0; i<edges.size(); i++) {

Node m = edges.elementAt(i);

if(!visited[m.index]) {

if(m.color == temp.color)

return false;

m.color = !temp.color;

visited[m.index] = true;

q.add(m);

}

}

}

return true;

}

* 1. **All Pairs Shortest Paths**

public void floydWarshall(int[][] graph) {

int length = graph[0].length;

for(int k=0; k<length; k++)

for(int i=0; i<length; i++)

for(int j=0; j<length; j++)

graph[i][j] = Math.min(graph[i][j], graph[i][k] +

graph[k][j]);

}

* 1. **Minimum Spanning Tree**

class Edge implements Comparable<Edge> {

Node a, b;

double weight;

public Edge(Node c, Node d, double w) {

a = c;

b = d;

weight = w;

}

public int compareTo(Edge e) {

if(Double.compare(weight, e.weight) != 0)

return Double.compare(weight, e.weight);

else if(a.compareTo(e.a) != 0)

return a.compareTo(e.a);

else

return b.compareTo(e.b);

}

}

public double mstKruskal(PriorityQueue<Edge> edgeList, int nodes) {

UFDSet set = new UFDSet();

double cost = 0;

set.initSet(nodes);

while(!edgeList.isEmpty()) {

Edge temp = edgeList.poll();

double a = temp.a;

double b = temp.b;

if(!set.isSameSet(a, b)) {

cost +=a;

set.unionSet(a, b);

}

}

return cost;

}

* 1. **Single-Source Shortest Path (Positive Weights)**

public Vector<Integer> dijkstra(Vector<Node> graph, Node s) {

PriorityQueue<Node> pq = new PriorityQueue<Node>();

Vector<Integer> dist = new Vector<Integer>();

dist.set(s.index, 0);

pq.offer(s);

Vector<Node> edges;

while(!pq.isEmpty()) {

Node temp = pq.poll();

if(temp.dist <= dist.elementAt(temp.index)) {

edges = temp.edges;

for(int i=0; i<edges.size(); i++) {

Node t2 = edges.elementAt(i);

if(dist.elementAt(temp.index) +

t2.dist < dist.elementAt(t2.index))

{

dist.set(t2.index, dist.get(temp.index) + t2.dist);

pq.offer(t2);

}

}

}

}

return dist;

}

* 1. **Single-Source Shortest Path (Negative Weights)**

public Vector<Integer> bellmanFord(Vector<Node> graph, Node s) {

Vector<Integer> dist = new Vector<Integer>();

dist.set(s.index, 0);

for(int i=0; i<graph.size()-1; i++)

for(int j=0; j<graph.size(); j++) {

Node temp = graph.elementAt(j);

Vector<Node> edges = temp.edges;

for(int k=0; k<edges.size(); k++) {

Node t2 = edges.elementAt(k);

dist.set(t2.index, Math.min(dist.get(t2.index),

dist.get(temp.index) + temp.weight);

}

}

boolean nega\_cycle = false;

for(int i=0; i<graph.size(); i++) {

Node temp = graph.elementAt(i);

Vector<Node> edges = temp.edges;

for(int j=0; j<edges.size(); j++) {

Node t2 = edges.elementAt(j);

if(dist.get(t2.index) > dist.get(temp.index) + t2.dist)

nega\_cycle = true;

}

}

if(nega\_cycle)

return new Vector<Integer>();

else

return dist;

}

* 1. **Maximum Flow**

int maxv = // get the max vertices that can be made

int[][] res = new int[maxv][];

Node s, t;

int f;

Vector<Integer> p = new Vector<Integer>();

public void augment(Node v, int minEdge) {

if(v.equals(s))

f = minEdge;

else if(p.get(v.index) != -1) {

augment(p.get(v.index), Math.min(minEdge,

res[p.get(v.index)][v.index]));

res[p.get(v.index)][v.index] -= f;

res[v.index][p.get(v.index)] += f;

}

}

public int edmondsKarp(Vector<Node> graph) {

int mf = 0;

while(true) {

f = 0;

Queue<Node> queue = new LinkedList<Node>();

Vector<Integer> dist = new Vector<Integer>();

dist.addAll(Collections.nCopies(graph.size(), 2000000000));

q.offer(s);

dist.set(s.index, 0);

p.clear();

p.addAll(Collections.nCopies(graph.size(), -1));

while(!q.isEmpty()) {

Node u = q.poll();

if(u.equals(t)) break;

for(int v=0; v<maxv; v++)

if(res[u.index][v] > 0 && dist.get(v) == INF) {

dist.set(v, dist.get(u.index) + 1);

q.offer(v);

p.set(v, u.index);

}

}

augment(t, 2000000000);

if(f == 0) break;

mf += f;

}

return mf;

}

l. **Strongly Connected Components**

int lowest[MAXV], num[MAXV], visited[MAXV], comp[MAXV];

int prev\_edge[MAXE], last\_edge[MAXE], adj[MAXE], nedges;

int cur\_num, cur\_comp;

Stack<Integer> visiting;

public void init() {

for(int i=0; i<last\_edge.length; i++) last\_edge[i] = -1;

nedges = 0;

visiting = new Stack<Integer>();

}

public void edge(int v, int w) {

prev\_edge[nedges] = last\_edge[v];

adj[nedges] = w;

last\_edge[v] = nedges++;

}

public int tarjan\_dfs(int v) {

lowest[v] = num[v] = cur\_num++;

visiting.push(v);

visited[v] = 1;

for(int i=last\_edge[v]; i!=-1; i=prev\_edge[i]) {

int w = adj[i];

if(visited[w] == 0) lowest[v] = Math.min(lowest[v], tarjan\_dfs(w));

else if(visited[w] == 1) lowest[v] = Math.min(lowest[v], nu[w]);

}

if(lowest[v] == num[v]) {

int last = -1;

while(last != v) {

comp[last = visiting.top()] = cur\_comp;

visited[last] = 2;

visiting.pop();

}

++cur\_comp;

}

return lowest[v];

}

public void tarjan\_scc(int num\_v = MAXV) {

visiting = new Stack<Integer>();

for(int i=0; i<visited.length; i++) visited[i] = 0;

cur\_num = cur\_comp = 0;

for(int i=0; i<num\_v; ++i)

if(!visited[i]) tarjan\_dfs(i);

}

1. **Shortest Path Faster Algorithm**

public void SPFA(Vector<Node> graph, Node source) {

int[] d = new int[graph.size()];

for(int i=0; i<graph.size(); i++)

if(!graph.elementAt(i).equals(source))

d[i] = 2000000000;

d[s.index] = 0;

Queue<Node> queue = new LinkedList<Node>();

boolean[] visited = new Boolean[graph.size()];

visited[s.index] = true;

queue.add(s);

while(queue.size() != 0) {

Node u = queue.pop();

Vector<Node> edges = u.edges;

Vector<Integer> weights = u.weights;

for(int i=0; i<edges.size(); i++) {

Node v = edges.elementAt(i);

int w = weights.elementAt(i);

if(d[u.index]+w < d[v.index]) {

d[v.index] = d[u.index]+w;

if(!visited[v.index]) {

visited[v.index] = true;

queue.add(v);

}

}

}

}

}

1. **Mathematics**
   1. **LCM**

public int lcm(int a, int b) {

int temp = gcd(a, b);

return temp ? (a/temp\*b) : 0;

}

* 1. **Cycle Finding**

public void cycleFinding(int x0) {

int t=f(x0), h=f(f(x0)), start=0, length=1;

while(t != h) {

t = f(t);

h = f(f(h));

}

h = t;

t = x0;

while(t != h) {

t = f(t);

h = f(h);

++start;

}

h = f(t);

while(t != h) {

h = f(h);

++length;

}

}

* 1. **Binomial Coefficients**

public int C(int n, int k) {

if(k == 0 || k == n)

return 1;

if(table[n][k] != 0)

return table[n][k];

table[n][k] = C(n-1, k-1) + C(n-1, k);

return table[n][k];

}

* 1. **Prime Factors (Pollard’s rho)**

public long mulmod(long a, long b, long c) {

long x = 0, y = a % c;

while(b > 0) {

if(b%2 == 1) x = (x+y) % c;

y = (y\*2) % c;

b /= 2;

}

return x%c;

}

public long gcd(long a, long b) {

return b == 0 ? a : gcd(b, a%b);

}

public long pollard\_rho(long n) {

int i = 0, k = 2;

long x = 3, y = 3;

while(true) {

i++;

x = (mulmod(x, x, n) + n – 1) % n;

long d = gcd(Math.abs(y-x), n);

if(d != 1 && d != n) return d;

if(i == k) {

y = x;

k \*= 2;

}

}

}

* 1. **Relative Prime Factors of N (Euler’s Phi)**

public long eulerPhi(long n) {

long PF\_idx = 0, PF = primes[PF\_idx], ans = N;

while(PF \* PF <= N) {

if(N % PF == 0) ans -= ans/PF;

while(N % PF == 0) N /= PF;

PF = primes[++PF\_idx];

}

if(N != 1) ans -= ans/N;

return ans;

}

* 1. **Diophantine Equations (Extended Euclid)**

// x, y, d are global variables

public void extendedEuclid(int a, int b) {

if(b == 0) {

x = 1;

y = 0;

d = a;

return;

}

extendedEuclid(b, a%b);

int x1 = y;

int y1 = x – (a/b) \* y;

x = x1;

y = y1;

}

**g. Catalan Numbers**

1. **Mathematical Sums**

sum(k) = n(n+1)/2

sum(k)a\_to\_b = (a+b)(b-a+1)/2

sum(k^2) = n(n+1)(2n+1)/6

sum(k^3) = n^2(n+1)^2/4

sum(k^4) = (6n^5+15n^4+10n^3-n)/30

sum(k^5) = (2n^6+6n^5+5n^4-n^2)/12

sum(x^k) = (x^(n+1)-1)/(x-1)

sum(kx^k( = (x-(n+1)x^(n+1) + nx^(n+2))/(x-1)^2

1. **Pythagorean Triples**

x = 2mn, y = m^2-n^2, z = m^2+n^2 where m>n, gcd(m,n)=1 and m~n(mod2)

1. **Geometry**
   1. **Line Intersection**
   2. **Line Segment Intersection Algorithm**

public boolean doesIntersect(Line a, Line b) {

double denominator = ((b.Y2 - b.Y1) \* (a.X2 - a.X1)) - ((b.X2 - b.X1) \*

(a.Y2 - a.Y1));

double numeratorA = ((b.X2 - b.X1) \* (a.Y1 - b.Y1)) - ((b.Y2 - b.Y1) \*

(a.X1 - b.X1));

double numeratorB = ((a.X2 - a.X1) \* (a.Y1 - b.Y1)) - ((a.Y2 - a.Y1) \*

(a.X1 - b.X1));

if(denominator == 0)

return false;

double uA = numeratorA / denominator;

double uB = numeratorB / denominator;

if(uA >= 0 && uA <= 1 && uB >= 0 && uB <= 1)

return true;

return false;

}

1. **2D Point Implementation**

public class Point implements Comparable<Point> {

double x, y;

final Comparator<Point> POLAR\_ORDER = new PolarOrder();

public Point(double x, double y) {

this.x = x;

this.y = y;

}

public static int ccw(Point a, Point b, Point c) {

double area2 = (b.x-a.x)\*(x.y-a.y) – (b.y-a.y)\*(c.x-a.x);

if(area < 0) return -1;

else if(area2 > 0) return 1;

else return 0;

}

private class PolarOrder implements Comparator<Point> {

public int compare(Point q1, Point q2) {

double dx1 = q1.x-x;

double dy1 = q1.y-y;

double dx2 = q2.x-x;

double dy2 = q2.y-y;

if(dy1 >= 0 && dy2 < 0) return -1;

else if(dy2 >= 0 && dy1 < 0) return 1;

else if(dy1 == 0 && dy2 == 0) {

if(dx1 >= 0 && dx2 < 0) return -1;

else if(dx2 >= 0 && dx < 0) return 1;

else return 0;

}

else return –ccw(Point.this, q1, q2);

}

}

}

1. **Convex Hull**

public class GrahamScan {

Stack<Point> hull = new Stack<Point>();

public GrahamScan(Point[] pts) {

int n = pts.length;

Point[] points = new Point[n];

for(int i=0; i<n; i++)

points[i] = pts[i];

Arrays.sort(points);

Arrays.sort(points, 1, n, points[0].POLAR\_ORDER);

hull.push(points[0]);

int k1;

for(k1=1; k1<n; k1++)

if(!points[0].equals(points[k1])) break;

if(k1 == n) return;

int k2;

for(k2=k1+1; k2<n; k2++)

if(Point.ccw(points[0], points[k1], points[k2]) != 0) break;

hull.push(points[k2-1]);

for(int i=k2; i<n; i++) {

Point top = hull.pop();

while(Point.ccw(hull.peek(), top, points[i]) <= 0)

top = hull.pop();

hull.push(top);

hull.push(points[i]);

}

assert isConvex();

}

public Iterable<Point> hull() {

Stack<Point> s = new Stack<Point>();

for(Point p : hull) s.push(p);

return s;

}

private boolean isConvex() {

int n = hull.size();

if(n <= 2) return true;

Point[] points = new Point[n];

int n=0;

for(Point p : hull())

points[n++] = p;

for(int i=0; i<n; i++)

if(Point.ccw(points[i], points[(i+1) % n], points[(i+2) % n])

<= 0) return false;

return true;

}

}

1. **Area of a Polygon**

public double area(Vector<Point> p) {

double result = 0;

double x1, y1, x2, y2;

for(int i=0; i<p.size(); i++) {

x1 = p.elementAt(i).x;

x2 = p.elementAt((i+1) % p.size()).x;

y1 = p.elementAt(i).y;

y2 = p.elementAt((i+1) % p.size()).y;

result += (x1 \* y2 – x2 \* y1);

}

return Math.abs(result)/2.0;

}

1. **Points/Lines**

public class Point implements Comparable<Point> {

double x, y;

public Point(double \_x, double \_y) {

x = \_x;

y = \_y;

}

public int compareTo(Point other) {

if(Math.abs(x-other.x) > 1e-9)

return (int)Math.ceil(x-other.x);

else if(Math.abs(y-other.y) > 1e-9)

return (int)Math.ceil(y-other.y);

else return 0;

}

}

public class Line {

double a, b, c;

double m, c;

}

public class Vec {

double x, y;

}

// methods

public double distance(Point p1, Point p2) {

return Math.hypot(p1.x-p2.x, p1.y-p2.y);

}

public Point rotate(Point p, double theta) {

double rad = Math.toRadians(theta);

return new Point(p.x\*Math.cos(rad) – p.y\*Math.sin(rad),

p.x\*Math.sin(rad) + p.y\*Math.cos(rad));

}

public void pointsToLine(Point p1, Point p2, Line l) {

if(Math.abs(p1.x-p2.x) < 1e-9) {

l.a = 1.0;

l.b = 0.0;

l.c = -p1.x;

}

else {

l.a = -(double)(p1.y-p2.y)/(p1.x-p2.x);

l.b = 1.0;

l.c = -(double)(l.a\*p1.x)-p1.y;

}

}

public int pointsToLine2(Point p1, Point p2, Line l) {

if(Math.abs(p1.x-p2.x) < 1e-9) {

l.m = INF;

l.c = p1.x;

return 0; // vertical line

}

else {

l.m = (double)(p1.y-p2.y)/(p1.x-p2.x);

l.c = p1.y-l.m\*p1.x;

return 1;

}

}

public boolean areParallel(Line l1, Line l2) {

return (Math.abs(l1.a-l2.a) < 1e-9) && (Math.abs(l1.b-l2.b) < 1e-9);

}

public boolean areSame(Line l1, Line l2) {

return areParallel(l1, l2) && (Math.abs(l1.c-l2.c) < 1e-9);

}

public boolean areIntersect(Line l1, Line l2, Point p) {

if(areParallel(l1, l2))

return false;

p.x = (l2.b\*l1.c-l1.b\*l2.c)/(l2.a\*l1.b-l1.a\*l2.b);

if(Math.abs(l1.b) > 1e-9) p.y = -(l1.a\*p.x+l1.c);

else p.y = -(l2.a\*p.x+l2.c);

return true;

}

public Vec toVec(Point a, Point b) {

return new Vec(b.x-a.x, b.y-a.y);

}

public Vec scale(Vec v, double s) {

return new Vec(v.x\*s, v.y\*s);

}

public Point translate(Point p, Vec v) {

return new Point(p.x+v.x, p.y+v.y);

}

1. **Circles**

public boolean circle2PtsRad(Point p1, Point p2, double r, Point ans) {

double d2 = (p1.x-p2.x)\*(p1.x-p2.x) + (p1.y-p2.y)\*(p1.y-p2.y);

double det = r\*r/d2-0.25;

if(det < 0.0) return false;

double h = Math.sqrt(det);

c.x = (p1.x+p2.x)\*0.5 + (p1.y-p2.y)\*h;

c.y = (p1.y+p2.y)\*0.5 + (p2.x-p1.x)\*h;

return true;

}

1. **Triangles**

public double dist(Point p1, Point p2) {

return Math.hypot(p1.x-p2.x, p1.y-p2.y);

}

public double perimeter(Point a, Point b, Point c) {

return dist(a, b) + dist(b, c) + dist(c, a);

}

public double area(Point a, Point b, Point c) {

double s = 0.5\*perimeter(a, b, c);

return Math.sqrt(a)\*Math.sqrt(s-dist(a,b))\*

Math.sqrt(s-dist(b,c))\*Math.sqrt(s-dist(c,a));

}

public double rInCircle(Point a, Point b, Point c) {

return area(a,b,c)/(0.5\*perimeter(a,b,c));

}

public int inCircle(Point p1, Point p2, Point p3, Point ctr, double r) {

r = rInCircle(p1, p2, p3);

if(Math.abs(r) < 1e-9) return 0;

Line l1 = new Line();

Line l2 = new Line();

double ratio = dist(p1, p2)/dist(p1, p3);

Point p = translate(p2, scale(toVec(p2, p3), ratio/(1+ratio)));

pointsToLine(p1, p, l1);

ratio = dist(p2, p1)/dist(p2, p3);

p = translate(p1, scale(toVec(p1, p3), ratio/(1+ratio)));

pointsToLine(p2, p, l2);

areIntersect(l1, l2, ctr);

return 1;

}

public double rCircumCircle(Point a, Point b, Point c) {

return dist(a, b) \* dist(b, c) \* dist(c, a) / (4.0\*area(a, b, c));

}

public double circumCircle(Point p1, Point p2, Point p3, Point ctr) {

double a = p2.x-p1.x, b = p2.y-p1.y;

double c = p3.x-p1.x, d = p3.y-p1.y;

double e = a\*(p1.x+p2.x) + b\*(p1.y+p2.y);

double f = c\*(p1.x+p3.x) + d\*(p1.y+p3.y);

double g = 2.0\*(a\*(p3.y-p2.y)-b\*(p3.x-p2.x));

if(Math.abs(g) < 1e-9) return 0;

ctr.x = (d\*e – b\*f)/g;

ctr.y = (a\*f – c\*e)/g;

return dist(p1, ctr);

}

public boolean inCircumCircle(Point a, Point b, Point c, Point d) {

return ((a.x-d.x)\*(b.y-d.y)\*((c.x-d.x)\*(c.x-d.x)

+ (c.y-d.y)\*(c.y-d.y)) + (a.y-d.y)\*((b.x-d.x)\*(b.x-d.x)

+ (b.y-d.y)\*(b.y-d.y))\*(c.x - d.x)

+ ((a.x-d.x)\*(a.x-d.x) + (a.y-d.y)\*(a.y-d.y))\*(b.x-d.x)\*(c.y-d.y)

- ((a.x-d.x)\*(a.x-d.x) + (a.y-d.y)\*(a.y-d.y))\*(b.y-d.y)\*(c.x-d.x)

- (a.y-d.y)\*(b.x-d.x)\*((c.x-d.x)\*(c.x-d.x)

+ (c.y-d.y)\*(c.y-d.y)) - (a.x-d.x)\*((b.x-d.x)\*(b.x-d.x)

+ (b.y-d.y)\*(b.y-d.y))\*(c.y-d.y)) > 0.0;

}

public boolean canFormTriangle(double a, double b, double c) {

return (a+b>c) && (a+c>b) && (b+c>a);

}

1. **Polygons**

public double dot(Vec a, Vec b) {

return (a.x\*b.x + a.y\*b.y);

}

public double norm\_sq(Vec v) {

return v.x\*v.x + v.y\*v.y;

}

public double angle(Point a, Point o, Point b) {

Vec oa = toVec(o, a), ob = toVec(o, b);

return Math.acos(dot(oa, ob) / Math.sqrt(norm\_sq(oa) \* norm\_sq(ob)));

}

public double cross(Vec a, Vec b) {

return a.x\*b.y – a.y\*b.x;

}

public boolean ccw(Point p, Point q, Point r) {

return cross(toVec(p, q), toVec(p, r)) > 0; // >= 0 to check collinear

}

public boolean collinear(Point p, Point q, Point r) {

return Math.abs(cross(toVec(p, q), toVec(p, r))) < 1e-9;  
}

public boolean isConvex(List<Point p) {

int sz = (int)P.size();

if(sz <= 3) return false;

boolean isLeft = ccw(P.get(0), P.get(1), P.get(2));

for(int i=1; i<sz-1; i++)

if(ccw(P.get(i), P.get(i+1), P.get((i+2)==sz?1:i+2)) != isLeft)

return false;

return true;

}

public boolean inPolygon(Point pt, List<Point> P) {

if((int)P.size() == 0) return false;

double sum = 0;

for(int i=0; i<(int)P.size()-1; i++) {

if(ccw(pt, P.get(i), P.get(i+1)))

sum += angle(P.get(i), pt, P.get(i+1));

else sum -= angle(P.get(i), pt, P.get(i+1));

}

return Math.abs(Math.abs(sum)-2\*Math.PI) < 1e-9;

}

public Point lineIntersectSeg(Point p, Point q, Point A, Point B) {

double a = B.y – A.y;

double b = A.x – B.x;

double c = B.x \* A.y – A.x \* B.y;

double u = Math.abs(a\*p.x + b\*p.y + c);

double v = Math.abs(a\*q.x + b\*q.y + c);

return new Point((p.x\*v + q.x\*u) / (u+v), (p.y\*v + q.y\*u)/(u+v));

}

6. **String Manipulation**

a. **Regular Expressions**

**Extracting and Capturing**

import java.util.regex.\*;

public static void main(String[] args) {

String input = "I have a cat, but I like my dog better.";

Pattern p = Pattern.compile("(mouse|cat|dog|wolf|bear|human)");

Matcher m = p.matcher(input);

List<String> animals = new ArrayList<String>();

while (m.find()) {

System.out.println("Found a " + m.group() + ".");

animals.add(m.group());

}

}

**Modifying and Substitution**

public static void main(String[] args) {

String input = "User clientId=23421. Some more text

clientId=33432. This clientNum=100";

Pattern p = Pattern.compile("(clientId=)(\\d+)");

Matcher m = p.matcher(input);

StringBuffer result = new StringBuffer();

while (m.find()) {

System.out.println("Masking: " + m.group(2));

m.appendReplacement(result, m.group(1) + "\*\*\*masked\*\*\*");

}

m.appendTail(result);

System.out.println(result);

}

b. **String Matching (KMP-Strong)**

private static void preprocess() {

int k=1;

for(int j=2; j<pattern.length(); j++) {

if(pattern.charAt(j) == pattern.charAt(k))

next[j] = next[k];

else

next[j] = k;

while(k > 0 && pattern.charAt(j) != pattern.charAt(k))

k = next[k];

k++;

}

}

private static void process(String[] lines) {

for(int i=0; i<lines.length; i++) {

String line = lines[i];

String temp = "" + line;

j++;

if(Collections.binarySearch(options, 'I') >= 0) {

line = line.toLowerCase();

pattern = pattern.toLowerCase();

}

int i=0, k=1;

while(i < line.length() && k < pattern.length()) {

while(k >= 1 && line.charAt(i) != pattern.charAt(k))

k = next[k];

i++;

k++;

}

if(k == pattern.length()) {

lines.addElement("[" + j + "]" + temp);

count++;

}

}

}

* 1. **String Alignment**

// dp table is global

public int strAlign(char[] A, char[] B) {

int i, j, n = A.length, m = B.length;

for(i=1; i<=n; i++) table[i][0] = i\*-1;

for(j=1; j<=m; j++) table[0][j] = j\*-1;

for(i=1; i<=n; i++)

for(j=1; j<=m; j++) {

table[i][j] = table[i-1][j-1] + (A[i-1] == B[j-1] ? 2 : -1);

table[i][j] = Math.max(table[i][j], table[i-1][j]-1);

table[i][j] = Math.max(table[i][j], table[i][j-1]-1);

}

return table[n][m];

}