To make some calculations (which are rather tedious to do by hand) feasible, we will be using the computer algebra system GAP as a calculation tool. You are also welcome to use GAP (or any other computer algebra system) for calculations, as long as they do not render the problem trivial. For example, if a problem asks to calculate a GCD, you could use GAP to do the division with remainders. As long as you use the system purely for aritmetic, you can just say that you are using it. If you do more compliacted calculations, you should indicate in your homework solution the commands that you have entered.

You can download a Windows or OSX installer for GAP for your private machine from `http://www.math.colostate.edu/~hulpke/CGT/education.html`. Installers work as usual for the respective operating system. If you are using the lab (Weber 205/6), make sure the domain is set to `MATHSTAT2` and log in as user **m366**. The password is the one given in the lecture. This lab account is to be used *only* for use in this class.
You can now start GAP with the `ggap` icon (in Windows on the desktop, under OSX in the `Applications` folder). The program will start up and you will get window into which you can input text at a prompt `>`. We will write this prompt here as `gap>` to indicate that it is input to the program.

```
gap>
```

You now can type in commands (followed by a semicolon) and GAP will print out the result.
To leave GAP you can either call `quit;` or close the window.
You should be able to use the cursor keys for editing lines. Note that changing prior entries in the worksheet does not change the prior calculation, but just executes the same command again. If you want to repeat a sequence of commands, pasting the lines in is a better approach.

You can use the online help to get documentation, it opens in a separate window.

```
gap> ?gcd
```

You might want to create a transcript of your session in a text file. On the laboratory machines the best place for this is the `My Documents` folder. You create the log by issuing the command

```
LogTo("C:/Documents and Settings/m366/My Documents/logfile.txt");
```

Two caveats:

- You must type *forward* slashes (`/`) even though the standard Windows convention is for backslashes.
- You all share the same account – files with the same name will overwrite each other. A good idea therefore is to have files include e.g. your eName to be unique, e.g. call a log file *`yourname`*`log.txt`.

You can end logging with the command

```
LogTo();
```

Saving and Loading

If you want to enter a set of commands (or even write your own code) it is convenient to write this in a text file (`.txt`), for example using `NotePad`. (Careful with using `Word`, which usually creates more complicated files!)
You can read in such a file using the `Read` command:

```
gap> Read("C:/Documents and Settings/m366/My Documents/myprogram.txt");
```

(Again careful with name clashes, if you are using the common class account!)
It also is possible to save an existing session and start it again later. You can do so by `Saveing` the session (make sure you do not change the default option away from *Workspace*) (either in the menu, or via the disk icon.) This file contains all details of your current session, you can load it later (even on another machine) and continue. To load such a file, use `Open` (again, either in the menu, or via the folder icon) and load a saved workspace. Again careful with name clashes of workspaces on the lab machines. Also, as workspace files can be rather large, please delete unused workspace files.

Numbers

GAP knows integers of arbitrary length and rational numbers:

```
gap> 17 - 23;
-6
gap> 123456/7891011+1;
2671489/2630337
```

GAP knows a precedence between operators that may be overridden by parentheses and can compare objects:

```
gap> (9 - 7) * 5 = 9 - 7  * 5;
false
gap> 5/3<2;
true
```

You can assign numbers (or more general: every GAP object) to variables, by using the assignment operator `:=`. Once a variable is assigned to, you can refer to it as if it was a number. The special variables `last`, `last2`, and `last3` contain the results of the last three commands.

```
gap> a:=2^16-1;
65535
gap> b:=a/(2^4+1);
3855
```

The following commands show some useful integer calculations related to quotient and remainder:

```
gap> Int(8/3); # round down
2
gap> QuoInt(76,23); # integral part of quotient
3
gap> QuotientRemainder(76,23);
[ 3, 7 ]
gap> 76 mod 23; # remainder (note the blanks)
7
gap> 1/5 mod 7;
3
gap> Gcd(64,30);
2
gap> rep:=GcdRepresentation(64,30);
[ -7, 15 ]
```

```
gap> rep[1]*64+rep[2]*30;
2

gap> Factors(2^64-1);
[ 3, 5, 17, 257, 641, 65537, 6700417 ]
```

Lists

Objects separated by commas and enclosed in square brackets form a list.
Collections of objects are represented by such lists. Lists are also used to represent sets.

```
gap> l:=[5,3,99,17,2]; # create a list
[ 5, 3, 99, 17, 2 ]
gap> l[4]; # access to list entry
17
gap> l[3]:=22; # assignment to list entry
22
gap> l;
[ 5, 3, 22, 17, 2 ]
gap> Length(l);
5
gap> 3 in l; # element test
true
gap> 4 in l;
false
gap> Position(l,2);
5
gap> Add(l,17); # extension of list at end
gap> l;
[ 5, 3, 22, 17, 2, 17 ]
gap> s:=Set(l); # new list, sorted, duplicate free
[ 2, 3, 5, 17, 22 ]
```

Results that consist of several numbers are represented as list.

```
gap> DivisorsInt(96);
[ 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 96 ]
```

There are powerful list functions that often can save programming loops: `List`, `Filtered`, `ForAll`, `ForAny`, `First`. The notation `i -> xyz` is a shorthand for a one parameter function.

```
gap> l:=[5,3,99,17,2];
[ 5, 3, 99, 17, 2 ]
gap> List(l,IsPrime);
[ true, true, false, true, true ]
gap> List(l,i -> i^2);
[ 25, 9, 9801, 289, 4 ]
gap> Filtered(l,IsPrime);
```

```
[ 5, 3, 17, 2 ]
gap> ForAll(l,i -> i>10);
false
gap> ForAny(l,i -> i>10);
true
gap> First(l,i -> i>10);
99
```

A special case of lists are *ranges*, indicated by double dots.

```
gap> l:=[10..100];
[ 10 .. 100 ]
gap> Length(l);
91
```

For example to solve homework problem 2b, we want to find years, for which $d = 1$ and $e = 0$. To do so, we first solve the value for a from the defining formula $d = (19a + 24) mod 30$. We then search for years which give this $a = Y mod 19$ value.

```
gap> a:=(1-24)/19 mod 30;
13
gap> years:=Filtered([1900..2099],y->y mod 19=a);
[ ...]
```

We now calculate for each of these years the value of e, and check when $e = 0$:

```
gap> Filtered(years,y-> (2*(y mod 4)+4*(y mod 7) +6+5) mod 7=0);
[ ... ]
```

Polynomials

To create polynomials, we need to create the variable first. Note that the name we give is the printed name (which could differ from the variable we assign it to). As GAP does not know the real numbers we do it over the rationals

```
gap> x:=X(Rationals,"x");
x
```

If we wanted, we could define several different variables this way. Now we can do all the usual arithmetic with polynomials:

```
gap> f:=x^7+3*x^6+x^5+3*x^3-4*x^2-4*x;
x^7+3*x^6+x^5+3*x^3-4*x^2-4*x
gap> g:=x^5+2*x^4-x^2-2*x;
x^5+2*x^4-x^2-2*x
gap> f+g;f*g;
x^7+3*x^6+2*x^5+2*x^4+3*x^3-5*x^2-6*x
x^12+5*x^11+7*x^10+x^9-2*x^8-5*x^7-14*x^6-11*x^5-2*x^4+12*x^3+8*x^2
```

The same operations as for integers hold for polynomials

```
gap> QuotientRemainder(f,g);
[ x^2+x-1, 3*x^4+6*x^3-3*x^2-6*x ]
gap> f mod g;
3*x^4+6*x^3-3*x^2-6*x
gap> Gcd(f,g);
x^3+x^2-2*x
gap> GcdRepresentation(f,g);
[ -1/3*x, 1/3*x^3+1/3*x^2-1/3*x+1 ]
gap> rep:=GcdRepresentation(f,g);
[ -1/3*x, 1/3*x^3+1/3*x^2-1/3*x+1 ]
gap> rep[1]*f+rep[2]*g;
x^3+x^2-2*x
gap> Factors(g);
[ x-1, x, x+2, x^2+x+1 ]
```

Matrices

Lists are used to form vectors and matrices:
A *vector* is simply a list of numbers. A list of (row) vectors is a matrix. GAP knows matrix arithmetic.

```
gap> mat:=[[1,2,3],[5,6,7],[9,10,12]];;
gap> mat^5;
[ [ 289876, 342744, 416603 ], [ 766848, 906704, 1102091 ],
  [ 1309817, 1548698, 1882429 ] ]
gap> DeterminantMat(mat);
-4
```

The command `Display` can be used to get a nicer output:

```
gap> 3*mat^2-mat;
[ [ 113, 130, 156 ], [ 289, 342, 416 ], [ 492, 584, 711 ] ]
gap> Display(last);
[ [  113,  130,  156 ],
  [  289,  342,  416 ],
  [  492,  584,  711 ] ]
```

Integers modulo residue

If we want to compute in the integers modulo n (what we called $\mathbb{Z}_n$ in the lecture) without need to always type `mod` we can create obhjects that immediately reduce their arithmetic modulo n:

```
gap> im:=Integers mod 6; # represent numbers for ‘‘modulo 6’’ calculations
(Integers mod 6)
```

To convert “ordinary” integers to residue classes, we have to multiply them with the“One” of these residue classes, the command `Int` converts back to ordinary integers:

```
gap> a:=5*One(im);
ZmodnZObj( 5, 6 )
```

```
gap> b:=3*One(im);
ZmodnZObj( 3, 6 )
gap> a+b;
ZmodnZObj( 2, 6 )
gap> Int(last);
2
```

(If one wants one can get all residue classes or – for example test which are invertible).

```
gap> Elements(im);
[ ZmodnZObj(0,6),ZmodnZObj(1,6),ZmodnZObj(2,6),ZmodnZObj(3,6),
  ZmodnZObj(4,6),ZmodnZObj(5,6) ]
gap> Filtered(last,x->IsUnit(x));
[ ZmodnZObj( 1, 6 ), ZmodnZObj( 5, 6 ) ]
gap> Length(last);
2
```

If we calculate modulo a *prime* the default output looks a bit different.

```
gap> im:=Integers mod 7;
GF(7)
```

(The name `GF` stands for *Galois Field*, explanation later.) Also elements display differently. Therefore it is convenient to once issue the command

```
gap> TeachingMode(true);
```

which (amongst others) will simplify the display.

Matrices and Polynomials modulo a prime

We can use these rings to calculate in matrices modulo a number. `Display` again brings it into a nice form:

```
gap> mat:=[[1,2,3],[5,6,7],[9,10,12]]*One(im);
[ [ ZmodnZObj( 1, 7 ), ZmodnZObj( 2, 7 ), ZmodnZObj( 3, 7 ) ],
  [ ZmodnZObj( 5, 7 ), ZmodnZObj( 6, 7 ), ZmodnZObj( 0, 7 ) ],
  [ ZmodnZObj( 2, 7 ), ZmodnZObj( 3, 7 ), ZmodnZObj( 5, 7 ) ] ]
gap> Display(mat);
 1 2 3
 5 6 .
 2 3 5
gap> mat^-1+mat;
[ [ ZmodnZObj( 4, 7 ), ZmodnZObj( 4, 7 ), ZmodnZObj( 4, 7 ) ],
  [ ZmodnZObj( 6, 7 ), ZmodnZObj( 1, 7 ), ZmodnZObj( 5, 7 ) ],
  [ ZmodnZObj( 3, 7 ), ZmodnZObj( 1, 7 ), ZmodnZObj( 6, 7 ) ] ]
```

In the same way we can also work with polynomials modulo a number. For this one just needs to define a suitable variable. For example suppose we want to work with polynomials modulo 2:

```
gap> r:=Integers mod 2;
GF(2)
gap> x:=X(r,"x");
x
gap> f:=x^2+x+1;
x^2+x+Z(2)^0
gap> Factors(f);
[ x^2+x+Z(2)^0 ]
```

As 2 is a prime (and therefore every nonzero remainder invertible), we can now work with polynomials modulo f. The possible remainders now are all the polynomials of strictly smaller degree (with coefficients suitable reduced):

```
gap> o:=One(r);
Z(2)^0
gap> elms:=[0*x,0*x+o,x,x+o];
[ 0*Z(2), Z(2)^0, x, x+Z(2)^0 ]
```

We can now build addition and multiplication tables for these four objects modulo f:

```
gap> Display(List(elms,a->List(elms,b->Position(elms,a+b mod f))));
[ [  1,  2,  3,  4 ],
  [  2,  1,  4,  3 ],
  [  3,  4,  1,  2 ],
  [  4,  3,  2,  1 ] ]
gap> Display(List(elms,a->List(elms,b->Position(elms,a*b mod f))));
[ [  1,  1,  1,  1 ],
  [  1,  2,  3,  4 ],
  [  1,  3,  4,  2 ],
  [  1,  4,  2,  3 ] ]
```

This process yields extremely interesting structures.