Inhaltsverzeichnis

Standard IO system

```
import java.util.*;

class Main
{
	public static void main(String... strings)
	{
		Scanner sc = new Scanner(System.in);

		int n = sc.nextInt();

		System.out.println(String.format("%d", n));
	}
}
```

Complex Numbers

z = r (cos(phi) + i sin(phi)) = r e^(i phi) = (a + b i)
r = sqrt(a^2 + b^2)
phi = +- arctan2(b / a)

Adding Matrix Powers

A^1+A^2+...+A^n = S[n]
S[n] = S[n/2] + A^(n/2) * S[n/2]; // when n is even
S[n] = A + A * S[n-1]; // when n is odd

Square and Multiply

```
// computes a number to the power of e.
public Number pow(int e)
{
	Number result = this;
	List<Number> tomult = new ArrayList<Number>();
	while (e > 1)
	{
		if ((e & 1) == 0)
		{
			result = result.mult(result);
			e >>= 1;
		}
		else
		{
			tomult.add(new Number(this.clone()));
			e--;
		}
	}
	for (Number c : tomult)
		result = result.mult(c);
	return result;
}
```

Backtracking

```
// start with track(new ArrayList<Integer>(), 0, 0);
// it will fill „solutions“ with all column-solutions sorted by row.
private static List<List<Integer>> solutions;
private static void track(List<Integer> mySolution, int row, int max)
{
	if(row >= max)
	{
		solutions.add(mySolution);
		return;
	}

	for(int col=0; col<max; col++)
	{
		if(possibleSolution(mySolution, row, col))
		{
			List<Integer> s = new ArrayList<Integer>(mySolution);
			s.add(col);
			track(s, row + 1, max);
		}
	}
}
// you can give the criteria for a possible solution here
private static void possibleSolution(List<Integer> mySolution, int row, int col)
{
	if( „adding an element to (row,col) is possible in mySolution“ )
		return true;
	else
		return false;
}
```

Number of Digits

Number of Digits(N) = Math.floor(Math.log10(N)) + 1

Point Class for Graham Scan

```
public static Point firstPoint;
// Point class representing one point in a cloud of points. Useful for graham scan.
public static class Point implements Comparable<Point>
{
	public int y;
	public int x;

	public Point(int x, int y)
	{
		this.x = x;
		this.y = y;
	}

	// determines the polar angle between this and a predefined point.
	public double angle()
	{
		return Math.atan2((double) (y - firstPoint.y), (double) (x - firstPoint.x));
	}

	public double size()
	{
		return Math.sqrt((long) (x - firstPoint.x) * (long) (x - firstPoint.x) + (long) (y -
firstPoint.y) * (long) (y - firstPoint.y));
	}
```

```
	@Override
	public int compareTo(Point p2)
	{
		if (Math.abs(this.cotangent() - p2.cotangent()) < 1e-5)
			if (this.size() < p2.size())
				return -1;
			else if (this.size() > p2.size())
				return 1;
			else
				return 0;
		if (this.cotangent() < p2.cotangent())
			return -1;
		return 1;
	}
}
```

Convex Hull (Graham Scan)

```
List<Point> pointList = new ArrayList<Point>();
// start testcase, read in all points and add them to pointList.
// also determine the point with the lowest y coordinate (and most left x if y is the same).
...
// move the point with lowest y  coordinate to index 0
Collections.swap(pointList, 0, pointList.indexOf(pWithLowestY));

// save the first point for size and angle measurement
firstPoint = pWithLowestY;

// sort points by polar coordinate angle, see „point class for graham scan"
Collections.sort(pointList);

// add the first point at the end to close the polygon (not absolutely necessary)
pointList.add(new Point(pWithLowestY.x, pWithLowestY.y));

// start with graham scan, always take the most right point from the current point.
int i = 2;
while (i < pointList.size())
{
	// if the points are in one line, remove the middle point
	if (ccw(pointList.get(i - 2), pointList.get(i - 1), pointList.get(i)) == 0)
	{
		pointList.remove(i - 1);
	}
	else
	{
		// if the points are directed to the left, its ok
		if (ccw(pointList.get(i - 2), pointList.get(i - 1), pointList.get(i)) > 0)
		{
			i++;
		}
		// if they are directed "concave", remove the middle one and go back
		else
		{
			pointList.remove(i - 1);
			i--;
		}
	}
}

// now the pointList contains only points on the hull
for (Point p : pointList)
	System.out.println(String.format("%d %d", p.x, p.y));
```

Binomial Sums

$$\sum_{k=0}^{n}\binom{n}{k}=2^n$$

$$\sum_{k=1}^{n}k\binom{n}{k}=n2^{n-1}$$

$$(x+1)^n=\sum_{i=1}^{n}\binom{n}{i}x^i$$

Matrix Class

```
// Matrix class. Can add and multiply other matrizes. For computing A^n, see „Square and Multiply“.
public class Matrix
{
	private final int[][] elements;
	private final int rows,cols;
	public Matrix(int n, int m)
	{
		elements = new int[n][m];
		this.rows = n;
		this.cols = m;
	}

	// multiplicates a matrix with the current and returns the result
	public Matrix mult(Matrix matrix)
	{
		Matrix result = new Matrix(cols, matrix.rows);

		for(int lrows=0; lrows < rows; lrows++)
		{
			for(int lcols=0; lcols < cols; lcols++)
			{
				result.elements[lrows][lcols] = 0;
				for(int k=0;k<cols; k++)
					result.elements[lrows][lcols] += elements[lrows][k]*matrix.elements[k]
[lcols];
			}
		}
		return result;
	}

	// adds a matrix to the current and returns the result
	public Matrix add(Matrix matrix)
	{
		Matrix result = new Matrix(cols, rows);
		for(int lrows=0; lrows<rows; lrows++)
			for(int lcols=0; lcols<cols; lcols++)
				result.elements[lrows][lcols] = this.elements[lrows][lcols] + matrix.elements[lrows]
[lcols];
		return result;
	}
}
```

Fast prime generation

```
// Calculate all prime numbers in the range 1 to MAX_PRIME
// using the sieve of Eratosthenes.
// numbers in the "numbers" array that are not prime are are set to -1.
static final int MAX_PRIME = 5000000;
static int[] numbers = new int[MAX_PRIME+5];

private static void generatePrimes()
{
	numbers[0] = -1;
	numbers[1] = -1;
	for (int i = 2; i <= MAX_PRIME+1; i++)
	{
		if (numbers[i] >= 0)
		{
			numbers[i] = i;
			for (int k = 2; k * i <= MAX_PRIME+1; k++)
			{
				numbers[k * i] = -1;
			}
		}
	}
}
```

Special Sums

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{\infty} \frac{x^i}{i} = -\ln(1-x)$$