~/.vimrc Configuration File:

```
set shiftwidth=2
set tabstop=2
set nu
set autoindent
syntax on
```

Control Mode Commands:

- Repeat insertion n times: `n i TYPE_WHILE_IN_INSERT_MODE ESCAPE`
- Repeat last command: `.`
- Indent current line: `>>`
- Indent n lines, use ‘<’ for unindent, ‘>>’ for two indents, etc.: `>n`
- Replace ‘foo’ with ‘bar’, including ‘%’ searches the entire file: `:%s/foo/bar/g`
- Pastes n times: `np`
- Undo: `u`
- Return to console, putting vim in background: `ctrl+z` (NOTE: Running `fg` in the terminal returns to vim)
- Duplicate line: `yy` to copy and `p` to paste.
- Print current file (incuding syntax highlighting): `:hardcopy`

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Java J2SE “Regular Expressions” Cheat Sheet v 0.1

Metacharacters

```
([{\^$|)?*+.
```

Character Classes

[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z, or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z] (subtraction)

Predefined Character Classes

.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

Quantifiers

Greedy	Reluctant	Possessive	Meaning
X?	X??	X?+	X, once or not at all
X*	X*?	X*+	X, zero or more times
X+	X+?	X++	X, one or more times
X{n}	X{n}?	X{n}+	X, exactly n times
X{n,}	X{n,}?	X{n,}+	X, at least n times
X{n,m}	X{n,m}?	X{n,m}+	X, at least n but not more than m times

Boundary Matchers

^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

Class Pattern Fields

CANON_EQ	Enables canonical equivalence.
CASE_INSENSITIVE	Enables case-insensitive matching.
COMMENTS	Permits whitespace and comments in pattern.
DOTALL	Enables dotall mode.
MULTILINE	Enables multiline mode.
UNICODE_CASE	Enables Unicode-aware case folding.
UNIX_LINES	Enables Unix lines mode.

Class Matcher Methods

static Pattern compile(String regex)
Compiles the given regular expression into a pattern.

static Pattern compile(String regex, int flags)
Compiles the given regular expression into a pattern with the given flags.

int flags()
Returns this pattern's match flags.

Matcher matcher(CharSequence input)
Creates a matcher that will match the given input against this pattern.

static Boolean matches(String regex, CharSeq input)
Compiles the given regular expression and attempts to match the given input against it.

String pattern()
Returns the regular expression from which this pattern was compiled.

String[] split(CharSequence input)
Splits the given input sequence around matches of this pattern.

String[] split(CharSequence input, int limit)
Splits the given input sequence around matches of this pattern.

Class Matcher Methods

Matcher appendReplacement(StringBuffer sb, String replacement)
Implements a non-terminal append-and-replace step.

StringBuffer appendTail(StringBuffer sb)
Implements a terminal append-and-replace step.

int end()
Returns the index of the last character matched, plus one.

int end(int group)
Returns the index of the last character, plus one, of the subsequence captured by the given group during the previous match operation.

boolean find()
Attempts to find the next subsequence of the input sequence that matches the pattern.

boolean find(int start)
Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.

String group()
Returns the input subsequence matched by the previous match.

String group(int group)
Returns the input subsequence captured by the given group during the previous match operation.

int groupCount()
Returns the number of capturing groups in this matcher's pattern.

boolean lookingAt()
Attempts to match the input sequence, starting at the beginning, against the pattern.

boolean matches()
Attempts to match the entire input sequence against the pattern.

Pattern pattern()
Returns the pattern that is interpreted by this matcher.

String replaceAll(String replacement)
Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.

String replaceFirst(String replacement)
Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.

Matcher reset()
Resets this matcher.

Matcher reset(CharSequence input)
Resets this matcher with a new input sequence.

int start()
Returns the start index of the previous match.

int start(int group)
Returns the start index of the subsequence captured by the given group during the previous match operation.

Number Series:

- Fibonacci: `1,1,2,3,5,8,13,21,34,55,89...`
- Catalan: `1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796...`

This series has many appearances in combinatorics. Starting with C_0 = 1, we can calculate using the recurrence relation `C_(n+1) = ((2(2n+1))/(n+2))*C_n`, or the following formulas can be used:

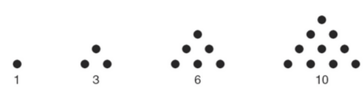
$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k} \qquad \text{for } n \geq 0.$$

- Lazy Caterer: `1, 2, 4, 7, 11, 16, 22, 29, 37, 46, 56, 67, 79, 92, 106, 121, 137...`

The maximum number of pieces of a circle that can be made with a given number of straight cuts. For example, three cuts across a circle will produce six pieces if the cuts all meet at a common point, but seven if they do not. Calculated using: `p = (n^2 + n + 2)/2 for n >= 0`

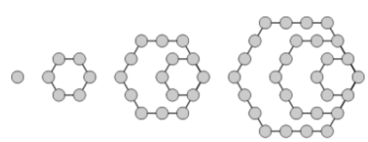
- Trianglar: `0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55...`

$$T_n = \sum_{k=1}^{n} k = 1+2+3+\cdots+n = \frac{n(n+1)}{2} = \binom{n+1}{2}$$



- Hexagonal: `1, 6, 15, 28, 45, 66, 91, 120, 153, 190, 231...`

$$h_n = 2n^2 - n = n(2n-1) = \frac{2n \times (2n-1)}{2}$$



- Subfactorial: `0 1 2 9 44 265 1854 14833...`

Denoted !n, this represents a lot of common patterns, notably the number of ways elements can be arranged such that each element is not found in it's starting position.

$$!n = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!}.$$

Command line flags:

- Set maximum Java heap size: `-Xmx<size>` (**For example:** `-Xmx1024k`, `-Xmx512m`, `-Xmx8g`)
- Set maxmimum Java thread stack size: `-Xss<size>`

Input:

- Fastest known way to read a large number of space-separated integers from a line (was tested with up to 200000 integers on one lines). NOTE: This method of using a BufferedReader is faster than Scanner for other uses too.

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
StringTokenizer token  = new StringTokenizer(br.readLine());
for (int i = 0; i < n; i++)
    int n = Integer.parseInt(token.nextToken());
```

Conversions:

- Character to Integer: `int val = Character.getNumericValue(char c);`
- Integer to Character: `Character c = i + '0;'`
- ArrayList to Set: `Set<Foo> listName = new HashSet<Foo>(arrayListName);`
- Set to ArrayList: `ArrayList<Integer> arrayListName = new ArrayList<Integer>(mySet);`
- Base x to Base 10 (where 2 <= x <= 36): `int base10 = Integer.parseInt(strBaseX, x);`

Comparator:

```
// Example: Sort YourObjects by ID
class YourComparator implements Comparator<YourObject> {
    @Override public int compare(YourObject a, YourObject b) {
        return (new Integer(a.id)).compareTo(b.id);
    }
}
```

Derivatives

Basic Properties/Formulas/Rules

$\frac{d}{dx}\left(cf(x)\right)=cf'(x)$, c is any constant. $\left(f(x)\pm g(x)\right)'=f'(x)\pm g'(x)$

$\frac{d}{dx}\left(x^n\right)=nx^{n-1}$, n is any number. $\frac{d}{dx}(c)=0$, c is any constant.

$(f\,g)'=f'\,g+f\,g'$ – **(Product Rule)** $\left(\frac{f}{g}\right)'=\frac{f'\,g-f\,g'}{g^2}$ – **(Quotient Rule)**

$\frac{d}{dx}\left(f\left(g(x)\right)\right)=f'\left(g(x)\right)g'(x)$ **(Chain Rule)**

$\frac{d}{dx}\left(\mathbf{e}^{g(x)}\right)=g'(x)\mathbf{e}^{g(x)}$ $\frac{d}{dx}\left(\ln g(x)\right)=\frac{g'(x)}{g(x)}$

Common Derivatives

Polynomials

$\frac{d}{dx}(c)=0$ $\frac{d}{dx}(x)=1$ $\frac{d}{dx}(cx)=c$ $\frac{d}{dx}\left(x^n\right)=nx^{n-1}$ $\frac{d}{dx}\left(cx^n\right)=ncx^{n-1}$

Trig Functions

$\frac{d}{dx}(\sin x)=\cos x$ $\frac{d}{dx}(\cos x)=-\sin x$ $\frac{d}{dx}(\tan x)=\sec^2 x$

$\frac{d}{dx}(\sec x)=\sec x\tan x$ $\frac{d}{dx}(\csc x)=-\csc x\cot x$ $\frac{d}{dx}(\cot x)=-\csc^2 x$

Inverse Trig Functions

$\frac{d}{dx}\left(\sin^{-1}x\right)=\frac{1}{\sqrt{1-x^2}}$ $\frac{d}{dx}\left(\cos^{-1}x\right)=-\frac{1}{\sqrt{1-x^2}}$ $\frac{d}{dx}\left(\tan^{-1}x\right)=\frac{1}{1+x^2}$

$\frac{d}{dx}\left(\sec^{-1}x\right)=\frac{1}{|x|\sqrt{x^2-1}}$ $\frac{d}{dx}\left(\csc^{-1}x\right)=-\frac{1}{|x|\sqrt{x^2-1}}$ $\frac{d}{dx}\left(\cot^{-1}x\right)=-\frac{1}{1+x^2}$

Exponential/Logarithm Functions

$\frac{d}{dx}\left(a^x\right)=a^x\ln(a)$ $\frac{d}{dx}\left(\mathbf{e}^x\right)=\mathbf{e}^x$

$\frac{d}{dx}\left(\ln(x)\right)=\frac{1}{x},\ x>0$ $\frac{d}{dx}\left(\ln|x|\right)=\frac{1}{x},\ x\neq 0$ $\frac{d}{dx}\left(\log_a(x)\right)=\frac{1}{x\ln a},\ x>0$

Hyperbolic Trig Functions

$\frac{d}{dx}(\sinh x)=\cosh x$ $\frac{d}{dx}(\cosh x)=\sinh x$ $\frac{d}{dx}(\tanh x)=\operatorname{sech}^2 x$

$\frac{d}{dx}(\operatorname{sech} x)=-\operatorname{sech} x\tanh x$ $\frac{d}{dx}(\operatorname{csch} x)=-\operatorname{csch} x\coth x$ $\frac{d}{dx}(\coth x)=-\operatorname{csch}^2 x$

Integrals

Basic Properties/Formulas/Rules

$\int cf(x)\,dx = c\int f(x)\,dx$, c is a constant. $\int f(x) \pm g(x)\,dx = \int f(x)\,dx \pm \int g(x)\,dx$

$\int_a^b f(x)\,dx = F(x)\Big|_a^b = F(b) - F(a)$ where $F(x) = \int f(x)\,dx$

$\int_a^b cf(x)\,dx = c\int_a^b f(x)\,dx$, c is a constant. $\int_a^b f(x) \pm g(x)\,dx = \int_a^b f(x)\,dx \pm \int_a^b g(x)\,dx$

$\int_a^a f(x)\,dx = 0$ $\int_a^b f(x)\,dx = -\int_b^a f(x)\,dx$

$\int_a^b f(x)\,dx = \int_a^c f(x)\,dx + \int_c^b f(x)\,dx$ $\int_a^b c\,dx = c(b-a)$

If $f(x) \geq 0$ on $a \leq x \leq b$ then $\int_a^b f(x)\,dx \geq 0$

If $f(x) \geq g(x)$ on $a \leq x \leq b$ then $\int_a^b f(x)\,dx \geq \int_a^b g(x)\,dx$

Common Integrals

Polynomials

$\int dx = x + c$ $\int k\,dx = k\,x + c$ $\int x^n dx = \frac{1}{n+1}x^{n+1} + c,\ n \neq -1$

$\int \frac{1}{x}dx = \ln|x| + c$ $\int x^{-1}\,dx = \ln|x| + c$ $\int x^{-n}dx = \frac{1}{-n+1}x^{-n+1} + c,\ n \neq 1$

$\int \frac{1}{ax+b}dx = \frac{1}{a}\ln|ax+b| + c$ $\int x^{\frac{p}{q}}dx = \frac{1}{\frac{p}{q}+1}x^{\frac{p}{q}+1} + c = \frac{q}{p+q}x^{\frac{p+q}{q}} + c$

Trig Functions

$\int \cos u\,du = \sin u + c$ $\int \sin u\,du = -\cos u + c$ $\int \sec^2 u\,du = \tan u + c$

$\int \sec u \tan u\,du = \sec u + c$ $\int \csc u \cot u\,du = -\csc u + c$ $\int \csc^2 u\,du = -\cot u + c$

$\int \tan u\,du = \ln|\sec u| + c$ $\int \cot u\,du = \ln|\sin u| + c$

$\int \sec u\,du = \ln|\sec u + \tan u| + c$ $\int \sec^3 u\,du = \frac{1}{2}\left(\sec u \tan u + \ln|\sec u + \tan u|\right) + c$

$\int \csc u\,du = \ln|\csc u - \cot u| + c$ $\int \csc^3 u\,du = \frac{1}{2}\left(-\csc u \cot u + \ln|\csc u - \cot u|\right) + c$

Exponential/Logarithm Functions

$\int \mathbf{e}^u\,du = \mathbf{e}^u + c$ $\int a^u\,du = \frac{a^u}{\ln a} + c$ $\int \ln u\,du = u\ln(u) - u + c$

$\int \mathbf{e}^{au}\sin(bu)\,du = \frac{\mathbf{e}^{au}}{a^2+b^2}\left(a\sin(bu) - b\cos(bu)\right) + c$ $\int u\mathbf{e}^u du = (u-1)\mathbf{e}^u + c$

$\int \mathbf{e}^{au}\cos(bu)\,du = \frac{\mathbf{e}^{au}}{a^2+b^2}\left(a\cos(bu) + b\sin(bu)\right) + c$ $\int \frac{1}{u\ln u}du = \ln|\ln u| + c$

Trigonometry:

$\sin A = \dfrac{\text{opp leg}}{\text{hypotenuse}}$

$\cos A = \dfrac{\text{adj leg}}{\text{hypotenuse}}$

$\tan A = \dfrac{\text{opp leg}}{\text{adj leg}} = \dfrac{\sin A}{\cos A}$

$\csc A = \dfrac{\text{hypotenuse}}{\text{opp leg}} = \dfrac{1}{\sin A}$

$\sec A = \dfrac{\text{hypotenuse}}{\text{adj leg}} = \dfrac{1}{\cos A}$

$\cot A = \dfrac{\text{adj leg}}{\text{opp leg}} = \dfrac{1}{\tan A}$

Values to Memorize:

$\sin 30° = \dfrac{1}{2} = \cos 60°$

$\cos 30° = \dfrac{\sqrt{3}}{2} = \sin 60°$

$\tan 30° = \dfrac{\sqrt{3}}{3} = \cot 60°$

$\sin 45° = \cos 45° = \dfrac{\sqrt{2}}{2}$

$\tan 45° = 1$

$\sin 15° = \dfrac{\sqrt{6} - \sqrt{2}}{4} = \cos 75°$

$\cos 15° = \dfrac{\sqrt{6} + \sqrt{2}}{4} = \sin 75°$

$\tan 15° = 2 - \sqrt{3}$, $\tan 75° = 2 + \sqrt{3}$

Golden rectangle & regular pentagon.

$\sin 18° = \cos 72° = \dfrac{\sqrt{5} - 1}{4}$

$\cos 36° = \sin 54° = \dfrac{\sqrt{5} + 1}{4}$

Pythagorean Identities

$\sin^2 A + \cos^2 A = 1$

$1 + \tan^2 A = \sec^2 A$

$1 + \cot^2 A = \csc^2 A$

Odd-Even Functions:

$\sin(-A) = -\sin(A)$

$\cos(-A) = \cos(A)$

$\tan(-A) = -\tan(A)$

Complements A & B

$\sin^2 A + \sin^2 B = 1$

$\sin A = \cos B$, etc

Sum & Difference Identities

$\sin(A \pm B) = \sin A \cdot \cos B \pm \cos A \cdot \sin B$

$\cos(A \pm B) = \cos A \cdot \cos B \mp \sin A \cdot \sin B$

$\tan(A \pm B) = \dfrac{\tan A \pm \tan B}{1 \mp \tan A \cdot \tan B}$

Double Angle Identities

$\sin 2A = 2 \sin A \cdot \cos A$

$\cos 2A = \cos^2 A - \sin^2 A$

or $= 1 - 2\sin^2 A = 2\cos^2 A - 1$

$\tan 2A = \dfrac{2 \tan A}{1 - \tan^2 A}$

Sum to Product

$\sin A + \sin B = 2 \sin \dfrac{A+B}{2} \cos \dfrac{A-B}{2}$

$\sin A - \sin B = 2 \sin \dfrac{A-B}{2} \cos \dfrac{A+B}{2}$

$\cos A + \cos B = 2 \cos \dfrac{A+B}{2} \cos \dfrac{A-B}{2}$

$\cos A - \cos B = -2 \sin \dfrac{A+B}{2} \sin \dfrac{A-B}{2}$

$\tan A \pm \tan B = \dfrac{\sin(A \pm B)}{\cos A \cdot \cos B}$

Triple Angle Identities

$\sin 3A = 3 \sin A - 4 \sin^3 A$

$\cos 3A = 4 \cos^3 A - 3 \cos A$

$\tan 3A = \dfrac{\tan A \cdot (\tan^2 A - 3)}{3 \tan^2 A - 1}$

Half Angle Formulas

$\sin \dfrac{A}{2} = \pm \sqrt{\dfrac{1 - \cos A}{2}}$

$\cos \dfrac{A}{2} = \pm \sqrt{\dfrac{1 + \cos A}{2}}$

$\tan \dfrac{A}{2} = \dfrac{\sin A}{1 + \cos A} = \dfrac{1 - \cos A}{\sin A}$

Product to Sum

$\sin A \cdot \sin B = \frac{1}{2}[\cos(A-B) - \cos(A+B)]$

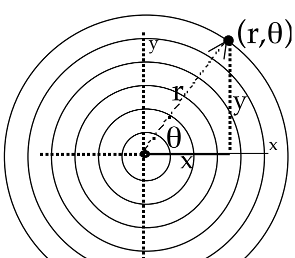
$\cos A \cdot \cos B = \frac{1}{2}[\cos(A-B) + \cos(A+B)]$

$\sin A \cdot \cos B = \frac{1}{2}[\sin(A-B) + \sin(A+B)]$

$\tan A \cdot \tan B = \dfrac{\cos(A-B) - \cos(A+B)}{\cos(A-B) + \cos(A+B)}$

Polar Coordinates

Points are represented in terms of (r,θ) rather than (x,y)



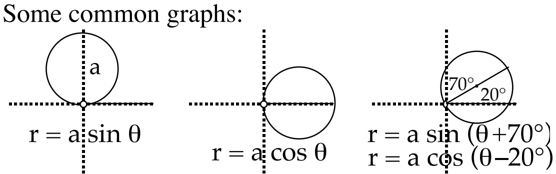
$x^2 + y^2 = r^2$

$\tan \theta = \dfrac{y}{x}$

$x = r \cos \theta$

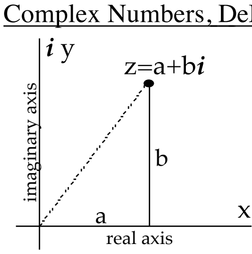
$y = r \sin \theta$

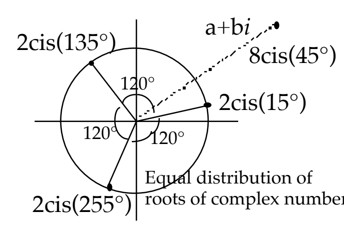
Some common graphs:



$r = a \sin \theta$ $\quad$ $r = a \cos \theta$ $\quad$ $r = a \sin(\theta + 70°)$, $r = a \cos(\theta - 20°)$

Complex Numbers, DeMoivre's Thm, Euler's Thm & C*I* S





$Z = a + b\boldsymbol{i} = r \operatorname{cis} \theta$ (polar form of complex number)

The magnitude, $r = |a + b\boldsymbol{i}| = \sqrt{a^2 + b^2}$

$e^{\boldsymbol{i}\theta} = \cos\theta + \boldsymbol{i}\sin\theta = \operatorname{cis}\theta$ (Euler)

$\operatorname{cis}(A + B) = \operatorname{cis} A \cdot \operatorname{cis} B$ $\quad$ $\operatorname{cis}(A - B) = \dfrac{\operatorname{cis} A}{\operatorname{cis} B}$

DeMoivre's Theorems: (see illustration above)

$(a + b\boldsymbol{i})^n = (r \operatorname{cis} \theta)^n = r^n \operatorname{cis}(n \cdot \theta)$ for $n =$ pos int

$\sqrt[n]{r \cdot \operatorname{cis} \theta} = \sqrt[n]{r} \cdot \operatorname{cis}\left(\dfrac{2\pi k + \theta}{n}\right)$ for $k = 0,1,2,3\ldots,n-1$

Combinatorics

Sums

$\sum_{k=0}^{n} k = n(n+1)/2$ $\quad$ $\sum_{k=a}^{b} k = (a+b)(b-a+1)/2$
$\sum_{k=0}^{n} k^2 = n(n+1)(2n+1)/6$ $\quad$ $\sum_{k=0}^{n} k^3 = n^2(n+1)^2/4$
$\sum_{k=0}^{n} k^4 = (6n^5+15n^4+10n^3-n)/30$ $\quad$ $\sum_{k=0}^{n} k^5 = (2n^6+6n^5+5n^4-n^2)/12$
$\sum_{k=0}^{n} x^k = (x^{n+1}-1)/(x-1)$ $\quad$ $\sum_{k=0}^{n} kx^k = (x-(n+1)x^{n+1}+nx^{n+2})/(x-1)^2$
$1+x+x^2+\cdots = 1/(1-x)$

Binomial coefficients

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1												
1	1	1											
2	1	2	1										
3	1	3	3	1									
4	1	4	6	4	1								
5	1	5	10	10	5	1							
6	1	6	15	20	15	6	1						
7	1	7	21	35	35	21	7	1					
8	1	8	28	56	70	56	28	8	1				
9	1	9	36	84	126	126	84	36	9	1			
10	1	10	45	120	210	252	210	120	45	10	1		
11	1	11	55	165	330	462	462	330	165	55	11	1	
12	1	12	66	220	495	792	924	792	495	220	66	12	1
	0	1	2	3	4	5	6	7	8	9	10	11	12

$\binom{n}{k} = \frac{n!}{(n-k)!k!}$

$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

$\binom{n+1}{k} = \frac{n+1}{n-k+1}\binom{n}{k}$

$\binom{n}{k+1} = \frac{n-k}{k+1}\binom{n}{k}$

$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$

$\binom{n}{k} = \frac{n-k+1}{k}\binom{n}{k-1}$

$12! \approx 2^{28.8}$
$20! \approx 2^{61.1}$

Number of ways to pick a multiset of size k from n elements: $\binom{n+k-1}{k}$
Number of n-tuples of non-negative integers with sum s: $\binom{s+n-1}{n-1}$, at most s: $\binom{s+n}{n}$
Number of n-tuples of positive integers with sum s: $\binom{s-1}{n-1}$
Number of lattice paths from $(0,0)$ to (a,b), restricted to east and north steps: $\binom{a+b}{a}$

Multinomial theorem. $(a_1+\cdots+a_k)^n = \sum \binom{n}{n_1,\ldots,n_k} a_1^{n_1}\ldots a_k^{n_k}$, where $n_i \geqslant 0$ and $\sum n_i = n$.
$\binom{n}{n_1,\ldots,n_k} = M(n_1,\ldots,n_k) = \frac{n!}{n_1!\ldots n_k!}$. $M(a,\ldots,b,c,\ldots) = M(a+\cdots+b,c,\ldots)M(a,\ldots,b)$

Catalan numbers. $C_n = \frac{1}{n+1}\binom{2n}{n}$. $\quad C_0 = 1$, $C_n = \sum_{i=0}^{n-1} C_i C_{n-1-i}$. $\quad C_{n+1} = C_n \frac{4n+2}{n+2}$.
$C_0, C_1, \ldots = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, \ldots$
C_n is the number of: properly nested sequences of n pairs of parentheses; rooted ordered binary trees with $n+1$ leaves; triangulations of a convex $(n+2)$-gon.

Derangements. Number of permutations of $n = 0, 1, 2, \ldots$ elements without fixed points is $1, 0, 1, 2, 9, 44, 265, 1854, 14833, \ldots$ Recurrence: $D_n = (n-1)(D_{n-1}+D_{n-2}) = nD_{n-1} + (-1)^n$. Corollary: number of permutations with exactly k fixed points is $\binom{n}{k}D_{n-k}$.

Stirling numbers of 1^{st} kind. $s_{n,k}$ is $(-1)^{n-k}$ times the number of permutations of n elements with exactly k permutation cycles. $|s_{n,k}| = |s_{n-1,k-1}| + (n-1)|s_{n-1,k}|$. $\quad \sum_{k=0}^{n} s_{n,k}\, x^k = x^{\underline{n}}$

Stirling numbers of 2^{nd} kind. $S_{n,k}$ is the number of ways to partition a set of n elements into exactly k non-empty subsets. $S_{n,k} = S_{n-1,k-1} + kS_{n-1,k}$. $S_{n,1} = S_{n,n} = 1$. $x^n = \sum_{k=0}^{n} S_{n,k}\, x^{\underline{k}}$

Bell numbers. B_n is the number of partitions of n elements. $B_0, \ldots = 1, 1, 2, 5, 15, 52, 203, \ldots$
$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k = \sum_{k=1}^{n} S_{n,k}$. Bell triangle: $B_r = a_{r,1} = a_{r-1,r-1}$, $a_{r,c} = a_{r-1,c-1} + a_{r,c-1}$.

NOTE: Java's Math class has methods for toRadians(double degrees) and toDegrees(double radians). It also has a method called hypot(double x, double y), which returns sqrt(x^2 +y^2).

Find Center of Circle given two points and a radius:

```
// Note: plus=TRUE gives one possible point, plus=FALSE gives the other possible point
static double cx(double a, double b, double c, double d, double r, boolean plus) {
  double q = Math.sqrt((a-c)*(a-c) + (b-d)*(b-d));
  double x3 = (a+c)/2.0;
  double y3 = (b+d)/2.0;
  if (plus)
    return x3 + Math.sqrt(r*r-(q/2)*(q/2))*(b-d)/q;
  return x3 - Math.sqrt(r*r-(q/2)*(q/2))*(b-d)/q;
}

static double cy(double a, double b, double c, double d, double r, boolean plus) {
  double q = Math.sqrt((a-c)*(a-c) + (b-d)*(b-d));
  double y3 = (b+d)/2.0;
  if (plus)
    return y3 + Math.sqrt(r*r-(q/2.0)*(q/2.0))*(c-a)/q;
  return y3 - Math.sqrt(r*r-(q/2.0)*(q/2.0))*(c-a)/q;
}
```

Find Center of Circle given three points:

```
static Point2D findCenter(double x1, double y1, double x2, double y2, double x3, double y3)
    if (x1 == x2 && x1 == x3)
        return null; // No circle exists (points are on the same line)
    // Hack to avoid division by zero (for a vertical slope)
    if (x2 == x1 || x3 == x2) return findCenter(x3, y3, x1, y1, x2, y2);
    double ma = (y2 - y1)/(x2 - x1);
    double mb = (y3 - y2)/(x3 - x2);
    if (ma == mb)
        return null; // No circle exists (points are on the same line)
    double x = ((ma*mb*(y1 - y3)) + (mb*(x1 + x2)) - (ma*(x2 + x3))) / (2.0*(mb - ma));
    double y;
    if (ma != 0)
        y = ((y1 + y2)/2.0) - ((x - (x1 + x2)/2.0)/ma);
    else
        y = ((y2 + y3)/2.0) - ((x - (x2 + x3)/2.0)/mb);
    return new Point2D.Double(x, y);
}
```

Inradius:

For a triangle,

$$r = \frac{1}{2}\sqrt{\frac{(b+c-a)(c+a-b)(a+b-c)}{a+b+c}}$$

Incenter:

For a triangle with Cartesian vertices (x_1, y_1), (x_2, y_2), (x_3, y_3), the Cartesian coordinates of the incenter are given by

$$(x_I, y_I) = \left(\frac{a x_1 + b x_2 + c x_3}{a+b+c}, \frac{a y_1 + b y_2 + c y_3}{a+b+c}\right).$$

Convert Polar Co-ordinate to Cartesian Co-ordinate:

```
static Point2D polarToCartesian(double degrees, double radius) {
  double radians = Math.toRadians(degrees);
  return new Point2D.Double(radius*Math.cos(radians), radius*Math.sin(radians));
}
```

Angle from Point A to B:

```
// Find the angle from point A to point B in radians
// NOTE: Not entirely tested yet
static double findAngleBetweenPoints(Point2D a, Point2D b) {
  return Math.atan2(b.getY() - a.getY(), b.getX() - a.getX());
}
```

Area of Triangle:

```
static double area(Point2D a, Point2D b, Point2D c) {
  double temp1 = a.getX()*(b.getY() - c.getY());
  double temp2 = b.getX()*(c.getY() - a.getY());
  double temp3 = c.getX()*(a.getY() - b.getY());
  return Math.abs(temp1 + temp2 + temp3)/2.0;
}
```

Triangle Contains Point:

```java
static boolean containsPoint(Point2D a, Point2D b, Point2D c,
                             double area, double x, double y) {
  double ABC = Math.abs (a.getX()*(b.getY()-c.getY())
               + b.getX()*(c.getY()-a.getY()) + c.getX()*(a.getY()-b.getY()));
  double ABP = Math.abs (a.getX()*(b.getY()-y)
               + b.getX()*(y-a.getY()) + x*(a.getY()-b.getY()));
  double APC = Math.abs (a.getX()*(y-c.getY())
               + x*(c.getY()-a.getY()) + c.getX()*(a.getY()-y));
  double PBC = Math.abs (x*(b.getY()-c.getY())
               + b.getX()*(c.getY()-y) + c.getX()*(y-b.getY()));
  return ABP + APC + PBC == ABC;
}
```

Area of Polygon:

```java
// Points must be ordered (either clockwise or counter-clockwise)
static double findAreaOfPolygon(Point2D[] pts) {
  double area = 0;
  for (int i = 1; i + 1 < pts.length; i++)
    area += areaOfTriangulation(pts[0], pts[i], pts[i+1]);
  return Math.abs(area);
}

// May return positive or negative value (important for polygon method)
static double areaOfTriangulation(Point2D a, Point2D b, Point2D c) {
  return crossProduct(subtract(a, b), subtract(a, c))/2.0;
}

// Find the difference between two points
static Point2D subtract(Point2D a, Point2D b) {
  return new Point2D.Double(a.getX() - b.getX(), a.getY() - b.getY());
}

static double crossProduct(Point2D a, Point2D b) {
  return a.getX()*b.getY() - a.getY()*b.getX();
}
```

Convex Hull:

```java
import java.util.*;
import java.awt.geom.*;
public class ConvexHull {
```

```java
static Stack<Point2D> hull; // Clear this each time
static Scanner sc = new Scanner(System.in);
static void convexHull(Point2D[] pts) {
    hull = new Stack<Point2D>();
    int N = pts.length;
    Point2D[] points = new Point2D.Double[N];
    for (int i = 0; i < N; i++) points[i] = pts[i];
    Arrays.sort(points, new PointOrder());
    Arrays.sort(points, 1, N, new PolarOrder(points[0]));
    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 (ccw(points[0], points[k1], points[k2]) != 0) break;
    hull.push(points[k2-1]);
    for (int i = k2; i < N; i++) {
        Point2D top = hull.pop();
        while (ccw(hull.peek(), top, points[i]) <= 0)
            top = hull.pop();
        hull.push(top);
        hull.push(points[i]);
    }
}

// Compare other points relative to polar angle (between 0 and 2pi) they make with this
static class PolarOrder implements Comparator<Point2D> {
    Point2D pt;
    public PolarOrder(Point2D pt) { this.pt = pt; }
    @Override public int compare(Point2D q1, Point2D q2) {
        double dx1 = q1.getX() - pt.getX();
        double dy1 = q1.getY() - pt.getY();
        double dx2 = q2.getX() - pt.getX();
        double dy2 = q2.getY() - pt.getY();
        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 && dx1 < 0) return +1;
            else                          return  0;
        }
        else return -ccw(pt, q1, q2);
    }
}
```

```java
    // Put lower Y co-ordinates first, with lower X in the case of ties
    static class PointOrder implements Comparator<Point2D> {
        @Override public int compare(Point2D q1, Point2D q2) {
            if (q1.getY() < q2.getY()) return -1;
            if (q1.getY() == q2.getY()) {
                if (q1.getX() < q2.getX()) return -1;
                else if (q1.getX() > q2.getX()) return 1;
                else return 0;
            }
            return 1;
        }
    }

    static int ccw(Point2D a, Point2D b, Point2D c) {
        double area = (b.getX() - a.getX())*(c.getY()
                    - a.getY()) - (b.getY() - a.getY())*(c.getX() - a.getX());
        return (int) Math.signum(area);
    }

    // check that boundary of hull is strictly convex
    static boolean isConvex() {
        int N = hull.size();
        if (N <= 2) return true;
        Point2D[] points = new Point2D.Double[N];
        int n = 0;
        for (Point2D p : hull) points[n++] = p;
        for (int i = 0; i < N; i++)
            if (ccw(points[i], points[(i+1) % N], points[(i+2) % N]) <= 0)
                return false;
        return true;
    }

    // Sample Usage
    public static void main(String[] args) {
        int N = sc.nextInt();
        Point2D[] points = new Point2D.Double[N];
        for (int i = 0; i < N; i++)
            points[i] = new Point2D.Double(sc.nextInt(), sc.nextInt());
        convexHull(points);
        for (Point2D p : hull)
            System.out.println( p.getX() + " " + p.getY());
    }
}
```

Dijkstra: Finds shortest path from a specified node to another specified node using a priority queue (or simply breadth first search if unweighted). Complexity: O(V*log(E)) using priority queue. Only works with non-negative weights. Works on both directed and undirected graphs. Can work for multiple-source or multiple-destination by modifying original graph (by adding new node and adding edges of weight 0).

```java
// Using adjacency matrix
static int dijkstra(Integer[][] weights, int n, int start, int end) {
    int[] dist = new int[n];
    Arrays.fill(dist, INFINITY);
    dist[start] = 0;
    PriorityQueue<Node> q = new PriorityQueue<Node>();
    q.offer(new Node(start, 0));
    while (q.size() > 0) {
        Node node = q.poll();
        if (dist[node.index] < node.dist) continue; // Check to see if its stale
        if (node.index == end) return node.dist; // Reached destination
        for (int i = 0; i < n; i++)
            if (weights[node.index][i] != null) {
                int newDist = dist[node.index] + weights[node.index][i];
                if (newDist < dist[i]) {
                    dist[i] = newDist;
                    q.offer(new Node(i, newDist));
                }
            }
    }
    return -1; // Does not connect
}
class Node implements Comparable<Node> {
    int index, dist;
    public Node(int index, int dist) {
        this.index = index; this.dist = dist;
    }
    @Override public int compareTo(Node other) {
        return ((Integer) dist).compareTo(other.dist);
    }
}
```

Floyd-Warshall: Complexity: O(V^3). Computes all-pairs shortest paths. Works with negative weights (acyclic). Works on both directed and undirected graphs. Can be modified to detect negative-weight cycles (if running the algorithm on it a second time would result in anything being changed, then there is at least one negative-weight cycle).

```java
static void floydWarshall(Double[][] dist, int n) {
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (dist[i][k] != null && dist[k][j] != null)
                    if (dist[i][j] == null || dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j];
}
```

Eularian Path/Circuit:

- Difference: An Euler path starts and ends at different vertices. An Euler circuit starts and ends at the same vertex.
- Task: Given an undirected or a directed graph, find a path or circuit that passes through each edge exactly once.
- Complexity: O(V + E)

Conditions for an undirected graph:

- An undirected graph has an eulerian circuit if and only if it is connected and each vertex has an even degree (degree is the number of edges that are adjacent to that vertex).
- An undirected graph has an eulerian path if and only if it is connected and all vertices except 2 have even degree. One of those 2 vertices that have an odd degree must be the start vertex, and the other one must be the end vertex.

Conditions for a directed graph:

- A directed graph has an eulerian circuit if and only if it is connected and each vertex has the same in-degree as out-degree.
- A directed graph has an eulerian path if and only if it is connected and each vertex except 2 have the same in-degree as out-degree, and one of those 2 vertices has out-degree with one greater than in-degree (this is the start vertex), and the other vertex has in-degree with one greater than out-degree (this is the end vertex).

Algorithm for undirected graphs: (Note that obtained circuit will be in reverse order - from end to start**)**

- Start with an empty stack and an empty circuit (eulerian path).
    - If all vertices have even degree - choose any of them.
    - If there are exactly 2 vertices having an odd degree - choose one of them.
    - Otherwise no euler circuit or path exists.
- If current vertex has no neighbors - add it to circuit, remove the last vertex from the stack and set it as the current one. Otherwise (in case it has neighbors) - add the vertex to the stack, take any of its neighbors, remove the edge between selected neighbor and that vertex, and set that neighbor as the current vertex.
- Repeat step 2 until the current vertex has no more neighbors and the stack is empty.

Algorithm for directed graphs: (Note that obtained circuit will be in reverse order - from end to start**)**

- Start with an empty stack and an empty circuit (eulerian path).
    - If all vertices have same out-degrees as in-degrees - choose any of them.
    - If all but 2 vertices have same out-degree as in-degree, and one of those 2 vertices has out-degree with one greater than its in-degree, and the other has in-degree with one greater than its out-degree - then choose the vertex that has its out-degree with one greater than its in-degree.
    - Otherwise no euler circuit or path exists.
- If current vertex has no out-going edges (i.e. neighbors) - add it to circuit, remove the last vertex from the stack and set it as the current one. Otherwise (in case it has out-going edges, i.e. neighbors) - add the vertex to the stack, take any of its neighbors, remove the edge between that vertex and selected neighbor, and set that neighbor as the current vertex.
- Repeat step 2 until the current vertex has no more out-going edges (neighbors) and the stack is empty.

Prim's algorithm: Finds the minimum spanning tree of an undirected graph. Can be modified to find the minimum spanning forest. O(n^2) using adjacency matrix, however the naive implementation is O(n^3). NOTE: If we use a priority queue we might be able to get this to O(n*log(m)).

```
// PARTIALLY TESTED
static long prims(int[][] dist) {
    long total = 0;
    boolean[] isConnected = new boolean[n];
    isConnected[0] = true;
    // Initialize the minimum distances from the starting node
    int[] minDist = new int[n];
    for (int i = 1; i < n; i++)
        minDist[i] = dist[0][i];
    // Greedily add shortest edge from connected part to disconnect part each time
    for (int nConnected = 1; nConnected < n; nConnected++) {
        // Find smallest distance
        int smallest = Integer.MAX_VALUE;
        int index = -1;
        for (int i = 0; i < n; i++)
            if (!isConnected[i] && minDist[i] < smallest) {
                smallest = minDist[i];
                index = i;
            }
        // Connect to selected node
        isConnected[index] = true;
        total += smallest;
        // Update minimum distances
        for (int i = 0; i < n; i++)
            if (!isConnected[i])
                minDist[i] = Math.min(minDist[i], dist[index][i]);
    }
    return total;
}
```

Ford-Fulkerson: Solves maximum flow problems. Can be used to solve problems with more than one source or sink by modifying the original graph (adding a new node and edges with capacities that will not inhibit the flow). Can also be used to solve problems where nodes have a capacity (split nodes in half and add a new edge between them with the desired capacity). Can also be used to find the minimum cut by modifying a few lines (see below).

```
static long fordFulkerson(Node source, Node target, int nNodes) {
  long maxFlow = 0;
  boolean pathWasFound = true;
  while (pathWasFound) {
    pathWasFound = false;
    Long bottleneck = getBottleNeck(source, target, new boolean[nNodes], Long.MAX_VALUE);
    if (bottleneck != null && bottleneck > 0) {
      pathWasFound = true;
      maxFlow += bottleneck;
    }
  }
  return maxFlow;
}
```

```java
static Long getBottleNeck(Node current, Node target, boolean[] visited, long currentBottleNeck) {
  if (visited[current.index]) return null;
  if (current.index == target.index) return currentBottleNeck;
  visited[current.index] = true;
  for (Edge edge : current.adj)
    if (edge.capacityLeft > 0) {
      Long bottleneck = getBottleNeck(
        edge.target, target, visited, Math.min(currentBottleNeck, edge.capacityLeft));
      if (bottleneck != null) {
        edge.capacityLeft -= bottleneck;
        edge.opposite.capacityLeft += bottleneck;
        return bottleneck;
      }
    }
  return null; // Dead-end
}
class Node {
  List<Edge> adj = new ArrayList<Edge>();
  int index;
  public Node(int index) { this.index = index; }
  public static void addEdge(Node initial, Node target, long capacity) {
    Edge edge = new Edge(initial, target, capacity);
    edge.originalCapacity = capacity;
    initial.adj.add(edge);
    Edge reversed = new Edge(target, initial, 0);
    target.adj.add(reversed);
    edge.opposite = reversed;
    reversed.opposite = edge;
  }
}
class Edge {
  Edge opposite = null;
  Node initial, target;
  // Residual edges should have originalCapacity=null
  // so that this can be used to determine which nodes are the originals
  Long originalCapacity = null;
  long capacityLeft;
  public Edge(Node initial, Node target, long capacity) {
    this.initial = initial; this.target = target; this.capacityLeft = capacity;
  }
}
```

This algorithm can easily be modified to return the minimum cut:

```java
// NOTE: This only passed 12/49 tests on Kattis (Minimum Cut problem), but failed due to a time-out
static boolean[] fordFulkerson(Node source, Node target, int n) {
    boolean pathWasFound = true;
    boolean[] minCut = new boolean[n];
    while (pathWasFound) {
      pathWasFound = false;
      Arrays.fill(minCut, false);
      Long bottleneck = getBottleNeck(source, target, Long.MAX_VALUE, minCut);
      if (bottleneck != null && bottleneck > 0) pathWasFound = true;
    }
    return minCut;
}
```

Find Strongly Connected Components:

The complexity of this algorithm (which we came up with) is O(m*(n+m)) to remove the "bridges". After the "bridges" have been removed, we are left with connected components (indicating those which were originally strongly connected), and we can simply find the components in O(n+m). **NOTE:** It may be possible to improve our algorithm by removing all bridges at once, instead of one at a time. **NOTE:** Our use of the word "bridge" is slightly different than the normal definition. We're using it to refer to any edge that when followed, you would never be able to return to the starting spot (so we are dealing with directed graphs).

```
// Remove all bridges
boolean removed = true;
while (removed) {
    removed = false;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (arr[i][j] && isBridge(arr, j, i, new boolean[n])) {
                arr[i][j] = false;
                removed = true;
            }
}

// Find largest connected component
int max = 0;
boolean[] visited = new boolean[n];
for (int i = 0; i < n; i++)
    if (!visited[i])  {
        int size = findSize(arr, i, visited);
        max = Math.max(max, size);
    }

static int findSize(boolean[][] arr, int cur, boolean[] visited) {
    int total = 0;
    visited[cur] = true;
    for (int i = 0; i < arr.length; i++)
        if (!visited[i] && arr[cur][i]) total += findSize(arr, i, visited);
    return total + 1;
}
static boolean isBridge(boolean[][] arr, int cur, int target, boolean[] visited) {
    if (cur == target) return false;
    visited[cur] = true;
    for (int i = 0; i < arr.length; i++)
        if (!visited[i] && arr[cur][i]) {
            boolean result = isBridge(arr, i, target, visited);
            if (!result) return false;
        }
    return true;
}
```

Prime Checker:

```java
static boolean isPrime(final long n) {
     if (n < 2) return false;
     if (n == 2 || n == 3) return true;
     if (n % 2 == 0 || n % 3 == 0) return false;
     int limit = (int) Math.sqrt(n);
     for (int i = 5; i <= limit; i += 6)
        if (n % i == 0 || n % (i + 2) == 0)
            return false;
     return true;
}
```

Series: $S = a_1 + a_2 + a_3 + \ldots + a_n + \ldots$

•Arithmetic: Constant Difference $d = a_{n+1} - a_n$

$$a_n = a_1 + (n-1)\cdot d \quad \& \quad S_n = \frac{n}{2}(a_1 + a_n)$$

•Geometric: Constant ratio $r = \frac{a_{n+1}}{a_n}$

$$a_n = a_1 \cdot r^{n-1}, \quad S_n = \frac{a - a\,r^{n-1}}{1 - r} \quad \& \quad S_\infty = \frac{a_1}{1-r}$$

$(-1 < r < 1)$

Pascal's Triangle:

```java
// Note: Switch to BigInteger if you want to generate more than 67 rows
static long[][] generatePascalTriangle(int nRows) {
    long[][] arr = new long[nRows][nRows];
    arr[0][0] = 1;
    for (int y = 1; y < nRows; y++) {
        arr[y][0] = arr[y - 1][0];
        for (int x = 1; x <= y; x++)
            arr[y][x] = arr[y - 1][x - 1] + arr[y - 1][x];
    }
    return arr;
}
```

GCF (Greatest Common Factor):

```java
static int gcf( int a, int b ) {
    if (b == 0) return a;
    return gcf( b, a % b );
}
```

Factorization:

```java
// Returns the factors of a given number UNSORTED.
// Where n >= 0, does not account for negative numbers!
static ArrayList<Integer> factors(int n) {

    ArrayList<Integer> divs = new ArrayList<Integer>();
    divs.add(1);

    if (n > 1) {
        divs.add(n);
        for (int f = 2; f < ((int)Math.pow(n, 0.5))+1; f++) {
            if (n % f == 0) {
                int c = n / f;
                if (c != f) {
                    divs.add(f);
                    divs.add(c);
                } else {
                    divs.add(f);
                }
            }
        }
    }

    return divs;
}
```

Co-Prime:

```java
// Co-prime is a fancy way of saying that two numbers share no factors
static boolean areCoprime(int a, int b) {
    return gcf(a, b) == 1;
}
```

Sieve of Eratosthenes (Prime Sieve):

```java
// Gets all primes up to, but not including limit, return as a list of primes
static ArrayList<Integer> sieve(int limit) {

    // See: http://mathworld.wolfram.com/PrimeCountingFunction.html
    final int numPrimes = (limit > 1 ? (int)(1.25506 * limit / Math.log((double)limit)) : 0);
    ArrayList<Integer> primes = new ArrayList<Integer>(numPrimes);

    boolean [] isComposite = new boolean [limit];   // all false
    final int sqrtLimit    = (int)Math.sqrt(limit); // floor
    for (int i = 2; i <= sqrtLimit; i++) {
        if (!isComposite [i]) {
            primes.add(i);
            for (int j = i*i; j < limit; j += i) // `j+=i` can overflow
                isComposite [j] = true;
        }
    }
    for (int i = sqrtLimit + 1; i < limit; i++)
        if (!isComposite [i])
            primes.add(i);
    return primes;
}
```

```java
// Gets all primes up to, but not including limit, return as a boolean array
static boolean[] sieve(int limit) {

  boolean[] isPrime = new boolean[limit];
  Arrays.fill(isPrime, true);
  if (limit >= 1)
    isPrime[0] = false;
  if (limit >= 2)
    isPrime[1] = false;

  final int sqrtLimit = (int)Math.sqrt(limit);
  for (int i = 2; i <= sqrtLimit; i++) {
      if (isPrime[i])
        for (int j = i*i; j < limit; j += i)
          isPrime[j] = false;
  }

  return isPrime;

}
```

Euler's phi function (aka Euler's totient function):

```java
public static HashSet<Integer> getDistinctPrimeFactors( int n ) {
    HashSet <Integer> set = new HashSet<Integer>();
    for(int f = 2; f < Math.pow(n, 0.5) + 1; f++ ) {
        if (n % f == 0) {
            int c = n / f;
            if (c != f) {
                if (isprime(f)) set.add(f);
                if (isprime(c)) set.add(c);
            } else {
                if (isprime(f)) set.add(f);
            }
        }
    }
    return set;
}

public static int phi(int n) {
    // Phi of a prime is prime-1
    if (isprime(n)) return n-1;
    int phi = n;
    for (int p : getDistinctPrimeFactors(n))
        phi -= (phi / p);
    return phi;
}
```

<u>Logarithms</u>

$$\log_b N = p \Leftrightarrow b^p = N$$

$$\log_b N = \frac{\log_a N}{\log_a b} \quad \text{"change base"}$$

$$\log \frac{m \cdot n}{q} = \log m + \log n - \log q$$

$$\log N^p = p \log N$$

$$\log_b a = \frac{1}{\log_a b}$$

Prime Factorization:

```
static int pollard_rho(int n) {
    if (n % 2 == 0) return 2;
    // Get a number between [2, 10^6] inclusive
    int x = 2 + (int) ( ((1000000-2)+1) * Math.random());
    int c = 2 + (int) ( ((1000000-2)+1) * Math.random());
    int y = x;
    int d = 1;
    while (d == 1) {
        x = (x * x + c) % n;
        y = (y * y + c) % n;
        y = (y * y + c) % n;
        d = gcf(Math.abs(x - y), n);
        if (d == n) break;
    }
    return d;
}

static ArrayList<Integer> primeFactorization(int n) {
    ArrayList<Integer> factors = new ArrayList<Integer> ();
    if (n <= 0) throw new IllegalArgumentException();
    else if (n == 1) return factors;
    PriorityQueue <Integer> divisorQueue = new PriorityQueue<Integer>();
    divisorQueue.add(n);
    while (!divisorQueue.isEmpty()) {
        int divisor = divisorQueue.remove();
        if (isPrime(divisor)) {
            factors.add(divisor);
            continue;
        }
        int next_divisor = pollard_rho(divisor);
        if (next_divisor == divisor) {
            divisorQueue.add(divisor);
        } else {
            divisorQueue.add(next_divisor);
            divisorQueue.add(divisor/next_divisor);
        }
    }
    return factors;
}
```

Java's version of C++'s next_permutation:

NOTE: This method accounts for duplicates (which is wherever compareTo() returns 0). By modifying the two lines indicated below, this method can also be used to get the previous permutation.

```
private static <T extends Comparable<? super T>> T[] nextPermutation(final T[] c) {
  int first = getFirst(c);
  if (first == -1) return null;
  int toSwap = c.length - 1;
  while (c[first].compareTo(c[toSwap]) >= 0) // Change to <= for descending
    --toSwap;
  swap(c, first++, toSwap);
  toSwap = c.length - 1;
  while (first < toSwap)
    swap(c, first++, toSwap--);
  return c;
}

private static <T extends Comparable<? super T>> int getFirst(final T[] c) {
  for (int i = c.length - 2; i >= 0; --i)
    if (c[i].compareTo(c[i + 1]) < 0) // Change to > for descending
      return i;
  return -1;
}

private static <T extends Comparable<? super T>> void swap(final T[] c, final int i, final
  final T tmp = c[i];
  c[i] = c[j];
  c[j] = tmp;
}
```

Find a particular permutation efficiently:

Given a string, find a specified permutation based on its index (sorted lexographically), without generating all of the permutations. The complexity of this algorithm is O(nm), where is the length of the string, and m is the number of possible characters. The following implementation was designed for the characters A-Z, but it could clearly be modified for other character sets, or even other types of objects. NOTE: This algorithm ensures that each permutation is only counted once (so a string like "ALL" won't mess it up).

```java
static long[] factorial;

// Finds the specified perm (1-based index)
static String findPerm(String str, long permIndex) {
    // Pre-compute factorial values
    factorial = new long[str.length() + 1];
    factorial[0] = 1;
    for (int i = 1; i < factorial.length; i++)
        factorial[i] = factorial[i-1]*i;
    // Count the occurrences of each character
    char[] arr = str.toCharArray();
    int[] count = new int[26];
    for (char ch : arr)
        count[ch - 'A']++;
    return findPermHelper(count, str.length(), permIndex);
}

static String findPermHelper(int[] count, int nLeft, long permIndex) {
    if (nLeft == 0) return "";
    // Try placing each character at the front, recurse once we've found the right one
    for (int i = 0; i < count.length; i++) {
        if (count[i] == 0) continue;
        count[i]--;
        long nCombinations = factorial[nLeft-1]/accountForRepeatedCharacters(count);
        if (nCombinations >= permIndex)
            return (char) ('A' + i) + findPermHelper(count, nLeft - 1, permIndex);
        count[i]++;
        permIndex -= nCombinations;
    }
    return null;
}

static long accountForRepeatedCharacters(int[] count) {
    long n = 1;
    for (int i : count)n *= factorial[i];
    return n;
}
```

Number Of Power Strings:

Given a string `str`, find `n` such that `substr^n = str`, where `substr` is some substring of `str`. Example of string exponentiation: `"abf"^4 = "abfabfabfabf"`. The following algorithm (which I came up with) is O(n). The naive solution is O(n^2).

```
static int findMaxNumberOfPowerStrings(String str) {
  char[] arr = str.toCharArray();
  int end = 1, len = 1;
  while (end < arr.length) {
    if (arr[end] == arr[end-len])
      end++;
    else if (len == end) {
      len++;
      end++;
    } else
      len = end;
  }
  return arr.length/len;
}
```

Knuth–Morris–Pratt algorithm:

Given string `str` and substring `substr`, count the number of occurences of `substr` in `str` (which can be overlapping). The naive solution is O(n*m) where n is the length of `str` and m is the length of `substr`. The KMP algorithm is difficult to understand, but it is able to solve this problem in O(n).

NOTE: This can easily be modified to work with other things such as integers, for example, instead of characters. For example, I used KMP as part of my solution for the following problem: https://open.kattis.com/problems/clockpictures.

```
// returns -1 if not found, otherwise the start index of the pattern in the string
static int kmp(String string, String pattern) {
  char[] str = string.toCharArray();
  char[] pat = pattern.toCharArray();
  int[] arr = kmpHelper(pat);
  int i = 0, j = 0;
  while (i < str.length) {
    if (str[i] == pat[j]) {
      i++;
```

```java
        j++;
      } else if (j == 0)
        i++;
      else
        j = arr[j-1];
      if (j == pat.length)
        return i - j;
    }
    return -1;
  }

  static int[] kmpHelper(char[] ch) {
    int[] arr = new int[ch.length];
    int i = 1, j = 0;
    while (i < ch.length) {
      if (ch[i] == ch[j]) arr[i++] = j++ + 1;
      else if (j == 0) i++;
      else j = arr[j-1];
    }
    return arr;
  }
```

Suffix Tree:

```java
  class Node {
    Node[] nodes = new Node[26];
    void add(String str, int start) {
      if (start == str.length()) return;
      int index = str.charAt(start) - 'a';
      if (nodes[index] == null) nodes[index] = new Node();
      nodes[index].add(str, start + 1);
    }
```

Setup:

```java
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  String str = br.readLine();
  Node root = new Node();
  for (int i = 0; i < str.length(); i++)
    root.add(str, i);
```