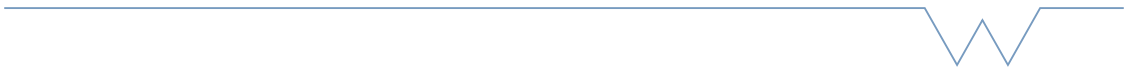

Warwick Research Software Engineering

Catalogue of Common Sorts of Bug

H. Ratcliffe and C.S. Brady
Senior Research Software Engineers

"The Angry Penguin", used under creative commons licence
from Swantje Hess and Jannis Pohlmann.

October 2, 2018

1 Bug Catalogue

1.1 Logic or algorithm bugs

Logic bugs are a catch-all for when your program does what it ought to, but not what you wanted; effectively you have written the wrong correct program. They can be very tricky to find, because they often arise from some misunderstanding of what you are trying to achieve. You may find it helpful to look over your plan (**you did make a plan, right?**)

Symptoms:

- incorrect answers
- poor performance

Examples:

- Finding the minimum of a list the wrong way: if your data is sorted, the minimum value must be the first in the list but sorting the data to just find the minimum item is silly.[1]
- Missing parts of a range:

```
INTEGER a, c
a = GET_NEXT_INTEGER()
IF ( a < 0 ) THEN
   c = 1
ELSE IF (a > 0 AND a < 5) THEN
   c = 0
ELSE IF ( a >= 5) THEN
   c = 2
END
```

  Notice that the case of a == 0 has been missed, and in this case c is undefined.

- Single branch of sqrt: calculating a from something like $a^2 = 9$, and forgetting that a can be either 3 or -3, introducing a sign error in further calculations
- Most typos: mistyped names, function assignment rather than call (Python), missing semicolon (C). Mis-use of operators, for example using & and && (C).
- The lampposts problem: for n slots, there are n+1 lampposts

1.2 Numerics bugs

1.2.1 Floating Point Basics

Computers store numbers in a fixed number of bits, using a binary representation. Usually, floating point numbers are stored as a number and a power of 2 to multiply it by,

[1] This exact issue showed up in actual proprietary code. Company must remain anonymous

analogous to scientific notation.[2] Most languages define several types of float, currently 32 bit and 64 bit being common. For 32-bit floats, 8 bits are used for the exponent, 23 for the significand[3] and one for the sign. This introduces several limitations. Firstly, for large numbers it can be the case that $X + 1 == X$ to the computer[4], because with only 23 bits, some of the number has been truncated. Secondly, they can't exactly represent all decimals, even those which terminate in normal, base-10 representation.[5] And finally, sufficiently large numbers cannot be stored at all, and will overflow.

1.2.2 Rounding Errors and Truncation

You may recall from school that for most calculations it is recommended to do all intermediate steps to as many significant figures as possible, before rounding the final result. In code, any time a number is stored into a variable it is converted to the correct type and any additional information is thrown away. For example in this snippet

```
INTEGER a = 3.0/2.0
PRINT a // Prints 1
```

the value 3.0/2.0 is calculated 1.5 and then stored into the integer a by truncating the non-integer part. Most programming languages truncate rather than round, i.e. they throw away the decimal part. This can lead to odd results in combination with floating point errors, for example the example below where 0.5/0.01 can be slightly less than 50, and so when truncated becomes 49.

Rounding errors are one of the reasons it is a bad idea to use floating point numbers as loop indexes, even if your language allows it. This snippet

```
REAL i
FOR i = 0.0, 1.0, 0.01 DO //Loop stride is 0.01
  PRINT i
END
```

can end up with i slightly less than 1 at the expected end point, and you may then sometimes get an entire extra iteration.

1.2.3 Combining Big and Small Numbers

When you add two numbers, you must first line up the places (like in long-hand arithmetic, matching units to units, tens to tens, hundreds to hundreds and so on), so because of the limited precision, if you add a small number to a large one, you lose precision in the smaller. In most cases, this wont matter much, because it will make only small differences to the final answer. There are two common issues however. Firstly, the order of the terms in the sum may make subtle changes to the final answer. In the worst cases this breaks associativity of addition,[6] because of the rounding at each

[2] E.g. $1.2 \times 10^3 == 1200$

[3] Number being multiplied, 1.2 in previous note

[4] See machine epsilon

[5] Any number with denominator a power of 2 is exact, all others aren't

[6] Associativity is the property that $a + b + c = (a + b) + c = a + (b + c)$

stage. If this matters, there exist special algorithms for dealing with such sums, to reduce these rounding and truncation errors. Secondly, if you take two such sums and subtract them, you can get strange cancellation results and unexpected zero or non-zero answers, see the example below.

1.2.4 Overflow and Underflow Bugs

All numeric types have an upper limit on the number they can store. Some languages, such as Python, may hide this but it is important to be aware of. You can look up the limits for your platform and data type. If you try and store a larger number (positive or negative) you will get an overflow error.

For integers, most languages define two kinds, signed and unsigned. Unsigned integers cannot be less than zero, and can therefore be roughly twice as large. In Python, integers will automatically become large-ints if they get too big, which can lead to significant slow-down. In C-like languages, signed integer overflow is well defined and should wrap around to negative values. Unsigned overflow is undefined behaviour although it often seems to work.

According to the IEEE (I triple-E) standards, there are strict standards for floating point operations, in particular which combinations are infinity or **NaN**(see also below). Fig 1 shows these. Note that there is a "signed zero" which has to result from some operations, such as $1/-Inf$

Underflow means that a number gets too small (not too-large-and-negative). There is a wrinkle here, which is explored a little in the accompanying exercise, called "denormal" numbers. Usually the base part of the number always has leading bit set, so is a number like $1.abcde$ as in normal scientific notation but base 2.[7] In denormal numbers the exponent is already as small as possible, so the significant is made a pure decimal. This allows the representation of smaller numbers, but at lower precision, because there is a finite number of bits available.[8]

1.2.5 Mixed Mode arithmetic

Mixed-mode arithmetic means using different types of number in an expression. For example in C, Fortran or Python 2[9]

```
INTEGER a = 1, b = 2
REAL d = 1, e = 2
REAL c = a / b
PRINT a / b, c, d / e // Prints 0, 0, 0.5
```

a/b is 0.5 exactly, as is d/e. However, in the first case, a and b are integers so the result of their division is converted to an integer, here by truncating (throwing away the fractional part) to give 0. This is the case even when we ask it be stored into a

[7] In normalised scientific notation 1.00×10^x and 9.99×10^x are both valid but e.g. 0.10×10^x is not.

[8] Suppose 5 "slots" are available: 1.2345×10^x has more sig.fig than $0.0123 \times 10^{x+3}$

[9] Python 3 promotes all divisions to be float, and uses the special operator // for integer division.

decimal, c. d and e are floating point numbers, so behave as we expect. In complicated expressions it can be difficult to work out which parts might be integer divisions, so it is a good idea to explicitly convert to floating point to be sure. Note that similar conversions occur if you mix 32 and 64 bit (or any other sizes) of number. The rules can be complicated, so again it is a good idea to keep everything the same or you may get unexpected loss of precision.

1.2.6 What is NaN?

NaN(nan in Python) is a special value indicating that something is Not a Number. **NaN**can result from e.g taking square-root of a negative number, as there is no real valued solution. **NaN**is contagious - any calculation involving a **NaN**will have **NaN**result. However **NaN**has one special property - it is not equal to any other number, including itself, and does not compare to any of them either. Beware though: this is true for any other comparison too, so **NaN**is not less than **NaN**nor greater than it. Figure 1 shows which arithmetic operations can result in **NaN**.

Symptoms:

- subtly wrong answers
- severe changes to answers on small edits
- underflow and overflow exceptions/warnings/signals (see Sec **??**)
- **NaN**, Inf or other special values in results

Examples:

- Non exact numbers: 1.0/33.0 = 0.030303030303030304 Where did that 4 come from? In C, if we ask for 1/33 to 45 dp we get 0.030303030303030303871381079261482227593660355
- The commonest non-bug in gcc: https://gcc.gnu.org/bugs/#nonbugs

```
#include <iostream>
int main()
{
  double a = 0.5;
  double b = 0.01;
  std::cout << (int)(a / b) << std::endl; //Prints either 49 or 50
     depending on optimisation etc
  return 0;
}
```

- In really bad cases this even breaks associativity: 1.0 - (1/33.0 + 2/33.0 + 29/33.0) - 1/33.0 may not give the same answer as 1.0 - (1/33.0 + 2/33.0 + 30/33.0)
- Divide by zero: in Python this is an exception. In gfortran you can set floating-point exception traps using -ffpe-trap etc

Left operand is in the column, right along the top

+	-inf	-1.0	-0.0	0.0	1.0	inf	nan
-inf	-inf	-inf	-inf	-inf	-inf	nan	nan
-1.0	-inf	-2.0	-1.0	-1.0	0.0	inf	nan
-0.0	-inf	-1.0	-0.0	0.0	1.0	inf	nan
0.0	-inf	-1.0	0.0	0.0	1.0	inf	nan
1.0	-inf	0.0	1.0	1.0	2.0	inf	nan
inf	nan	inf	inf	inf	inf	inf	nan
nan	nan	nan	nan	nan	nan	nan	nan

*	-inf	-1.0	-0.0	0.0	1.0	inf	nan
-inf	inf	inf	nan	nan	-inf	-inf	nan
-1.0	inf	1.0	0.0	-0.0	-1.0	-inf	nan
-0.0	nan	0.0	0.0	-0.0	-0.0	nan	nan
0.0	nan	-0.0	-0.0	0.0	0.0	nan	nan
1.0	-inf	-1.0	-0.0	0.0	1.0	inf	nan
inf	-inf	-inf	nan	nan	inf	inf	nan
nan	nan	nan	nan	nan	nan	nan	nan

-	-inf	-1.0	-0.0	0.0	1.0	inf	nan
-inf	nan	-inf	-inf	-inf	-inf	-inf	nan
-1.0	inf	0.0	-1.0	-1.0	-2.0	-inf	nan
-0.0	inf	1.0	0.0	-0.0	-1.0	-inf	nan
0.0	inf	1.0	0.0	0.0	-1.0	-inf	nan
1.0	inf	2.0	1.0	1.0	0.0	-inf	nan
inf	inf	inf	inf	inf	inf	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan

/	-inf	-1.0	-0.0	0.0	1.0	inf	nan
-inf	nan	inf	inf	-inf	-inf	nan	nan
-1.0	0.0	1.0	inf	-inf	-1.0	-0.0	nan
-0.0	0.0	0.0	nan	nan	-0.0	-0.0	nan
0.0	-0.0	-0.0	nan	nan	0.0	0.0	nan
1.0	-0.0	-1.0	-inf	inf	1.0	0.0	nan
inf	nan	-inf	-inf	inf	inf	nan	nan
nan	nan	nan	nan	nan	nan	nan	nan

Figure 1: IEEE requirements for floating point number operations

- Sums: Adding a small thing to a large thing
- Small differences of large numbers: (1.0 + 1e-15 + 3e-15) - (1.0 + 4e-15) is not zero in e.g. Python2, but (1.0 + 1e-15 + 3e-15) - (1.0 + 4e-15 + 2e-16) is.
- Integer division: discussed under Mixed-Mode above. Result of 1/2 is different to 1.0/2.0
- In Python all real numbers are "double precision" so run up to about 1.8e308. Integers will automatically grow to hold their content BUT they will get potentially a lot slower when they do. In C or Fortran you specify type, float/double and int/long. Try this:

```
int j=2147483647;
printf("%d\n", j);
printf("%d\n", j+1);
```

  The value j is set to is the maximum possible 64 bit integer, so the +1 "overflows". The reason it becomes negative is because of how integers are represented. Note there are also "unsigned" integers, which overflow to 0.

- Instability: some algorithms work analytically but when numerical effects like rounding are included can become unstable and give a wrong or no answer

1.3 Initialisation Bugs

These types of bugs occur when you create a variable and forget or fail to set it (such as in the range-error example in Sec 1.1), or when you accidentally create a local variable and think you're working with a global (or vice versa). In C-type languages where you manage memory they can occur when you forget to allocate memory, or when you forget to nullify a pointer and then try and work with it. These latter are more serious as you can accidentally overwrite other parts of your program's data.

Symptoms:

- segmentation (seg) faults or crashing
- different answers run-to-run
- severe changes to behaviour on small edits
- rogue connections between supposedly independent bits of code
- changes with optimisation level

Examples:

- Undefined variable: variable gets whatever value that bit of memory happened to have. Code may give the wrong answer, may vary run to run, may be changed if unrelated code is edited

- Undefined pointer: a pointer gets whatever value the memory had. Usually leads to a segmentation (seg) fault if you try to access it. Always nullify your pointers.
- Unallocated memory: memory isn't allocated, but the pointer or reference to it "seems" OK and doesn't crash. Can overwrite other parts of your program's data etc and lead to anything from an infinite loop (if you accidentally clobber your loop index) to wrong answers, crashing at seemingly unrelated times, etc

1.4 Memory Bugs

Note: overlaps with Sec 1.3

These types of bugs occur when try and create very large objects, or very many objects, when you try to access beyond the range of an object such as an array, when you try to use an unitialized or already freed pointer in languages where you manage your own memory or lifetimes, e.g. C or Fortran, or when you rapidly create and free objects in a language with a garbage collector. A similar sort of error exists where you use resources like file identifiers.

Symptoms:

- hanging (if you try to obtain too much of a resource)
- changes to wrong data
- crashes (usually only if you write to completely invalid memory, or exceed available memory)

Examples:

- Off-by-1 error in array access:

```
ARRAY, REAL(100) arr
arr[0:100] = 1
```

  Remember that different languages use 0 or 1 for the first element of an array.

- Buffer overrun errors: (C strings, C++ if you use C-style strings etc)

```
  char name[10] = "Abcdefghij"; // No space for null terminator,
     string is now invalid
  printf("%s", name); //See string content and then some junk
```

- Using a freed pointer: When objects are freed, their memory is marked free, but is not changed until it is allocated to something else.[10] If you are lucky, use of a freed pointer will segfault or show up fast. If you are unlucky, it will work perfectly as the "ghost" of the data is still there, and show up only as subtle bugs. **Always nullify pointers after freeing the memory they point to**

[10] This may be familiar if you have ever accidentally deleted a file - often the file is still there on the disk and can be restored, but any new files written may overwrite it.

- Severely bad writes (out of program arena): common error in C, possible in Fortran but uncommon, can occur when you attempt to allocate a very very large memory block. For extra fun these can crash tools like Valgrind (later) when you try to diagnose.

- Mis-allocation of memory:

```
int * start = malloc(1024*1024*1024*3); //Exceeds MAX_INT, actually
    attempts to allocate a negative number
```

  Note that your compiler is unlikely to warn you about this, even if it detects the overflow

1.5 Speed Bugs

Slowness is not a bug in itself, but it can be symptom of various logic-type bugs. We include these here as they have many of the same characteristics. The commonest cause is repeating work unnecessarily, for instance inside a loop.
Examples:

- File IO

```
INTEGER i
ARRAY data[100] = GET_NAMES()
FILE output_file
FOR i = 0, 99 DO
  OPENW output_file //Open file for writing
   data[i] = TO_UPPER_CASE(data[i])
  WRITE output_file, data[i]
  CLOSE output_file
END
```

- Multiple loops where one would do, and/or loops where array ops would do

```
INTEGER i
ARRAY data[100] = GET_INTEGERS()
FOR i = 0, 99 DO
   data[i] = data[i] + 1
END
FOR i = 0, 99 DO
   data[i] = data[i] *2
END
```

  Usually the overhead of the loop is not much, but in these cases with very little work to do, it can be significant. These examples could also be replaced with a single array operation (in Fortran, Python, C++) which can be much more efficient.

- Wrong algorithms (see also Sec 1.1)

- Not breaking early:

```
1 INTEGER i
2 ARRAY data[100] = GET_NAMES()
3 FLAG found = FALSE
4 FOR i = 0, 99 DO
5   IF (data[i] == "Bob") THEN
6     found = TRUE
7     //Bob found
8   END
9 END
```

  Since the only purpose of this loop is to identify whether the given value exists in the data, we may as well BREAK on line 7, as continuing the loop is needless work.[11]

Glossary - Testing and Debugging

code path Aka control flow path. The path taken through your source code when your program runs. For example, each "if" statement causes a branch into two paths, true and false. The complexity grows exponentially: a second if following the first gives up to 4 paths and a 3rd gives up to 8. The use of "up-to" is because in general many paths will be indistinguishable because the branches are independent. Ideally all code paths should be tested.

correct A program that uses all language features that it uses correctly. It does not necessarily give the answer that you expect. 1

floating point numbers Decimals, where the position of the decimal point is allowed to vary. This gives them the same relative precision over a very large range of values. Scientific notation is technically a floating point notation: although you always write a fixed number of decimals, the scaling factor 10^x means the absolute accuracy changes. 1

garbage collector When variables go out of scope the memory they occupy should be made available for reuse. In languages that allow references or pointers to variables, there is a problem with this, as you only want this to happen after the last reference to a given item is lost. This is done by the "garbage collector" in languages which have one. Usually, this runs every so often, or when available memory is getting tight, and cleans up all the now dead values. Note that languages like C don't have this, you must free memory when the last pointer or

[11] In some security critical code, early breaking introduces the potential for a "timing attack" where one gets information based on how long code takes to execute. For instance, in the example here, we might be able to guess whether "Bob" is definitely in the list, because those cases would stop slightly quicker.

reference goes. In Fortran you must manually clean up Pointers, but Allocatables are automatically deallocated (since F95) and cleaned up like regular variables. 7

invariant A condition which is always fulfilled. For example, within any for loop (FOR i =0, 10) you know that i is between 0 and 10. Often it is very useful to know that a number is positive, non-zero etc as this allows you use code that relies on this (e.g. if you're passing the number to a sqrt function, or dividing by it respectively).

machine epsilon Floating point numbers are stored in the exponential format $a \times 2^b$. With a limited number of bits, there is a finite step from one number to the next that can be represented. For example, in base-10 with a 5 digit significand (a) we have 1.0000×10^b and the next number we can show is 1.0001×10^b. For $b = 0$ the difference between these is 0.0001. For $b = 4$ the difference is 1. This step-size is called the machine epsilon, and is usually quoted for numbers close to 1. For very large numbers, the step size can be much greater than 1. This is one reason why you should not using floating-point numbers for large integers. 1

minimum working example A small program that demonstrates something with as little extra code as possible. They are very useful when trying out a new library or code, or when reporting a problem or bug. If somebody is trying to help, they wont appreciate wading through reams of code to find the problem, so producing a small program that demonstrates your issue is very useful.

no-op Code that does nothing. Stands for no-operation, and can exist for many reasons. Can include incomplete statements, like `a;` in C or conditionals like `if(false)`.

overflow A numerical error caused by exceeding the largest number (positive or negative) a type can store. 3

precondition and postcondition Guarantees about the inputs (precondition) or the outputs (postcondition) of a function. E.g. for a sqrt function, you may have to be sure that the input is positive, and the function may promise to return only the positive branch ($\sqrt{}(9)$ is plus OR minus 3).

regression Going backwards in code fitness, e.g. reintroducing a bug which was already fixed, making answer quality worse, breaking a working feature etc.

segmentation (seg) fault A severe error in code causing it to attempt to read or write invalid memory. 6

syntax error An error in the sequence of characters in a piece of code. For example, a misspelled name, or a missing bracket, making the piece invalid and unreadable to the computer.

underflow A numerical error caused by attempting to store a number that is too small, often causing it to be rounded to 0. 3

unreachable code Code that never runs. This can be a function that is never called, or a condition that is always true or false, so its body is unreachable.