Fast square and multiply modular exponentiation (FSMMAE)

Abstract

Modular exponentiation: (a^b) mod n can be computed using square and multiply exponentiation algorithm[1]. The present algorithm in this repository describes a faster variant stopping the computation when intermediary modulus is 0.

Observations

The modulus of a number by a number that begins with a number it begins with is the same as the modulus of the number by the number it ends with

```
2555 mod 25 = 55 mod 25
```

The number begins 25, so the modulus of 2555 by 25 is the same as the modulus of 55 by 25. Both are 5.

[1] Christof Paar, Jan Pelzl, Understanding Cryptography, section 7.4

Why? Because `2555 = 25 * 100 + 55` and modulus of `25 * 100 by 25 is 0`, so the modulus of 2555 by 25 is the same as the modulus of 55 by 25.

A generalization of the previous observation is that a modulus of a number is equal to the modulus of this number plus another number which modulus is equal to 0

```
5055 mod 25 = 55 mod 25
```

So can we compute only the smaller number modulus to be faster?

Yes, using the square and multiply modular exponentiation algorithm and storing the last modulus. If the current one is equal to 0, the computation can stop, and the algorithm can return the last modulus stored.

Python implementation

Prototyping is done in Python for the ease of development and testing. The corresponding module is `pysma.py`. The reference implementation is `pow` Python built-in method.

C implementation

For a portable implementation and low-level profiling of the executables, C is used to test than new implementation. It is a transpile of the Python code. The classic square and multiply modular exponentiation is implemented in `sma.c`. The fast square and multiply modular exponentiation is implemented in `fsma.c`. Again the reference implementation is `pow` Python built-in method.

Dependencies

Compilation and tests need LLVM, cmake, Clang/cc/gcc or any compiler, Valgrind and Python3.

Compilation

Both sources are compiled as an executable to be used in CLI and shared library to be loaded using Python for testing purposes. The compilation is done using the following command:

```
$ make all
```

Testing

The tests are done using a number inferior to a C unsigned long integer, the type used in C programs, as base, exponent, and modulus. Running the corresponding Python module should end in return code 0 if the result is the same as the reference implementation `pow` otherwise an `assertionError` is raised.

Test results Both `sma.c` and `fsma.c` compiled shared libraries returned results equal to the reference implementation `pow` for all the numbers inferior to a C unsigned long integer tested.

Hypothesis

The FSMMEA algorithm is generally faster than the SMA algorithm because of the computation ending sooner in case a temporary modulus is equal to 0. The impact of storing the last modulus and comparing the current to 0 is negligible.

Performance analysis

Counting instructions using Valgrind Has expected, comparing the temporary remainder to 0 has a cost. When no intermediate remainder is 0, the

computing function itself execute 20% more instructions. In this case the percentage is constant.

The classic implementation counts 567 instructions:

```
$ valgrind --tool=callgrind --toggle-collect=sma ./sma-prof <<<"97032574325492 4294965003 35
==1559769== Callgrind, a call-graph generating cache profiler
==1559769== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==1559769== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==1559769== Command: ./sma-prof
==1559769==
==1559769== For interactive control, run 'callgrind_control -h'.
1040
==1559769==
==1559769== Events    : Ir
==1559769== Collected : 567
==1559769==
==1559769== I   refs:      567
```

The currently described implementation counts 679 instructions:

```
$ valgrind --tool=callgrind --toggle-collect=fsma ./fsma-prof <<<"97032574325492 4294965003
...
==1559866== I   refs:      679
```

When there is an intermediary remainder equal to 0, the number of instructions is reduced by proportion that changes depending on the number of square and multiply left to compute. Here above an example with an 82% decrease of the number of instructions executed.

The classical implementation counts 457 instructions:

```
$ valgrind --tool=callgrind --toggle-collect=sma ./sma-prof <<<"294 98725745 98"
...
==1577575== Events    : Ir
==1577575== Collected : 457
==1577575==
==1577575== I   refs:      457
```

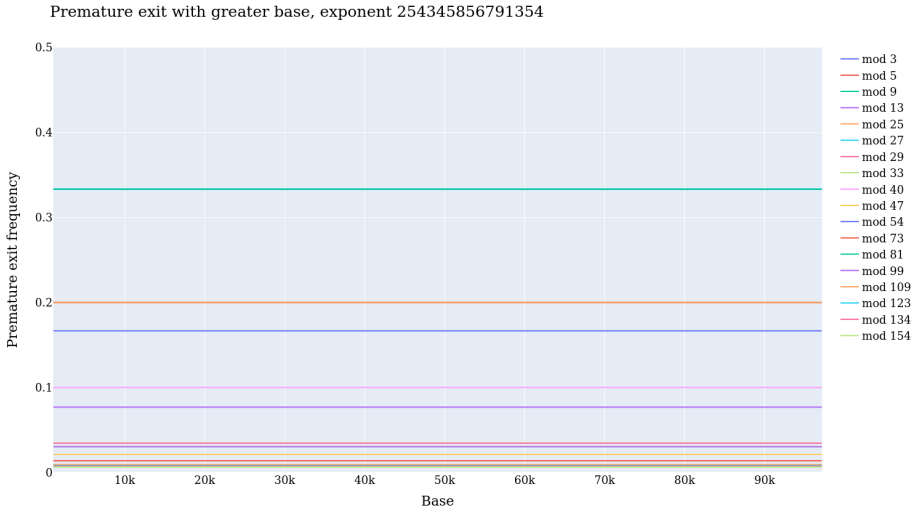
The currently described implementation counts 84 instructions:

```
$ valgrind --tool=callgrind --toggle-collect=fsma ./fsma-prof <<<"294 98725745 98"
...
==1577460== Events    : Ir
==1577460== Collected : 84
==1577460==
==1577460== I   refs:      84
```

The proportion of instructions executed is not constant and the chance of having an intermediary remainder equal to 0 is not constant either. It surely increases with the number of square and multiply to compute. To measure those

phenomenons, the C is not adapted. A more high-level language like Python with it data science tools is needed.

Analysing the premature exit of the algorithm The frequency of the premature exit, i.e., intermediary remainder equal to 0 of the algorithm does not change why the base. But it changes with the modulus. The following graph shows the frequency (linear regression to suppress variation du to sampling) of premature exit of the algorithm function of the base.



But the frequency of premature exit tend to zero with the modulo getting greater.