

C++ Classes for Group Theoretical Computations

Jeffrey W. Clark
Department of Mathematics, Elon College
Elon College, NC 27244
clarkj@@numen.elon.edu

December 20, 1996

1 Introduction

GROUPCPP is a collection of class definitions intended to facilitate group programming in C++ by students and faculty. The code is neither the swiftest nor the most efficient; there are existing packages that are better by these criteria. Rather, the code is as simple and open as possible, so that users can see how it works and modify as they see fit.

The classes include permutations and invertible matrices over four types of rings: finite fields, rational numbers, the integers, and the integers mod n for some n . There is a good deal of repetition in the code; I chose not to use templates in the hope of keeping the code simple.

I have avoided wherever possible portability problems by defining my own functions (except for pre-defined input/output and string-related functions). I have also included all working code in the relevant header files, for simplicity of use and examination. This code was developed using a Borland C++ compiler, and was intended for use in a DOS environment. I would urge users to use the file **testalg.cpp** to verify that all of GROUPCPP compiles satisfactorily on other compilers.

The GROUPCPP package is available via anonymous ftp at ftp.elon.edu, in directory pub/MATH. I would welcome any comments or suggestions via e-mail at clarkj@@numen.elon.edu.

2 Files in GROUPCPP

GROUPCPP consist of the following files:

errmsg.h contains code for error messages.

example1.cpp shows how to use GROUPCPP to find a group generated by two permutations.

example2.cpp shows how to use GROUPECPP to find the conjugacy classes of a subgroup of $GL(2, \mathbb{Z}/3\mathbb{Z})$.

example3.cpp shows how to use GROUPECPP to find the commutator subgroup of a subgroup of $GL(2, \mathbb{F})$ where $\mathbb{F} = (\mathbb{Z}/3\mathbb{Z})[x]/(x^2 + 2)$.

ffield.h defines the **ffield** class of elements of a fixed finite field.

groupcpp.doc is an ASCII version of this documentation.

groupcpp.tex is a L^AT_EX version of this documentation.

groupcpp.zip is a compressed version of the rest of this set of files.

integer.h defines the **integer** class, which contains integers of arbitrary size.

invmatff.h defines the **invmatff** class, which contains invertible matrices with elements from **ffield**.

invmatq.h defines the **invmatq** class, which contains invertible matrices with elements from **rational**.

invmatz.h defines the **invmatz** class, which contains invertible matrices with elements from **integer**.

invmatzm.h defines the **invmatzm** class, which contains invertible matrices with elements from **zmod**.

natural.h defines the **natural** class, which contains natural numbers of arbitrary size.

perm.h defines the **perm** class, which contains permutations.

rational.h defines the **rational** class, which contains rational numbers with numerators and denominators of arbitrary size.

testalg.cpp contains test code for all of the classes in GROUPECPP.

zmod.h defines the **zmod** class, which contains the integers mod n for some n .

zmodpoly.h defines the **zmodpoly** class, which contains polynomials with coefficients in **zmod**.

3 Group Classes and Common Operations

3.1 Group Classes

invmatff is a class for invertible matrices over a finite field.

invmatq is a class for invertible matrices over the rational numbers.

invmatz is a class for invertible matrices over the integers.

invmatzm is a class for invertible matrices over the integers modulo n for some n .

perm is a class for permutations.

3.2 Common Operations

Let x and y be arbitrary elements of the same group, and let n be any integer. Aside from the basic operations ($=$, $!$, $<$) the following operations are defined:

$x * y$ returns the product of x and y .

$x \wedge y$ returns the conjugate of x by y , i.e., $y^{-1}xy$.

$x \wedge n$ returns x^n , i.e., the n th power of x .

commutator(x,y) returns the commutator of x and y , i.e., $x^{-1}y^{-1}xy$.

x .**inverse**() returns the inverse of x .

x .**order**() returns the order of x , and is only defined for **invmatff**, **invmatzm**, and **perm**, since the order may be infinite for elements of **invmatq** and **invmatz**.

4 Auxiliary Classes

All of the auxiliary classes (except for **zmodpoly**) are number classes, meant to extend the functionality of existing data types such as **int**, **unsigned int**, etc. They share arithmetic operators ($+$, $-$, $*$, $/$, $\%$, \wedge , $+=$, $-=$, $*=$, $/=$, $\%=$) as well as relation operators ($=$, $!$, $<$, $<=$, $>$, $>=$) where appropriate.

ffield is a class for working with finite fields. The field is assumed to be of the form $(\mathbb{Z}/p\mathbb{Z})/(f(x))$ for some prime p and some irreducible polynomial $f(x)$. p and $f(x)$ must be initialized before the class can be used; no check is made to verify that p is a prime or that $f(x)$ is irreducible.

integer is a class for working with integers of arbitrary length.

natural is a class for working with natural numbers of arbitrary length.

rational is a class for working with rational numbers.

zmod is a class for working with the integers mod n for some n .

zmodpoly is a class for working with polynomials with coefficients in **zmod**.

5 Usage

5.1 `ffield`

5.1.1 Definition

Defines elements of a finite field $(\mathbb{Z}/p\mathbb{Z})[x]/(f(x))$ for a prime p and an irreducible polynomial $f(x)$. The size of the field is $p^{\deg(f)}$. The class does *not* verify that p is prime or that $f(x)$ is irreducible.

5.1.2 Initialization

1. `ffield::set_prime(p)`—where p is a prime and can be entered as an unsigned int or as a string: `ffield::set_prime(103)` or `ffield::set_prime("103")`.
2. `ffield::set_minpoly(f(x))`—where $f(x)$ is entered as a string, e.g., `ffield::set_minpoly("x^2 + 2")`.

5.1.3 Constructors

1. `ffield()`—default value is 0.
2. `ffield(unsigned int n)` for $n > 0$, e.g., `ffield(5)`.
3. `ffield(string)` where *string* is a polynomial written as a string, e.g., `ffield("x^3 - 3x^2 + x + 1")`.

5.1.4 Operators

1. Assignment/Output: `=`, `<<`
2. Arithmetic: `+`, `-`, `*`, `/`, `^`, `+=`, `-=`, `*=`, `/=`
3. Relations: `==`, `!=`

5.1.5 Member Functions

1. `x.inverse()` returns the reciprocal of x .

5.2 `integer`

5.2.1 Definition

Class contains integers, stored as strings, that can be of arbitrary length. All of the relevant operations already defined on the data type `int` have been defined for `integer`.

5.2.2 Constructors

1. `integer()`—default value is 0.
2. `integer(int n)` for $n \neq 0$, e.g., `integer(5)`.
3. `integer(string)` where *string* is an integer written as a string, e.g., `integer("-1234567890")`.
4. `integer(natural n)` converts from class **natural** to class **integer**.

5.2.3 Operators

1. Assignment/Output: `=`, `<<`
2. Arithmetic: `+`, `-`, `*`, `/`, `%`, `^`, `++`, `--`, `+=`, `-=`, `*=`, `/=`, `%=`
3. Relations: `==`, `!=`, `<`, `<=`, `>`, `>=`. The convention with `/`, `%`, `/=`, `%=` is that a remainder upon division is always nonnegative.

5.2.4 Member Functions

1. `x.abs()` returns the absolute value of x as type **natural**.

5.2.5 Other Functions

1. `gcd(m,n)` returns the greatest common divisor of m and n .

5.3 invmatff

5.3.1 Definition

Class contains invertible matrices with entries in a finite field (**ffield**).

5.3.2 Initialization

1. `ffield::set_prime(p)`—where p is a prime and can be entered as an unsigned int or as a string: `ffield::set_prime(103)` or `ffield::set_prime("103")`.
2. `ffield::set_minpoly($f(x)$)`—where $f(x)$ is entered as a string, e.g., `ffield::set_minpoly("x ^ 2 + 2")`.

5.3.3 Constructors

1. `invmatff(unsigned int n)` for $n > 0$ creates an $n \times n$ identity matrix, e.g., `invmatff(5)`.
2. `invmatff(string)` where *string* is of the form `"[[a,b,c],[d,e,f],[g,h,i]]"`, e.g., `invmatff("[[1,x],[x,1]]")`.

5.3.4 Operators

1. Assignment/Output: =, <<
2. Group: * (multiplication), ^ (conjugation), ^ (exponentiation)
3. Relations: ==, !=

5.3.5 Member Functions

1. `m.inverse()` returns the inverse of `m`.
2. `m.order()` returns the order of `m`.

5.3.6 Other Functions

1. `commutator(x,y)` returns the commutator of `x` and `y`, i.e., $x^{-1}y^{-1}xy$.

5.4 invmatq

5.4.1 Definition

Class contains invertible matrices with rational entries using **rational**.

5.4.2 Constructors

1. `invmatq(unsigned int n)` for $n > 0$ creates an $n \times n$ identity matrix, e.g., `invmatq(5)`.
2. `invmatq(string)` where `string` is of the form
`"[[a1/a2,b1/b2,c1/c2],[d1/d2,e1/e2,f1/f2],[g1/g2,h1/h2,i1/i2]]"`,
 e.g., `invmatq("[[1/2, 3/4],[-2/3,-4/5]]")`.

5.4.3 Operators

1. Assignment/Output: =, <<
2. Group: * (multiplication), ^ (conjugation), ^ (exponentiation)
3. Relations: ==, !=

5.4.4 Member Functions

1. `m.inverse()` returns the inverse of `m`.

5.4.5 Other Functions

1. `commutator(x,y)` returns the commutator of `x` and `y`, i.e., $x^{-1}y^{-1}xy$.

5.5 `invmatz`

5.5.1 Definition

Class contains invertible matrices with integer entries using **integer**.

5.5.2 Constructors

1. `invmatz(unsigned int n)` for $n > 0$ creates an $n \times n$ identity matrix, e.g., `invmatz(5)`.
2. `invmatz(string)` where *string* is of the form “[[a,b,c],[d,e,f],[g,h,i]]”, e.g., `invmatz(“[[1,3],[5,7]]”)`.

5.5.3 Operators

1. Assignment/Output: `=`, `<<`
2. Group: `*` (multiplication), `^` (conjugation), `^` (exponentiation)
3. Relations: `==`, `!=`

5.5.4 Member Functions

1. `m.inverse()` returns the inverse of *m*.

5.5.5 Other Functions

1. `commutator(x,y)` returns the commutator of *x* and *y*, i.e., $x^{-1}y^{-1}xy$.

5.6 `invmatzm`

5.6.1 Definition

Class contains invertible matrices with entries in **zmod**.

5.6.2 Constructors

1. `invmatzm(unsigned int n)` for $n > 0$ creates an $n \times n$ identity matrix, e.g., `invmatzm(5)`.
2. `invmatzm(string)` where *string* is of the form “[[a,b,c],[d,e,f],[g,h,i]]”, e.g., `invmatzm(“[[1,2],[3,4]]”)`.

5.6.3 Operators

1. Assignment/Output: `=`, `<<`
2. Group: `*` (multiplication), `^` (conjugation), `^` (exponentiation)
3. Relations: `==`, `!=`

5.6.4 Member Functions

1. `m.inverse()` returns the inverse of m .
2. `m.order()` returns the order of m .

5.6.5 Other Functions

1. `commutator(x,y)` returns the commutator of x and y , i.e., $x^{-1}y^{-1}xy$.

5.7 natural

5.7.1 Definition

Class contains naturals, stored as strings, that can be of arbitrary length. All of the relevant operations already defined on the data type **unsigned int** have been defined for **natural**. The idea for this class came from a chapter in [?].

5.7.2 Constructors

1. `natural()`—default value is 0.
2. `natural(unsigned int n)` for $n > 0$, e.g., `natural(5)`.
3. `natural(string)` where *string* is an natural written as a string, e.g., `“1234567890”`, e.g., `natural(“1234567890”)`.

5.7.3 Operators

1. Assignment/Output: `=`, `<<`
2. Arithmetic: `+`, `-`, `*`, `/`, `%`, `^`, `++`, `--`, `+=`, `-=`, `*=`, `/=`, `%=`
3. Relations: `==`, `!=`, `<`, `<=`, `>`, `>=`

5.7.4 Functions

1. `gcd(m,n)` returns the greatest common divisor of m and n .

5.8 perm

5.8.1 Definition

Class contains permutations acting on the positive natural numbers from the left. The permutations are treated as members of the direct limit of S_n as $n \rightarrow \infty$, so that any two permutations can be multiplied.

5.8.2 Constructors

1. `perm()` is the identity permutation.
2. `perm(string)` where *string* is of the form “(a b c)(d e f g)”, e.g., `perm(“(1 2 3)(4 5)”`). Entries in the cycles must be positive integers.

5.8.3 Operators

1. Assignment/Output: `=`, `<<`
2. Group: `*` (multiplication), `^` (conjugation), `^` (exponentiation)
3. Relations: `==`, `!=`

5.8.4 Member Functions

1. `p.inverse()` returns the inverse of *p*.
2. `p.order()` returns the order of *p*.

5.8.5 Other Functions

1. `commutator(x,y)` returns the commutator of *x* and *y*, i.e., $x^{-1}y^{-1}xy$.

5.9 rational

5.9.1 Definition

Class contains rationals that can have numerators and denominators of arbitrary size. All of the relevant operations already defined on the data type `int` have been defined for `rational`.

5.9.2 Constructors

1. `rational()`—default value is 0.
2. `rational(int n)` for $n \neq 0$, e.g., `rational(5)`.
3. `rational(string)` where *string* is an rational written as a string, e.g., `rational(“-1234567890/257”)`.

5.9.3 Operators

1. Assignment/Output: `=`, `<<`
2. Arithmetic: `+`, `-`, `*`, `/`, `^`, `++`, `--`, `+=`, `-=`, `*=`, `/=`
3. Relations: `==`, `!=`, `<`, `<=`, `>`, `>=`

5.10 zmod

5.10.1 Definition

Defines elements of the integers mod n for some positive integer n , which need not be a prime number.

5.10.2 Initialization

1. `zmod::set_modulus(n)`—where n can be entered as an unsigned int or as a string: `zmod::set_modulus(103)` or `zmod::set_modulus("103")`.

5.10.3 Constructors

1. `zmod()`—default value is 0.
2. `zmod(int n)`, e.g., `zmod(5)`.
3. `zmod(string)` where *string* is an integer written as a string, e.g., `zmod("1234567890")`.
4. `zmod(integer)`.

5.10.4 Operators

1. Assignment/Output: `=`, `<<`
2. Arithmetic: `+`, `-`, `*`, `/`, `^`, `+=`, `-=`, `*=`, `/=`. The `/` operator relies on the `inverse()` member function.
3. Relations: `==`, `!=`

5.10.5 Member Functions

1. `x.inverse()` returns the reciprocal of x . Based on extended Euclidean algorithm for finding a gcd as per Knuth [?].

5.11 zmodpoly

5.11.1 Definition

Defines polynomials with coefficients in `zmod`.

5.11.2 Initialization

1. `zmodpoly::set_modulus(n)`—where n can be entered as an unsigned int or as a string: `zmodpoly::set_modulus(103)` or `zmodpoly::set_modulus("103")`.

5.11.3 Constructors

1. `zmodpoly()`—default value is 0.
2. `zmodpoly(unsigned int n)`, e.g., `zmodpoly(5)`.
3. `zmodpoly(string)` where *string* is a polynomial written as a string, e.g., `zmodpoly("12x \wedge 3 - 45x \wedge 2 + 67x - 890")`.

5.11.4 Operators

1. Assignment/Output: `=`, `<<`
2. Arithmetic: `+`, `-`, `*`, `/`, `%`, `\wedge` , `+=`, `-=`, `*=`, `/=`, `%`.
3. Relations: `==`, `!=`

5.11.5 Member Functions

1. `f.degree()` returns the degree of f .
2. `f.lc()` returns the leading coefficient of f .

6 Examples

6.1 Example #1

```

/*****
 * example1.cpp
 *
 * Example of use of GROUPCPP code:  solution to the
 * following problem:
 *
 * Find the size of the group generated by (1 2 3)(4 5 6)
 * and (1 2)(3 4)(5 6) and list its elements.
 *
 * Jeffrey W. Clark, Elon College, clarkj@numen.elon.edu
 *
 * Last revised: 12/17/96
 *****/

#include <fstream.h>
#include "perm.h"

const MAX_SIZE = 100;

void main(void)
{
    ofstream fout;
    fout.open("example1.out", ios::trunc);

    perm a("(1 2 3)(4 5 6)");
    perm b("(1 2)(3 4)(5 6)");

    perm group[MAX_SIZE];

    perm id;
    group[0] = id;

    unsigned group_size = 1;

    // form all possible products from a, b, starting with identity

    for (unsigned int i = 0; i < group_size; i++)
    {
        perm x = group[i];
        perm xa = x*a;
        perm xb = x*b;
    }
}

```

```
// flags as to whether xa, xb are already in group
unsigned int a_flag = 0;
unsigned int b_flag = 0;

for (unsigned int j = 0; j < group_size; j++)
    if (xa == group[j])
        {
            a_flag = 1;
            break;
        }
if (a_flag == 0)
    {
        group[group_size] = xa;
        group_size++;
    }
for (unsigned int k = 0; k < group_size; k++)
    if (xb == group[k])
        {
            b_flag = 1;
            break;
        }
if (b_flag == 0)
    {
        group[group_size] = xb;
        group_size++;
    }
}
fout << endl << "The size of the group is " << group_size << ".";
for (unsigned int k = 0; k < group_size; k++)
    fout << endl << '#' << (k+1) << ":\t" << group[k];
}
```

6.2 Example #2

```

/*****
 * example2.cpp
 *
 * Example of use of GROUPCPP code:  solution to the
 * following problem:
 *
 * Find the conjugacy classes of the group generated the
 * following elements of  $GL(2, Z/3Z)$ :   $[[1,1],[0,1]]$  and
 *  $[[1,0],[1,1]]$ .
 *
 * Jeffrey W. Clark, Elon College, clarkj@numen.elon.edu
 *
 * Last revised: 12/17/96
 *****/

#include <fstream.h>
#include "invmatzm.h"

const MAX_SIZE = 48;          // size of  $GL(2, Z/3Z)$ 

unsigned int construct_group(const invmatzm& a, const invmatzm& b,
                             invmatzm*& group);

unsigned int location(const invmatzm& iff, invmatzm* group,
                     const unsigned int& group_size);

void main(void)
{
    ofstream fout;
    fout.open("example2.out", ios::trunc);

    zmod::set_modulus(3);
    invmatzm a("[[1,1],[0,1]]");
    invmatzm b("[[1,0],[1,1]]");
    invmatzm* group = NULL;

    // recycle code from example1.cpp to construct group
    unsigned int group_size = construct_group(a,b,group);

    // construct array of flags for elements already done
    unsigned int* done_flag = new(unsigned int[group_size]);
    for (unsigned int i = 0; i < group_size; i++) done_flag[i] = 0;

    // array to store elements of current conjugacy class

```

```

invmatzm* conj_class = new(invmatzm[group_size]);
unsigned int conj_class_size;

unsigned int conj_class_number = 1;

for (unsigned int j = 0; j < group_size; j++)
{
    if (done_flag[j] == 1) continue;
    fout << endl << endl << "Conjugacy class #" << conj_class_number;
    fout << ":" << endl;
    conj_class[0] = group[j];
    fout << endl << "#1: " << conj_class[0];
    conj_class_size = 1;
    done_flag[j] = 1;

    // form all possible conjugates with a, b,
    // starting with group[j]

    for (unsigned int k = 0; k < conj_class_size; k++)
    {
        invmatzm x = conj_class[k];
        invmatzm xa = x^a;
        done_flag[location(xa,group, group_size)] = 1;
        invmatzm xb = x^b;
        done_flag[location(xb,group, group_size)] = 1;

        // flags as to whether x^a, x^b are already in class
        unsigned int a_flag = 0;
        unsigned int b_flag = 0;

        for (unsigned int l = 0; l < conj_class_size; l++)
            if (xa == conj_class[l])
                {
                    a_flag = 1;
                    break;
                }
        if (a_flag == 0)
            {
                conj_class[conj_class_size] = xa;
                conj_class_size++;
                fout << endl << '#' << conj_class_size << ": " << xa;
            }
        for (unsigned int m = 0; m < conj_class_size; m++)
            if (xb == conj_class[m])
                {
                    b_flag = 1;
                }
    }
}

```

```

                break;
            }
        if (b_flag == 0)
        {
            conj_class[conj_class_size] = xb;
            conj_class_size++;
            fout << endl << '#' << conj_class_size << ": " << xb;
        }
    }
    conj_class_number++;
}
return;
}

unsigned int location(const invmatzm& iff, invmatzm* group,
    const unsigned int& group_size)
{
    for (unsigned int i = 0; i < group_size; i++)
        if (iff == group[i]) return i;
    return group_size;
}

unsigned int construct_group(const invmatzm& a, const invmatzm& b,
    invmatzm*& group)
{
    group = new(invmatzm[MAX_SIZE]);

    invmatzm id(2);
    group[0] = id;

    unsigned group_size = 1;

    // form all possible products from a, b, starting with identity
    for (unsigned int i = 0; i < group_size; i++)
    {
        invmatzm x = group[i];
        invmatzm xa = x*a;
        invmatzm xb = x*b;

        // flags as to whether xa, xb are already in group
        unsigned int a_flag = 0;
        unsigned int b_flag = 0;

        for (unsigned int j = 0; j < group_size; j++)

```

```
        if (xa == group[j])
            {
                a_flag = 1;
                break;
            }
    if (a_flag == 0)
        {
            group[group_size] = xa;
            group_size++;
        }
    for (unsigned int k = 0; k < group_size; k++)
        if (xb == group[k])
            {
                b_flag = 1;
                break;
            }
    if (b_flag == 0)
        {
            group[group_size] = xb;
            group_size++;
        }
    }
    return group_size;
}
```

6.3 Example #3

```

/*****
 * example3.cpp
 *
 * Example of use of GROUPCPP code: solution to the
 * following problem:
 *
 * Find the commutator subgroup of the group generated by
 *  $[[1,x],[0,1]]$  and  $[[1,0],[x,1]]$  in  $GL(2,F)$ , where
 *  $F = (Z/5Z)[x]/(x^2+2)$ 
 *
 * Jeffrey W. Clark, Elon College, clarkj@numen.elon.edu
 *
 * Last revised: 12/20/96
 *****/

#include <fstream.h>          // for file operation
#include "invmatff.h"        // for data manipulation

const MAX_SIZE = 480;       //  $GL(2,(Z/5Z)[x]/(x^2+2))$  has size 480
                             // =  $(25-1)*(25-5)$ 
unsigned int construct_group(invmatff*& gen_array,
                             const unsigned int& no_gens, invmatff*& group);

unsigned int find_conj_class(const invmatff& c, const invmatff& a,
                             const invmatff& b, invmatff*& conj_class);

void main(void)
{
   ffield::set_prime(5);
   ffield::set_minpoly("x^2+2");

    invmatff a("[[1,x],[0,1]]");
    invmatff b("[[1,0],[x,1]]");

    ofstream fout;
    fout.open("example3.out", ios::trunc);

    // commutator subgroup is smallest normal subgroup containing
    // [a,b], i.e., the group generated by [a,b] and all of its
    // conjugates

    // need to find conjugacy class of [a,b] recycling code from
    // example2.cpp, then recycle code from example1.cpp to
    // construct group

```

```

    invmatff* conj_class = NULL;
    invmatff c = commutator(a,b);
    unsigned int conj_class_size = find_conj_class(c,a,b,conj_class);

    invmatff* commutator_group = NULL;
    unsigned int commutator_group_size = construct_group(conj_class,
        conj_class_size, commutator_group);

    fout << endl << "The commutator group has size ";
    fout << commutator_group_size << ".";

    for (unsigned int k = 0; k < commutator_group_size; k++)
        fout << endl << '#' << (k+1) << ": " << commutator_group[k];
    return;
}

unsigned int construct_group(invmatff*& gen_array,
    const unsigned int& no_gens, invmatff*& group)
{
    group = new(invmatff[MAX_SIZE]);

    invmatff id(2);
    group[0] = id;

    unsigned group_size = 1;

    // form all possible products from generators starting with identity
    for (unsigned int i = 0; i < group_size; i++)
    {
        invmatff x = group[i];
        invmatff* xmults = new(invmatff[no_gens]);
        for (unsigned index = 0; index < no_gens; index++)
            xmults[index] = x*gen_array[index];

        // flags as to whether x*gen is already in group
        unsigned int* flags = new(unsigned int[no_gens]);
        for (unsigned index2 = 0; index2 < no_gens; index2++)
            flags[index2] = 0;

        for (unsigned index3 = 0; index3 < no_gens; index3++)
        {
            for (unsigned int j = 0; j < group_size; j++)
                if (xmults[index3] == group[j])
                    {

```

```

                                flags[index3] = 1;
                                break;
                                }
    if (flags[index3] == 0)
    {
        group[group_size] = xmults[index3];
        group_size++;
    }
}
return group_size;
}

```

```

unsigned int find_conj_class(const invmatff& c, const invmatff& a,
    const invmatff& b, invmatff*& conj_class)
{
    conj_class = new(invmatff[MAX_SIZE]);
    conj_class[0] = c;
    unsigned int conj_class_size = 1;
    for (unsigned int j = 0; j < conj_class_size; j++)
    {

        // form all possible conjugates with a, b,
        // starting with c

        invmatff x = conj_class[j];
        invmatff xa = x^a;
        invmatff xb = x^b;

        // flags as to whether x^a, x^b are already in class
        unsigned int a_flag = 0;
        unsigned int b_flag = 0;

        for (unsigned int k = 0; k < conj_class_size; k++)
            if (xa == conj_class[k])
            {
                a_flag = 1;
                break;
            }
        if (a_flag == 0)
        {
            conj_class[conj_class_size] = xa;
            conj_class_size++;
        }
        for (unsigned int l = 0; l < conj_class_size; l++)
            if (xb == conj_class[l])

```

```
        {
            b_flag = 1;
            break;
        }
    if (b_flag == 0)
        {
            conj_class[conj_class_size] = xb;
            conj_class_size++;
        }
    }
    return conj_class_size;
}
```