Republic of Iraq

Ministry of Higher Education and Scientific Research University of Baghdad

Computational Group Theory

الزمر الحسابية

محاضرات تدرس لطلبة الدراسات الاولية

المرحلة الثالثة-الفصل الدراسي الثاني

قسم الرياضيات-كلية العلوم-جامعة بغداد

بواسطة

الدكتور علي عبد عبيد

2021-2020

Computational Group Theory

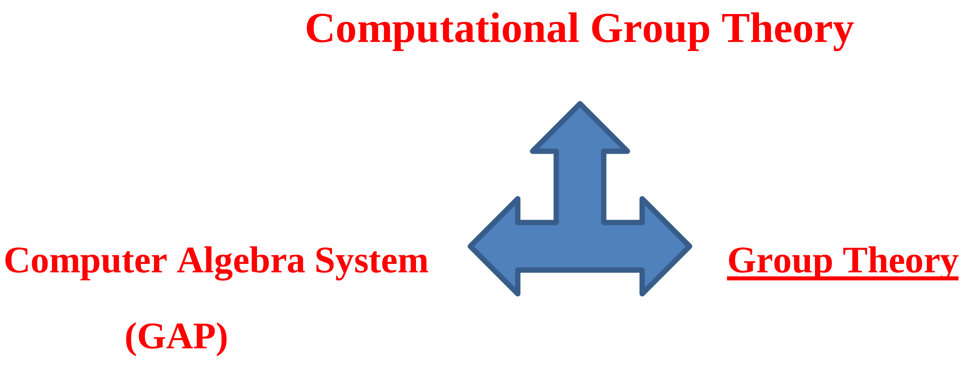
Definition 1: Computational Group theory (CGT) is the study of algorithms for groups. It aims to produce algorithms to answer questions about concrete groups, given for example by generators or as symmetries of a certain algebraic or combinatorial structures.

Interest in this comes from (at least) three areas:

1-Interest in developing algorithms.

2- Concrete questions about concrete groups.

3- Complexity theory.



References:

1-The GAP Group, GAP Groups, Algorithms, and Programming, Version 4.4.12, http://www.gap-system.org, (2008).

2- H. Kurzweil B. Stellmacher. The Theory of Finite Groups An Introduction. Springer, Universitext, (2003).

Section 1: Short Introduction To GAP

Definition 2: **GAP** (Groups, Algorithms and Programming) is a computer algebra system for computational discrete algebra with particular emphasis on computational group theory.

GAP is a system for discrete computational algebra, with particular emphasis on Computational Group Theory. GAP provides a programming language, a library of thousands of functions implementing algebraic algorithms written in the GAP language as well as large data libraries of algebraic objects, for example the small groups library which contains, among others.

Structure of GAP

GAP has a kernel written in C. It implements

1-The GAP language,

2-An interactive environment for developing and using GAP programs,

3-Memory management.

4-Fast versions of time critical operations for various data types.

The **GAP** system will run on any machine with a UNIX-like or recent Windows or Mac OS X operating system and with a reasonable amount of RAM and disk space. The current version is **GAP 4**, but **GAP 3** is also available.

About Starting and Leaving GAP

If the program is correctly installed then you usually start **GAP** by simply typing gap at the prompt of your operating system followed by the **Return** key, sometimes this is also called the **Newline** key.

```
$ gap
```

GAP answers your request with its beautiful banner and then it shows its own prompt gap> asking you for further input.

```
gap>
```

The usual way to end a **GAP** session is to type quit; at the gap> prompt. Do not omit the semicolon!

```
gap> quit;
$
```

On some systems you could type **Ctrl-D** to yield the same effect. In any situation **GAP** is ended by typing **Ctrl-C** twice within a second. Here as always, a combination like **Ctrl-D** means that you have to press the **D** key while you hold down the **Ctrl** key.

A simple calculation with **GAP** is as easy as one can imagine. You type the problem just after the prompt, terminate it with a semicolon and then pass the problem to the program with the **Return** key. For example, to multiply the difference between 9 and 7 by the sum of 5 and 6, that is to calculate (9 - 7) * (5 + 6), you type exactly this last sequence of symbols followed by ; and **Return**.

```
gap> (9 - 7) * (5 + 6);
22
gap>
```

Then **GAP** echoes the result 22 on the next line and shows with the prompt that it is ready for the next problem. Henceforth, we will no longer print this additional prompt.

If you make a mistake while typing the line, but *before* typing the final **Return**, you can use the **Delete** key (or sometimes **Backspace** key) to delete the last typed character. You can also move the cursor back and forward in the line with **Ctrl-B** and **Ctrl-F** and insert or delete characters anywhere in the line. The line editing commands are fully described in section Reference: Line Editing.

If you did omit the semicolon at the end of the line but have already typed **Return**, then **GAP** has read everything you typed, but does not know that the command is complete. The program is waiting for further input and indicates this with a partial prompt >. This problem is solved by simply typing the missing semicolon on the next line of input. Then the result is printed and the normal prompt returns.

```
gap> (9 - 7) * (5 + 6)
> ;
22
```

So the input can consist of several lines, and **GAP** prints a partial prompt > in each input line except the first, until the command is completed with a semicolon. (**GAP** may already evaluate part of the input when **Return** is typed, so for long calculations it might take some time until the partial prompt appears). Whenever you see the partial prompt and you cannot decide what **GAP** is still waiting for, then you have to type semicolons until the normal prompt returns. In every situation the exact meaning of the prompt gap> is that the program is waiting for a new problem.

But even if you mistyped the command more seriously, you do not have to type it all again. Suppose you mistyped or forgot the last closing parenthesis. Then your command is syntactically incorrect and **GAP** will notice it, incapable of computing the desired result.

```
gap> (9 - 7) * (5 + 6;
Syntax error: ) expected
(9 - 7) * (5 + 6;
                ^
```

Instead of the result an error message occurs indicating the place where an unexpected symbol occurred with an arrow sign ^under it. As a computer program cannot know what your intentions really were, this is only a hint. But in this case **GAP** is right by claiming that there should be a closing parenthesis before the semicolon. Now you can type **Ctrl-P** to recover the last line of input. It will be written after the prompt with the cursor in the first position. Type **Ctrl-E** to take the cursor to the end of the line, then **Ctrl-B** to move the cursor one character back. The cursor is now on the position of the semicolon. Enter the missing parenthesis by simply typing). Now the line is correct and may be passed to **GAP** by hitting the **Return** key. Note that for this action it is not necessary to move the cursor past the last character of the input line.

Each line of commands you type is sent to **GAP** for evaluation by pressing **Return** regardless of the position of the cursor in that line. We will no longer mention the **Return** key from now on.

Sometimes a syntax error will cause **GAP** to enter a *break loop*. This is indicated by the special prompt brk>. If another syntax error occurs while **GAP** is in a break loop, the prompt will change to brk_02>, brk_03> and so on. You can leave the current break loop and exit to the next outer one by either typing quit; or by hitting **Ctrl-D**. Eventually **GAP** will return to its normal state and show its normal prompt gap> again.

Constants and Operators

In an expression like (9 - 7) * (5 + 6) the constants 5, 6, 7, and 9 are being composed by the operators +, * and - to result in a new value.

There are three kinds of operators in **GAP**, arithmetical operators, comparison operators, and logical operators. You have already seen that it is possible to form the sum, the difference, and the product of two integer values. There are some more operators applicable to integers in **GAP**. Of course integers may be divided by each other, possibly resulting in noninteger rational values.

```
gap> 12345/25;
2469/5
```

Note that the numerator and denominator are divided by their greatest common divisor and that the result is uniquely represented as a division instruction.

The next self-explanatory example demonstrates negative numbers.

```
gap> -3; 17 - 23;
-3
-6
```

The exponentiation operator is written as ^. This operation in particular might lead to very large numbers. This is no problem for **GAP** as it can handle numbers of (almost) any size.

```
gap> 3^132;
9550049507968252368931907017744140119199351389743431298368538
41
```

The mod operator allows you to compute one value modulo another.

```
gap> 17 mod 3;
2
```

Note that there must be whitespace around the keyword mod in this example since 17mod3 or 17mod would be interpreted as identifiers. The whitespace around operators that do not consist of letters, e.g., the operators * and -, is not necessary.

GAP knows a precedence between operators that may be overridden by parentheses.

```
gap> (9 - 7) * 5 = 9 - 7  * 5;
false
```

Besides these arithmetical operators there are comparison operators in **GAP**. A comparison results in a *boolean value* which is another kind of constant. The comparison operators =, <>, <, <=, > and >=, test for equality, inequality, less than, less than or equal, greater than and greater than or equal, respectively.

```
gap> 10^5 < 10^4;
false
```

The boolean values true and false can be manipulated via logical operators, i. e., the unary operator not and the binary operators and and or. Of course boolean values can be compared, too.

```
gap> not true; true and false; true or false;
false
false
true
gap> 10 > 0 and 10 < 100;
true
```

Another important type of constants in **GAP** are *permutations*. They are written in cycle notation and they can be multiplied.

```
gap> (1,2,3);
(1,2,3)
gap> (1,2,3) * (1,2);
(2,3)
```

The inverse of the permutation (1,2,3) is denoted by (1,2,3)^-1. Moreover the caret operator ^ is used to determine the image of a point under a permutation and to conjugate one permutation by another.

```
gap> (1,2,3)^-1;
(1,3,2)
gap> 2^(1,2,3);
3
gap> (1,2,3)^(1,2);
(1,3,2)
```

Variables versus Objects

A **GAP** command *sequence_of_letters_and_digits* := *meaning*, where the sequence on the left hand side is called the *identifier* of the variable and it serves as its name. The meaning on the right hand side can be a constant like an integer or a permutation, but it can also be almost any other **GAP** object. From now on, we will use the term *object* to denote something that can be assigned to a variable.

There must be no whitespace between the : and the = in the assignment operator. Also do not confuse the assignment operator with the single equality sign = which in **GAP** is only used for the test of equality.

```
gap> a:= (9 - 7) * (5 + 6);
22
gap> a;
22
gap> a * (a + 1);
506
gap> a = 10;
false
gap> a:= 10;
```

```
10
gap> a * (a + 1);
110
```

After an assignment the assigned object is echoed on the next line. The printing of the object of a statement may be in every case prevented by typing a double semicolon.

```
gap> w:= 2;;
```

After the assignment the variable evaluates to that object if evaluated. Thus it is possible to refer to that object by the name of the variable in any situation.

There are some further interesting variables one of which will be introduced now.

Whenever **GAP** returns an object by printing it on the next line this object is assigned to the variable last. So if you computed

```
gap> (9 - 7) * (5 + 6);
22
```

and forgot to assign the object to the variable a for further use, you can still do it by the following assignment.

```
gap> a:= last;
22
```

About Functions

A program written in the **GAP** language is called a *function*. Functions are special **GAP** objects. Most of them behave like mathematical functions. They are applied to objects and will return a new object depending on the input. The function Factorial(Reference: Factorial), for example, can be applied to an integer and will return the factorial of this integer.

```
gap> Factorial(17);
355687428096000
```

Applying a function to arguments means to write the arguments in parentheses following the function. Several arguments are separated by commas, as for the function Gcd (Reference: Gcd) which computes the greatest common divisor of two integers.

```
gap> Gcd(1234, 5678);
2
```

There are other functions that do not return an object but only produce a side effect, for example changing one of their arguments. These functions are sometimes called procedures. The function Print (Reference: Print) is only called for the side effect of printing something on the screen.

```
gap> Print(1234, "\n");
1234
```

A comfortable way to define a function yourself is the *maps-to* operator -> consisting of a minus sign and a greater sign with no whitespace between them. The function cubed which maps a number to its cube is defined on the following line.

```
gap> cubed:= x -> x^3;
function( x ) ... end
```

After the function has been defined, it can now be applied.

```
gap> cubed(5);
125
```

Lists and Sets

A *list* is a collection of objects separated by commas and enclosed in brackets. Let us for example construct the list primes of the first ten prime numbers.

```
gap> primes:= [2, 3, 5, 7, 11, 13, 17, 19, 23, 29];
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

The next two primes are 31 and 37. They may be appended to the existing list by the function Append which takes the existing list as its first and another list as a second argument. The second argument is appended to the list primes and no value is returned. Note that by appending another list the object primes is changed.

```
gap> Append(primes, [31, 37]);
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 ]
```

You can as well add single new elements to existing lists by the function Add which takes the existing list as its first argument and a new element as its second argument. The new element is added to the list primes and again no value is returned but the list primes is changed.

```
gap> Add(primes, 41);
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 ]
```

Single elements of a list are referred to by their position in the list. To get the value of the seventh prime, that is the seventh entry in our list primes, you simply type

```
gap> primes[7];
17
```

This value can be handled like any other value, for example multiplied by 2 or assigned to a variable. On the other hand this mechanism allows one to assign a value to a position in a list. So the next prime 43 may be inserted in the list directly after the last occupied position of primes. This last occupied position is returned by the function Length.

```
gap> Length(primes);
13
gap> primes[14]:= 43;
43
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43 ]
```

Note that this operation again has changed the object primes. The next position after the end of a list is not the only position capable of taking a new value. If you know that 71 is the 20th prime, you can enter it right now in the 20th position of primes. This will result in a list with holes which is however still a list and now has length 20.

```
gap> primes[20]:= 71;
71
gap> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,,,,,, 71 ]
gap> Length(primes);
20
```

Sets

GAP knows several special kinds of lists. A *set* in **GAP** is a list that contains no holes (such a list is called *dense*) and whose elements are strictly sorted w.r.t. <; in particular, a set cannot contain duplicates. (More precisely, the elements of a set in **GAP** are required to lie in the same *family*, but roughly this means that they can be compared using the < operator.)

The elements of the sets used in the examples of this section are strings.

```
gap> fruits:= ["apple", "strawberry", "cherry", "plum"];
[ "apple", "strawberry", "cherry", "plum" ]
gap> IsSSortedList(fruits);
false
gap> fruits:= Set(fruits);
[ "apple", "cherry", "plum", "strawberry" ]
```

For and While Loops

Given a list pp of permutations we can form their product by means of a for loop instead of writing down the product explicitly.

```
gap> pp:= [ (1,3,2,6,8)(4,5,9), (1,6)(2,7,8), (1,5,7)(2,3,8,6),
>           (1,8,9)(2,3,5,6,4), (1,9,8,6,3,4,7,2)];;
gap> prod:= ( );
()
gap> for p in pp do
>       prod:= prod*p;
>    od;
gap> prod;
(1,8,4,2,3,6,5,9)
```

First a new variable prod is initialized to the identity permutation ( ). Then the loop variable p takes as its value one permutation after the other from the list pp and is multiplied with the present value of prod resulting in a new value which is then assigned to prod.

The for loop has the following syntax

for *var* in *list* do *statements* od;

The effect of the for loop is to execute the *statements* for every element of the *list*. A for loop is a statement and therefore terminated by a semicolon. The list of *statements* is enclosed by the keywords do and od (reverse do). A for loop returns no value. Therefore we had to ask explicitly for the value of prod in the preceding example.

The for loop can loop over any kind of list, even a list with holes. In many programming languages the for loop has the form

for *var* from *first* to *last* do *statements* od;

In **GAP** this is merely a special case of the general for loop as defined above where the *list* in the loop body is a range :

for *var* in [*first*..*last*] do *statements* od;

You can for instance loop over a range to compute the factorial 15! of the number 15 in the following way.

```
gap> ff:= 1;
1
gap> for i in [1..15] do
>       ff:= ff * i;
>    od;
gap> ff;
1307674368000
```

The while loop has the following syntax

while *condition* do *statements* od;

The while loop loops over the *statements* as long as the *condition* evaluates to true. Like the for loop the while loop is terminated by the keyword od followed by a semicolon.

We can use our list primes to perform a very simple factorization. We begin by initializing a list factors to the empty list. In this list we want to collect the prime factors of the number 1333. Remember that a list has to exist before any values can be assigned to positions of the list. Then we will loop over the list primes and test for each prime whether it divides the number. If it does we will divide the number by that prime, add it to the list factors and continue.

```
gap> n:= 1333;;
gap> factors:= [];;
gap> for p in primes do
>      while n mod p = 0 do
>        n:= n/p;
>        Add(factors, p);
>      od;
>    od;
gap> factors;
[ 31, 43 ]
gap> n;
1
```

As n now has the value 1 all prime factors of 1333 have been found and factors contains a complete factorization of 1333. This can of course be verified by multiplying 31 and 43.

Writing Functions

Writing a function that prints hello, world. on the screen is a simple exercise in **GAP**.

```
gap> sayhello:= function()
> Print("hello, world.\n");
> end;
function(  ) ... end
```

This function when called will only execute the Print statement in the second line. This will print the string hello, world. on the screen followed by a newline character \n that causes the **GAP** prompt to appear on the next line rather than immediately following the printed characters.

The function definition has the following syntax.

function(*arguments*) *statements* end

A function definition starts with the keyword function followed by the formal parameter list *arguments* enclosed in parenthesis ( ). The formal parameter list may be empty as in the example. Several parameters are separated by commas. Note that there must be *no* semicolon behind the closing parenthesis. The function definition is terminated by the keyword end.

A **GAP** function is an expression like an integer, a sum or a list. Therefore it may be assigned to a variable. The terminating semicolon in the example does not belong to the function definition but terminates the assignment of the function to the name sayhello. Unlike in the case of integers, sums, and lists the value of the function sayhello is echoed in the abbreviated fashion function( ) ... end. This shows the most interesting part of a function: its formal parameter list (which is empty in this example). The complete value of sayhello is returned if you use the function Print (Reference: Print).

```
gap> Print(sayhello, "\n");
function ( )
   Print( "hello, world.\n" );
   return;
end
```

Note the additional newline character "\n" in the Print (Reference: Print) statement. It is printed after the object sayhello to start a new line. The extra return statement is inserted by **GAP** to simplify the process of executing the function.

The newly defined function sayhello is executed by calling sayhello() with an empty argument list.

```
gap> sayhello();
hello, world.
```

However, this is not a typical example as no value is returned but only a string is printed.

If Statements

In the following example we define a function sign which determines the sign of an integer.

```
gap> sign:= function(n)
>       if n < 0 then
>          return -1;
>       elif n = 0 then
>          return 0;
>       else
>          return 1;
>       fi;
>    end;
function( n ) ... end
gap> sign(0); sign(-99); sign(11);
0
-1
1
```

This example also introduces the if statement which is used to execute statements depending on a condition. The if statement has the following syntax.

if *condition* then *statements* elif *condition* then *statements* else *statements* fi

There may be several elif parts. The elif part as well as the else part of the if statement may be omitted. An if statement is no expression and can therefore not be assigned to a variable. Furthermore an if statement does not return a value.

Fibonacci numbers are defined recursively by f(1) = f(2) = 1 and f(n) = f(n-1) + f(n-2) for n $\geq$ 3. Since functions in **GAP** may call themselves, a function fib that computes Fibonacci numbers can be implemented basically by typing the above equations. (Note however that this is a very inefficient way to compute f(n).)

```
gap> fib:= function(n)
```

```
>      if n in [1, 2] then
>         return 1;
>      else
>         return fib(n-1) + fib(n-2);
>      fi;
>    end;
function( n ) ... end
gap> fib(15);
610
```

There should be additional tests for the argument n being a positive integer. This function fib might lead to strange results if called with other arguments. Try inserting the necessary tests into this example.

Local Variables

A function gcd that computes the greatest common divisor of two integers by Euclid's algorithm will need a variable in addition to the formal arguments.

```
gap> gcd:= function(a, b)
       local c;
       while b <> 0 do
          c:= b;
         b:= a mod b;
         a:= c;
       od;
       return c;
    end;
function( a, b ) ... end
gap> gcd(30, 63);
3
```

Repeat

repeat *statements* until *bool-expr*;

The repeat loop executes the statement sequence *statements* until the condition *bool-expr* evaluates to true.

The repeat loop in the following example has the same purpose as the while loop in the preceding example, namely to sum up the squares 1^2, 2^2, ... until the sum exceeds 200.

```
gap> i := 0;; s := 0;;
gap> repeat
>    i := i + 1; s := s + i^2;
>  until s > 200;
gap> s;
204
```

Break

break;

The statement break; causes an immediate exit from the innermost loop enclosing it.

```
gap> g := Group((1,2,3,4,5),(1,2)(3,4)(5,6));
Group([ (1,2,3,4,5), (1,2)(3,4)(5,6) ])
gap> for x in g do
> if Order(x) = 3 then
> break;
> fi; od;
gap> x;
(1,5,2)(3,4,6)
```

It is an error to use this statement other than inside a loop.

```
gap> break;
Syntax error: 'break' statement not enclosed in a loop
```

Continue

continue;

The statement continue; causes the rest of the current iteration of the innermost loop enclosing it to be skipped.

```
gap> g := Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> for x in g do
> if Order(x) = 3 then
> continue;
> fi; Print(x,"\n"); od;
()
(2,3)
(1,3)
(1,2)
```

It is an error to use this statement other than inside a loop.

```
gap> continue;
Syntax error: 'continue' statement not enclosed in a loop
```

Line Editing

GAP allows one you to edit the current input line with a number of editing commands. Those commands are accessible either as control keys or as escape keys. You enter a control key by pressing the **Ctrl** key, and, while still holding the **Ctrl** key down, hitting another key key. You enter an escape key by hitting **Esc** and then hitting another key key. Below we denote control keys by **Ctrl-**key and escape keys by **Esc-**key. The case of key does not matter, i.e., **Ctrl-A** and **Ctrl-a** are equivalent.

Normally, line editing will be enabled if the input is connected to a terminal. Line editing can be enabled or disabled using the command line options -f and -n respectively (see 3.1), however this is a machine dependent feature of **GAP**.

Typing **Ctrl-key** or **Esc-key** for characters not mentioned below always inserts **Ctrl-**key resp. **Esc-**key at the current cursor position.

The first few commands allow you to move the cursor on the current line.

Ctrl-A

move the cursor to the beginning of the line.

Esc-B

move the cursor to the beginning of the previous word.

Ctrl-B

move the cursor backward one character.

Ctrl-F

move the cursor forward one character.

Esc-F

move the cursor to the end of the next word.

Ctrl-E

move the cursor to the end of the line.

The next commands delete or kill text. The last killed text can be reinserted, possibly at a different position, with the "yank" command **Ctrl-Y**.

Ctrl-H or del

delete the character left of the cursor.

Ctrl-D

delete the character under the cursor.

Ctrl-K

kill up to the end of the line.

Esc-D

kill forward to the end of the next word.

Esc-del

kill backward to the beginning of the last word.

Ctrl-X

kill entire input line, and discard all pending input.

Ctrl-Y

insert (yank) a just killed text.

The next commands allow you to change the input.

Ctrl-T

exchange (twiddle) current and previous character.

Esc-U

uppercase next word.

Esc-L

lowercase next word.

Esc-C

capitalize next word.

The **Tab** character, which is in fact the control key **Ctrl-I**, looks at the characters before the cursor, interprets them as the beginning of an identifier and tries to complete this identifier. If there is more than one possible completion, it completes to the longest common prefix of all those completions. If the characters to the left of the cursor are already the longest common prefix of all completions hitting **Tab** a second time will display all possible completions.

tab

complete the identifier before the cursor.

The next commands allow you to fetch previous lines, e.g., to correct typos, etc.

Ctrl-L

> insert last input line before current character.

Ctrl-P

> redisplay the last input line, another **Ctrl-P** will redisplay the line before that, etc. If the cursor is not in the first column only the lines starting with the string to the left of the cursor are taken.

Ctrl-N

> Like **Ctrl-P** but goes the other way round through the history.

Esc-<

> goes to the beginning of the history.

Esc->

> goes to the end of the history.

Ctrl-O

> accepts this line and perform a **Ctrl-N**.

Finally there are a few miscellaneous commands.

Ctrl-V

> enter next character literally, i.e., enter it even if it is one of the control keys.

Ctrl-U

> execute the next line editing command 4 times.

Esc-num

execute the next line editing command num times.

Esc-Ctrl-L

redisplay input line.

The four arrow keys (cursor keys) can be used instead of **Ctrl-B**, **Ctrl-F**, **Ctrl-P**, and **Ctrl-N**, respectively.

Display

Displays the object *obj* in a nice, formatted way which is easy to read (but might be difficult for machines to understand). The actual format used for this depends on the type of *obj*. Each method should print a newline character as last character.

```
gap> Display( [ [ 1, 2, 3 ], [ 4, 5, 6 ] ] * Z(5) );
 2 4 1
 3 . 2
```

Function

function([*arg-ident* {, *arg-ident*}])

  [local *loc-ident* {, *loc-ident*} ;]

  statements

end

A function is in fact a literal and not a statement. Such a function literal can be assigned to a variable or to a list element or a record component. The following is an example of a function definition. It is a function to compute values of the Fibonacci sequence.

```
gap> fib := function ( n )
>     local f1, f2, f3, i;
>    f1 := 1; f2 := 1;
>    for i in [3..n] do
>      f3 := f1 + f2;
>      f1 := f2;
>      f2 := f3;
>    od;
>    return f2;
>  end;;
gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

File Operations

1- Read

‣ Read(*filename*) (operation)

reads the input from the file with the filename *filename*, which must be given as a string.

Read first opens the file *filename*. If the file does not exist, or if **GAP** cannot open it, e.g., because of access restrictions, an error is signalled.

2- ReadAsFunction

‣ ReadAsFunction(*filename*) (operation)

reads the file with filename *filename* as a function and returns this function.

Example

Suppose that the file /tmp/example.g contains the following

```
local a;

a := 10;
return a*10;
```

Reading the file as a function will not affect a global variable a.

```
gap> a := 1;
1
gap> ReadAsFunction("/tmp/example.g")();
100
gap> a;
1
```

3- PrintTo and AppendTo

‣ PrintTo(*filename*[, *obj1*, ...]) (function)

‣ AppendTo(*filename*[, *obj1*, ...]) (function)

PrintTo works like Print , except that the arguments *obj1*, ... (if present) are printed to the file with the name *filename* instead of the standard output. This file must of course be writable by **GAP**. Otherwise an error is signalled. Note that PrintTo will *overwrite* the previous contents of this file if it already existed; in particular, PrintTo with just the *filename* argument empties that file.

AppendTo works like PrintTo, except that the output does not overwrite the previous contents of the file, but is appended to the file.

There is an upper limit of 15 on the number of output files that may be open simultaneously.

4- LogTo

‣ LogTo(*filename*) (operation)

‣ LogTo( ) (operation)

Calling LogTo with a string *filename* causes the subsequent interaction to be logged to the file with the name *filename*, i.e., everything you see on your terminal will also appear in this file. LogTo may also be used to log to a stream.) This file must of course be writable by **GAP**, otherwise an error is signalled. Note that LogTo will overwrite the previous contents of this file if it already existed.

Called without arguments, LogTo stops logging to a file or stream.

5- InputLogTo

- InputLogTo(*filename*) (operation)

- InputLogTo( ) (operation)

Calling InputLogTo with a string *filename* causes the subsequent input to be logged to the file with the name *filename*, i.e., everything you type on your terminal will also appear in this file. Note that InputLogTo and LogTo cannot be used at the same time while InputLogTo and OutputLogTo can. Note that InputLogTo will overwrite the previous contents of this file if it already existed.

Called without arguments, InputLogTo stops logging to a file or stream.

6- OutputLogTo

- OutputLogTo(*filename*) (operation)

- OutputLogTo( ) (operation)

Calling OutputLogTo with a string *filename* causes the subsequent output to be logged to the file with the name *filename*, i.e., everything **GAP** prints on your terminal will also appear in this file. Note that OutputLogTo and LogTo cannot be used at the same time while InputLogTo and OutputLogTo can. Note that OutputLogTo will overwrite the previous contents of this file if it already existed.

Called without arguments, OutputLogTo stops logging to a file or stream.

Section 2 : A Brief introduction to Group Theory

Roughly speaking, a group is a set of objects with a rule of combination. Given any two elements of the group, the rule yields another group element, which depends on the two elements chosen. Addition illustrates some of the properties which a group combination rule must have, including that it is associative and that there is an element that, like 0, doesn't change any element when combined with it.

Definition 2.1 A group is a non-empty set G together with a rule that assigns to each pair g, h of elements of G an element $g * h$ such that:

- $g * h \in G$. We say that G is closed under $*$.
- $g * (h * k) = (g * h) * k$ for all $g, h, k \in G$. We say that $*$ is associative.
- There exists an identity element $e \varepsilon G$ such $e * g = g * e = g$ for all $g \in G$.
- Every element $g \varepsilon G$ has an inverse $g^{-1} \varepsilon G$ such that $g * g^{-1} = g^{-1} * g = e$.

Note: we may set e={1}.

Definition 2.2 [1] Let S be a nonempty subset of a group G. If

S_1: $a;b \in S \rightarrow ab \in S$, and

S_2: $a \in S \rightarrow a^{-1} \in S$; Then S is a subgroup of G.

We write $S \leq G$ to indicate that S is a subgroup of G which is possibly equal to G itself. We write $S < G$ for a subgroup which is not equal to G. The subsets {1} and G are subgroups of G.

Definition 2.3 Suppose G is a group and $H \leq G$. For $a \in G$ we define the right coset of H by $Ha = \{ha \mid h \in H\}$ ($\subseteq G$). While lift coset H by $aH = \{ah \mid h \in H\}$ ($\subseteq G$).

Definition 2.4 If G is a group then we define the centre of G as

$$Z(G) = \{g \in G \mid gx = xg , \forall x \in G\}$$

The **center** of a group, G, is the set of elements that commute with every element of G.

Definition 2.5 A **finite group** is one with only a finite number of elements. The order of a finite group, written |G|, is the number of elements in G. (Note that if X is a set, we also often write |X| to be the number of elements in X.)

Definition 2.6. The order of an element $g \in G$ is the smallest positive integer n such that $g^n=e$.

Definition 2.7 Let G be a group and X subset of G. Then the subgroup generated by X which denoted by <X> is the intersection of all subgroups of G containing X.

Definition 2.8 : Cyclic group is group generated by single element. Let $a \in G$ then the subgroup group of G generated by a is $\langle a \rangle = \{a^n : n \in Z\}$.

Definition 2.9: Involution elements of group G is an element of order 2 in G (i.e., an element g of a group such that $g^2=e$, where e is the identity element).

Definition 2.10 : The **centralizer** of an element z of a group G is the set of elements of G which commute with z, $C_G(z)=\{ x \in G , x z=z x \}$.

Likewise, the centralizer of a subgroup H of a group G is the set of elements of G which commute with every element of H,

$$C_G(H)=\{ x \in G , \forall h \in H , x h=h x \}.$$

Definition 2.11: Normal subgroups a subgroup N of G is normal, denoted $N \lhd G$, if $gNg^{-1} = N$ for all $g \in G$. We should note that the identity element and G itself are normal subgroup of G.

EXAMPLE 1.1.12:

1- **Generalized quaternion group** Q_{4n} with presentation $\langle x,y|x^{2n}=y^4=1,x^n=y^2,\ \ y^{-1}xy=x^{-1}\rangle$ for $n\geq 2$. For n=2 is the well-known group which is the **quaternion group** Q_8.

2-**The dihedral group D_{2n}**with presentation $\langle x,y|x^n=y^2=1,xy=yx^{-1}\rangle$ for $n\geq 1$.

3- Let $(R,\cdot)$ the group of nonzero real numbers under multiplication. Let $H=\{1,-1\}$ then $H \lhd R$.

Definition 2.13 : The **symmetric** group S_n is the group of bijections from any set of n objects, which we usually call simply **{1,2,3…,n}** to itself. An element of this group is called a **permutation** of **{1,2,3…,n}**. The group operation in **Sn** is composition of mappings.

Definition 1.1.14:(Fix of permutation) For $\beta \in S_n$ we define fix(β) to be the set of elements in $\Omega = \{1, 2, . . . n\}$ which fixed by β . So that $x\in \Omega$ be in the fix(β) if and only if $\beta(x)=x$. The set of elements of **Ω** which are fixed by β can be denoted Fix(β).

Remark 2.15: 1- Cyclic notation $(\alpha_1, \alpha_2, ..., \alpha_r)$ where $\alpha_1, \alpha_2, ..., \alpha_r$ are distinct elements of $\Omega = \{1, 2, . . . n\}$ and $r \leq n$. denotes the following permutation in S_n:

$\alpha_1 \rightarrow \alpha_2$

$\alpha_2 \rightarrow \alpha_3$

.

.

.
$\alpha_{r-1} \to \alpha_r$
$\alpha_r \to \alpha_1$
And $\alpha \to \alpha\ \forall \alpha \in \Omega \setminus \{\alpha_1, \alpha_2, ..., \alpha_r\}$. A r-cycle $(\alpha_1, \alpha_2, ..., \alpha_r)$. Also, r-Cycles $(\alpha_1, \alpha_2, ..., \alpha_r)$ and , s-Cycles $(\beta_1, \beta_2, ..., \beta_s)$ are disjoint cycles if and only if $\{\alpha_1, \alpha_2, ..., \alpha_r\} \cap \{\beta_1, \beta_2, ..., \beta_s\} = \emptyset$.

3-A 2-cycle is also called a **transposition.** For example**,** (1,2) or (5,6) in S_5.

2-Every permutation can be written as a product of disjoint cycles — cycles that all have no elements in common. Disjoint cycles commute.

Definition 2.16: Let S be a finite set and $\sigma : S \to S$ be a permutation. The cycle type of σ is the data of how many cycles of each length are present in the cycle decomposition of σ .

Example 2.17:- The conjugacy class of (1 2)(3 4) in S_4 consists of all the elements of cycle-type (2, 2) and is
{(1 2)(3 4), (1 3)(2 4), (1 4)(2 3)}

Theorem 2.18: [1]For $x \in S_n$, the conjugacy class x^{Sn} of x in Sn consists of all permutations in Sn which have the same cycle-shape as x.

Example 2.19 The conjugacy classes of S_3 are

Class	Cycle-shape
{1}	(1)
{(1 2), (1 3), (2 3)}	(3)
{(1 2 3), (1 3 2)}	(2)

Definition 2.20 It is the group of even permutations of a finite set. The alternating group on a set of n elements is called the alternating group of degree n, or the alternating group on n letters and denoted by A_n or Alt(n). The group An is a normal subgroup of Sn.

EXAMPLE 2 .21:$G=S_3$, then G={1,(1, 2),(1 ,3),(2, 3),(1, 2, 3),(1 , 3 ,2)}, then

(1,2)(1,3)=(1,2,3)

(1,2,3)(1,3,2)=( ), the identity element.

We also see that A_3={1,(1,2,3),(1,3,2)} is normal subgroup of $S_{3.}$

Definition 2 .22: $S^g = \{^{g\text{-}1xg} \mid x \in S\}$ Call S^g a conjugate (in G) of S.

Say R and S are conjugate in G if g∃ ∈ G such that $S^g = R$

*** Special case:**

S ={x}, R = {y}.

Then R and S are conjugate (in G) ⇔g ∈ G such that $g^{-1}xg = y$

We say x and y are conjugate in G.

Notation • $x^g = g^{-1}xg$

• Conjugacy class of x in G: $x^G = \{^{g\text{-}1xg} \mid g \in G\}$

Remarks 2 .23 :G a group. S⊆G, g ∈ G.

(1) $|S| = |S^g|$ because the map $x \mapsto g^{-1}xg$ is a map from S to S^g which is a bijection.

(2) If $S \leq G$, then $S^g \leq G$

(3) If x; y∈G and x and y are conjugate in G, then x and y have the same order.

(4) $1^G = \{1^g \mid g \in G\} = \{g^{-1}1_g \mid g \in G\} = \{1\}$

(5) If x ∈ Z(G), then $x^G = \{x\}$

$x^G = \{g^{-1}xg \mid g \in G\} = \{g^{-1}gx \mid g \in G\} = \{x\}$

In particular if G is abelian then all conjugacy classes contain just one element.

Theorem 2.24: Every group is a union of conjugacy classes, and distinct conjugacy classes are disjoint.

Theorem 2.25: Let $x \in G$. Then the size of the conjugacy class x^G is given $|x^G|=|G, C_G(x)|=|G|/|C_G(x)|$, In particular, $|x^G|$ divides $|G|$

The Class Equation: Let $x_1,..., x_n$ be representatives of the conjugacy classes of G. Then

$|G|=|Z(G)|+\sum_{xi \notin Z(G)} |x_i^G|$,Where $|x_i^G|=|G:C_G(x_i)|$, and both $|x_i^G|$ and $|Z(G)|$ divide $|G|$.

- **Example 2.26:** In the abelian group, all the conjugacy classes are singleton sets of size 1. This is because for any x,g, we have $gxg^{-1}=x$
- **Example 2.27 :** More generally, for any group G , and any element in the center of G , the conjugacy class of that element has size 1.
- **Example 2.28:**The group S3 is the smallest non-Abelian group. This group has three conjugacy classes, the class of the identity element (size 1), the class of the (transpositions (2-Cycles)) (size 3) and the class of the 3-cycles (size 2).

Now we give a basis definitions in regard with the group action.

Definition 2 .29: Let G be a group, and let X be a nonempty set. Then a (left)

action of G on X is a map $G \times X \mapsto X$,written $(g, x) \mapsto g \cdot x$, such that

$g_1 \cdot (g_2 \cdot x) = (g_1g_2) \cdot x$ and $e \cdot x = x$,for all $g_1, g_2 \in G$ and all $x \in X$

Definition 2.30: Suppose G is a group which acts on a set S. If $s \in S$,

let $O(s) = \{gs| g \in G\}$.

The set O(s) is called the orbit of s. The stabilizer of s is the subset

$G_s = \{g \in G \mid gs = s\}$of G.

Transversal of s ={g ∊ G | gs = t} for t ∊ O(s).

Definition 2.31: A action of a group on a set is called transitive when the set is nonempty and there is exactly one orbit.

Lemma 2.32: Let G act on a set S. If s ∊ S, then the stabilizer G_s of s is a subgroup of G.

Proof. Note that Gs is nonempty since e ∊Gs. Furthermore, if g, h ∊ Gs then gs = hs = s, so (gh)s = g(hs) = gs = s, so gh ∊Gs. Finally, if g ∊Gs then g-1s = g^{-1}(gs) = (g^{-1}g)s = es = s. Thus g^{-1} ∊ Gs. Therefore Gs is a subgroup of G.

(Orbit-Stabilizer Theorem) 2.33 :Let G be a finite group acting on a set S, and let x ∈ S. Then the number of elements in the orbit x^G is equal to [G : StabG(x)].

Section 3: Algorithms

An algorithm is a set of instructions designed to perform a specific task. This can be a simple process, such as multiplying two numbers, or a complex operation, such as playing a compressed video file. Search engines use proprietary algorithms to display the most relevant results from their search index for specific queries.

In computer programming, algorithms are often created as functions. These functions serve as small programs that can be referenced by a larger program. For example, an image viewing application may include a library of functions that each use a custom algorithm to render different image file formats. An image editing program may contain algorithms designed to process image data. Examples of image processing algorithms include cropping, resizing, sharpening, blurring, red-eye reduction, and color enhancement.

Algorithm 3.1 The "**plain vanilla**" orbit algorithm.

```
Input: A group G, given by a generating set g = {$g_1$, . . . , gm}, acting on a
domain $\Omega$. Also a point $\omega \in \Omega$.
Output: return the orbit $\omega^G$.
begin
1: $\Delta = [\omega]$;
2: for $\delta \in \Delta$ do
3: for i $\in$ {1, . . . ,m} do
4: $\gamma = \delta^{g_i}$;
5: if $\gamma \notin \Delta$ then
6: Append $\gamma$ to $\Delta$;
7: fi;
8: od;
9: od;
10: return $\Delta$;
end
```

Algorithm 3.2: **Orbit** algorithm with **transversal** computation

Input: A group G, given by a generating set **g** = {g_l , . . . , gm}, acting on a domain Ω. Also a point $\omega \in \Omega$.
Output: return the orbit ω^G and a transversal T.

```
begin
1: $\Delta$= [$\omega$];
2: T= [1];
3: for $\delta \in \Delta$ do
4: for i $\in$ {1, . . . ,m} do
5: $\gamma=\delta^{gi}$ ;
6: if $\gamma \notin \Delta$ then
7: Append $\gamma$ to $\Delta$ ;
8: Append T[$\delta$].$g_i$ to T ;
9: fi;
10: od;
11: od;
12: return $\Delta$,T;
end
```

Algorithm 3.3: **Orbit/Stabilizer** algorithm

Input: A group G, given by a generating set **g** = {g_l , . . . , gm}, acting on a domain Ω. Also a point $\omega \in \Omega$.
Output: return the orbit ω^G and a transversal T.

```
begin
1: $\Delta$= [$\omega$];
2: T= [1];
3:S=<1>;
4: for $\delta \in \Delta$ do
5: for i $\in$ {1, . . . ,m} do
6: $\gamma=\delta^{gi}$ ;
7: if $\gamma \notin \Delta$ then
8: Append $\gamma$ to $\Delta$ ;
9: Append T[$\delta$].$g_i$ to T ;
10:else
11:S=< S,T[$\delta$].$g_i$ . T[$\gamma$]$^{-1}$ >
12: fi;
13: od;
14: od;
15: return $\Delta$,T,S;
end
```

Definition 3.1: Let $\Delta= \omega^G$ (again considered as a list). A Schreier vector (or factored transversal) is a list S of length $|\Delta|$ with the following properties:

• The entries of S are generators of G (or the identity element). (In fact the entries are pointers to generators, thus requiring only one pointer per entry instead of one group element.)
• $S[\omega] = 1$
• If $S[\delta] = g$ and $\delta^{gi} = \gamma$ then γ precedes δ in the orbit.

Schreier vectors can take the place of a transversal:

Algorithm 3.4: If S is a Schreier vector for a point $\omega \in \Omega$., the following algorithm computes for $\delta \in \omega^G$ a representative r such that $\omega^r = \delta$.

```
begin
1: $\gamma=\delta$;
2: r= 1;
3:while $\gamma \neq \omega$ do
4: g:=S[ $\gamma$];
5: r:=g.r;
6: $\gamma=\gamma^{gi(-1)}$ ;
14: od;
15: return r;
end
```

Section 4: Examples

Ex1: Let G=S_3 the Symmetric group of degree 3. Then find

1- The Size of G .

2-The elements of G.

3-Conjugacy classes of G.

4- let Ω={1,2,3}, the G act on Ω by usual manner. if ω=3 $\in\Omega$, then calculate

a-The orbits ω^G.

b-The transversal of ω^G.

5- Find the Stabilizer(G, ω).

6-Find Fix((1,2)).

Solution:

1- Size of the G=S_3 is 3!=6

2-G={( ),(1,2),(1,3),(2,3),(1,2,3),(1,3,2)}

(1,2)=(2,1) because in (1,2) 1---2 and in (2,1) 2---1 which is the same

(1,2,3)=(3,1,2)=(2,3,1) because

In (1,2,3) we have 1---2 , 2---3, 3---1

In (3,1,2) we have 3---1 , 1---2, 2---3

In (2,3,1) we have 2---3, 3---1,1---2

3- Conjugacy classes of G

$( )^G$ ----{ ( )}

$(1,2)^G$---{(1,2),(1,3),(2,4)}

$(1,2,3)^G$---{(1,2,3),(1,3,2)}

4-

a-The orbits $\omega^G=\{3^{(\,)},3^{(1,2)},3^{(1,3)},3^{(2,3)},3^{(1,2,3)},3^{(1,3,2)}\}=[3,3,1,2,1,2]=\{1,2,3\}$

b- The transversal of $\omega^G=\{(1,3),(2,3),(\,)\}$

5- Stabilizer(G,3)={( ),(1,2)}.

6-Fix((1,2))={1,2,3}\{1,2}={3}

Ex2: In Ex1 use Gap to calculate all the requests.

```
G:=SymmetricGroup(3);
Sym( [ 1 .. 3 ] )
```

1-

```
Size(G);
6
```

2-

```
Elements(G);
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
```

3-

```
ConjugacyClasses(G);
[ ()^G, (1,2)^G, (1,2,3)^G ]
```

4-

a-

```
 Orbit(G,3);
[ 1, 3, 2 ]
```

b-

```
T:=[ ];
for i in [1,2,3] do
Add(T,RepresentativeAction(G,3,i));
od;
T;
[ (1,3), (2,3), () ]
```

5-

```
Elements(Stabilizer(G,3));
[ (), (1,2) ]
```

6-

```
Difference([1,2,3],MovedPoints((1,2)));
[ 3 ]
```

Ex3: Let G=A4 the Alternating group of degree 4. Then find

1- The Size of G .

2-The elements of G.

3- let Ω={1,2,3,4}, the G act on Ω by usual manner. if ω=2 ∈Ω, then calculate :

a-The orbits ω^G.

b-The transversal of ω^G.

4- Find the Stabilizer(G, ω).

5-Find Fix((1,2,3)).

Solution:

1-The size of A_4 is 4!/2=12.

2-The element of A_4 are the even permutations of Sn

A_4={( ),(1,2)(3,4),(1,3)(2,4),(1,4)(2,3), (1,2,3), (1,2,4), (1,3,2), (1,3,4), (1,4,2), (1,4,3)}

3-

a-ω^G=[2,1,4,3,3,4,1,2,1,2]={1,2,3,4}

b- The transversal of ω^G={( ),(1,2)(3,4),(1,4)(2,3),(1,3)(2,4)};

4- Stabilizer(G, ω)={( ), (1,3,4), (1,4,3)}

5- Fix((1,2,3))={1,2,3,4}\{1,2,3}={4}

Ex4: In Ex3 use Gap to calculate all the requests.

```
G:= AlternatingGroup(4);

 Alt( [ 1 .. 4 ] )

1-
Size(G);
12

2-
Elements(G);
 [ (), (2,3,4), (2,4,3), (1,2)(3,4), (1,2,3), (1,2,4), (1,3,2), (1,3,4), (1,3)(2,4),
(1,4,2), (1,4,3), (1,4)(2,3) ]

3-
a-
 Orbit(G,2);
[ 1, 2, 3,4 ]




b-

T:=[ ];
```

```
for i in [1,2,3,4] do
Add(T,RepresentativeAction(G,2,i));
od;
T;
[ (1,3,2), (1,2,3), (1,2,4), () ]
```

4-

```
Elements(Stabilizer(G,2));
[ (), (1,3,4), (1,4,3) ]
```

5-

```
Difference([1,2,3,4],MovedPoints((1,2,3)));
[ 4 ]
```

Ex5: Let G be a group with the following elements

[(), (1,2,4,7,6,11,5,3,10,9,8), (1,3,7,8,5,4,9,11,2,10,6), (1,4,6,5,10,8,2,7,11,3,9), (1,5,2,3,4,10,7,9,6,8,11), (1,6,10,2,11,9,4,5,8,7,3), (1,7,5,9,2,6,3,8,4,11,10), (1,8,9,10,3,5,11,6,7,4,2), (1,9,3,11,7,2,8,10,5,6,4), (1,10,11,4,8,3,6,2,9,5,7), (1,11,8,6,9,7,10,4,3,2,5)]

Then Calculate the following:

1- The Size of G .

2- let Ω={1,2,3,4,5,6,7,8,9,10,11}, the G act on Ω by usual manner. if ω=4 ∊Ω, then calculate :

a-The orbits ω^G.

b-The transversal of ω^G.

3- Find the Stabilizer(G, ω).

4-Find Fix((1,11,8,6,9,7,10,4,3,2,5)).

Solution:

1-Size(G) or |G|=11.

2-

a-

4^G=[4,7,9,6,10,5,11,2,1,8,3]={1,2,3,4,5,6,7,8,9,10,11}

b-

The transversal of ω^G=G.

3- Stabilizer(G, 4)={( )}.

4- Fix((1,11,8,6,9,7,10,4,3,2,5))={1,2,3,4,5,6,7,8,9,10,11}

{ 1,11,8,6,9,7,10,4,3,2,5}=∅.

Ex6: In Ex5 use Gap to calculate all the requests

```
G:=Subgroup(SymmetricGroup(11),[ (), (1,2,4,7,6,11,5,3,10,9,8),
(1,3,7,8,5,4,9,11,2,10,6), (1,4,6,5,10,8,2,7,11,3,9), (1,5,2,3,4,10,7,9,6,8,11),
(1,6,10,2,11,9,4,5,8,7,3), (1,7,5,9,2,6,3,8,4,11,10), (1,8,9,10,3,5,11,6,7,4,2),
(1,9,3,11,7,2,8,10,5,6,4), (1,10,11,4,8,3,6,2,9,5,7), (1,11,8,6,9,7,10,4,3,2,5) ]
);
permutation group with 11 generators
```

1-

```
Size(G);
11
```

2-

a-

```
 Orbit(G,4);
[ 1, 2, 3,4,5,6,7,8,9,10,11 ]
```

b-

```
T:=[ ];

for i in [1,2,3,4,5,6,7,8,9,10,11] do
Add(T,RepresentativeAction(G,4,i));
od;
T;
[ (), (1,2,4,7,6,11,5,3,10,9,8), (1,3,7,8,5,4,9,11,2,10,6),

(1,4,6,5,10,8,2,7,11,3,9), (1,5,2,3,4,10,7,9,6,8,11), (1,6,10,2,11,9,4,5,8,7,3),

 (1,7,5,9,2,6,3,8,4,11,10), (1,8,9,10,3,5,11,6,7,4,2), (1,9,3,11,7,2,8,10,5,6,4),

(1,10,11,4,8,3,6,2,9,5,7), (1,11,8,6,9,7,10,4,3,2,5) ]
```

4-

```
Elements(Stabilizer(G,4));
[ () ]
```

5-

```
Difference([1,2,3,4,5,6,7,8,9,10,11],MovedPoints((1,11,8,6,9,7,10,4,3,2,5))
);
[  ]
```

Ex7: Let G be a group with the following elements

[(), (5,6), (3,4), (3,4)(5,6), (1,2), (1,2)(5,6), (1,2)(3,4), (1,2)(3,4)(5,6), (1,3)(2,4), (1,3)(2,4)(5,6), (1,3,2,4), (1,3,2,4)(5,6), (1,4,2,3), (1,4,2,3)(5,6), (1,4)(2,3), (1,4)(2,3)(5,6)]

1- The Size of G .

2- let Ω={1,2,3,4,5,6}, the G act on Ω by usual manner. if ω=3 ∈Ω, then calculate :

a-The orbits ω^G.

b-The transversal of ω^G.

3- Find the Stabilizer(G, ω).

4-Find Fix((5,6)).

<u>Solution</u>:

1-Size(G) or |G|=16.

2-

a-

3^G={1,2,3,4}

b-

The transversal of ω^G=[(1,3)(2,4), (1,3,2,4), (), (3,4)].

3- Stabilizer(G, 3)={ (), (5,6), (1,2), (1,2)(5,6) }.

4- Fix((5,6))={1,2,3,4}

Ex8: In Ex7 use Gap to calculate all the requests

```
G:=Subgroup(SymmetricGroup(6),[ [ (), (5,6), (3,4), (3,4)(5,6), (1,2),
(1,2)(5,6), (1,2)(3,4), (1,2)(3,4)(5,6), (1,3)(2,4), (1,3)(2,4)(5,6), (1,3,2,4),
(1,3,2,4)(5,6), (1,4,2,3), (1,4,2,3)(5,6), (1,4)(2,3), (1,4)(2,3)(5,6) ]
);
permutation group with 6 generators
```

1-
```
Size(G);
16
```

2-
a-
```
 Orbit(G,3);
```

```
[ 1, 2, 3,4 ]
b-

T:=[ ];

for i in [1,2,3,4] do
Add(T,RepresentativeAction(G,3,i));
od;
T;
[ (1,3)(2,4), (1,3,2,4), (), (3,4) ]
```

4-

```
Elements(Stabilizer(G,3));
[ (), (5,6), (1,2), (1,2)(5,6) ]
```

5- `Difference([1,2,3,4,5,6],MovedPoints((5,6)));`

```
[1,2,3,4 ]
```

Ex9: Let G be a group with the following elements

[(), (4,5,6), (4,6,5), (1,2,3), (1,2,3)(4,5,6), (1,2,3)(4,6,5), (1,3,2), (1,3,2)(4,5,6), (1,3,2)(4,6,5)]

Then Calculate the following:

1- The Size of G .

2- let $\Omega=\{1,2,3,4,5,6\}$, the G act on Ω by usual manner. if $\omega=6 \in\Omega$, then calculate :

a-The orbits ω^G.

b-The transversal of ω^G.

3- Find the Stabilizer(G, ω).

4-Find Fix((1,2,3)).

<u>Solution</u>:

1-Size(G) or |G|=9.

2-

a-

6^G={4,5,6}

b-

The transversal of ω^G= [(4,5,6), (4,6,5), ()].

3- Stabilizer(G, 6)={ (), (1,2,3), (1,3,2) }.

4- Fix((1,2,3))={4,5,6}

Ex10: In Ex9 use Gap to calculate all the requests

```
G:=Subgroup(SymmetricGroup(6), [ (), (4,5,6), (4,6,5), (1,2,3),
(1,2,3)(4,5,6), (1,2,3)(4,6,5), (1,3,2), (1,3,2)(4,5,6), (1,3,2)(4,6,5) ]);
```

```
permutation group with 6 generators
```

1-

```
Size(G);
9
```

2-

a-

```
 Orbit(G,3);
[ 4,5,6 ]
```

b-

```
T:=[ ];

for i in [4,5,6] do
Add(T,RepresentativeAction(G,6,i));
od;
T;
```

[(4,5,6), (4,6,5), ()]

4-

Elements(Stabilizer(G,6));
[(), (1,2,3), (1,3,2)]

5- Difference([1,2,3,4,5,6],MovedPoints((1,2,3)));
[4,5,6]

Ex11: Let G be a group with the following elements

[(), (1,2,3,4,5), (1,3,5,2,4), (1,4,2,5,3), (1,5,4,3,2)]

Then Calculate the following:

1- The Size of G .

2- let Ω={1,2,3,4,5}, the G act on Ω by usual manner. if ω=1 ∈Ω, then calculate :

a-The orbits ω^G.

b-The transversal of ω^G.

3- Find the Stabilizer(G, ω).

4-Find Fix(()).

Solution:

1-Size(G) or |G|=5.

2-

a-

1^G={1,2,3,4,5}

b-

The transversal of ω^G= G.

3- Stabilizer(G, 1)={ ()}.

4- Fix(())={1,2,3,4,5}

Ex12: In Ex11 use Gap to calculate all the requests

```
G:=Subgroup(SymmetricGroup(5), [ (), (1,2,3,4,5), (1,3,5,2,4),
(1,4,2,5,3), (1,5,4,3,2) ]);
```

permutation group with 5 generators

1-

```
Size(G);
5
```

2-

a-

```
 Orbit(G,1);
[ 1,2,3,4,5 ]
```

b-

```
T:=[ ];

for i in [1,2,3,4,5] do
Add(T,RepresentativeAction(G,1,i));
od;
T;
[ (), (1,2,3,4,5), (1,3,5,2,4), (1,4,2,5,3), (1,5,4,3,2) ]
```

4-

```
Elements(Stabilizer(G,1));
[()]
```

5- Difference([1,2,3,4,5],MovedPoints(()));

```
[1,2,3,4,5 ]
```

Ex13: Let G be a group with the following elements

[(), (1,2,3,4,5,6), (1,3,5)(2,4,6), (1,4)(2,5)(3,6), (1,5,3)(2,6,4), (1,6,5,4,3,2)]

Then Calculate the following:

1- The Size of G .

2- let Ω={1,2,3,4,5,6}, the G act on Ω by usual manner. if ω=4 ∈Ω, then calculate :

a-The orbits ω^G.

b-The transversal of ω^G.

3- Find the Stabilizer(G, ω).

4-Find Fix(()).

Solution:

1-Size(G) or |G|=6.

2-

a-

4^G={1,2,3,4,5,6}

b-

The transversal of ω^G= G.

3- Stabilizer(G, 4)={ ()}.

4- Fix(())={1,2,3,4,5,6}

Ex14: In Ex13 use Gap to calculate all the requests

```
G:=Subgroup(SymmetricGroup(5), [ (), (1,2,3,4,5,6), (1,3,5)(2,4,6),
(1,4)(2,5)(3,6), (1,5,3)(2,6,4), (1,6,5,4,3,2) ]);
```

```
permutation group with 6 generators
```

1-

```
Size(G);
6
```

2-

a-

```
 Orbit(G,4);
[ 1,2,3,4,5,6 ]
```

b-

```
T:=[ ];

for i in [1,2,3,4,5,6] do
Add(T,RepresentativeAction(G,4,i));
od;
T;
[(), (1,2,3,4,5,6), (1,3,5)(2,4,6), (1,4)(2,5)(3,6), (1,5,3)(2,6,4), (1,6,5,4,3,2) ]
```

4-

```
Elements(Stabilizer(G,4));
[()]
```

5- `Difference([1,2,3,4,5,6],MovedPoints(()));`

```
[1,2,3,4,5,6 ]
```