

quickcheck

Randomised Testing for **GAP** Functions

1.1.0

1 October 2026

Christopher Jefferson

Christopher Jefferson Email: caj21@st-andrews.ac.uk

Homepage: <https://heather.cafe/>

Address: Jack Cole Building, North Haugh, St Andrews, Fife,
KY16 9SX, Scotland

Contents

1	Introduction	3
2	Tutorial	4
2.1	Using QuickCheck	4
2.2	Valid argument types for QuickCheck	5
2.3	Adding new types to QuickCheck	7
3	Functionality	8
3.1	Methods	8
3.2	Argument descriptions	10
	Index	11

Chapter 1

Introduction

QuickCheck is a property-based testing package for GAP that automatically validates functions against randomly generated inputs. The package can either verify that a function consistently returns 'true' across inputs, or confirm that two functions produce identical results for the same inputs.

The strength of QuickCheck lies in its ability to rapidly test functions with diverse inputs, including edge cases that are often overlooked in manual testing, such as empty lists, trivial groups, or boundary values. By generating hundreds of test cases automatically, QuickCheck helps discover bugs and inconsistencies that might not be apparent from inspecting code or writing traditional tests.

This approach is particularly valuable for mathematical algorithms and group theory implementations where exhaustive testing is impractical.

Chapter 2

Tutorial

2.1 Using QuickCheck

The idea behind QuickCheck is to write functions which should always return true, or two functions which should always return the same value. We then feed those functions many different inputs, and see if that behaviour holds.

While this cannot detect all bugs, it can catch many issues. In particular, the code can catch issues with small inputs, or edge conditions. This tutorial will walk through the major functionality of the QuickCheck package through examples.

As a first example, let's test if GAP's integers are commutative -- that is if $a * b = b * a$. We first make a function which takes two arguments and returns true if the inputs are commutative. We test this using `QC_Check`. `QC_Check` has two mandatory arguments, firstly the types of the arguments of the functions we want to test, and secondly the function to test.

Example

```
gap> testFunc := function(a,b)
>   return a*b = b*a;
> end;;
gap> QC_Check([IsInt, IsInt], testFunc);
true
```

Great! We can use the same function to test if other types are commutative:

Example

```
gap> QC_Check([IsPerm, IsPerm], testFunc);
Test 60 of 500 failed:
  Input: [ (1,3), (1,3,2) ]
  Output: false
false
```

Turns out multiplication of permutations is not commutative! This isn't a surprise, of course...

If we want to analyse those arguments, we can access them through `QC_LastFailure`.

Example

```
gap> QC_LastFailure();
rec( args := [ (1,3), (1,3,2) ], func := function( a, b ) ... end )
```

Rather than running a single function and checking if it returns `true`, we can run two functions and test if they produce the same output. For example, let's compare two methods of intersecting groups -- GAP's optimised implementation and a "brute force" method which just finds the elements in both groups:

Example

```
gap> slowIntersection := function(g,h)
>   return Group(Filtered(g, p -> p in h));
> end;;
gap> QC_CheckEqual([IsPermGroup, IsPermGroup], Intersection, slowIntersection);
true
```

Could we be more efficient? How about if we just test which generators of g are in h ?

Example

```
gap> genIntersection := function(g,h)
>   return Group(Filtered(GeneratorsOfGroup(g), p -> p in h), ());
> end;;
gap> QC_CheckEqual([IsPermGroup, IsPermGroup], Intersection, genIntersection);
Test 97 of 500 failed:
  Input: [ Group( [ (1,2,3), (2,3,4) ] ), Group( [ (1,4)(2,3), (1,2)(3,4) ] ) ]
  Output: Return values differ: Group( [ (1,4)(2,3), (1,2)(3,4) ] ) and Group( () )
false
```

No, unfortunately not! You may be surprised it took until test 97 to find this bug (you may also get different results). This is because QuickCheck starts by making many very small tests, which are fast to execute (finding this bad example takes less than a tenth of a second).

In some cases, the inputs given might not make sense for the function we are testing. For example, imagine we want to test if $b * (a/b) = a$. This statement isn't valid for $b = 0$. We can skip invalid values by return the special value `QC_Skip`.

Example

```
gap> checkDiv := function(a,b)
>   if b = 0 then return QC_Skip; fi;
>   return b*(a/b);
> end;;
gap> justA := function(a,b)
>   return a;
> end;;
gap> QC_CheckEqual([IsInt, IsInt], checkDiv, justA);
true
```

In some cases we might want to return an explanation of why a test failed. Rather than returning `false` from a function tested by `QC_Check`, you can instead return a string. Returning a string is treated as a failure, and the string is printed out to the user.

2.2 Valid argument types for QuickCheck

The set of elements which can be given to QuickCheck tests is listed below. The list is always growing, and please submit an issue on GitHub if you have any specific requests.

The currently allowed argument types for QuickCheck are:

- IsInt : An integer
- IsPosInt : A positive integer
- IsNegInt : A negative integer
- IsRat : Rational number
- IsPosRat : Positive rational number
- IsNegRat : Negative rational number
- IsPerm : A permutation
- IsTransformation : A transformation
- IsPartialPerm : A partial permutation
- IsPermGroup : A permutation group
- IsDigraph : A digraph (requires the Digraph package)

There is also a generic function to take a random member from a collection:

- QC_ElementOf(x) : Return a random member of the list x

This can be used to take a random element of a small list of options (like [1,2,3]), or a random element of a collection which is not explicitly listed above (like AlternatingGroup(7)). The disadvantage of using QC_ElementOf is that it does not handle 'limit' well, but will instead take a random element from x.

There are also ways of defining lists or sets of arguments, recursively:

- QC_ListOf(x) : A (possibly empty) list of items of type x
- QC_SetOf(x) : A (possibly empty) set of items of type x
- QC_PairOf(x) : A pair of items of type x
- QC_FixedLengthListOf(x, len) : A list of length len of items of type x

In some cases this may not be sufficient, for example an algorithm may require an integer which is not prime. The function can return the special value QC_Skip, which skips this test. If too many tests are skipped, the tester will eventually stop generating new arguments. For example, if we want to check the AlternatingGroup(n) is a strict subgroup of the SymmetricGroup(n) for $n > 2$.

Example

```
gap> func := function(x)
>   local a, s;
>   if x < 2 then
>     return QC_Skip;
>   fi;
>   a := AlternatingGroup(x);
>   s := SymmetricGroup(x);
>   if not IsSubgroup(s, a) then return "Not a subgroup"; fi;
> end;
```

```

>   if s = a then return "Equal!"; fi;
>   return true;
> end;;
gap> QC_Check([IsInt], func);
true

```

2.3 Adding new types to QuickCheck

QuickCheck takes a list of the arguments for the function to be tested, so it needs to know how to generate values of any given type. There is a built-in list of types and new ones can be easily added.

All value generators are a function which takes two arguments -- the first is a RandomSource, and the second is a positive integer representing the maximum "size" of the value created (each type can choose how to interpret this).

For example, here are generators for a positive integer and a permutation.

Example

```

gap> makePosInt := function(rs, limit)
>   return Random(rs, [1..limit]);
> end;;
gap> makePerm := function(rs, limit)
>   return Random(rs, SymmetricGroup(limit));
> end;;

```

These functions can be used immediately in QC_Check:

Example

```

gap> QC_Check([makePosInt, makePerm, makePerm], {r,p1,p2} -> (r^p1)^p2 = r^(p1*p2));
true

```

These functions can also be registered with the filter which they implement, using QC_RegisterFilterGen:

Example

```

gap> gap> QC_RegisterFilterGen(IsPosInt, makePosInt);

```

Chapter 3

Functionality

3.1 Methods

This section will describe the methods of QuickCheck

3.1.1 QC_MakeRandomArgument

▷ `QC_MakeRandomArgument(ObjectDescription, RandomSource, limit)` (function)

Create a random object as described by *ObjectDescription* of size at most *limit* using *RandomSource*. How *limit* is interpreted will vary depending on the type of object.

3.1.2 QC_Check

▷ `QC_Check(arguments, function[, config])` (function)

Run tests on *function* with arguments as described in *arguments*.

3.1.3 QC_CheckEqual

▷ `QC_CheckEqual(arguments, functionL, functionR[, config])` (function)

Check that, given the same list of arguments as described in *arguments*, *functionL* and *functionR* return the same value.

3.1.4 QC_LastFailure

▷ `QC_LastFailure()` (function)

Return the function called, and arguments given, if the most recent call to `QC_Check` or `QC_CheckEqual` failed on a particular input.

Returns a record containing *args* (the arguments) and a function *func* (if `QC_Check` failed) or a list of functions *funcs* (if `QC_CheckEqual` failed). Returns `false` otherwise.

3.1.5 QC_RerunLastFailure

▷ `QC_RerunLastFailure()` (function)

Rerun the last test which failed, as given by `QC_LastFailure` (3.1.4). This is most useful if a test in a `.tst` file failed, as this will allow the test to enter the break loop. Returns `fail` if `QC_LastFailure` (3.1.4) returns `false`.

3.1.6 QC_SetConfig

▷ `QC_SetConfig(config)` (function)

Set config options for QuickCheck globally, by passing a record. It is not required to set all options. Current options are:

- `tests`: Number of tests to run (default 500)
- `limit`: The size of the largest object to create (default 9)
- `ramp`: Number of tests, including skipped ones, to run at each size before increasing it (default 30)
- `seed`: Initial random seed (default 1)
- `catchErrors`: If `true`, an error in the tested function counts as a failure; if `false`, it enters the break loop (default `true`)

3.1.7 QC_GetConfig

▷ `QC_GetConfig()` (function)

Get the current global configuration for QuickCheck, as a record

3.1.8 QC_Skip

▷ `QC_Skip` (global variable)

A function tested by `QC_Check` (3.1.2) or `QC_CheckEqual` (3.1.3) can return `QC_Skip` if its arguments do not satisfy its requirements (for example, if it needs an intransitive group, or an integer which is not prime). Skipped tests do not count as failures or towards the number of tests. To avoid infinite loops, if 100 times the requested number of tests are skipped, the check stops and returns `false`.

3.1.9 QC_RegisterFilterGen

▷ `QC_RegisterFilterGen(filter, gen)` (function)

Register *gen* as a generator for arguments described by the filter *filter*. *gen* is called as `gen(rs, limit)`, where *rs* is a random source and *limit* a positive integer bounding the size of the value. It is up to *gen* to decide how to interpret *limit*. Smaller values of *limit* are used first, so simple inputs are tested before complex ones. Values not in *filter* are discarded and regenerated,

with an error after 100 attempts. This lets a generator serve a more specific filter, for example an abelian permutation group when only a *gen* for permutation groups has been installed.

3.2 Argument descriptions

An argument description is either a filter with a registered generator (see `QC_RegisterFilterGen` (3.1.9)), or a function `gen(rs, limit)`. The functions below build descriptions from other descriptions.

3.2.1 QC_ListOf

▷ `QC_ListOf(desc)` (function)

Describes a list of between 0 and `limit` values described by `desc`.

Example

```
gap> QC_Check([QC_ListOf(QC_PairOf(IsPosInt))],
>             1 -> ForAll(1, p -> Length(p) = 2 and ForAll(p, IsPosInt)));
true
```

3.2.2 QC_FixedLengthListOf

▷ `QC_FixedLengthListOf(desc, len)` (function)

Describes a list of `len` values described by `desc`.

3.2.3 QC_PairOf

▷ `QC_PairOf(desc)` (function)

Describes a list of length two, whose entries are described by `desc`.

3.2.4 QC_SetOf

▷ `QC_SetOf(desc)` (function)

Describes a set of between 0 and `limit` values described by `desc`.

3.2.5 QC_ElementOf

▷ `QC_ElementOf(coll)` (function)

Describes a random element of the collection `coll`. This ignores `limit`.

Index

QC_Check, [8](#)
QC_CheckEqual, [8](#)
QC_ElementOf, [10](#)
QC_FixedLengthListOf, [10](#)
QC_GetConfig, [9](#)
QC_LastFailure, [8](#)
QC_ListOf, [10](#)
QC_MakeRandomArgument, [8](#)
QC_PairOf, [10](#)
QC_RegisterFilterGen, [9](#)
QC_RerunLastFailure, [9](#)
QC_SetConfig, [9](#)
QC_SetOf, [10](#)
QC_Skip, [9](#)