TIFA

0.1.0(devel:20110617)

Generated by Doxygen 1.5.5

Fri Jun 17 11:10:11 2011

Contents

1 TIFA

1.1 About the TIFA library

TIFA is an acronym standing for "Tools for Integer FActorisation". As its (utterly unoriginal) name implies **TIFA** is a open source library for composite integer factorization. Its goal is to provide portable and reasonably fast implementations for several algorithms, with a particular emphasis on the factorization of small to medium-sized composites, say from 40 bits to about 200 bits.

Although it obviously won't break any record by itself, **TIFA** may be a good companion to more ambitious factorization attempts such as a distributed implementation of the Number Field Sieve, where it could be used to factor the numerous smaller-sized by-products.

1.2 License

1.3 Content of the TIFA package

Actually, **TIFA** is a little bit more than a library *per se*. The **TIFA** package supplies:

- a **C99** library providing implementations for the following factorization algorithms:
  - CFRAC (Continued FRACtion factorization)
  - ECM (Elliptic Curve Method)
  - Fermat (McKee's "fast" variant of Fermat's algorithm)

  - SIQS (Self-Initializing Quadratic Sieve)
  - SQUFOF (SQUare FOrm Factorization)
- a set of stand-alone factorization programs for each algorithm implemented:
  - `cfrac_program`
  - `ecm_program`
  - `fermat_program`
  - `siqs_program`
  - `squfof_program`
- a set of **Perl 5** scripts wrappers and launchers;
- a basic benchmarking framework written in **Perl 5** used to assess the performance of **TIFA**'s implementations.

1.4 Documentation

A complete user's guide is in preparation.

In the interim, the best source of documentation (apart from this Doxygen documentation generated during the build process) is the included (infamous) `readme.txt` file in the `readme` directory.

Also worth a look is the (unfortunately not empty) `issues.txt` file.

2 Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

3 File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

4 Data Structure Documentation

4.1 struct_approximer_t Struct Reference

Structure used to find number approximation.

`#include <lib/utils/include/approx.h>`

Data Fields

- mpz_t target
- float targetlog
- float dlog_tolerance
- uint32_array_t ∗ facpool
- uint32_t nfactors
- int32_t imin
- int32_t imax
- uint32_t keven
- uint32_t neven
- uint32_t kodd
- uint32_t nodd
- uint32_t nsubsets_odd
- uint32_t ∗ subset_odd
- uint32_t rank
- uint32_tuple_t ∗ tuples
- uint32_t ntuples

4.1.1 Detailed Description

Structure used to find number approximation.

The structure `approximer_t` (and its associated functions) is used to approximate a target value by multiplying a given number of factors from a given base. Each factor is allowed to appear only once in the decomposition of the approximation on the given base.

Definition at line 116 of file approx.h.

4.1.2 Field Documentation

4.1.2.1 mpz_t struct_approximer_t::target

The target number to approximate.

Definition at line 120 of file approx.h.

4.1.2.2 float struct_approximer_t::targetlog

Logarithm in base 10 of the target number.

Definition at line 124 of file approx.h.

4.1.2.3 float struct_approximer_t::dlog_tolerance

The logarithm in base 10 of the approximation should be within `dlog_tolerance` of `targetlog`.

Definition at line 129 of file approx.h.

4.1.2.4 uint32_array_t* struct_approximer_t::facpool

Pool of potential factors.

Definition at line 133 of file approx.h.

4.1.2.5 uint32_t struct_approximer_t::nfactors

Numbers of factors from `facpool` in the approximation.

Definition at line 137 of file approx.h.

4.1.2.6 int32_t struct_approximer_t::imin

`facpool[imin]` is the smallest factor allowed.

Definition at line 141 of file approx.h.

4.1.2.7 int32_t struct_approximer_t::imax

`facpool[imax]` is the largest factor allowed.

Definition at line 145 of file approx.h.

4.1.2.8 uint32_t struct_approximer_t::keven

Number of factors to choose with even indexes.

Definition at line 149 of file approx.h.

4.1.2.9 uint32_t struct_approximer_t::neven

Number of available factors with even indexes.

Definition at line 153 of file approx.h.

4.1.2.10 uint32_t struct_approximer_t::kodd

Number of factors to choose with odd indexes.

Definition at line 157 of file approx.h.

4.1.2.11 uint32_t struct_approximer_t::nodd

Number of available factors with odd indexes.

Definition at line 161 of file approx.h.

4.1.2.12 uint32_t struct_approximer_t::nsubsets_odd

Number of distinct combination of `kodd` factors from the pool of odd indexed factors (in other words C(`nodd`, `kodd`)).

Definition at line 167 of file approx.h.

4.1.2.13 uint32_t* struct_approximer_t::subset_odd

A subset of `kodd` odd-indexes.

Definition at line 171 of file approx.h.

4.1.2.14 uint32_t struct_approximer_t::rank

The rank of the subset of `kodd` odd-indexes in lexicographic order.

Definition at line 176 of file approx.h.

4.1.2.15 uint32_tuple_t* struct_approximer_t::tuples

List of tuples obtained by precomputing all combinations of `keven` factors.

Definition at line 181 of file approx.h.

4.1.2.16 uint32_t struct_approximer_t::ntuples

Number of all possible combinations of `keven` factors.

Definition at line 185 of file approx.h.

The documentation for this struct was generated from the following file:

- approx.h

4.2 struct_binary_array_t Struct Reference

Defines an array of bits.

`#include <lib/utils/include/array.h>`

Data Fields

- uint32_t alloced

- uint32_t length
- TIFA_BITSTRING_T * data

4.2.1 Detailed Description

Defines an array of bits.

This structure defines an array of bits which knows its current length and its allocated memory space.

Note:

Internally, bits are packed in a `TIFA_BITSTRING_T` array.

Definition at line 1127 of file array.h.

4.2.2 Field Documentation

4.2.2.1 uint32_t struct_binary_array_t::alloced

Memory space allocated for this array's `data` field, given as a multiple of `sizeof(TIFA_BITSTRING_T)`. This is the maximum number of `TIFA_BITSTRING_T` that the array can accommodate. The number of bits that the array can hold is hence `CHAR_BIT * sizeof(TIFA_BITSTRING_T)` times this value (`CHAR_BIT` being the number of bits used to represent a `char`, usually 8 on most current architectures).

Definition at line 1137 of file array.h.

4.2.2.2 uint32_t struct_binary_array_t::length

Current number of bits hold in the array pointed by the structure's `data` field.

Definition at line 1142 of file array.h.

4.2.2.3 TIFA_BITSTRING_T* struct_binary_array_t::data

Array of `TIFA_BITSTRING_T` whose size is given by the `alloced` field.

Definition at line 1147 of file array.h.

Referenced by flip_array_bit(), get_array_bit(), set_array_bit_to_one(), and set_array_bit_to_zero().

The documentation for this struct was generated from the following file:

- array.h

4.3 struct_binary_matrix_t Struct Reference

Defines a matrix of bits.

```
#include <lib/utils/include/matrix.h>
```

Data Fields

- uint32_t nrows_alloced
- uint32_t ncols_alloced

- uint32_t nrows
- uint32_t ncols
- TIFA_BITSTRING_T ** data

4.3.1 Detailed Description

Defines a matrix of bits.

This structure defines a matrix of bits which knows its current dimensions and its allocated memory space.

Note:

Internally, a matrix of bits is represented as a matrix of `TIFA_BITSTRING_T` elements.

Definition at line 71 of file matrix.h.

4.3.2 Field Documentation

4.3.2.1 uint32_t struct_binary_matrix_t::nrows_alloced

Maximum number of rows of the matrix.

Definition at line 75 of file matrix.h.

4.3.2.2 uint32_t struct_binary_matrix_t::ncols_alloced

Maximum number of columns of the matrix. Since bits are packed in `TIFA_BITSTRING_T` elements, the maximum number of bits per column is 8 * `sizeof(TIFA_BITSTRING_T)` * `nrows_alloced`.

Hence the total allocated memory for the `data` field is `nrows_alloced` * `ncols_alloced` * `sizeof(TIFA_BITSTRING_T)` bytes.

Definition at line 86 of file matrix.h.

4.3.2.3 uint32_t struct_binary_matrix_t::nrows

Current number of rows of the matrix.

Definition at line 90 of file matrix.h.

Referenced by first_row_with_one_on_col().

4.3.2.4 uint32_t struct_binary_matrix_t::ncols

Current number of columns of the matrix.

Definition at line 94 of file matrix.h.

4.3.2.5 TIFA_BITSTRING_T** struct_binary_matrix_t::data

2D array of `TIFA_BITSTRING_T` elements whose dimensions are given by the `nrows_alloced` and `ncols_alloced` fields.

Definition at line 99 of file matrix.h.

Referenced by first_row_with_one_on_col(), flip_matrix_bit(), get_matrix_bit(), set_matrix_bit_to_one(), and set_matrix_bit_to_zero().

The documentation for this struct was generated from the following file:

- matrix.h

4.4 struct_byte_array_t Struct Reference

Defines an array of bytes.

`#include <lib/utils/include/array.h>`

Data Fields

- uint32_t alloced
- uint32_t length
- unsigned char * data

4.4.1 Detailed Description

Defines an array of bytes.

This structure defines a special kind of byte array (actually `unsigned char` array) which knows its current length and its allocated memory space.

Definition at line 100 of file array.h.

4.4.2 Field Documentation

4.4.2.1 uint32_t struct_byte_array_t::alloced

Memory space allocated for this array's `data` field, given as a multiple of `sizeof(unsigned char)`. This is the maximum number of bytes that the array can accommodate.

Definition at line 106 of file array.h.

4.4.2.2 uint32_t struct_byte_array_t::length

Current number of bytes hold in the array pointed by the structure's `data` field.

Definition at line 111 of file array.h.

Referenced by is_in_sorted_byte_array().

4.4.2.3 unsigned char* struct_byte_array_t::data

Array of `unsigned char` whose size is given by the `alloced` field.

Definition at line 116 of file array.h.

The documentation for this struct was generated from the following file:

- array.h

4.5 struct_byte_matrix_t Struct Reference

Defines a matrix of bytes.

`#include <lib/utils/include/matrix.h>`

Data Fields

- uint32_t nrows_alloced
- uint32_t ncols_alloced
- uint32_t nrows
- uint32_t ncols
- unsigned char ** data

4.5.1 Detailed Description

Defines a matrix of bytes.

This structure defines a matrix of bytes which knows its current dimensions and its allocated memory space.

Note:

Internally, a matrix of bytes is represented as a matrix of `unsigned` char elements.

Definition at line 351 of file matrix.h.

4.5.2 Field Documentation

4.5.2.1 uint32_t struct_byte_matrix_t::nrows_alloced

Maximum number of rows of the matrix.

Definition at line 355 of file matrix.h.

4.5.2.2 uint32_t struct_byte_matrix_t::ncols_alloced

Maximum number of columns of the matrix.

Definition at line 359 of file matrix.h.

4.5.2.3 uint32_t struct_byte_matrix_t::nrows

Current number of rows of the matrix.

Definition at line 363 of file matrix.h.

4.5.2.4 uint32_t struct_byte_matrix_t::ncols

Current number of columns of the matrix.

Definition at line 367 of file matrix.h.

4.5.2.5 unsigned char** struct_byte_matrix_t::data

2D array of `unsigned` char elements whose dimensions are given by the `nrows_alloced` and `ncols_alloced` fields.

Definition at line 372 of file matrix.h.

The documentation for this struct was generated from the following file:

- matrix.h

4.6 struct_cfrac_params_t Struct Reference

Defines the variable parameters used in the CFRAC algorithm.

`#include <lib/algo/include/cfrac.h>`

Data Fields

- uint32_t nprimes_in_base
- uint32_t nprimes_tdiv
- uint32_t nrelations
- linalg_method_t linalg_method
- bool use_large_primes
- smooth_filter_method_t filter_method
- unsigned short int nsteps_early_abort

4.6.1 Detailed Description

Defines the variable parameters used in the CFRAC algorithm.

This structure defines the set of the variable parameters used in the CFRAC algorithm.

Definition at line 93 of file cfrac.h.

4.6.2 Field Documentation

4.6.2.1 uint32_t struct_cfrac_params_t::nprimes_in_base

Number of prime numbers composing the factor base on which to factor the residues.

Definition at line 98 of file cfrac.h.

4.6.2.2 uint32_t struct_cfrac_params_t::nprimes_tdiv

Number of the first primes to use in the trial division of the residues known to be smooth.

Warning:

`nprimes_tdiv` should be greater than or equal to 1.

Definition at line 105 of file cfrac.h.

4.6.2.3 uint32_t struct_cfrac_params_t::nrelations

Number of linear dependences to find.

Definition at line 109 of file cfrac.h.

4.6.2.4 linalg_method_t struct_cfrac_params_t::linalg_method

Linear system resolution method to use.

Definition at line 113 of file cfrac.h.

4.6.2.5 bool struct_cfrac_params_t::use_large_primes

True if we use the single large prime variation. False otherwise.

Definition at line 118 of file cfrac.h.

4.6.2.6 smooth_filter_method_t struct_cfrac_params_t::filter_method

Method to use to detect smooth residues and relations.

Definition at line 122 of file cfrac.h.

4.6.2.7 unsigned short int struct_cfrac_params_t::nsteps_early_abort

Number of steps in the early abort strategy. If zero, no early abort is performed. Only used is `linalg_method` is set to `TDIV` or `TDIV_EARLY_ABORT`.

Note:

`nsteps` should be less than or equal to `MAX_NSTEPS`, as defined in smooth_filter.h.

Definition at line 131 of file cfrac.h.

The documentation for this struct was generated from the following file:

- cfrac.h

4.7 struct_cont_frac_state_t Struct Reference

An ad-hoc structure for the computation of the continued fraction of a square root.

`#include <sqrt_cont_frac.h>`

Data Fields

- mpz_t a
- mpz_t p
- mpz_t q
- mpz_t t
- mpz_t sqrtn
- mpz_t n
- uint32_t nsteps_performed

4.7.1 Detailed Description

An ad-hoc structure for the computation of the continued fraction of a square root.

lib/utils/include/sqrt_cont_frac.h This ad-hoc structure defines the variables needed for the computation of the expansion of the continued fraction of the square root of a non perfect square.

Note:

Since the denominators of the fraction are not needed in the CFRAC algorithm, they are not computed in this particular implementation.

Definition at line 69 of file sqrt_cont_frac.h.

4.7.2 Field Documentation

4.7.2.1 mpz_t struct_cont_frac_state_t::a

Current numerator of the continued fraction approximating `sqrtn`, given modulo `n`.

Definition at line 78 of file sqrt_cont_frac.h.

4.7.2.2 mpz_t struct_cont_frac_state_t::p

(Used in the computation of `a`).

Definition at line 87 of file sqrt_cont_frac.h.

4.7.2.3 mpz_t struct_cont_frac_state_t::q

One has: (-1)^(nsteps_performed) q = `a*a - b*b*n`

Definition at line 91 of file sqrt_cont_frac.h.

4.7.2.4 mpz_t struct_cont_frac_state_t::t

The current term of the expansion of the continued fraction.

Definition at line 95 of file sqrt_cont_frac.h.

4.7.2.5 mpz_t struct_cont_frac_state_t::sqrtn

The (truncated) square root of which to compute the continued fraction.

Definition at line 100 of file sqrt_cont_frac.h.

4.7.2.6 mpz_t struct_cont_frac_state_t::n

The non perfect square integer whose square root will be approximated by the computation of a continued fraction.

Definition at line 105 of file sqrt_cont_frac.h.

4.7.2.7 uint32_t struct_cont_frac_state_t::nsteps_performed

The number of non trivial terms of the continued fraction already computed.

Definition at line 110 of file sqrt_cont_frac.h.

The documentation for this struct was generated from the following file:

- sqrt_cont_frac.h

4.8 struct_ecm_params_t Struct Reference

Defines the variable parameters used in ECM.

`#include <lib/algo/include/ecm.h>`

Data Fields

- uint32_t b1
- uint32_t b2
- uint32_t ncurves

4.8.1 Detailed Description

Defines the variable parameters used in ECM.

This structure defines the set of the variable parameters used in the elliptic curve method (ECM).

Definition at line 67 of file ecm.h.

4.8.2 Field Documentation

4.8.2.1 uint32_t struct_ecm_params_t::b1

Bound used in the first phase of ECM.

Definition at line 71 of file ecm.h.

4.8.2.2 uint32_t struct_ecm_params_t::b2

Bound used in the second phase of ECM. If set to 0, no second phase is performed.

Warning:

> Due to a current limitation in the code, it is `required` than `b2` == 0 (no second phase) or `b2` > 105. Failure to assess such a condition will lead to unpredictable behaviour.

Definition at line 80 of file ecm.h.

4.8.2.3 uint32_t struct_ecm_params_t::ncurves

Number of curves to try before giving up the factorization when using the SINGLE_RUN factoring mode.

Definition at line 85 of file ecm.h.

The documentation for this struct was generated from the following file:

- ecm.h

4.9 struct_factoring_machine Struct Reference

Defines a structure to represent the logic behind all factorization algorithms.

```
#include <factoring_machine.h>
```

Data Fields

- mpz_t n
- factoring_mode_t mode
- void ∗ context
- void ∗ params
- ecode_t(∗ init_context_func)(struct struct_factoring_machine ∗const)
- ecode_t(∗ perform_algo_func)(struct struct_factoring_machine ∗const)
- ecode_t(∗ update_context_func)(struct struct_factoring_machine ∗const)
- ecode_t(∗ clear_context_func)(struct struct_factoring_machine ∗const)
- ecode_t(∗ recurse_func)(mpz_array_t ∗const, uint32_array_t ∗const, const mpz_t, factoring_mode_t)
- mpz_array_t ∗ factors
- uint32_array_t ∗ multis
- bool success

4.9.1 Detailed Description

Defines a structure to represent the logic behind all factorization algorithms.

tools/include/factoring_machine.h

This structure defines a set of data and functions to represent the logic behind all factorization algorithms. The idea is to be able to write down the factoring process' boilerplate once and for all so that actual factorization algorithm can use such a structure by merely "filling-in" the gaps.

Definition at line 118 of file factoring_machine.h.

4.9.2 Field Documentation

4.9.2.1 mpz_t struct_factoring_machine::n

The integer to factor.

Definition at line 122 of file factoring_machine.h.

4.9.2.2 factoring_mode_t struct_factoring_machine::mode

The factoring mode to use.

Definition at line 126 of file factoring_machine.h.

4.9.2.3 void∗ struct_factoring_machine::context

The context of the factorization algorithm. This is a pointer to an algorithm implementation dependant structure holding all variables, data, and functions needed by the implementation.

Definition at line 132 of file factoring_machine.h.

4.9.2.4 void* struct_factoring_machine::params

The parameters used by the factorization algorithm. This is a pointer to an algorithm implementation dependant structure holding the algorithm parameters needed by the implementation.

Definition at line 138 of file factoring_machine.h.

4.9.2.5 ecode_t(* struct_factoring_machine::init_context_func)(struct struct_factoring_machine *const)

A pointer to a function initializing the algorithm context.

Parameters:

(unnamed) A pointer to the `factoring_machine_t` used by the actual algorithm implementation.

4.9.2.6 ecode_t(* struct_factoring_machine::perform_algo_func)(struct struct_factoring_machine *const)

A pointer to a function performing the actual factorization stage of the factorization algorithm (by opposition to the initialization stage for example).

Parameters:

(unnamed) A pointer to the `factoring_machine_t` used by the actual algorithm implementation.

4.9.2.7 ecode_t(* struct_factoring_machine::update_context_func)(struct struct_factoring_machine *const)

A pointer to a function updating the context of the factorization algorithm. This function is responsible of the definition of the factorization strategy should something bad happens (e.g. what to do if no factors are found after the first run?).

Parameters:

(unnamed) A pointer to the `factoring_machine_t` used by the actual algorithm implementation.

4.9.2.8 ecode_t(* struct_factoring_machine::clear_context_func)(struct struct_factoring_machine *const)

A pointer to a function clearing all memory space used by the context.

Parameters:

(unnamed) A pointer to the `factoring_machine_t` used by the actual algorithm implementation.

4.9.2.9 ecode_t(* struct_factoring_machine::recurse_func)(mpz_array_t *const, uint32_array_t *const, const mpz_t, factoring_mode_t)

A pointer to the function to use to perform recursive factorization (i.e. factorization of the non-prime factors found).

Parameters:

(unnamed) A pointer to an `mpz_array_t` to hold the found factors.

(unnamed) A pointer to an `uint32_array_t` to hold the multiplicities.

(unnamed) The non-prime factor to factorize.

(unnamed) The `factoring_mode_t` to use.

4.9.2.10 mpz_array_t* struct_factoring_machine::factors

A pointer to a `mpz_array_t` to hold the found factors.

Definition at line 191 of file factoring_machine.h.

4.9.2.11 uint32_array_t* struct_factoring_machine::multis

A pointer to a `uint32_array_t` to hold the multiplicities of the found factors.

Definition at line 196 of file factoring_machine.h.

4.9.2.12 bool struct_factoring_machine::success

true if the algorithm succeeds. false otherwise.

Note:

The notion of success is given by the `factoring_mode_t` mode used and not on whether or not some factors are found.

Definition at line 204 of file factoring_machine.h.

The documentation for this struct was generated from the following file:

- factoring_machine.h

4.10 struct_factoring_program Struct Reference

Defines a structure to represent the logic behind all factorization programs.

`#include <factoring_program.h>`

Data Fields

- int argc
- char ** argv
- mpz_t n
- factoring_mode_t mode
- int verbose

- int timing
- uint32_t nprimes_tdiv
- uint32_t nfactors
- char ∗ algo_name
- void ∗ params
- void(∗ print_usage_func)(struct struct_factoring_program ∗const)
- void(∗ print_params_func)(struct struct_factoring_program ∗const)
- void(∗ process_args_func)(struct struct_factoring_program ∗const)
- ecode_t(∗ factoring_algo_func)(mpz_array_t ∗const factors, uint32_array_t ∗const multis, const mpz_t n, const void ∗const params, factoring_mode_t mode)
- void(∗ set_params_to_default_func)(struct struct_factoring_program ∗const)

4.10.1 Detailed Description

Defines a structure to represent the logic behind all factorization programs.

tools/include/factoring_program.h

This structure defines a set of data and functions to represent the logic behind all factorization programs. The idea is to be able to write down the actual factoring process once and for all so that actual factorization program can use such a structure by merely "filling-in" the gaps.

Definition at line 58 of file factoring_program.h.

4.10.2 Field Documentation

4.10.2.1 int struct_factoring_program::argc

Argument count as given by the "main" function.

Definition at line 62 of file factoring_program.h.

4.10.2.2 char∗∗ struct_factoring_program::argv

Argument values as given by the "main" function.

Definition at line 66 of file factoring_program.h.

4.10.2.3 mpz_t struct_factoring_program::n

Integer to factor.

Definition at line 70 of file factoring_program.h.

4.10.2.4 factoring_mode_t struct_factoring_program::mode

Factoring mode to use.

Definition at line 74 of file factoring_program.h.

4.10.2.5 int struct_factoring_program::verbose

Verbosity option. Should be either 1 (be verbose) or 0 (stay laconic).

Definition at line 78 of file factoring_program.h.

4.10.2.6 int struct_factoring_program::timing

Timing option. Should be either 1 (proceed with timings) or 0 (do not perform timings).

Definition at line 83 of file factoring_program.h.

4.10.2.7 uint32_t struct_factoring_program::nprimes_tdiv

Number of primes used in the trial division of the number to factor.

Definition at line 87 of file factoring_program.h.

4.10.2.8 uint32_t struct_factoring_program::nfactors

The maximum number of factors to collect (excluding the factors found in the trial division stage).

Definition at line 92 of file factoring_program.h.

4.10.2.9 char* struct_factoring_program::algo_name

Name of the factoring algorithm to use (preferably a short acronym).

Definition at line 96 of file factoring_program.h.

4.10.2.10 void* struct_factoring_program::params

A pointer to the parameters of the factoring algorithm to use.

Definition at line 100 of file factoring_program.h.

4.10.2.11 void(* struct_factoring_program::print_usage_func)(struct struct_factoring_program *const)

A pointer to a function printing the usage of the factoring program.

Parameters:

A pointer to the `factoring_program_t` used by the actual factoring program.

4.10.2.12 void(* struct_factoring_program::print_params_func)(struct struct_factoring_program *const)

A pointer to a function printing the values of the parameters used by the actual factoring program.

Parameters:

A pointer to the `factoring_program_t` used by the actual factoring program.

4.10.2.13 void(* struct_factoring_program::process_args_func)(struct struct_factoring_program *const)

A pointer to a function reading arguments on the command line and setting the parameters of the actual factoring program.

Parameters:

A pointer to the `factoring_program_t` used by the actual factoring program.

4.10.2.14 ecode_t(∗ struct_factoring_program::factoring_algo_func)(mpz_array_t ∗const factors, uint32_array_t ∗const multis, const mpz_t n, const void ∗const params, factoring_mode_t mode)

A pointer to a function implementing the factorization algorithm to use.

Parameters:

factors An `mpz_array_t` to hold the found factors.

factors A `uint32_array_t` to hold the found multiplicities.

n The integer to factor.

params A pointer to the parameters to be used in the factorization algorithm.

mode The factorization mode to use.

4.10.2.15 void(∗ struct_factoring_program::set_params_to_default_func)(struct struct_-factoring_program ∗const)

A pointer to a function setting the algorithm's parameters to some default values.

Parameters:

A pointer to the `factoring_program_t` used by the actual factoring program.

The documentation for this struct was generated from the following file:

- factoring_program.h

4.11 struct_fermat_params_t Struct Reference

Defines the variable parameters used in Fermat's algorithm (dummy structure).

`#include <lib/algo/include/fermat.h>`

Data Fields

- unsigned int _dummy_variable_

4.11.1 Detailed Description

Defines the variable parameters used in Fermat's algorithm (dummy structure).

This structure is intended to define the set of the variable parameters used in Fermat's algorithm.

Warning:

For the time being, this is a completely unused dummy structure which is kept only as a placeholder should the need for user parameters arise in future code revisions.

Definition at line 69 of file fermat.h.

4.11.2 Field Documentation

4.11.2.1 unsigned int struct_fermat_params_t::_dummy_variable_

Unused dummy variable.

Definition at line 73 of file fermat.h.

The documentation for this struct was generated from the following file:

- fermat.h

4.12 struct_hashtable_entry_t Struct Reference

The structure of a hashtable's entry.

`#include <lib/utils/include/hashtable.h>`

Data Fields

- void ∗ key
- void ∗ data

4.12.1 Detailed Description

The structure of a hashtable's entry.

This structure defines a hashtable entry, i.e. some data and its associated key.

Definition at line 110 of file hashtable.h.

4.12.2 Field Documentation

4.12.2.1 void∗ struct_hashtable_entry_t::key

Key associated to this entry, as a pointer to `void`.

Definition at line 114 of file hashtable.h.

4.12.2.2 void∗ struct_hashtable_entry_t::data

Data associated to this entry, as a pointer to `void`.

Definition at line 118 of file hashtable.h.

The documentation for this struct was generated from the following file:

- hashtable.h

4.13 struct_hashtable_t Struct Reference

A basic implementation of a hashtable.

`#include <lib/utils/include/hashtable.h>`

Data Fields

- uint32_t alloced
- uint32_t nentries
- linked_list_t ∗ buckets
- int(∗ cmp_func)(const void ∗const key_a, const void ∗const key_b)
- uint32_t(∗ hash_func)(const void ∗const key)

4.13.1 Detailed Description

A basic implementation of a hashtable.

This structure defines a simple generic hashtable. It can store any type of elements, provided that suitable comparison and hash functions exist for the type of the keys used.

This hashtable implementation uses a simple sequential search in a linked list to solve the collisions.

Warning:

Due to limitations in the current implementation, it is strongly advised to use *only* pointers to integers or strings as keys.

Definition at line 59 of file hashtable.h.

4.13.2 Field Documentation

4.13.2.1 uint32_t struct_hashtable_t::alloced

Number of allocated buckets (always a power of two).

Definition at line 63 of file hashtable.h.

4.13.2.2 uint32_t struct_hashtable_t::nentries

Current number of entries in the hashtable.

Definition at line 67 of file hashtable.h.

4.13.2.3 linked_list_t∗ struct_hashtable_t::buckets

Array of `linked_list_t` of size `alloced` used to store the hashtable's entries.

Definition at line 72 of file hashtable.h.

4.13.2.4 int(∗ struct_hashtable_t::cmp_func)(const void ∗const key_a, const void ∗const key_b)

Pointer to a comparison function for the keys stored in the hashtable. The function's signature should be: `int cmp_func(const void* const, const void* const)`. The function should take two keys of *identical* type passed as pointers to `void`.

For now, the only requirement if that the pointed function should return 0 if two identical keys are compared.

4.13.2.5 uint32_t(∗ struct_hashtable_t::hash_func)(const void ∗const key)

Pointer to the hash function used by the hashtable. The hash function's signature should be: `uint32_t hash_func(const void* const)`. The function should take a key passed as a pointer to `void`.

Needless to say, the real type of the key handled by this function should be the same as the one handled by the comparison function pointed by `cmp_int`.

The documentation for this struct was generated from the following file:

- hashtable.h

4.14 struct_int32_array_t Struct Reference

Defines an array of `int32`.

`#include <lib/utils/include/array.h>`

Data Fields

- uint32_t alloced
- uint32_t length
- int32_t ∗ data

4.14.1 Detailed Description

Defines an array of `int32`.

This structure defines a special kind of `int32` array which knows its current length and its allocated memory space.

Definition at line 621 of file array.h.

4.14.2 Field Documentation

4.14.2.1 uint32_t struct_int32_array_t::alloced

Memory space allocated for this array's `data` field, given as a multiple of `sizeof(int32_t)`. This is the maximum number of `int32_t` that the array can accommodate.

Definition at line 627 of file array.h.

4.14.2.2 uint32_t struct_int32_array_t::length

Current number of `int32_t` hold in the array pointed by the structure's `data` field.

Definition at line 632 of file array.h.

Referenced by is_in_sorted_int32_array().

4.14.2.3 int32_t∗ struct_int32_array_t::data

Array of `int32_t` whose size is given by the `alloced` field.

Definition at line 636 of file array.h.

The documentation for this struct was generated from the following file:

- array.h

4.15 struct_linked_list_node_t Struct Reference

A basic implementation of a linked list node.

`#include <linked_list.h>`

Data Fields

- struct struct_linked_list_node_t * next
- void * data

4.15.1 Detailed Description

A basic implementation of a linked list node.

lib/utils/include/linked_list.h This structure defines a simple linked list node, i.e. some data together with a pointer to the next node.

Definition at line 51 of file linked_list.h.

4.15.2 Field Documentation

4.15.2.1 struct struct_linked_list_node_t* struct_linked_list_node_t::next `[read]`

Pointer to the next node of the linked list.

Definition at line 55 of file linked_list.h.

4.15.2.2 void* struct_linked_list_node_t::data

Pointer to the node data.

Definition at line 59 of file linked_list.h.

The documentation for this struct was generated from the following file:

- linked_list.h

4.16 struct_linked_list_t Struct Reference

A basic implementation of a linked list.

`#include <lib/utils/include/linked_list.h>`

Data Fields

- struct struct_linked_list_node_t * head
- struct struct_linked_list_node_t * tail
- int(* cmp_func)(const void *const data_a, const void *const data_b)
- uint32_t length

4.16.1 Detailed Description

A basic implementation of a linked list.

This structure defines a simple linked list. It can store any type of elements, provided that there is a suitable comparison function for the type of the data used.

Definition at line 70 of file linked_list.h.

4.16.2 Field Documentation

4.16.2.1 struct struct_linked_list_node_t* struct_linked_list_t::head `[read]`

Pointer to the head of the linked list.

Definition at line 74 of file linked_list.h.

4.16.2.2 struct struct_linked_list_node_t* struct_linked_list_t::tail `[read]`

Pointer to the tail of the linked list.

Definition at line 78 of file linked_list.h.

4.16.2.3 int(* struct_linked_list_t::cmp_func)(const void *const data_a, const void *const data_b)

Pointer to the comparison function used to compare the node data. This function, which take two pointers to the data to compare, should return:

- 0 if the datas pointed by `data_a` and `data_b` are the same.
- A positive number if the data pointed by `data_a` is greater than the data pointed by `data_b`.
- A negative number if the data pointed by `data_a` is less than the data pointed by `data_b`.

4.16.2.4 uint32_t struct_linked_list_t::length

Number of nodes composing the linked list.

Definition at line 93 of file linked_list.h.

The documentation for this struct was generated from the following file:

- linked_list.h

4.17 struct_mpz_array_list_t Struct Reference

Defines a list of `mpz_array_t`.

`#include <x_array_list.h>`

Data Fields

- uint32_t alloced
- uint32_t length
- mpz_array_t ** data

4.17.1 Detailed Description

Defines a list of mpz_array_t.

lib/utils/include/x_array.h

This structure defines an array of pointers to mpz_array_t elements. Its name is a bit confusing since it is actually more of an array than a list strictly speaking.

Definition at line 170 of file x_array_list.h.

4.17.2 Field Documentation

4.17.2.1 uint32_t struct_mpz_array_list_t::alloced

This is the maximum number of mpz_array_t pointers that the array can accommodate.

Definition at line 175 of file x_array_list.h.

4.17.2.2 uint32_t struct_mpz_array_list_t::length

Current number of mpz_array_t pointers hold in the array pointed by the structure's data field.

Definition at line 180 of file x_array_list.h.

Referenced by add_entry_in_mpz_array_list().

4.17.2.3 mpz_array_t** struct_mpz_array_list_t::data

Array of pointers to mpz_array_t whose size is given by the alloced field.

Definition at line 185 of file x_array_list.h.

Referenced by add_entry_in_mpz_array_list().

The documentation for this struct was generated from the following file:

- x_array_list.h

4.18 struct_mpz_array_t Struct Reference

Defines an array of mpz_t elements from the GMP library.

`#include <lib/utils/include/array.h>`

Data Fields

- uint32_t alloced
- uint32_t length
- mpz_t * data

4.18.1 Detailed Description

Defines an array of mpz_t elements from the GMP library.

This structure defines a special kind of mpz array which knows its current length and its allocated memory space.

Definition at line 857 of file array.h.

4.18.2 Field Documentation

4.18.2.1 uint32_t struct_mpz_array_t::alloced

Memory space allocated for this array's `data` field, given as a multiple of `sizeof(mpz_t)`. This is the maximum number of `mpz_t` elements that the array can accommodate.

Definition at line 863 of file array.h.

4.18.2.2 uint32_t struct_mpz_array_t::length

Current number of "useful" `mpz_t` elements hold in the array pointed by the structure's `data` field.

Warning:

Prior to version 1.2, the `length` field also indicated which positions had been `mpz_init'ed` in the `data` field. Since version 1.2 this is no longer true. Now all positions in the `data` array are `mpz_init'ed` and `length` only gives which part of the array is useful from the client standpoint.

Definition at line 874 of file array.h.

Referenced by is_in_sorted_mpz_array().

4.18.2.3 mpz_t* struct_mpz_array_t::data

Array of `mpz_t` elements whose size is given by the `alloced` field.

Definition at line 879 of file array.h.

The documentation for this struct was generated from the following file:

- array.h

4.19 struct_mpz_pair_t Struct Reference

A pair of `mpz_t` integers.

`#include <lib/utils/include/gmp_utils.h>`

Data Fields

- mpz_t x
- mpz_t y

4.19.1 Detailed Description

A pair of `mpz_t` integers.

This very simple structure defines a pair of `mpz_t` integers.

Definition at line 50 of file gmp_utils.h.

4.19.2 Field Documentation

4.19.2.1 mpz_t struct_mpz_pair_t::x

The first `mpz_t` integer of the pair.

Definition at line 54 of file gmp_utils.h.

Referenced by clear_mpz_pair(), and init_mpz_pair().

4.19.2.2 mpz_t struct_mpz_pair_t::y

The second `mpz_t` integer of the pair.

Definition at line 58 of file gmp_utils.h.

Referenced by clear_mpz_pair(), and init_mpz_pair().

The documentation for this struct was generated from the following file:

- gmp_utils.h

4.20 struct_mult_data_t Struct Reference

Ad hoc structure used in the computation of the multiplier to use.

`#include <lib/utils/include/funcs.h>`

Data Fields

- uint32_t multiplier
- uint32_t count
- double sum_inv_pi

4.20.1 Detailed Description

Ad hoc structure used in the computation of the multiplier to use.

This ad-hoc structure defines several variables needed in the determination of the best multiplier, as described by M. A. Morrison and J. Brillhart in the remark 5.3 of the paper "A Method of Factoring and the Factorization of F_7" (Mathematics of Computation, vol 29, #129, Jan 1975, pages 183-205).

Definition at line 87 of file funcs.h.

4.20.2 Field Documentation

4.20.2.1 uint32_t struct_mult_data_t::multiplier

The multiplier to use in the factoring algorithms.

Definition at line 91 of file funcs.h.

4.20.2.2 uint32_t struct_mult_data_t::count

The number of primes p_i less than or equal to the `MAX_IPRIME_IN_MULT_CALC`-th prime for which the legendre symbol (`k*N`/p_i) is 0 or 1 *and* for which either (`k*N`/3) or (`k*N`/5) (or both) is 0 or 1.

Definition at line 99 of file funcs.h.

4.20.2.3 double struct_mult_data_t::sum_inv_pi

The sum of 1/p_i where {p_i} is the set of primes previously described.

Definition at line 104 of file funcs.h.

The documentation for this struct was generated from the following file:

- funcs.h

4.21 struct_siqs_params_t Struct Reference

Defines the variable parameters used in the SIQS algorithm.

`#include <lib/algo/include/siqs.h>`

Data Fields

- uint32_t sieve_half_width
- uint32_t nprimes_in_base
- uint32_t threshold
- uint32_t nprimes_tdiv
- uint32_t nrelations
- linalg_method_t linalg_method
- bool use_large_primes

4.21.1 Detailed Description

Defines the variable parameters used in the SIQS algorithm.

This structure defines the set of the variable parameters used in the SIQS algorithm.

Definition at line 89 of file siqs.h.

4.21.2 Field Documentation

4.21.2.1 uint32_t struct_siqs_params_t::sieve_half_width

Sieve's half-width, i.e. the SIQS will sieve in the interval `[-sieve_half_width, sieve_half_-width]`.

Definition at line 94 of file siqs.h.

4.21.2.2 uint32_t struct_siqs_params_t::nprimes_in_base

Number of prime numbers composing the factor base on which to factor the residues.

Definition at line 99 of file siqs.h.

4.21.2.3 uint32_t struct_siqs_params_t::threshold

Sieve threshold.

Definition at line 103 of file siqs.h.

4.21.2.4 uint32_t struct_siqs_params_t::nprimes_tdiv

Number of the first primes to use in the trial division of the residues.

Warning:

`nprimes_tdiv` should be greater than or equal to 1.

Definition at line 111 of file siqs.h.

4.21.2.5 uint32_t struct_siqs_params_t::nrelations

Number of congruence relations to find before attempting the factorization of the large integer.

Definition at line 116 of file siqs.h.

4.21.2.6 linalg_method_t struct_siqs_params_t::linalg_method

Linear system resolution method to use.

Definition at line 120 of file siqs.h.

4.21.2.7 bool struct_siqs_params_t::use_large_primes

True if we use the large prime variation. False otherwise.

Definition at line 125 of file siqs.h.

The documentation for this struct was generated from the following file:

- siqs.h

4.22 struct_siqs_poly_t Struct Reference

Defines polynomials used by SIQS.

`#include <lib/utils/include/siqs_poly.h>`

Data Fields

- mpz_t a
- mpz_t b
- mpz_t c
- mpz_t n
- uint32_t loga
- uint32_t logb
- uint32_t logc
- approximer_t * approximer
- uint32_array_t * factor_base
- uint32_array_t * sqrtm_pi
- int32_array_t * sol1
- int32_array_t * sol2
- mpz_array_t * Bl
- uint32_t ** Bainv2

- uint32_t npolys
- uint32_t polyno
- uint32_t nprimes_in_a
- uint32_t * idx_of_a
- uint32_t nfullpolyinit

4.22.1 Detailed Description

Defines polynomials used by SIQS.

This structure defines the polynomials used by SIQS together with all its associated data.

A polynomial **P** is given by **a**X^2 + **b**X + **c**.

Definition at line 53 of file siqs_poly.h.

4.22.2 Field Documentation

4.22.2.1 mpz_t struct_siqs_poly_t::a

The **a** coefficient

Definition at line 57 of file siqs_poly.h.

4.22.2.2 mpz_t struct_siqs_poly_t::b

The **b** coefficient

Definition at line 61 of file siqs_poly.h.

4.22.2.3 mpz_t struct_siqs_poly_t::c

The **c** coefficient

Definition at line 65 of file siqs_poly.h.

4.22.2.4 mpz_t struct_siqs_poly_t::n

The number to factor (or a small multiple if a multiplier is used)

Definition at line 69 of file siqs_poly.h.

4.22.2.5 uint32_t struct_siqs_poly_t::loga

Logarithm of **a** in base 2

Definition at line 73 of file siqs_poly.h.

4.22.2.6 uint32_t struct_siqs_poly_t::logb

Logarithm of **b** in base 2

Definition at line 77 of file siqs_poly.h.

4.22.2.7 uint32_t struct_siqs_poly_t::logc

Logarithm of **c** in base 2

Definition at line 81 of file siqs_poly.h.

4.22.2.8 approximer_t∗ struct_siqs_poly_t::approximer

An **approximer_t** used to determine suitable values of the **a** coefficient

Definition at line 86 of file siqs_poly.h.

4.22.2.9 uint32_array_t∗ struct_siqs_poly_t::factor_base

The factor base

Definition at line 90 of file siqs_poly.h.

4.22.2.10 uint32_array_t∗ struct_siqs_poly_t::sqrtm_pi

The modular square roots of `n` modulo each primes in the factor base

Definition at line 94 of file siqs_poly.h.

4.22.2.11 int32_array_t∗ struct_siqs_poly_t::sol1

The first solution to the equation **P**(x) = 0 mod **pi** for each prime **pi** in the factor base

Definition at line 99 of file siqs_poly.h.

4.22.2.12 int32_array_t∗ struct_siqs_poly_t::sol2

The second solution to the equation **P**(x) = 0 mod **pi** for each prime **pi** in the factor base

Definition at line 104 of file siqs_poly.h.

4.22.2.13 mpz_array_t∗ struct_siqs_poly_t::Bl

For all **l** in [0, `npolys`-1] solutions to

- `Bl`^2 = `n` mod **q_l**
- `Bl` = 0 mod **q_j** for all **j** != **l**

where **q_i** are the primes from the factor base such that **a** = **q_0** x **q_1** x ... x **q_**`npolys`

Definition at line 114 of file siqs_poly.h.

4.22.2.14 uint32_t∗∗ struct_siqs_poly_t::Bainv2

`Bainv2[i][j]` = 2 x `Bl[i]` x inv(`a`) mod **pj** for **i** in [0, `npolys`-1] and **pj** in the factor base

Definition at line 119 of file siqs_poly.h.

4.22.2.15 uint32_t struct_siqs_poly_t::npolys

Number of different polynomials having the same `a` coefficient

Definition at line 123 of file siqs_poly.h.

4.22.2.16 uint32_t struct_siqs_poly_t::polyno

Current polynomial number (from 1 to `npolys`)

Definition at line 127 of file siqs_poly.h.

4.22.2.17 uint32_t struct_siqs_poly_t::nprimes_in_a

Number of primes in the prime decomposition of `a`

Definition at line 131 of file siqs_poly.h.

4.22.2.18 uint32_t* struct_siqs_poly_t::idx_of_a

Indexes (in the factor base) of the (prime) factors of `a`

Definition at line 135 of file siqs_poly.h.

4.22.2.19 uint32_t struct_siqs_poly_t::nfullpolyinit

Number of "full" polynomial initializations performed

Definition at line 139 of file siqs_poly.h.

The documentation for this struct was generated from the following file:

- siqs_poly.h

4.23 struct_siqs_sieve_t Struct Reference

Defines the sieve used by SIQS.

`#include <lib/utils/include/siqs_sieve.h>`

Data Fields

- uint32_t nchunks
- uint32_t chunk_size
- int32_t next_chunkno_to_fill
- uint32_t scan_begin
- uint32_t threshold
- byte_array_t * log_primes
- byte_array_t * sieve
- siqs_poly_t * poly
- int32_array_t * sol1
- int32_array_t * sol2
- uint32_t endlast
- bool use_buckets
- uint32_t nprimes_no_buckets
- uint32_t buckets_first_prime
- buckets_t * buckets_positive
- buckets_t * buckets_negative
- stopwatch_t init_poly_timer
- stopwatch_t fill_timer
- stopwatch_t scan_timer

4.23.1 Detailed Description

Defines the sieve used by SIQS.

This structure defines the sieve used by SIQS together with all its associated data.

Definition at line 122 of file siqs_sieve.h.

4.23.2 Field Documentation

4.23.2.1 uint32_t struct_siqs_sieve_t::nchunks

Number of blocks to sieve on each side of zero (`nchunks` for positive **x** and `nchunks` for negative **x**)

Definition at line 127 of file siqs_sieve.h.

4.23.2.2 uint32_t struct_siqs_sieve_t::chunk_size

Size of a sieve chunk. The total sieving interval is thus given by `2 * chunk_size * nchunks`

Definition at line 132 of file siqs_sieve.h.

4.23.2.3 int32_t struct_siqs_sieve_t::next_chunkno_to_fill

The number of the next sieve chunk to fill (in [`nchunks`, 0[U]0, `nchunks`])

Definition at line 137 of file siqs_sieve.h.

4.23.2.4 uint32_t struct_siqs_sieve_t::scan_begin

The position to start scanning in the next sieve chunk to scan

Definition at line 141 of file siqs_sieve.h.

4.23.2.5 uint32_t struct_siqs_sieve_t::threshold

The sieve threshold. An **xi** will be tested for smoothness if `sieve[`**xi**`]` < `threshold`

Definition at line 146 of file siqs_sieve.h.

4.23.2.6 byte_array_t* struct_siqs_sieve_t::log_primes

Approximated logarithm (in base 2) of the primes in the factor base

Definition at line 150 of file siqs_sieve.h.

4.23.2.7 byte_array_t* struct_siqs_sieve_t::sieve

The actual sieve array, of size `chunk_size`

Definition at line 154 of file siqs_sieve.h.

4.23.2.8 siqs_poly_t* struct_siqs_sieve_t::poly

The SIQS polynomial

Definition at line 158 of file siqs_sieve.h.

4.23.2.9 int32_array_t∗ struct_siqs_sieve_t::sol1

The first solution to the equation **P**(x) = 0 mod **pi** for each prime **pi** in the factor base

Definition at line 163 of file siqs_sieve.h.

4.23.2.10 int32_array_t∗ struct_siqs_sieve_t::sol2

The first solution to the equation **P**(x) = 0 mod **pi** for each prime **pi** in the factor base

Definition at line 168 of file siqs_sieve.h.

4.23.2.11 uint32_t struct_siqs_sieve_t::endlast

The index of the last position to sieve in the last sieve chunk (on either side)

Definition at line 174 of file siqs_sieve.h.

4.23.2.12 bool struct_siqs_sieve_t::use_buckets

Should we use a bucket sieving for the largest primes in the base?

Definition at line 180 of file siqs_sieve.h.

4.23.2.13 uint32_t struct_siqs_sieve_t::nprimes_no_buckets

Index (in the factor base) of the largest prime for which a standard sieving procedure is use

Definition at line 185 of file siqs_sieve.h.

4.23.2.14 uint32_t struct_siqs_sieve_t::buckets_first_prime

Index (in the factor base) of the smallest prime for which the bucket sieving procedure is use

Definition at line 190 of file siqs_sieve.h.

4.23.2.15 buckets_t∗ struct_siqs_sieve_t::buckets_positive

Buckets for the positive **x**'s

Definition at line 194 of file siqs_sieve.h.

4.23.2.16 buckets_t∗ struct_siqs_sieve_t::buckets_negative

Buckets for the negative **x**'s

Definition at line 198 of file siqs_sieve.h.

4.23.2.17 stopwatch_t struct_siqs_sieve_t::init_poly_timer

Additional stopwatch (to get timing for the polynomial initializations)

Definition at line 202 of file siqs_sieve.h.

4.23.2.18 stopwatch_t struct_siqs_sieve_t::fill_timer

Additional stopwatch (to get timing for the sieve filling)

Definition at line 206 of file siqs_sieve.h.

4.23.2.19 stopwatch_t struct_siqs_sieve_t::scan_timer

Additional stopwatch (to get timing for the sieve scanning)

Definition at line 210 of file siqs_sieve.h.

The documentation for this struct was generated from the following file:

- siqs_sieve.h

4.24 struct_smooth_filter_t Struct Reference

Structure grouping variables needed for multi-step early abort strategy.

```
#include <smooth_filter.h>
```

Data Fields

- mpz_ptr n
- mpz_ptr kn
- smooth_filter_method_t method
- unsigned short int nsteps
- unsigned long int batch_size
- unsigned long int base_size
- uint32_array_t * complete_base
- uint32_array_t * factor_base [MAX_NSTEPS]
- mpz_array_t * candidate_xi
- mpz_array_t * candidate_yi
- mpz_array_t * candidate_ai
- mpz_array_t * accepted_xi
- mpz_array_t * accepted_yi
- mpz_array_t * accepted_ai
- mpz_array_t * filtered_xi [MAX_NSTEPS]
- mpz_array_t * filtered_yi [MAX_NSTEPS]
- mpz_array_t * filtered_ai [MAX_NSTEPS]
- mpz_array_t * cofactors [MAX_NSTEPS]
- mpz_t bounds [MAX_NSTEPS]
- mpz_t prod_pj [MAX_NSTEPS+1]
- hashtable_t * htable
- bool use_large_primes
- bool use_siqs_variant

4.24.1 Detailed Description

Structure grouping variables needed for multi-step early abort strategy.

lib/utils/include/smooth_filter.h

This structure and its associated functions implement the multi-step early abort strategy in a way reminiscent of Pomerance's suggestion in "Analysis and Comparison of Some Integer Factoring Algorithm" with the exception that the smoothness tests are performed by batch instead of trial division.

C. Pomerance, *Analysis and Comparison of Some Integer Factoring Algorithm*, in Mathematical Centre Tracts 154.

Definition at line 171 of file smooth_filter.h.

4.24.2 Field Documentation

4.24.2.1 mpz_ptr struct_smooth_filter_t::n

The number to factor.

Definition at line 175 of file smooth_filter.h.

4.24.2.2 mpz_ptr struct_smooth_filter_t::kn

The number to factor multiplied by a multiplier.

Definition at line 179 of file smooth_filter.h.

4.24.2.3 smooth_filter_method_t struct_smooth_filter_t::method

The method to use for smooth residue detection.

Definition at line 183 of file smooth_filter.h.

4.24.2.4 unsigned short int struct_smooth_filter_t::nsteps

The number of steps in the early abort strategy. If `nsteps == 0` no early abort is performed.

Note:

`nsteps` should be less than or equal to `MAX_NSTEPS`.

Definition at line 190 of file smooth_filter.h.

4.24.2.5 unsigned long int struct_smooth_filter_t::batch_size

Number of relations to accumulate before testing for smoothness.

Definition at line 194 of file smooth_filter.h.

4.24.2.6 unsigned long int struct_smooth_filter_t::base_size

Size of the complete factor base.

Definition at line 198 of file smooth_filter.h.

4.24.2.7 uint32_array_t* struct_smooth_filter_t::complete_base

Pointer to the complete factor base.

Definition at line 202 of file smooth_filter.h.

4.24.2.8 uint32_array_t* struct_smooth_filter_t::factor_base[MAX_NSTEPS]

Array giving the factor base to use at each step if we use the early-abort strategy.

Definition at line 207 of file smooth_filter.h.

4.24.2.9 mpz_array_t* struct_smooth_filter_t::candidate_xi

The candidate x's. Together with candidate_yi, stores candidate relations verifying x^2 (mod kn) == y (mod kn)

Note:

See candidate_ai if use_siqs_batch_variant is true. In that case the relations become x^2 (mod kn) == y * a (mod kn).

Definition at line 216 of file smooth_filter.h.

4.24.2.10 mpz_array_t* struct_smooth_filter_t::candidate_yi

The candidate y's. Together with candidate_xi, stores candidate relations verifying x^2 (mod kn) == y (mod kn)

Note:

See candidate_ai if use_siqs_batch_variant is true. In that case the relations become x^2 (mod kn) == y * a (mod kn).

Definition at line 225 of file smooth_filter.h.

4.24.2.11 mpz_array_t* struct_smooth_filter_t::candidate_ai

Used only if use_siqs_batch_variant is true.

candidate_ai stores the a's (i.e. the value of the first parameter of the SIQS polynomial used). Together with candidate_xi and candidate_yi, stores candidate relations verifying x^2 (mod kn) == y * a (mod kn).

Definition at line 234 of file smooth_filter.h.

4.24.2.12 mpz_array_t* struct_smooth_filter_t::accepted_xi

The accepted x's. Together with accepted_yi, stores 'good' relations verifying x^2 (mod kn) == y (mod kn) with y smooth over the factor base.

Note:

See accepted_ai if use_siqs_batch_variant is true. In that case the relations become x^2 (mod kn) == y * a (mod kn).

Definition at line 244 of file smooth_filter.h.

4.24.2.13 mpz_array_t* struct_smooth_filter_t::accepted_yi

The accepted y's. Together with accepted_xi, stores 'good' relations verifying x^2 (mod kn) == y (mod kn) with y smooth over the factor base.

Note:

See accepted_ai if use_siqs_batch_variant is true. In that case the relations become x^2 (mod kn) == y * a (mod kn).

Definition at line 254 of file smooth_filter.h.

4.24.2.14 mpz_array_t* struct_smooth_filter_t::accepted_ai

Used only if `use_siqs_batch_variant` is true.

The accepted a's (i.e. the value of the first parameter of the SIQS polynomial used). Together with `accepted_xi`, and `accepted_yi`, stores candidate relations verifying `x^2` (mod `kn`) == `y * a` (mod `kn`) with `y` smooth over the factor base.

Definition at line 264 of file smooth_filter.h.

4.24.2.15 mpz_array_t* struct_smooth_filter_t::filtered_xi[MAX_NSTEPS]

The `filtered_xi`[s] array gives the filtered x's after `s` early abort steps. Together with `filtered_yi`[s] and `cofactors`[s], stores relations verifying `x^2` (mod `kn`) == `y * cofactor` (mod `kn`) with `cofactor` smooth over the base composed by the partial_factor bases used in the early abort steps up to (and including) the `s-th` one, and with `y` less than `bounds`[s].

Note:

See `filtered_ai` if `use_siqs_batch_variant` is true. In that case the relations become `x^2` (mod `kn`) == `y * a * cofactor` (mod `kn`).

Definition at line 278 of file smooth_filter.h.

4.24.2.16 mpz_array_t* struct_smooth_filter_t::filtered_yi[MAX_NSTEPS]

The `filtered_yi`[s] array gives the filtered x's after `s` early abort steps. See filtered_xi.

Definition at line 283 of file smooth_filter.h.

4.24.2.17 mpz_array_t* struct_smooth_filter_t::filtered_ai[MAX_NSTEPS]

The `filtered_ai`[s] array gives the filtered a's after `s` early abort steps. See filtered_xi.

Definition at line 288 of file smooth_filter.h.

4.24.2.18 mpz_array_t* struct_smooth_filter_t::cofactors[MAX_NSTEPS]

The `cofactor`[s] array gives the cofactors of the y's in `filtered_yi`[s] after `s` early abort steps. See filtered_xi.

Definition at line 293 of file smooth_filter.h.

4.24.2.19 mpz_t struct_smooth_filter_t::bounds[MAX_NSTEPS]

`bounds`[s] is the upper bound of the `s-th` early abort step. A relation will not pass this step if it has a 'y' greater than this bound.

Definition at line 299 of file smooth_filter.h.

4.24.2.20 mpz_t struct_smooth_filter_t::prod_pj[MAX_NSTEPS+1]

`prod_pj`[s] is the product of all the elements in the partial factor base at the `s-th` early abort step.

Definition at line 304 of file smooth_filter.h.

4.24.2.21 hashtable_t* struct_smooth_filter_t::htable

Hashtable used if the large prime variation is used. Can be `NULL` if the variation is not used.

Definition at line 309 of file smooth_filter.h.

4.24.2.22 bool struct_smooth_filter_t::use_large_primes

true if and only if we are using the large prime variation.

Definition at line 313 of file smooth_filter.h.

4.24.2.23 bool struct_smooth_filter_t::use_siqs_variant

true if and only if we are using this `smooth_filter_t` with SIQS.

Definition at line 318 of file smooth_filter.h.

The documentation for this struct was generated from the following file:

- smooth_filter.h

4.25 struct_squfof_params_t Struct Reference

Defines the variable parameters used in the SQUFOF algorithm (dummy structure).

```
#include <lib/algo/include/squfof.h>
```

Data Fields

- unsigned int _dummy_variable_

4.25.1 Detailed Description

Defines the variable parameters used in the SQUFOF algorithm (dummy structure).

This structure is intended to define the set of the variable parameters used in the SQUFOF algorithm.

Warning:

For the time being, this is a completely unused dummy structure which is kept only as a placeholder should the need for user parameters arise in future code revisions.

Definition at line 79 of file squfof.h.

4.25.2 Field Documentation

4.25.2.1 unsigned int struct_squfof_params_t::_dummy_variable_

Unused dummy variable.

Definition at line 83 of file squfof.h.

The documentation for this struct was generated from the following file:

- squfof.h

4.26 struct_stopwatch_t Struct Reference

Defines a very basic stopwatch-like timer.

`#include <lib/utils/include/stopwatch.h>`

Data Fields

- struct rusage rsg [1]
- uint64_t started_usec
- uint64_t elapsed_usec
- bool is_running

4.26.1 Detailed Description

Defines a very basic stopwatch-like timer.

This structure defines very basic stopwatch-like timer based on the `rusage` structure.

Definition at line 55 of file stopwatch.h.

4.26.2 Field Documentation

4.26.2.1 struct rusage struct_stopwatch_t::rsg[1] `[read]`

An `rusage` structure.

Definition at line 59 of file stopwatch.h.

4.26.2.2 uint64_t struct_stopwatch_t::started_usec

The time (in microseconds) when the stopwatch started.

Definition at line 63 of file stopwatch.h.

4.26.2.3 uint64_t struct_stopwatch_t::elapsed_usec

The elapsed time accumulated (in microseconds).

Definition at line 67 of file stopwatch.h.

4.26.2.4 bool struct_stopwatch_t::is_running

`true` iif the stopwatch is currently running.

Definition at line 71 of file stopwatch.h.

The documentation for this struct was generated from the following file:

- stopwatch.h

4.27 struct_uint32_array_list_t Struct Reference

Defines a list of `uint32_array_t`.

`#include <x_array_list.h>`

Data Fields

- uint32_t alloced
- uint32_t length
- uint32_array_t ** data

4.27.1 Detailed Description

Defines a list of `uint32_array_t`.

lib/utils/include/x_array.h

This structure defines an array of pointers to `uint32_array_t` elements. Its name is a bit confusing since it is actually more of an array than a list strictly speaking.

Definition at line 62 of file x_array_list.h.

4.27.2 Field Documentation

4.27.2.1 uint32_t struct_uint32_array_list_t::alloced

This is the maximum number of `uint32_array_t` pointers that the array can accommodate.

Definition at line 67 of file x_array_list.h.

Referenced by add_entry_in_uint32_array_list().

4.27.2.2 uint32_t struct_uint32_array_list_t::length

Current number of `uint32_array_t` pointers hold in the array pointed by the structure's `data` field.

Definition at line 72 of file x_array_list.h.

Referenced by add_entry_in_uint32_array_list().

4.27.2.3 uint32_array_t** struct_uint32_array_list_t::data

Array of pointers to `uint32_array_t` whose size is given by the `alloced` field.

Definition at line 77 of file x_array_list.h.

Referenced by add_entry_in_uint32_array_list().

The documentation for this struct was generated from the following file:

- x_array_list.h

4.28 struct_uint32_array_t Struct Reference

Defines an array of `uint32`.

`#include <lib/utils/include/array.h>`

Data Fields

- uint32_t alloced
- uint32_t length

- uint32_t * data

4.28.1 Detailed Description

Defines an array of `uint32`.

This structure defines a special kind of `uint32` array which knows its current length and its allocated memory space.

Definition at line 361 of file array.h.

4.28.2 Field Documentation

4.28.2.1 uint32_t struct_uint32_array_t::alloced

Memory space allocated for this array's `data` field, given as a multiple of `sizeof(uint32_t)`. This is the maximum number of `uint32_t` that the array can accommodate.

Definition at line 367 of file array.h.

4.28.2.2 uint32_t struct_uint32_array_t::length

Current number of `uint32_t` hold in the array pointed by the structure's `data` field.

Definition at line 372 of file array.h.

Referenced by is_in_sorted_uint32_array().

4.28.2.3 uint32_t* struct_uint32_array_t::data

Array of `uint32_t` whose size is given by the `alloced` field.

Definition at line 376 of file array.h.

The documentation for this struct was generated from the following file:

- array.h

4.29 struct_uint32_tuple_t Struct Reference

Defines a tuple of integers together with a sorting key.

`#include <lib/utils/include/approx.h>`

Data Fields

- uint32_t tuple [MAX_NPRIMES_IN_TUPLE]
- float tlog

4.29.1 Detailed Description

Defines a tuple of integers together with a sorting key.

This structure defines a tuple of up to `MAX_NPRIMES_IN_TUPLE` integers together with a sorting key given as a float.

Definition at line 90 of file approx.h.

4.29.2 Field Documentation

4.29.2.1 uint32_t struct_uint32_tuple_t::tuple[MAX_NPRIMES_IN_TUPLE]

The value of the tuple.

Definition at line 94 of file approx.h.

4.29.2.2 float struct_uint32_tuple_t::tlog

Used as a key to sort tuples.

Definition at line 98 of file approx.h.

The documentation for this struct was generated from the following file:

- approx.h

5 File Documentation

5.1 approx.h File Reference

Approximate a value by multiplying some numbers from a pool.

```
#include <stdint.h>
#include <stdbool.h>
#include "exit_codes.h"
#include "array.h"
```

Data Structures

- struct struct_uint32_tuple_t

  Defines a tuple of integers together with a sorting key.

- struct struct_approximer_t

  Structure used to find number approximation.

Defines

- #define _TIFA_APPROX_H_
- #define MAX_NPRIMES_IN_TUPLE 3

Typedefs

- typedef struct struct_uint32_tuple_t uint32_tuple_t

  Equivalent to `struct_uint32_tuple_t`.

Functions

- approximer_t ∗ alloc_approximer (mpz_t target, uint32_array_t ∗const facpool, uint32_t nfactors)

  Allocates and returns a new `approximer_t`.
- void free_approximer (approximer_t ∗aximer)

  Frees a previously allocated `approximer_t`.
- void random_approximation (approximer_t ∗const aximer, mpz_t approxed, uint32_t ∗indexes)

  Generates a "random" approximation.

5.1.1 Detailed Description

Approximate a value by multiplying some numbers from a pool.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

This provides a structure `approximer_t` and associated functions that can be used to approximate a target value by multiplying a given number of factors from a given base. Each factor is allowed to appear only once in the decomposition of the approximation on the given base.

This is used in TIFA's SIQS implementation where we need to find a polynomial coefficient of a given order from the product of some prime numbers.

The strategy used to reach a good approximation is adapted from the Carrier-Wagstaff method.

- Let **t** be the target number to be obtained by multiplying **n** distinct numbers from a given base **B**=[p_-1, p_2, p_3, ...] (with p_i < p_{i+1}).
- Set **m** so that **B[m]** is roughly equal to the **n**th -root of **t**. Factors will be chosen from the set **B[m-d]**, ..., **B[m+d]** with **d** suitably chosen.
- The approximation **a** of the target **t** is obtained by choosing a random combination of (**n-i**) factors completed by **i** other factors so as to obtain the best approximation as possible.
- Since all numbers must be distinct the (**n-i**) randomly chosen factors are picked from the set of **B[j]** with **j** odd and the remaining **i** factors from the set of **B[k]** with **k** even.
- The best remaining **i** factors are obtained by precomputing and sorting all combinations of **i** factors, then picking the most suitable one.

Definition in file approx.h.

5.1.2 Define Documentation

5.1.2.1 #define _TIFA_APPROX_H_

Standard include guard.

Definition at line 65 of file approx.h.

5.1.2.2 #define MAX_NPRIMES_IN_TUPLE 3

Maximum number of factors in the sorted factor combinations.

Definition at line 81 of file approx.h.

5.1.3 Function Documentation

5.1.3.1 approximer_t∗ alloc_approximer (mpz_t *target*, uint32_array_t ∗const *facpool*, uint32_t *nfactors*)

Allocates and returns a new `approximer_t`.

Parameters:

target the target number to approximate.

facpool the pool of available factors.

nfactors the number of factors from `facpool` to use.

Returns:

A pointer to the newly allocated `approximer_t`.

5.1.3.2 void free_approximer (approximer_t ∗ *aximer*)

Frees a previously allocated `approximer_t`.

Frees all memory used by the pointed `approximer_t` and then frees the `aximer` pointer.

Warning:

Do not call `free(aximer)` in client code after a call to `free_approximer(aximer)`: it would result in an error.

Parameters:

aximer the `approximer_t` to free.

5.1.3.3 void random_approximation (approximer_t ∗const *aximer*, mpz_t *approxed*, uint32_t ∗ *indexes*)

Generates a "random" approximation.

Parameters:

← ***aximer*** the `approximer_t` to use.

→ ***approxed*** the approximation obtained.

→ ***indexes*** the (sorted) indexes of the factors making up the approximation.

5.2 array.h File Reference

Higher level arrays and associated functions.

```
#include "tifa_config.h"
#include <inttypes.h>
#include <stdbool.h>
#include <gmp.h>
#include "bitstring_t.h"
```

Data Structures

- struct struct_byte_array_t

  Defines an array of bytes.

- struct struct_uint32_array_t

  Defines an array of `uint32`.

- struct struct_int32_array_t

  Defines an array of `int32`.

- struct struct_mpz_array_t

  Defines an array of `mpz_t` *elements from the GMP library.*

- struct struct_binary_array_t

  Defines an array of bits.

Defines

- #define _TIFA_ARRAY_H_
- #define ELONGATION 16
- #define NOT_IN_ARRAY UINT32_MAX
- #define ARRAY_IS_FULL(ARRAY_PTR) ((ARRAY_PTR) → length == (ARRAY_PTR) → alloced)
- #define reset_byte_array(ARRAY) do {(ARRAY) → length = 0;} while (0)

  Resets a `byte_array_t`.

- #define reset_uint32_array(ARRAY) do {(ARRAY) → length = 0;} while (0)

  Resets a `uint32_array_t`.

- #define reset_int32_array(ARRAY) do {(ARRAY) → length = 0;} while (0)

  Resets an `int32_array_t`.

- #define reset_mpz_array(ARRAY) do {(ARRAY) → length = 0;} while (0)

  Resets an `mpz_array_t`.

- #define reset_binary_array(ARRAY) do {(ARRAY) → length = 0;} while (0)

  Resets a `binary_array_t`.

Typedefs

- typedef struct struct_byte_array_t byte_array_t

  Equivalent to `struct struct_byte_array_t`.

- typedef struct struct_uint32_array_t uint32_array_t

  Equivalent to `struct struct_uint32_array_t`.

- typedef struct struct_int32_array_t int32_array_t

  Equivalent to `struct struct_int32_array_t`.

- typedef struct struct_mpz_array_t mpz_array_t

  Equivalent to `struct struct_mpz_array_t`.

- typedef struct struct_binary_array_t binary_array_t

  Equivalent to `struct struct_binary_array_t`.

Functions

- byte_array_t ∗ alloc_byte_array (uint32_t length)

  Allocates and returns a new `byte_array_t`.

- void free_byte_array (byte_array_t ∗array)

  Frees a `byte_array_t`.

- void resize_byte_array (byte_array_t ∗const array, uint32_t alloced)

  Resizes the allocated memory of a `byte_array_t`.

- void append_byte_to_array (byte_array_t ∗array, const unsigned char to_append)

  Appends a `uint32_t` *to an* `byte_array_t`.

- void append_byte_array (byte_array_t ∗const array, const byte_array_t ∗const to_append)

  Appends the content of a `byte_array_t` *to another one.*

- void swap_byte_array (byte_array_t ∗const a, byte_array_t ∗const b)

  Swaps two `byte_array_t`*'s contents.*

- void print_byte_array (const byte_array_t ∗const array)

  Prints a `byte_array_t`.

- void ins_sort_byte_array (byte_array_t ∗const array)

  Sorts the elements of a `byte_array_t`.

- void qsort_byte_array (byte_array_t ∗const array)

  Sorts the elements of a `byte_array_t` *with a quick sort.*

- uint32_t index_in_byte_array (unsigned char to_find, const byte_array_t ∗const array)

  Returns the position of a byte in a `byte_array_t`.

- static bool is_in_byte_array (unsigned char to_find, const byte_array_t *const array)

  Returns true if a given byte is in a given array.

- uint32_t index_in_sorted_byte_array (unsigned char to_find, const byte_array_t *const sorted_array, uint32_t min_index, uint32_t max_index)

  Returns the position of an integer in a sorted portion of a `byte_array_t`.

- static bool is_in_sorted_byte_array (unsigned char to_find, const byte_array_t *const array)

  Returns true if a given byte is in a sorted `byte_array_t`.

- uint32_array_t * alloc_uint32_array (uint32_t length)

  Allocates and returns a new `uint32_array_t`.

- void free_uint32_array (uint32_array_t *array)

  Frees a `uint32_array_t`.

- void resize_uint32_array (uint32_array_t *const array, uint32_t alloced)

  Resizes the allocated memory of an `uint32_array_t`.

- void append_uint32_to_array (uint32_array_t *array, const uint32_t to_append)

  Appends a `uint32_t` *to an* `uint32_array_t`.

- void append_uint32_array (uint32_array_t *const array, const uint32_array_t *const to_append)

  Appends the content of a `uint32_array_t` *to another one.*

- void swap_uint32_array (uint32_array_t *const a, uint32_array_t *const b)

  Swaps two `uint32_array_t`*'s contents.*

- void print_uint32_array (const uint32_array_t *const array)

  Prints a `uint32_array_t`.

- void ins_sort_uint32_array (uint32_array_t *const array)

  Sorts the `uint32_t` *elements of a* `uint32_array_t`.

- void qsort_uint32_array (uint32_array_t *const array)

  Sorts the uint32_t elements of a `uint32_array_t` *with a quick sort.*

- uint32_t index_in_uint32_array (uint32_t to_find, const uint32_array_t *const array)

  Returns the position of an integer in a `uint32_array_t`.

- static bool is_in_uint32_array (uint32_t to_find, const uint32_array_t *const array)

  Returns true if a given integer is in a given array.

- uint32_t index_in_sorted_uint32_array (uint32_t to_find, const uint32_array_t *const sorted_array, uint32_t min_index, uint32_t max_index)

  Returns the position of an integer in a sorted portion of a `uint32_array_t`.

- static bool is_in_sorted_uint32_array (uint32_t to_find, const uint32_array_t *const array)

  Returns true if a given integer is in a given array.

- int32_array_t ∗ alloc_int32_array (uint32_t length)
  Allocates and returns a new `int32_array_t`.
- void free_int32_array (int32_array_t ∗array)
  Frees a `int32_array_t`.
- void resize_int32_array (int32_array_t ∗const array, uint32_t alloced)
  Resizes the allocated memory of an `int32_array_t`.
- void append_int32_to_array (int32_array_t ∗array, const int32_t to_append)
  Appends a `int32_t` *to an* `int32_array_t`.
- void append_int32_array (int32_array_t ∗const array, const int32_array_t ∗const to_append)
  Appends the content of an `int32_array_t` *to another one.*
- void swap_int32_array (int32_array_t ∗const a, int32_array_t ∗const b)
  Swaps two `int32_array_t`*'s contents.*
- void print_int32_array (const int32_array_t ∗const array)
  Prints a `int32_array_t`.
- uint32_t index_in_int32_array (int32_t to_find, const int32_array_t ∗const array)
  Returns the position of an integer in a `int32_array_t`.
- static bool is_in_int32_array (int32_t to_find, const int32_array_t ∗const array)
  Returns true if a given integer is in a given array.
- uint32_t index_in_sorted_int32_array (int32_t to_find, const int32_array_t ∗const sorted_array, uint32_t min_index, uint32_t max_index)
  Returns the position of an integer in a sorted portion of a `int32_array_t`.
- static bool is_in_sorted_int32_array (int32_t to_find, const int32_array_t ∗const array)
  Returns true if a given integer is in a given array.
- mpz_array_t ∗ alloc_mpz_array (uint32_t length)
  Allocates and returns a new `mpz_array_t`.
- void free_mpz_array (mpz_array_t ∗array)
  Frees a `mpz_array_t`.
- void resize_mpz_array (mpz_array_t ∗const array, uint32_t alloced)
  Resizes the allocated memory of an `mpz_array_t`.
- void swap_mpz_array (mpz_array_t ∗const a, mpz_array_t ∗const b)
  Swaps two `mpz_array_t`*'s contents.*
- void append_mpz_to_array (mpz_array_t ∗array, const mpz_t to_append)
  Appends an `mpz_t` *to an* `mpz_array_t`.
- void append_mpz_array (mpz_array_t ∗const array, const mpz_array_t ∗const to_append)

Appends the content of an `mpz_array_t` *to another one.*

- void print_mpz_array (const mpz_array_t *const array)

  Prints a `mpz_array_t`.

- uint32_t index_in_mpz_array (const mpz_t to_find, const mpz_array_t *const array)

  Returns the position of a `mpz_t` *in a* `mpz_array_t`.

- uint32_t index_in_sorted_mpz_array (const mpz_t to_find, const mpz_array_t *const sorted_array, uint32_t min_index, uint32_t max_index)

  Returns the position of an `mpz_t` *in a* sorted *portion of an* `mpz_array_t`.

- static bool is_in_mpz_array (const mpz_t to_find, const mpz_array_t *const array)

  Returns true if a given integer is in a given array.

- void ins_sort_mpz_array (mpz_array_t *const array)

  Sorts the mpz_t elements of a `mpz_array_t`.

- void qsort_mpz_array (mpz_array_t *const array)

  Sorts the mpz_t elements of a `mpz_array_t` *with a quick sort.*

- static bool is_in_sorted_mpz_array (const mpz_t to_find, const mpz_array_t *const array)

  Returns true if a given integer is in a given array.

- binary_array_t * alloc_binary_array (uint32_t length)

  Allocates and returns a new `binary_array_t`.

- void free_binary_array (binary_array_t *array)

  Frees a `binary_array_t`.

- void resize_binary_array (binary_array_t *const array, uint32_t alloced)

  Resizes the allocated memory of a `binary_array_t`.

- void append_bit_to_array (binary_array_t *array, const unsigned int to_append)

  Appends a bit to a `binary_array_t`.

- void print_binary_array (const binary_array_t *const array)

  Prints a `binary_array_t`.

- static uint8_t get_array_bit (uint32_t index, const binary_array_t *const array)

  Returns the value of a given bit in a `binary_array_t`.

- static void set_array_bit_to_one (uint32_t index, binary_array_t *const array)

  Sets a given bit to one in a `binary_array_t`.

- static void set_array_bit_to_zero (uint32_t index, binary_array_t *const array)

  Sets a given bit to zero in a `binary_array_t`.

- static void flip_array_bit (uint32_t index, binary_array_t *const array)

  Flips a given bit to zero in a `binary_array_t`.

5.2.1 Detailed Description

Higher level arrays and associated functions.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

This file defines higher level arrays together with some associated functions.

The `*_array_t` types and their associated functions are quite similar, the only differences being the type of the elements these arrays hold. Each `*_array_t` type is a structure composed of three fields:

- `alloced` - The maximum number of element the array can accomodate
- `length` - The current number of element in the array
- `data` - A pointer to the allocated memory space of `alloced` elements

Warning:

Since version 1.2.1 memory management changed. See the `alloc_*_array` and `clear_*_array` functions for more information.

Definition in file array.h.

5.2.2 Define Documentation

5.2.2.1 #define _TIFA_ARRAY_H_

Standard include guard.

Definition at line 47 of file array.h.

5.2.2.2 #define ARRAY_IS_FULL(ARRAY_PTR) ((ARRAY_PTR) → length == (ARRAY_PTR) → alloced)

Returns true if the `*_array_t` pointed by `ARRAY_PTR` is full (i.e. no more element can be added to the array without resizing it).

Returns false otherwise.

Definition at line 84 of file array.h.

5.2.2.3 #define ELONGATION 16

Incremental size used when automatically expanding the capacity of a `*_array_t`.

Note:

This is, of course, an hint to GMP's limbs and nails :-)

Definition at line 68 of file array.h.

5.2.2.4 #define NOT_IN_ARRAY UINT32_MAX

Value returned by the `index_in_*_array(x, array, ...)` functions if the element `x` is not in the array `array`.

Definition at line 75 of file array.h.

Referenced by is_in_byte_array(), is_in_int32_array(), is_in_mpz_array(), is_in_sorted_byte_array(), is_in_sorted_int32_array(), is_in_sorted_mpz_array(), is_in_sorted_uint32_array(), and is_in_uint32_-array().

5.2.2.5 #define reset_binary_array(ARRAY) do {(ARRAY) → length = 0;} while (0)

Resets a `binary_array_t`.

Resets the `length` field of `array` to zero.

Note that its `alloced` field is left unchanged and that memory for `alloced * CHAR_BIT * sizeof(TIFA_BITSTRING_T)` bits is still allocated.

Parameters:

← ***array*** A pointer to the `binary_array_t` to reset.

Definition at line 1197 of file array.h.

5.2.2.6 #define reset_byte_array(ARRAY) do {(ARRAY) → length = 0;} while (0)

Resets a `byte_array_t`.

Resets the `length` field of `array` to zero.

Note that its `alloced` field is left unchanged and that memory for `alloced` `byte_t` elements is still allocated.

Parameters:

← ***array*** A pointer to the `byte_array_t` to reset.

Definition at line 163 of file array.h.

5.2.2.7 #define reset_int32_array(ARRAY) do {(ARRAY) → length = 0;} while (0)

Resets an `int32_array_t`.

Resets the `length` field of `array` to zero.

Note that its `alloced` field is left unchanged and that memory for `alloced` `int32_t` elements is still allocated.

Parameters:

← ***array*** A pointer to the `int32_array_t` to reset.

Definition at line 683 of file array.h.

5.2.2.8 #define reset_mpz_array(ARRAY) do {(ARRAY) → length = 0;} while (0)

Resets an `mpz_array_t`.

Resets the `length` field of `array` to zero.

Note that its `alloced` field is left unchanged and that memory for `alloced` `mpz_t` elements is still allocated (all the elements remaining fully `mpz_init'ed`).

Warning:

Prior to 1.2 when the semantic was different, this function used to `mpz_clear` all positions in `array->data`. This is no longer true.

Parameters:

← ***array*** A pointer to the `mpz_array_t` to clear.

Definition at line 934 of file array.h.

5.2.2.9 #define reset_uint32_array(ARRAY) do {(ARRAY) → length = 0;} while (0)

Resets a `uint32_array_t`.

Resets the `length` field of `array` to zero.

Note that its `alloced` field is left unchanged and that memory for `alloced` `uint32_t` elements is still allocated.

Parameters:

← ***array*** A pointer to the `uint32_array_t` to reset.

Definition at line 423 of file array.h.

5.2.3 Function Documentation

5.2.3.1 binary_array_t∗ alloc_binary_array (uint32_t *length*)

Allocates and returns a new `binary_array_t`.

Allocates and returns a new `binary_array_t` such that:

- its `alloced` field is set to the minimum number of `TIFA_BITSTRING_T` variables needed to store length bits.
- its `length` field is set to zero.
- its `data` array is completely filled with zeroes.

Parameters:

← ***length*** The maximum bitlength of the `uint32_array_t` to allocate.

Returns:

A pointer to the newly allocated `uint32_array_t` structure. Note that this array may hold more that length bits if length is not a multiple of 8 ∗ `sizeof(TIFA_BITSTRING_T)`.

5.2.3.2 byte_array_t* alloc_byte_array (uint32_t *length*)

Allocates and returns a new `byte_array_t`.

Allocates and returns a new `byte_array_t` such that:

- its `alloced` field is set to the parameter length.
- its `length` field is set to zero.
- its `data` array is completely filled with zeroes.

Parameters:

← ***length*** The maximum length of the `byte_array_t` to allocate.

Returns:

A pointer to the newly allocated `byte_array_t` structure.

5.2.3.3 int32_array_t* alloc_int32_array (uint32_t *length*)

Allocates and returns a new `int32_array_t`.

Allocates and returns a new `int32_array_t` such that:

- its `alloced` field is set to the parameter length.
- its `length` field is set to zero.
- its `data` array is completely filled with zeroes.

Parameters:

← ***length*** The maximum length of the `int32_array_t` to allocate.

Returns:

A pointer to the newly allocated `int32_array_t` structure.

5.2.3.4 mpz_array_t* alloc_mpz_array (uint32_t *length*)

Allocates and returns a new `mpz_array_t`.

Allocates and returns a new `mpz_array_t` such that:

- its `alloced` field is set to the parameter length.
- its `length` field is set to zero.
- its `data` array is fully `mpz_init'ed`.

Parameters:

← ***length*** The maximum length of the `mpz_array_t` to allocate.

Returns:

A pointer to the newly allocated `mpz_array_t` structure.

Warning:

Since version 1.2, the `data` field is completely `mpz_init'ed` (from `data`[0] to `data[alloced` -1]) whereas older versions did not `mpz_init` anything. This change in behaviour was prompted by the need to avoid multiple memory deallocations and reallocations when using the same `mpz_array_t` repeatedly.

5.2.3.5 uint32_array_t* alloc_uint32_array (uint32_t *length*)

Allocates and returns a new `uint32_array_t`.

Allocates and returns a new `uint32_array_t` such that:

- its `alloced` field is set to the parameter length.
- its `length` field is set to zero.
- its `data` array is completely filled with zeroes.

Parameters:

← ***length*** The maximum length of the `uint32_array_t` to allocate.

Returns:

A pointer to the newly allocated `uint32_array_t` structure.

5.2.3.6 void append_bit_to_array (binary_array_t * *array*, const unsigned int *to_append*)

Appends a bit to a `binary_array_t`.

Appends a bit (set to one if `to_append != 0`, set to zero otherwise) to `array`. If `array` has not enough capacity to accommodate this extra element it will be resized via a call to `resize_binary_array` adding `ELONGATION * BITSTRING_T_BITSIZE` bit slots to avoid too frequent resizes.

Parameters:

← ***array*** A pointer to the recipient `binary_array_t`.

← ***to_append*** The bit to append (1 if `to_append != 0`, 0 otherwise).

5.2.3.7 void append_byte_array (byte_array_t *const *array*, const byte_array_t *const *to_append*)

Appends the content of a `byte_array_t` to another one.

Appends the content of the `to_append` array to the `byte_array_t` named `array`. If `array` has not enough capacity to accommodate all elements from `to_append`, it will be resized via a call to `resize_byte_array` with extra room for `ELONGATION` unused slots to avoid too frequent resizes.

Parameters:

← ***array*** A pointer to the recipient `byte_array_t`.

← ***to_append*** A pointer to the `byte_array_t` to append.

5.2.3.8 void append_byte_to_array (byte_array_t * *array*, const unsigned char *to_append*)

Appends a uint32_t to an byte_array_t.

Appends the byte to_append to array. If array has not enough capacity to accommodate this extra element it will be resized via a call to resize_byte_array adding ELONGATION byte slots to avoid too frequent resizes.

Parameters:

← ***array*** A pointer to the recipient byte_array_t.

← ***to_append*** The byte to append.

5.2.3.9 void append_int32_array (int32_array_t *const *array*, const int32_array_t *const *to_-append*)

Appends the content of an int32_array_t to another one.

Appends the content of the to_append array to the int32_array_t named array. If array has not enough capacity to accommodate all elements from to_append, it will be resized via a call to resize_-int32_array with extra room for ELONGATION unused int32_t slots to avoid too frequent resizes.

Parameters:

← ***array*** A pointer to the recipient int32_array_t.

← ***to_append*** A pointer to the int32_array_t to append.

5.2.3.10 void append_int32_to_array (int32_array_t * *array*, const int32_t *to_append*)

Appends a int32_t to an int32_array_t.

Appends the int32_t integer to_append to array. If array has not enough capacity to accommodate this extra element it will be resized via a call to resize_int32_array adding ELONGATION int32_t slots to avoid too frequent resizes.

Parameters:

← ***array*** A pointer to the recipient int32_array_t.

← ***to_append*** The integer to append.

5.2.3.11 void append_mpz_array (mpz_array_t *const *array*, const mpz_array_t *const *to_append*)

Appends the content of an mpz_array_t to another one.

Appends the content of the to_append array to the mpz_array_t named array. If array has not enough capacity to accommodate all elements from to_append, it will be resized via a call to resize_-mpz_array with extra room for ELONGATION unused mpz_t slots to avoid too frequent resizes.

Parameters:

← ***array*** A pointer to the recipient mpz_array_t.

← ***to_append*** A pointer to the mpz_array_t to append.

5.2.3.12 void append_mpz_to_array (mpz_array_t * *array*, const mpz_t *to_append*)

Appends an `mpz_t` to an `mpz_array_t`.

Appends the `mpz_t` integer `to_append` to `array`. If `array` has not enough capacity to accommodate this extra element it will be resized via a call to `resize_mpz_array` adding `ELONGATION` `mpz_t` slots to avoid too frequent resizes.

Parameters:

← ***array*** A pointer to the recipient `mpz_array_t`.

← ***to_append*** The `mpz_t` to append.

5.2.3.13 void append_uint32_array (uint32_array_t *const *array*, const uint32_array_t *const *to_append*)

Appends the content of a `uint32_array_t` to another one.

Appends the content of the `to_append` array to the `uint32_array_t` named `array`. If `array` has not enough capacity to accommodate all elements from `to_append`, it will be resized via a call to `resize_uint32_array` with extra room for `ELONGATION` unused `uint32_t` slots to avoid too frequent resizes.

Parameters:

← ***array*** A pointer to the recipient `uint32_array_t`.

← ***to_append*** A pointer to the `uint32_array_t` to append.

5.2.3.14 void append_uint32_to_array (uint32_array_t * *array*, const uint32_t *to_append*)

Appends a `uint32_t` to an `uint32_array_t`.

Appends the `uint32_t` integer `to_append` to `array`. If `array` has not enough capacity to accommodate this extra element it will be resized via a call to `resize_uint32_array` adding `ELONGATION` `uint32_t` slots to avoid too frequent resizes.

Parameters:

← ***array*** A pointer to the recipient `uint32_array_t`.

← ***to_append*** The integer to append.

5.2.3.15 static void flip_array_bit (uint32_t *index*, binary_array_t *const *array*) `[inline, static]`

Flips a given bit to zero in a `binary_array_t`.

Flips the `index`-th bit of the `binary_array_t` pointed to by `array`.

Parameters:

← ***index*** The position of the bit to flip.

← ***array*** A pointer to the `binary_array_t`.

Definition at line 1318 of file array.h.

References struct_binary_array_t::data.

5.2.3.16 void free_binary_array (binary_array_t * *array*)

Frees a `binary_array_t`.

Frees the `binary_array_t` pointed to by `array`, *i.e.* frees the memory space used by the C-style array pointed by `array->data` and frees the `array` pointer.

Warning:

Before version 1.2.1, the `array` pointer was not freed which required explicit calls to `free`(...) in client code.

Parameters:

← ***array*** A pointer to the `binary_array_t` to clear.

5.2.3.17 void free_byte_array (byte_array_t * *array*)

Frees a `byte_array_t`.

Frees the `byte_array_t` pointed to by `array`, *i.e.* frees the memory space used by the C-style array pointed by `array->data` and frees the `array` pointer.

Warning:

Before version 1.2.1, the `array` pointer was not freed which required explicit calls to `free`(...) in client code.

Parameters:

← ***array*** A pointer to the `byte_array_t` to clear.

5.2.3.18 void free_int32_array (int32_array_t * *array*)

Frees a `int32_array_t`.

Frees the `int32_array_t` pointed to by `array`, *i.e.* frees the memory space used by the C-style array pointed by `array->data` and frees the `array` pointer.

Warning:

Before version 1.2.1, the `array` pointer was not freed which required explicit calls to `free`(...) in client code.

Parameters:

← ***array*** A pointer to the `int32_array_t` to clear.

5.2.3.19 void free_mpz_array (mpz_array_t * *array*)

Frees a `mpz_array_t`.

Frees the `mpz_array_t` pointed to by `array`, *i.e.* frees the memory space used by the C-style array pointed by `array->data` and frees the `array` pointer.

Warning:

Before version 1.2.1, the array pointer was not freed which required explicit calls to free(...) in client code.

Parameters:

← ***array*** A pointer to the mpz_array_t to clear.

5.2.3.20 void free_uint32_array (uint32_array_t ∗ *array*)

Frees a uint32_array_t.

Frees the uint32_array_t pointed to by array, *i.e.* frees the memory space used by the C-style array pointed by array->data and frees the array pointer.

Warning:

Before version 1.2.1, the array pointer was not freed which required explicit calls to free(...) in client code.

Parameters:

← ***array*** A pointer to the uint32_array_t to clear.

5.2.3.21 static uint8_t get_array_bit (uint32_t *index*, const binary_array_t ∗const *array*) [inline, static]

Returns the value of a given bit in a binary_array_t.

Returns the value of the index-th bit of the binary_array_t pointed to by array, as either 0 or 1.

Parameters:

← ***index*** The position of the bit to read.

← ***array*** A pointer to the binary_array_t.

Returns:

The value of the index-th bit: either 0 or 1.

Definition at line 1248 of file array.h.

References struct_binary_array_t::data.

5.2.3.22 uint32_t index_in_byte_array (unsigned char *to_find*, const byte_array_t ∗const *array*)

Returns the position of a byte in a byte_array_t.

Returns the position of the byte to_find in the byte_array_t pointed to by array. If the byte to_find is not found in the byte_array_t, returns NOT_IN_ARRAY.

Note:

The NOT_IN_ARRAY value is actually -1 if interpreted as a signed int32_t.
If the array is already sorted, the more efficient function index_in_sorted_byte_array can be used as it uses a basic binary search instead of a complete scanning of the array.

Parameters:

← ***to_find*** The byte to find in the `byte_array_t`.

← ***array*** A pointer to the `byte_array_t`.

Returns:

The index of `to_find` in the array if `to_find` is found.
`NOT_IN_ARRAY` otherwise.

Referenced by is_in_byte_array().

5.2.3.23 uint32_t index_in_int32_array (int32_t *to_find*, const int32_array_t *const *array*)

Returns the position of an integer in a `int32_array_t`.

Returns the position of the integer `to_find` in the `int32_array_t` pointed to by `array`. If the integer `to_find` is not found in the `int32_array_t`, returns `NOT_IN_ARRAY`.

Note:

The `NOT_IN_ARRAY` value is actually -1 if interpreted as a signed `int32_t`.
If the array is already sorted, the more efficient function `index_in_sorted_int32_array` can be used as it uses a basic binary search instead of a complete scanning of the array.

Parameters:

← ***to_find*** The integer to find in the `int32_array_t`.

← ***array*** A pointer to the `int32_array_t`.

Returns:

The index of `to_find` in the array if `to_find` is found.
`NOT_IN_ARRAY` otherwise.

Referenced by is_in_int32_array().

5.2.3.24 uint32_t index_in_mpz_array (const mpz_t *to_find*, const mpz_array_t *const *array*)

Returns the position of a `mpz_t` in a `mpz_array_t`.

Returns the position of the `mpz_t` `to_find` in the `mpz_array_t` pointed to by `array`. If the integer `to_find` is not found in the `mpz_array_t`, returns `NOT_IN_ARRAY`.

Note:

The `NOT_IN_ARRAY` value is actually -1 if interpreted as a signed `int32_t`.

Parameters:

← ***to_find*** The `mpz_t` integer to find in the `mpz_array_t`.

← ***array*** A pointer to the `mpz_array_t`.

Returns:

The index of `to_find` in the array if `to_find` is found.
`NOT_IN_ARRAY` otherwise.

Referenced by is_in_mpz_array().

5.2.3.25 uint32_t index_in_sorted_byte_array (unsigned char *to_find*, const byte_array_t *const *sorted_array*, uint32_t *min_index*, uint32_t *max_index*)

Returns the position of an integer in a sorted portion of a `byte_array_t`.

Returns the position of the byte `to_find` in a *sorted* portion of the `byte_array_t` pointed to by `array`. If the byte `to_find` is not found in the portion delimited by `min_index` and `max_index`, returns `NOT_IN_ARRAY`.

Note:

The `NOT_IN_ARRAY` value is actually -1 if interpreted as a signed `int32_t`.

Parameters:

← ***to_find*** The byte to find in the `byte_array_t`.

← ***sorted_array*** A pointer to the `byte_array_t`.

← ***min_index*** The beginning of the sorted array portion to search in.

← ***max_index*** The end of the sorted array portion to search in.

Returns:

The index of `to_find` in the array if `to_find` is found in the sorted array portion. `NOT_IN_ARRAY` otherwise.

Referenced by is_in_sorted_byte_array().

5.2.3.26 uint32_t index_in_sorted_int32_array (int32_t *to_find*, const int32_array_t *const *sorted_array*, uint32_t *min_index*, uint32_t *max_index*)

Returns the position of an integer in a sorted portion of a `int32_array_t`.

Returns the position of the integer `to_find` in a *sorted* portion of the `int32_array_t` pointed to by `array`. If the integer `to_find` is not found in the portion delimited by `min_index` and `max_index`, returns `NOT_IN_ARRAY`.

Note:

The `NOT_IN_ARRAY` value is actually -1 if interpreted as a signed `int32_t`.

Parameters:

← ***to_find*** The integer to find in the `int32_array_t`.

← ***sorted_array*** A pointer to the `int32_array_t`.

← ***min_index*** The beginning of the sorted array portion to search in.

← ***max_index*** The end of the sorted array portion to search in.

Returns:

The index of `to_find` in the array if `to_find` is found in the sorted array portion. `NOT_IN_ARRAY` otherwise.

Referenced by is_in_sorted_int32_array().

5.2.3.27 uint32_t index_in_sorted_mpz_array (const mpz_t *to_find*, const mpz_array_t *const *sorted_array*, uint32_t *min_index*, uint32_t *max_index*)

Returns the position of an `mpz_t` in a *sorted* portion of an `mpz_array_t`.

Returns the position of the `mpz_t` `to_find` in the *sorted* portion of the `mpz_array_t` pointed to by `array`. If the integer `to_find` is not found in this portion, returns `NOT_IN_ARRAY`.

Note:

The `NOT_IN_ARRAY` value is actually -1 if interpreted as a signed `int32_t`.

Parameters:

← ***to_find*** The `mpz_t` integer to find in the `mpz_array_t`.

← ***sorted_array*** A pointer to the *sorted* `mpz_array_t`.

← ***min_index*** The beginning of the sorted array portion to search in.

← ***max_index*** The end of the sorted array portion to search in.

Returns:

The index of `to_find` in the array if `to_find` is found.
`NOT_IN_ARRAY` otherwise.

Referenced by is_in_sorted_mpz_array().

5.2.3.28 uint32_t index_in_sorted_uint32_array (uint32_t *to_find*, const uint32_array_t *const *sorted_array*, uint32_t *min_index*, uint32_t *max_index*)

Returns the position of an integer in a sorted portion of a `uint32_array_t`.

Returns the position of the integer `to_find` in a *sorted* portion of the `uint32_array_t` pointed to by `array`. If the integer `to_find` is not found in the portion delimited by `min_index` and `max_index`, returns `NOT_IN_ARRAY`.

Note:

The `NOT_IN_ARRAY` value is actually -1 if interpreted as a signed `int32_t`.

Parameters:

← ***to_find*** The integer to find in the `uint32_array_t`.

← ***sorted_array*** A pointer to the `uint32_array_t`.

← ***min_index*** The beginning of the sorted array portion to search in.

← ***max_index*** The end of the sorted array portion to search in.

Returns:

The index of `to_find` in the array if `to_find` is found in the sorted array portion.
`NOT_IN_ARRAY` otherwise.

Referenced by is_in_sorted_uint32_array().

5.2.3.29 uint32_t index_in_uint32_array (uint32_t *to_find*, const uint32_array_t ∗const *array*)

Returns the position of an integer in a `uint32_array_t`.

Returns the position of the integer `to_find` in the `uint32_array_t` pointed to by `array`. If the integer `to_find` is not found in the `uint32_array_t`, returns `NOT_IN_ARRAY`.

Note:

The `NOT_IN_ARRAY` value is actually -1 if interpreted as a signed `int32_t`.
If the array is already sorted, the more efficient function `index_in_sorted_uint32_array` can be used as it uses a basic binary search instead of a complete scanning of the array.

Parameters:

← ***to_find*** The integer to find in the `uint32_array_t`.

← ***array*** A pointer to the `uint32_array_t`.

Returns:

The index of `to_find` in the array if `to_find` is found.
`NOT_IN_ARRAY` otherwise.

Referenced by is_in_uint32_array().

5.2.3.30 void ins_sort_byte_array (byte_array_t ∗const *array*)

Sorts the elements of a `byte_array_t`.

Sorts the elements of a `byte_array_t` in natural order using a basic insertion sort.

Parameters:

← ***array*** A pointer to the `byte_array_t` to sort.

5.2.3.31 void ins_sort_mpz_array (mpz_array_t ∗const *array*)

Sorts the mpz_t elements of a `mpz_array_t`.

Sorts the mpz_t elements of a `mpz_array_t` in natural order using a basic insertion sort.

Parameters:

← ***array*** A pointer to the `mpz_array_t` to sort.

5.2.3.32 void ins_sort_uint32_array (uint32_array_t ∗const *array*)

Sorts the `uint32_t` elements of a `uint32_array_t`.

Sorts the uint32_t elements of a `uint32_array_t` in natural order using a basic insertion sort.

Parameters:

← ***array*** A pointer to the `uint32_array_t` to sort.

5.2.3.33 static bool is_in_byte_array (unsigned char *to_find*, const byte_array_t ∗const *array*) [inline, static]

Returns true if a given byte is in a given array.

Returns true if the byte `to_find` is in the `byte_array_t` pointed to by `array`. Returns false otherwise.

Note:

If the array is already sorted, the more efficient function `is_in_sorted_byte_array` can be used as it uses a basic binary search instead of a complete scanning of the array.

Parameters:

← ***to_find*** The integer to find in the `byte_array_t`.
← ***array*** A pointer to the `byte_array_t`.

Returns:

true if `to_find` is in the array `array`.
false otherwise.

Definition at line 296 of file array.h.

References index_in_byte_array(), and NOT_IN_ARRAY.

5.2.3.34 static bool is_in_int32_array (int32_t *to_find*, const int32_array_t ∗const *array*) [inline, static]

Returns true if a given integer is in a given array.

Returns true if the integer `to_find` is in the `int32_array_t` pointed to by `array`. Returns false otherwise.

Note:

If the array is already sorted, the more efficient function `is_in_sorted_int32_array` can be used as it uses a basic binary search instead of a complete scanning of the array.

Parameters:

← ***to_find*** The integer to find in the `int32_array_t`.
← ***array*** A pointer to the `int32_array_t`.

Returns:

true if `to_find` is in the array `array`.
false otherwise.

Definition at line 792 of file array.h.

References index_in_int32_array(), and NOT_IN_ARRAY.

5.2.3.35 static bool is_in_mpz_array (const mpz_t *to_find*, const mpz_array_t ∗const *array*) [inline, static]

Returns true if a given integer is in a given array.

Returns true if the `mpz_t` integer `to_find` is in the `mpz_array_t` pointed to by `array`. Returns false otherwise.

Note:

If the array is already sorted, the more efficient function `is_in_sorted_mpz_array` can be used as it uses a basic binary search instead of a complete scanning of the array.

Parameters:

← ***to_find*** The integer to find in the `mpz_array_t`.

← ***array*** A pointer to the `mpz_array_t`.

Returns:

true if `to_find` is in the array `array`.
false otherwise.

Definition at line 1062 of file array.h.

References index_in_mpz_array(), and NOT_IN_ARRAY.

5.2.3.36 static bool is_in_sorted_byte_array (unsigned char *to_find*, const byte_array_t ∗const *array*) `[inline, static]`

Returns true if a given byte is in a sorted `byte_array_t`.

Returns true if the byte `to_find` is in the (*already sorted*) `byte_array_t` pointed to by `array`. Returns false otherwise.

Parameters:

← ***to_find*** The byte to find in the `byte_array_t`.

← ***array*** A pointer to the *sorted* `byte_array_t`.

Returns:

true if `to_find` is in the array `array`.
false otherwise.

Definition at line 337 of file array.h.

References index_in_sorted_byte_array(), struct_byte_array_t::length, and NOT_IN_ARRAY.

5.2.3.37 static bool is_in_sorted_int32_array (int32_t *to_find*, const int32_array_t ∗const *array*) `[inline, static]`

Returns true if a given integer is in a given array.

Returns true if the integer `to_find` is in the (*already sorted*) `int32_array_t` pointed to by `array`. Returns false otherwise.

Parameters:

← ***to_find*** The integer to find in the `int32_array_t`.

← ***array*** A pointer to the *sorted* `int32_array_t`.

Returns:

true if `to_find` is in the array `array`.
false otherwise.

Definition at line 833 of file array.h.

References index_in_sorted_int32_array(), struct_int32_array_t::length, and NOT_IN_ARRAY.

5.2.3.38 static bool is_in_sorted_mpz_array (const mpz_t *to_find*, const mpz_array_t ∗const *array*) [inline, static]

Returns true if a given integer is in a given array.

Returns true if the `mpz_t` integer `to_find` is in the `mpz_array_t` pointed to by `array`. Returns false otherwise.

Parameters:

← ***to_find*** The integer to find in the `mpz_array_t`.
← ***array*** A pointer to the *sorted* `mpz_array_t`.

Returns:

true if `to_find` is in the array `array`.
false otherwise.

Definition at line 1102 of file array.h.

References index_in_sorted_mpz_array(), struct_mpz_array_t::length, and NOT_IN_ARRAY.

5.2.3.39 static bool is_in_sorted_uint32_array (uint32_t *to_find*, const uint32_array_t ∗const *array*) [inline, static]

Returns true if a given integer is in a given array.

Returns true if the integer `to_find` is in the (*already sorted*) `uint32_array_t` pointed to by `array`. Returns false otherwise.

Parameters:

← ***to_find*** The integer to find in the `uint32_array_t`.
← ***array*** A pointer to the *sorted* `uint32_array_t`.

Returns:

true if `to_find` is in the array `array`.
false otherwise.

Definition at line 598 of file array.h.

References index_in_sorted_uint32_array(), struct_uint32_array_t::length, and NOT_IN_ARRAY.

5.2.3.40 static bool is_in_uint32_array (uint32_t *to_find*, const uint32_array_t ∗const *array*) [inline, static]

Returns true if a given integer is in a given array.

Returns true if the integer `to_find` is in the `uint32_array_t` pointed to by `array`. Returns false otherwise.

Note:

If the array is already sorted, the more efficient function `is_in_sorted_uint32_array` can be used as it uses a basic binary search instead of a complete scanning of the array.

Parameters:

← ***to_find*** The integer to find in the `uint32_array_t`.

← ***array*** A pointer to the `uint32_array_t`.

Returns:

true if `to_find` is in the array `array`.
false otherwise.

Definition at line 557 of file array.h.

References index_in_uint32_array(), and NOT_IN_ARRAY.

5.2.3.41 void print_binary_array (const binary_array_t *const *array*)

Prints a `binary_array_t`.

Prints a `binary_array_t`'s on the standard output.

Parameters:

← ***array*** A pointer to the `binary_array_t` to print.

5.2.3.42 void print_byte_array (const byte_array_t *const *array*)

Prints a `byte_array_t`.

Prints a `byte_array_t`'s `data` elements on the standard output.

Note:

This function is mostly intended for debugging purposes as the output is not particularly well structured.

Parameters:

← ***array*** A pointer to the `byte_array_t` to print.

5.2.3.43 void print_int32_array (const int32_array_t *const *array*)

Prints a `int32_array_t`.

Prints a `int32_array_t`'s `data` elements on the standard output.

Note:

This function is mostly intended for debugging purposes as the output is not particularly well structured.

Parameters:

← ***array*** A pointer to the `int32_array_t` to print.

5.2.3.44 void print_mpz_array (const mpz_array_t ∗const *array*)

Prints a `mpz_array_t`.

Prints a `mpz_array_t`'s `data` elements on the standard output.

Note:

This function is mostly intended for debugging purposes as the output is not particularly well structured.

Parameters:

← ***array*** A pointer to the `mpz_array_t` to print.

5.2.3.45 void print_uint32_array (const uint32_array_t ∗const *array*)

Prints a `uint32_array_t`.

Prints a `uint32_array_t`'s `data` elements on the standard output.

Note:

This function is mostly intended for debugging purposes as the output is not particularly well structured.

Parameters:

← ***array*** A pointer to the `uint32_array_t` to print.

5.2.3.46 void qsort_byte_array (byte_array_t ∗const *array*)

Sorts the elements of a `byte_array_t` with a quick sort.

Sorts the elements of a `byte_array_t` in natural order using the quick sort algorithm.

Note:

This function relies on the C library implementation of the quick sort provided by the function `qsort`.

Parameters:

← ***array*** A pointer to the `byte_array_t` to sort.

5.2.3.47 void qsort_mpz_array (mpz_array_t ∗const *array*)

Sorts the mpz_t elements of a `mpz_array_t` with a quick sort.

Sorts the mpz_t elements of a `mpz_array_t` in natural order using the quick sort algorithm.

Note:

This function relies on the C library implementation of the quick sort provided by the function `qsort`.

Parameters:

← ***array*** A pointer to the `mpz_array_t` to sort.

5.2.3.48 void qsort_uint32_array (uint32_array_t *const *array*)

Sorts the uint32_t elements of a `uint32_array_t` with a quick sort.

Sorts the uint32_t elements of a `uint32_array_t` in natural order using the quick sort algorithm.

Note:

This function relies on the C library implementation of the quick sort provided by the function `qsort`.

Parameters:

← ***array*** A pointer to the `uint32_array_t` to sort.

5.2.3.49 void resize_binary_array (binary_array_t *const *array*, uint32_t *alloced*)

Resizes the allocated memory of a `binary_array_t`.

Resizes the storage available to an `binary_array_t` to make room for `alloced` integers, while preserving its content. If `alloced` is less than the length of the array, then obviously some of its content will be lost.

Parameters:

← ***alloced*** The new maximum length of the `binary_array_t` to resize.

← ***array*** A pointer to the `binary_array_t` to resize.

5.2.3.50 void resize_byte_array (byte_array_t *const *array*, uint32_t *alloced*)

Resizes the allocated memory of a `byte_array_t`.

Resizes the storage available to an `byte_array_t` to make room for `alloced` integers, while preserving its content. If `alloced` is less than the length of the array, then obviously some of its content will be lost.

Parameters:

← ***alloced*** The new maximum length of the `byte_array_t` to resize.

← ***array*** A pointer to the `byte_array_t` to resize.

5.2.3.51 void resize_int32_array (int32_array_t *const *array*, uint32_t *alloced*)

Resizes the allocated memory of an `int32_array_t`.

Resizes the storage available to an `int32_array_t` to make room for `alloced` integers, while preserving its content. If `alloced` is less than the length of the array, then obviously some of its content will be lost.

Parameters:

← ***alloced*** The new maximum length of the `int32_array_t` to resize.

← ***array*** A pointer to the `int32_array_t` to resize.

5.2.3.52 void resize_mpz_array (mpz_array_t ∗const *array*, uint32_t *alloced*)

Resizes the allocated memory of an `mpz_array_t`.

Resizes the storage available to an `mpz_array_t` to make room for `alloced` integers, while preserving its content. If `alloced` is less than the length of the array, then obviously some of its content will be freed and lost.

Parameters:

← ***alloced*** The new maximum length of the `mpz_array_t` to resize.

← ***array*** A pointer to the `mpz_array_t` to resize.

5.2.3.53 void resize_uint32_array (uint32_array_t ∗const *array*, uint32_t *alloced*)

Resizes the allocated memory of an `uint32_array_t`.

Resizes the storage available to an `uint32_array_t` to make room for `alloced` integers, while preserving its content. If `alloced` is less than the length of the array, then obviously some of its content will be lost.

Parameters:

← ***alloced*** The new maximum length of the `uint32_array_t` to resize.

← ***array*** A pointer to the `uint32_array_t` to resize.

5.2.3.54 static void set_array_bit_to_one (uint32_t *index*, binary_array_t ∗const *array*) `[inline, static]`

Sets a given bit to one in a `binary_array_t`.

Sets the `index`-th bit of the `binary_array_t` pointed to by `array` to 1.

Parameters:

← ***index*** The position of the bit to set.

← ***array*** A pointer to the `binary_array_t`.

Definition at line 1274 of file array.h.

References struct_binary_array_t::data.

5.2.3.55 static void set_array_bit_to_zero (uint32_t *index*, binary_array_t ∗const *array*) `[inline, static]`

Sets a given bit to zero in a `binary_array_t`.

Sets the `index`-th bit of the `binary_array_t` pointed to by `array` to 0.

Parameters:

← ***index*** The position of the bit to set.

← ***array*** A pointer to the `binary_array_t`.

Definition at line 1296 of file array.h.

References struct_binary_array_t::data.

5.2.3.56 void swap_byte_array (byte_array_t *const *a*, byte_array_t *const *b*)

Swaps two `byte_array_t`'s contents.

Swaps the contents of `a` and `b`, two `byte_array_t`'s.

Note:

In some case, pointer swapping is inappropriate (for example, if the pointers are passed as function arguments!), hence the need for such a swapping function.

Parameters:

← ***a*** A pointer to the first `byte_array_t` to swap.

← ***b*** A pointer to the second `byte_array_t` to swap.

5.2.3.57 void swap_int32_array (int32_array_t *const *a*, int32_array_t *const *b*)

Swaps two `int32_array_t`'s contents.

Swaps the contents of `a` and `b`, two `int32_array_t`'s.

Note:

In some case, pointer swapping is inappropriate (for example, if the pointers are passed as function arguments!), hence the need for such a swapping function.

Parameters:

← ***a*** A pointer to the first `int32_array_t` to swap.

← ***b*** A pointer to the second `int32_array_t` to swap.

5.2.3.58 void swap_mpz_array (mpz_array_t *const *a*, mpz_array_t *const *b*)

Swaps two `mpz_array_t`'s contents.

Swaps the contents of `a` and `b`, two `mpz_array_t`'s.

Note:

In some case, pointer swapping is inappropriate (for example, if the pointers are passed as function arguments!), hence the need for such a swapping function.

Parameters:

← ***a*** A pointer to the first `mpz_array_t` to swap.

← ***b*** A pointer to the second `mpz_array_t` to swap.

5.2.3.59 void swap_uint32_array (uint32_array_t *const *a*, uint32_array_t *const *b*)

Swaps two `uint32_array_t`'s contents.

Swaps the contents of `a` and `b`, two `uint32_array_t`'s.

Note:

In some case, pointer swapping is inappropriate (for example, if the pointers are passed as function arguments!), hence the need for such a swapping function.

Parameters:

← ***a*** A pointer to the first `uint32_array_t` to swap.

← ***b*** A pointer to the second `uint32_array_t` to swap.

5.3 bernsteinisms.h File Reference

Algorithms from two D. J. Bernstein's papers on the factorization of small integers.

```
#include <inttypes.h>
#include "array.h"
#include "x_array_list.h"
#include "hashtable.h"
#include "smooth_filter.h"
```

Defines

- #define _TIFA_BERNSTEINISMS_H_

Functions

- mpz_t ∗ bern_51 (uint32_t b, const mpz_t u)

  Daniel J. Bernstein's algorithm 5.1.

- mpz_t ∗ bern_53 (uint32_t b, const mpz_t u, const mpz_t x)

  Daniel J. Bernstein's algorithm 5.3.

- uint32_array_t ∗ bern_63 (const mpz_t x, const mpz_array_t ∗const tree)

  Daniel J. Bernstein's algorithm 6.3.

- void bern_71 (uint32_array_list_t ∗const decomp_list, const mpz_array_t ∗const to_be_factored, const uint32_array_t ∗const odd_primes)

  Daniel J. Bernstein's algorithm 7.1.

- uint32_t bern_21_rt (mpz_array_t ∗const smooth, const mpz_array_t ∗const xi, const mpz_t z)

  Daniel J. Bernstein's algorithm 2.1 (with computation of a remainder tree).

- uint32_t bern_21 (mpz_array_t ∗const smooth, const mpz_array_t ∗const xi, const mpz_t z)

  Daniel J. Bernstein's algorithm 2.1 (without computation of a remainder tree).

- uint32_t bern_21_rt_pairs (mpz_array_t ∗const xi, mpz_array_t ∗const smooth_yi, const mpz_array_t ∗const cand_xi, const mpz_array_t ∗const cand_yi, const mpz_t z)

  Daniel J. Bernstein's algorithm 2.1 modified.

- uint32_t bern_21_pairs (mpz_array_t ∗const xi, mpz_array_t ∗const smooth_yi, const mpz_array_t ∗const cand_xi, const mpz_array_t ∗const cand_yi, const mpz_t z)

  Daniel J. Bernstein's algorithm 2.1 modified (without computation of a remainder tree).

- uint32_t bern_21_rt_pairs_lp (const mpz_t n, hashtable_t ∗const htable, mpz_array_t ∗const xi, mpz_array_t ∗const smooth_yi, const mpz_array_t ∗const cand_xi, const mpz_array_t ∗const cand_yi, const mpz_t z)

  Daniel J. Bernstein's algorithm 2.1 modified, with large primes variation.

- uint32_t bern_21_pairs_lp (const mpz_t n, hashtable_t ∗const htable, mpz_array_t ∗const xi, mpz_array_t ∗const smooth_yi, const mpz_array_t ∗const cand_xi, const mpz_array_t ∗const cand_yi, const mpz_t z)

  Daniel J. Bernstein's algorithm 2.1 modified, with large primes variation (without computation of a remainder tree).

- uint32_t bern_21_rt_pairs_siqs (mpz_array_t ∗const xi, mpz_array_t ∗const smooth_yi, mpz_array_t ∗const a_for_smooth_gx, const mpz_array_t ∗const cand_xi, const mpz_array_t ∗const cand_yi, const mpz_array_t ∗const cand_a, const mpz_t z)

  Daniel J. Bernstein's algorithm 2.1 modified for SIQS (with computation of a remainder tree).

- uint32_t bern_21_rt_pairs_lp_siqs (const mpz_t n, hashtable_t ∗const htable, mpz_array_t ∗const xi, mpz_array_t ∗const smooth_yi, mpz_array_t ∗const a_for_smooth_yi, const mpz_array_t ∗const cand_xi, const mpz_array_t ∗const cand_yi, const mpz_array_t ∗const cand_a, const mpz_t z)

  Daniel J. Bernstein's algorithm 2.1 modified for SIQS, with large primes variation (with computation of a remainder tree).

- uint32_t djb_batch_rt (smooth_filter_t ∗const filter, unsigned long int step)

  Daniel J. Bernstein's algorithm 2.1 adapted to be used with a `smooth_filter_t`.

5.3.1 Detailed Description

Algorithms from two D. J. Bernstein's papers on the factorization of small integers.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Algorithms from two D. J. Bernstein's papers on the factorization of small integers:

- "How to find small factors of integers", `http://cr.yp.to/papers/sf.pdf`
- "How to find smooth parts of integers", `http://cr.yp.to/factorization/smoothparts-20040510.pdf`

Definition in file bernsteinisms.h.

5.3.2 Define Documentation

5.3.2.1 #define _TIFA_BERNSTEINISMS_H_

Standard include guard.

Definition at line 40 of file bernsteinisms.h.

5.3.3 Function Documentation

5.3.3.1 uint32_t bern_21 (mpz_array_t ∗const *smooth*, const mpz_array_t ∗const *xi*, const mpz_t *z*)

Daniel J. Bernstein's algorithm 2.1 (without computation of a remainder tree).

Given the prime numbers p_j listed by `pj` and the positive integers x_i listed by `xi`, determines the {p_j}-smooth part of each x_i and stores them in `smooth`, so that `smooth->data`[i] is the {p_j}-smooth part of `xi->data`[i].

The function stops when each integer from `xi` has been checked for smoothness or when the `smooth` array is completely filled. It then returns the index of the last integer in `xi` that has been checked for smoothness.

This is the algorithm 2.1 described in Daniel J. Bernstein's paper: "How to find smooth parts of integers".

Note:

This function differs from `bern_21_rt` only because no remainder tree is computed. This can sometimes be faster than the full fledged version.

See also:

Daniel J. Bernstein's paper: "How to find smooth parts of integers", `http://cr.yp.to/factorization/smoothparts-20040510.pdf`

Parameters:

→ ***smooth*** A pointer to the {p_j}-smooth parts of each x_i integer.

← ***xi*** A pointer to the list of the x_i integers.

← *z* The product of the the p_j prime numbers in the facotr base.

Returns:

The index of the last integer in `xi` that has been checked for smoothness.

5.3.3.2 uint32_t bern_21_pairs (mpz_array_t ∗const *xi*, mpz_array_t ∗const *smooth_yi*, const mpz_array_t ∗const *cand_xi*, const mpz_array_t ∗const *cand_yi*, const mpz_t *z*)

Daniel J. Bernstein's algorithm 2.1 modified (without computation of a remainder tree).

Given the prime numbers p_j listed by `pj` and the positive integers y_i listed by `cand_yi`, determines the y_i that are {p_j}-smooth and stores them in `smooth_yi`, so that `smooth_yi->data`[i] is indeed {p_j}-smooth.

In a typical factorization problem, we other found ourselves in situations where each y_i is associated to another integer x_i. The x_i associated to the {p_j}-smooth y_i are hence stored in `xi`.

The function stops when each integer from `cand_yi` has been checked for smoothness or when the `smooth_yi` array is completely filled. It then returns the index of the last integer in `cand_yi` that has been checked for smoothness.

This function uses the algorithm 2.1 described in Daniel J. Bernstein's paper: "How to find smooth parts of integers" except that this function has been tailored to better suit the factorization problem.

Note:

This function is very similar to `bern_21_rt_pairs`. The only difference is that in `bern_21_pairs` no remainder tree is computed. This can sometimes be faster than the full fledged version.

See also:

Daniel J. Bernstein's paper: "How to find smooth parts of integers", http://cr.yp.to/factorization/smoothparts-20040510.pdf

Parameters:

→ ***xi*** A pointer to the {x_i} associated to the {p_j}-smooth y_i integer.

→ ***smooth_yi*** A pointer to the {p_j}-smooth y_i integer.

← ***cand_xi*** A pointer to the list of the x_i integers.

← ***cand_yi*** A pointer to the list of the y_i integers.

← ***z*** The product of the the p_j prime numbers in the facotr base.

Returns:

The index of the last integer in `cand_yi` that has been checked for smoothness.

5.3.3.3 uint32_t bern_21_pairs_lp (const mpz_t *n*, hashtable_t ∗const *htable*, mpz_array_t ∗const *xi*, mpz_array_t ∗const *smooth_yi*, const mpz_array_t ∗const *cand_xi*, const mpz_array_t ∗const *cand_yi*, const mpz_t *z*)

Daniel J. Bernstein's algorithm 2.1 modified, with large primes variation (without computation of a remainder tree).

Given `z`, the product of prime numbers p_j and the positive integers y_i listed by `cand_yi`, determines the y_i that are {p_j}-smooth and stores them in `smooth_yi`, so that `smooth_yi->data`[i] is indeed {p_j}-smooth.

In a typical factorization problem, we other found ourselves in situations where each y_i is associated to another integer x_i. The x_i associated to the {p_j}-smooth y_i are hence stored in `xi`.

Moreover, this function implements the so-called large primes variation. If a given y_i is not {p_j}-smooth but is the product of a prime `p` by a {p_j}-smooth number, it is stored in the hashtable `htable`. Subsequently, if another y_j is the product of a {p_j}-smooth number by the same prime number `p`, then `y_i*y_j/(p^2)` is stored in `smooth_yi` and `x_i*x_j*pinv` is stored in `xi` where `pinv` is the inverse of `p` in Z/cZ.

The function stops when each integer from `cand_yi` has been checked for smoothness or when the `smooth_yi` array is completely filled. It then returns the index of the last integer in `cand_yi` that has been checked for smoothness.

This function uses the algorithm 2.1 described in Daniel J. Bernstein's paper: "How to find smooth parts of integers" except that this function has been tailored to better suit the factorization problem.

Note:

This function is very similar to `bern_21_rt_pairs_lp`. The only difference is that here no remainder tree is computed. This can sometimes be faster than the full fledged version.

See also:

Daniel J. Bernstein's paper: "How to find smooth parts of integers", http://cr.yp.to/factorization/smoothparts-20040510.pdf

Parameters:

← ***n*** The integer to factor.

← ***htable*** A pointer to the hashtable used for the large prime variation.

→ ***xi*** A pointer to the {x_i} associated to the {p_j}-smooth y_i integer.

→ ***smooth_yi*** A pointer to the {p_j}-smooth y_i integer.

← ***cand_xi*** A pointer to the list of the x_i integers.

← ***cand_yi*** A pointer to the list of the y_i integers.

← ***z*** The product of the the p_j prime numbers in the facotr base.

Returns:

The index of the last integer in `cand_yi` that has been checked for smoothness.

5.3.3.4 uint32_t bern_21_rt (mpz_array_t ∗const *smooth*, const mpz_array_t ∗const *xi*, const mpz_t *z*)

Daniel J. Bernstein's algorithm 2.1 (with computation of a remainder tree).

Given the prime numbers p_j listed by `pj` and the positive integers x_i listed by `xi`, determines the {p_j}-smooth part of each x_i and stores them in `smooth`, so that `smooth->data`[i] is the {p_j}-smooth part of `xi->data`[i].

The function stops when each integer from `xi` has been checked for smoothness or when the `smooth` array is completely filled. It then returns the index of the last integer in `xi` that has been checked for smoothness.

This is the algorithm 2.1 described in Daniel J. Bernstein's paper: "How to find smooth parts of integers".

See also:

Daniel J. Bernstein's paper: "How to find smooth parts of integers", http://cr.yp.to/factorization/smoothparts-20040510.pdf

Parameters:

→ ***smooth*** A pointer to the {p_j}-smooth parts of each x_i integer.

← ***xi*** A pointer to the list of the x_i integers.

← ***z*** The product of the the p_j prime numbers in the facotr base.

Returns:

The index of the last integer in `xi` that has been checked for smoothness.

5.3.3.5 uint32_t bern_21_rt_pairs (mpz_array_t *const *xi*, mpz_array_t *const *smooth_yi*, const mpz_array_t *const *cand_xi*, const mpz_array_t *const *cand_yi*, const mpz_t *z*)

Daniel J. Bernstein's algorithm 2.1 modified.

Given the prime numbers p_j listed by `pj` and the positive integers y_i listed by `cand_yi`, determines the y_i that are {p_j}-smooth and stores them in `smooth_yi`, so that `smooth_yi->data`[i] is indeed {p_j}-smooth.

In a typical factorization problem, we other found ourselves in situations where each y_i is associated to another integer x_i. The x_i associated to the {p_j}-smooth y_i are hence stored in `xi`.

The function stops when each integer from `cand_yi` has been checked for smoothness or when the `smooth_yi` array is completely filled. It then returns the index of the last integer in `cand_yi` that has been checked for smoothness.

This function uses the algorithm 2.1 described in Daniel J. Bernstein's paper: "How to find smooth parts of integers" except that this function has been tailored to better suit the factorization problem.

See also:

Daniel J. Bernstein's paper: "How to find smooth parts of integers", http://cr.yp.to/factorization/smoothparts-20040510.pdf

Parameters:

→ ***xi*** A pointer to the {x_i} associated to the {p_j}-smooth y_i integer.

→ ***smooth_yi*** A pointer to the {p_j}-smooth y_i integer.

← ***cand_xi*** A pointer to the list of the x_i integers.

← ***cand_yi*** A pointer to the list of the y_i integers.

← ***z*** The product of the the p_j prime numbers in the facotr base.

Returns:

The index of the last integer in `cand_yi` that has been checked for smoothness.

5.3.3.6 uint32_t bern_21_rt_pairs_lp (const mpz_t *n*, hashtable_t *const *htable*, mpz_array_t *const *xi*, mpz_array_t *const *smooth_yi*, const mpz_array_t *const *cand_xi*, const mpz_array_t *const *cand_yi*, const mpz_t *z*)

Daniel J. Bernstein's algorithm 2.1 modified, with large primes variation.

Given `z`, the product of prime numbers p_j and the positive integers y_i listed by `cand_yi`, determines the y_i that are {p_j}-smooth and stores them in `smooth_yi`, so that `smooth_yi->data`[i] is indeed {p_j}-smooth.

In a typical factorization problem, we other found ourselves in situations where each y_i is associated to another integer x_i. The x_i associated to the {p_j}-smooth y_i are hence stored in `xi`.

Moreover, this function implements the so-called large primes variation. If a given y_i is not {p_j}-smooth but is the product of a prime `p` by a {p_j}-smooth number, it is stored in the hashtable `htable`. Subsequently, if another y_j is the product of a {p_j}-smooth number by the same prime number `p`, then `y_i*y_j/(p^2)` is stored in `smooth_yi` and `x_i*x_j*pinv` is stored in `xi` where `pinv` is the inverse of `p` in Z/cZ.

The function stops when each integer from `cand_yi` has been checked for smoothness or when the `smooth_yi` array is completely filled. It then returns the index of the last integer in `cand_yi` that has been checked for smoothness.

This function uses the algorithm 2.1 described in Daniel J. Bernstein's paper: "How to find smooth parts of integers" except that this function has been tailored to better suit the factorization problem.

See also:

Daniel J. Bernstein's paper: "How to find smooth parts of integers", http://cr.yp.to/factorization/smoothparts-20040510.pdf

Parameters:

← ***n*** The integer to factor.

← ***htable*** A pointer to the hashtable used for the large prime variation.

→ ***xi*** A pointer to the {x_i} associated to the {p_j}-smooth y_i integer.

→ ***smooth_yi*** A pointer to the {p_j}-smooth y_i integer.

← ***cand_xi*** A pointer to the list of the x_i integers.

← ***cand_yi*** A pointer to the list of the y_i integers.

← ***z*** The product of the the p_j prime numbers in the facotr base.

Returns:

The index of the last integer in `cand_yi` that has been checked for smoothness.

5.3.3.7 uint32_t bern_21_rt_pairs_lp_siqs (const mpz_t *n*, hashtable_t ∗const *htable*, mpz_array_t ∗const *xi*, mpz_array_t ∗const *smooth_yi*, mpz_array_t ∗const *a_for_smooth_yi*, const mpz_array_t ∗const *cand_xi*, const mpz_array_t ∗const *cand_yi*, const mpz_array_t ∗const *cand_a*, const mpz_t *z*)

Daniel J. Bernstein's algorithm 2.1 modified for SIQS, with large primes variation (with computation of a remainder tree).

Given `z`, the product of prime numbers p_j and the positive integers y_i listed by `cand_yi`, determines the y_i that are {p_j}-smooth and stores them in `smooth_yi`, so that `smooth_yi->data`[i] is indeed {p_j}-smooth.

In a typical factorization problem, we other found ourselves in situations where each y_i is associated to another integer x_i. The x_i associated to the {p_j}-smooth y_i are hence stored in `xi`.

Moreover, this function implements the so-called large primes variation. If a given y_i is not {p_j}-smooth but is the product of a prime `p` by a {p_j}-smooth number, it is stored in the hashtable `htable`. Subsequently, if another y_j is the product of a {p_j}-smooth number by the same prime number `p`, then `y_i*y_j/(p^2)` is stored in `smooth_yi` and `x_i*x_j*pinv` is stored in `xi` where `pinv` is the inverse of `p` in Z/cZ.

The function stops when each integer from `cand_yi` has been checked for smoothness or when the `smooth_yi` array is completely filled. It then returns the index of the last integer in `cand_yi` that has been checked for smoothness.

This function uses the algorithm 2.1 described in Daniel J. Bernstein's paper: "How to find smooth parts of integers" except that this function has been tailored to better suit the factorization problem, particularly to our SIQS implementation where we need to keep track of additionnal `a_i` integers associated to each `y_i` integers. These extra integers are stored in `cand_a` whereas the a_i associated to the smooth y_i will be stored in the `a_for_smooth_gx` array.

Note:

The `a_i` integers are actually the values of the first parameter of the polynomials used in the SIQS algorithm.

See also:

Daniel J. Bernstein's paper: "How to find smooth parts of integers", http://cr.yp.to/factorization/smoothparts-20040510.pdf

Parameters:

← ***n*** The integer to factor.

← ***htable*** A pointer to the hashtable used for the large prime variation.

→ ***xi*** A pointer to the `x_i` associated to the {p_j}-smooth `y_i` integer.

→ ***smooth_yi*** A pointer to the {p_j}-smooth `y_i` integer.

→ ***a_for_smooth_yi*** A pointer to the array of the `a_i` integers.

← ***cand_xi*** A pointer to the list of the `x_i` integers.

← ***cand_yi*** A pointer to the list of the `y_i` integers.

← ***cand_a*** A pointer to the list of the `a_i` integers associated to the {p_j}-smooth y_i integers.

← ***z*** The product of the the `p_j` prime numbers in the factor base.

Returns:

The index of the last integer in `cand_yi` that has been checked for smoothness.

5.3.3.8 uint32_t bern_21_rt_pairs_siqs (mpz_array_t ∗const *xi*, mpz_array_t ∗const *smooth_yi*, mpz_array_t ∗const *a_for_smooth_gx*, const mpz_array_t ∗const *cand_xi*, const mpz_array_t ∗const *cand_yi*, const mpz_array_t ∗const *cand_a*, const mpz_t *z*)

Daniel J. Bernstein's algorithm 2.1 modified for SIQS (with computation of a remainder tree).

Given `z`, the product of prime numbers p_j and the positive integers y_i listed by `cand_yi`, determines the y_i that are {p_j}-smooth and stores them in `smooth_yi`, so that `smooth_yi->data`[i] is indeed {p_j}-smooth.

In a typical factorization problem, we other found ourselves in situations where each y_i is associated to another integer x_i. The x_i associated to the {p_j}-smooth y_i are hence stored in `xi`.

The function stops when each integer from `cand_yi` has been checked for smoothness or when the `smooth_yi` array is completely filled. It then returns the index of the last integer in `cand_yi` that has been checked for smoothness.

This function uses the algorithm 2.1 described in Daniel J. Bernstein's paper: "How to find smooth parts of integers" except that this function has been tailored to better suit the factorization problem, particularly to our SIQS implementation where we need to keep track of additionnal `a_i` integers associated to each `y_i` integers. These extra integers are stored in `cand_a` whereas the a_i associated to the smooth y_i will be stored in the `a_for_smooth_gx` array.

Note:

The `a_i` integers are actually the values of the first parameter of the polynomials used in the SIQS algorithm.

See also:

Daniel J. Bernstein's paper: "How to find smooth parts of integers", http://cr.yp.to/factorization/smoothparts-20040510.pdf

Parameters:

→ ***xi*** A pointer to the `x_i` associated to the {p_j}-smooth `y_i` integer.

→ ***smooth_yi*** A pointer to the {p_j}-smooth `y_i` integer.
→ ***a_for_smooth_gx*** A pointer to the array of the `a_i` integers.
← ***cand_xi*** A pointer to the list of the `x_i` integers.
← ***cand_yi*** A pointer to the list of the `y_i` integers.
← ***cand_a*** A pointer to the list of the `a_i` integers associated to the {p_j}-smooth y_i integers.
← ***z*** The product of the the `p_j` prime numbers in the factor base.

Returns:

The index of the last integer in `cand_yi` that has been checked for smoothness.

5.3.3.9 mpz_t∗ bern_51 (uint32_t *b*, const mpz_t *u*)

Daniel J. Bernstein's algorithm 5.1.

Given a positive integer `b` and an odd positive integer `u`, returns a non negative integer `v` < 2^`b` such that 1 + `u`∗`v` = 0 (mod 2^`b`).

This is the algorithm 5.1 described in Daniel J. Bernstein's paper: "How to find small factors of integers".

See also:

Daniel J. Bernstein's paper: "How to find small factors of integers", http://cr.yp.to/papers/sf.pdf

Parameters:

← ***b*** A positive integer.
← ***u*** An odd positive `mpz_t` integer.

Returns:

A non negative `mpz_t` integer `v` < 2^`b` such that 1 + `u`∗`v` = 0 (mod 2^`b`).

5.3.3.10 mpz_t∗ bern_53 (uint32_t *b*, const mpz_t *u*, const mpz_t *x*)

Daniel J. Bernstein's algorithm 5.3.

Given an odd positive integer `u` < 2^`c` and a non negative integer `x` < 2^(`b` + `c`), returns a non negative integer `r` < 2^(`c` + 1) such that `r`∗2^`b` = `x` (mod `u`).

This is the algorithm 5.3 described in Daniel J. Bernstein's paper: "How to find small factors of integers".

See also:

Daniel J. Bernstein's paper: "How to find small factors of integers", http://cr.yp.to/papers/sf.pdf

Parameters:

← ***b*** A positive integer.
← ***u*** An odd positive `mpz_t` integer.
← ***x*** An non negative `mpz_t` integer.

Returns:

A non negative `mpz_t` integer `r` such that `r`∗2^`b` = `x` (mod `u`).

5.3.3.11 uint32_array_t∗ bern_63 (const mpz_t *x*, const mpz_array_t ∗const *tree*)

Daniel J. Bernstein's algorithm 6.3.

Given a non negative integer `x` and given the product tree `tree` of a sequence of odd positive integers p_i, returns the integers p_i such that: `x` mod p_i = 0.

This is the algorithm 6.3 described in Daniel J. Bernstein's paper: "How to find small factors of integers".

See also:

Daniel J. Bernstein's paper: "How to find small factors of integers", http://cr.yp.to/papers/sf.pdf

Parameters:

← ***x*** A non negative positive integer.

← ***tree*** The product tree of a sequence of odd positive integers p_i.

Returns:

A pointer to an `uint32_array_t` holding the integers p_i such that: `x` mod p_i = 0.

5.3.3.12 void bern_71 (uint32_array_list_t ∗const *decomp_list*, const mpz_array_t ∗const *to_be_-factored*, const uint32_array_t ∗const *odd_primes*)

Daniel J. Bernstein's algorithm 7.1.

Given a sequence of odd primes p_j given by `odd_primes` and a set of integers n_i given by `to_be_-factored`, determines, for each n_i, the list of odd primes p_j such that (n_i mod p_j = 0) and stores them in `decomp_list`. Each entry in `decomp_list->data`[i] is a pointer to a `mpz_array_t` listing the p_j for the integer `to_be_factored->data[i]`.

This is the algorithm 7.1 described in Daniel J. Bernstein's paper: "How to find small factors of integers".

See also:

Daniel J. Bernstein's paper: "How to find small factors of integers", http://cr.yp.to/papers/sf.pdf

Parameters:

→ ***decomp_list*** A pointer to the list of matching p_j for each n_i.

← ***to_be_factored*** A pointer to the set of integers n_i.

← ***odd_primes*** A pointer to the set of integers p_j.

5.3.3.13 uint32_t djb_batch_rt (smooth_filter_t ∗const *filter*, unsigned long int *step*)

Daniel J. Bernstein's algorithm 2.1 adapted to be used with a `smooth_filter_t`.

If `filter->nsteps` **== 0** In such a case, no early abort strategy is performed. The effect of the function is the same as `bern_21_*_pairs_*` called with:

```
filter->n
filter->htable
```

```
filter->accepted_xi
filter->accepted_yi
filter->accepted_ai
filter->candidate_xi
filter->candidate_yi
filter->candidate_ai
filter->prod_pj[0]
```

If `filter->nsteps` != 0 An early abort strategy is performed.

- If 1 <= `step` < `filter->nsteps`:

  Relations at step `step-1` from `filter`, (`filter->filtered_*[step-1]`) are used as "candidate" arrays to populate either `filter->accepted_*` or `filter->filtered_*`[step].

- If `step` == 0:

  The candidate relations are taken from `filter->candidate_*`.

- If `step` == `filter->nsteps`:

  The candidate relations are taken from `filter->filtered_*[filter->nsteps` - 1] and 'good' relations will be stored in `filter->accepted_*`.

See also:

The bern_21_* functions.

Warning:

Using `filter->nsteps` != 0 is not recommended. First, it certainly does not make any sense to try to early-abort the batch. Second, even if it is useful (for some weird reasons that I'm not aware of), the cases `filter->nsteps` != 0 have not been tuned / fully debugged.

Parameters:

filter pointer to the `smooth_filter_t` to use

step the step number in the early abort strategy

Returns:

The number of relations used from the "candidate" arrays.

5.4 bitstring_t.h File Reference

Preprocessor defines for 'string of bit' type.

`#include "tifa_config.h"`

Defines

- #define _TIFA_BITSTRING_T_H_

5.4.1 Detailed Description

Preprocessor defines for 'string of bit' type.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Defines several preprocessor `define` symbols related to the type used to represent strings of bit. These symbolss are kept separately to avoid too much namespace pollution.

Definition in file bitstring_t.h.

5.4.2 Define Documentation

5.4.2.1 #define _TIFA_BITSTRING_T_H_

Standard include guard.

Definition at line 37 of file bitstring_t.h.

5.5 buckets.h File Reference

Structure and inline functions to implement bucket sieving.

`#include <stdlib.h>`

5.5.1 Detailed Description

Structure and inline functions to implement bucket sieving.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Definition in file buckets.h.

5.6 cfrac.h File Reference

The CFRAC factorization algorithm.

```
#include <stdbool.h>
#include <gmp.h>
#include "first_primes.h"
#include "array.h"
#include "lindep.h"
#include "smooth_filter.h"
#include "factoring_machine.h"
#include "exit_codes.h"
```

Data Structures

- struct struct_cfrac_params_t

  Defines the variable parameters used in the CFRAC algorithm.

Defines

- #define _TIFA_CFRAC_H_
- #define CFRAC_DFLT_NPRIMES_IN_BASE (NFIRST_PRIMES/16)
- #define CFRAC_DFLT_NPRIMES_TDIV (NFIRST_PRIMES/16)
- #define CFRAC_DFLT_NRELATIONS 32
- #define CFRAC_DFLT_LINALG_METHOD SMART_GAUSS_ELIM
- #define CFRAC_DFLT_USE_LARGE_PRIMES true

Typedefs

- typedef struct struct_cfrac_params_t cfrac_params_t

  Equivalent to `struct struct_cfrac_params_t`.

Functions

- void set_cfrac_params_to_default (const mpz_t n, cfrac_params_t ∗const params)

  Fills a `cfrac_params_t` *with "good" default values.*

- ecode_t cfrac (mpz_array_t ∗const factors, uint32_array_t ∗const multis, const mpz_t n, const cfrac_params_t ∗const params, const factoring_mode_t mode)

  Integer factorization via the continued fraction (CFRAC) algorithm.

5.6.1 Detailed Description

The CFRAC factorization algorithm.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

This is the TIFA library's implementation of the CFRAC factorization algorithm from M. A. Morrison and J. Brillhart, together with the large prime variation.

See also:

"A Method of Factoring and the Factorization of F_7", M. A. Morrison and J. Brillhart, *Mathematics of Computation*, vol 29, #129, Jan 1975, pages 183-205.

Definition in file cfrac.h.

5.6.2 Define Documentation

5.6.2.1 #define _TIFA_CFRAC_H_

Standard include guard.

Definition at line 41 of file cfrac.h.

5.6.2.2 #define CFRAC_DFLT_LINALG_METHOD SMART_GAUSS_ELIM

Default linear system resolution method to use.

Definition at line 79 of file cfrac.h.

5.6.2.3 #define CFRAC_DFLT_NPRIMES_IN_BASE (NFIRST_PRIMES/16)

Default number of prime numbers composing the factor base on which to factor the residues.

Definition at line 62 of file cfrac.h.

5.6.2.4 #define CFRAC_DFLT_NPRIMES_TDIV (NFIRST_PRIMES/16)

Default number of the first primes to use in the trial division of the residues.

Definition at line 68 of file cfrac.h.

5.6.2.5 #define CFRAC_DFLT_NRELATIONS 32

Default number of congruence relations to find before attempting the factorization of the large integer.

Definition at line 74 of file cfrac.h.

5.6.2.6 #define CFRAC_DFLT_USE_LARGE_PRIMES true

Use the large prime variation by default.

Definition at line 84 of file cfrac.h.

5.6.3 Function Documentation

5.6.3.1 ecode_t cfrac (mpz_array_t ∗const *factors*, uint32_array_t ∗const *multis*, const mpz_t *n*, const cfrac_params_t ∗const *params*, const factoring_mode_t *mode*)

Integer factorization via the continued fraction (CFRAC) algorithm.

Attempts to factor the non perfect square integer `n` with the CFRAC algorithm, using the set of parameters given by `params` and the factoring mode given by `mode`. Found factors are then stored in `factors`. Additionally, if the factoring mode used is set to FIND_COMPLETE_FACTORIZATION, factors' multiplicities are stored in the array `multis`.

Note:

If the factoring mode used is different from FIND_COMPLETE_FACTORIZATION, `multis` is allowed to be a NULL pointer. Otherwise, using a NULL pointer will lead to a fatal error.

Warning:

If the `factors` and `multis` arrays have not enough room to store the found factors (and the multiplicities, if any), they will be automatically resized to accommodate the data. This has to be kept in mind when trying to do ingenious stuff with memory management (hint: don't try to be clever here). The "no large primes" variant is currently disabled.

Parameters:

→ ***factors*** Pointer to the found factors of `n`.

→ ***multis*** Pointer to the multiplicities of the found factors (only computed if `mode` is set to FIND_-COMPLETE_FACTORIZATION).

← ***n*** The non perfect square integer to factor.

← ***params*** Pointer to the values of the parameters used in the CFRAC algorithm.

← ***mode*** The factoring mode to use.

Returns:

An exit code.

5.6.3.2 void set_cfrac_params_to_default (const mpz_t *n*, cfrac_params_t ∗const *params*)

Fills a `cfrac_params_t` with "good" default values.

Fills a `cfrac_params_t` with "good" default values choosen according to the size of the number n to factor.

Warning:

There is no guarantee that the choosen parameter values will be the best ones for a given number to factor. However, provided that the number to factor is between 40 and 200 bits long, the choosen values should be nearly optimal.

Parameters:

← ***n*** The `mpz_t` integer to factor.

→ ***params*** A pointer to the `cfrac_params_t` structure to fill.

5.7 common_funcs.h File Reference

Miscellaneous functions and macros used by the "tool" programs.

`#include "first_primes.h"`

Defines

- #define _TIFA_COMMON_FUNCS_H_
- #define PRINT_ABORT_MSG() fprintf(stderr, "Program aborted\n");
- #define PRINT_NAN_ERROR(X)
- #define PRINT_BAD_ARGC_ERROR()
- #define PRINT_ENTER_NUMBER_MSG() printf("> Enter the integer to factor: ")
- #define PRINT_USAGE_WARNING_MSG()
- #define MAX_NDIGITS 256
- #define NTRIES_MILLER_RABIN 32
- #define NPRIMES_TRIAL_DIV NFIRST_PRIMES

Functions

- void print_hello_msg (char *name)

  Function used by the "tool" programs to print a greeting message.

- void print_bye_msg ()

  Function used by the "tool" programs to print a bye-bye message.

5.7.1 Detailed Description

Miscellaneous functions and macros used by the "tool" programs.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Definition in file common_funcs.h.

5.7.2 Define Documentation

5.7.2.1 #define _TIFA_COMMON_FUNCS_H_

Standard include guard.

Definition at line 33 of file common_funcs.h.

5.7.2.2 #define MAX_NDIGITS 256

Maximal number of decimal digits of the number to factor.

Definition at line 90 of file common_funcs.h.

5.7.2.3 #define NPRIMES_TRIAL_DIV NFIRST_PRIMES

Default number of prime numbers used in trial division of number to factor.

Definition at line 101 of file common_funcs.h.

5.7.2.4 #define NTRIES_MILLER_RABIN 32

Number of iterations to use in the Miller-Rabin composition tests.

Definition at line 95 of file common_funcs.h.

5.7.2.5 #define PRINT_ABORT_MSG() fprintf(stderr, "Program aborted\n");

Macro printing an "program has aborted" message on the standard error.

Definition at line 50 of file common_funcs.h.

5.7.2.6 #define PRINT_BAD_ARGC_ERROR()

Value:

```
do {                             \
    fprintf(stderr, "\nERROR: Bad number of arguments!\n\n"); \
} while (0)
```

Macro printing a "bad number of argument" error message on the standard error.

Definition at line 67 of file common_funcs.h.

5.7.2.7 #define PRINT_ENTER_NUMBER_MSG() printf("> Enter the integer to factor: ")

Macro displaying a prompt asking the user to enter the integer to factor.

Definition at line 75 of file common_funcs.h.

5.7.2.8 #define PRINT_NAN_ERROR(X)

Value:

```
do {                             \
    fprintf(stderr, "\nERROR: %s is not an integer!\n", X); \
    PRINT_ABORT_MSG();                                       \
} while (0)
```

Macro printing a "X is not an integer" error message on the standard error.

Definition at line 57 of file common_funcs.h.

5.7.2.9 #define PRINT_USAGE_WARNING_MSG()

Value:

```
do {                                            \
    fprintf(stderr, "Please, use the perl wrapper factorize.pl instead of "); \
    fprintf(stderr, "a direct invocation\nof this program.\n");              \
} while (0)
```

Macro printing a boilerplate usage warning on the standard error.

Definition at line 81 of file common_funcs.h.

5.7.3 Function Documentation

5.7.3.1 void print_bye_msg ()

Function used by the "tool" programs to print a bye-bye message.

This function could be used by the "tool" programs to print a bye-bye message (it is not used right now).

5.7.3.2 void print_hello_msg (char ∗ *name*)

Function used by the "tool" programs to print a greeting message.

Parameters:

← ***name*** Name of the factoring program.

5.8 ecm.h File Reference

The elliptic curve method of integer factorization (ECM).

`#include <gmp.h>`

`#include "array.h"`

`#include "factoring_machine.h"`

`#include "exit_codes.h"`

Data Structures

- struct struct_ecm_params_t

  Defines the variable parameters used in ECM.

Defines

- #define _TIFA_ECM_H_

Typedefs

- typedef struct struct_ecm_params_t ecm_params_t
  Equivalent to `struct struct_ecm_params_t`.

Functions

- void set_ecm_params_to_default (const mpz_t n, ecm_params_t *const params)
  Fills an `ecm_params_t` *with "good" default values.*
- ecode_t ecm (mpz_array_t *const factors, uint32_array_t *const multis, const mpz_t n, const ecm_params_t *const params, const factoring_mode_t mode)
  Integer factorization with the elliptic curve method (ECM).

5.8.1 Detailed Description

The elliptic curve method of integer factorization (ECM).

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

This is the TIFA library's implementation of the 'ECM' factorization algorithm. The second phase of the algorithm follows the standard continuation and is implemented in a way reminiscent of the description given in the article "Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware" by Kris Gaj et al.

See also:

"Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware", K. Gaj et al., *Cryptographic Hardware and Embedded Systems - CHES 2006*.

Warning:

This is merely a toy-implementation of ECM without any smart optimizations. More work is certainly needed to make it competitive for small numbers. Large numbers are, of course, out of the scope of this library.

Definition in file ecm.h.

5.8.2 Define Documentation

5.8.2.1 #define _TIFA_ECM_H_

Standard include guard.

Definition at line 48 of file ecm.h.

5.8.3 Function Documentation

5.8.3.1 ecode_t ecm (mpz_array_t ∗const *factors*, uint32_array_t ∗const *multis*, const mpz_t *n*, const ecm_params_t ∗const *params*, const factoring_mode_t *mode*)

Integer factorization with the elliptic curve method (ECM).

Attempts to factor the non perfect square integer `n` with the ECM, using the set of parameters given by `params` and the factoring mode given by `mode`. Found factors are then stored in `factors`. Additionally, if the factoring mode used is set to FIND_COMPLETE_FACTORIZATION, factors' multiplicities are stored in the array `multis`.

Note:

If the factoring mode used is different from FIND_COMPLETE_FACTORIZATION, `multis` is allowed to be a NULL pointer. Otherwise, using a NULL pointer will lead to a fatal error.

Warning:

If the `factors` and `multis` arrays have not enough room to store the found factors (and the multiplicities, if any), they will be automatically resized to accommodate the data. This has to be kept in mind when trying to do ingenious stuff with memory management (hint: don't try to be clever here).

Parameters:

→ ***factors*** Pointer to the found factors of `n`.

→ ***multis*** Pointer to the multiplicities of the found factors (only computed if `mode` is set to FIND_-COMPLETE_FACTORIZATION).

← ***n*** The non perfect square integer to factor.

← ***params*** Pointer to the values of the parameters used in the ECM.

← ***mode*** The factoring mode to use.

Returns:

An exit code.

5.8.3.2 void set_ecm_params_to_default (const mpz_t *n*, ecm_params_t ∗const *params*)

Fills an `ecm_params_t` with "good" default values.

Fills an `ecm_params_t` with "good" default values choosen according to the size of the number n to factor.

Warning:

This is, for the time being, a dummy function. Parameters are *not* set to suitable values *at all!* Do *not* use it: for the time being, you should choose the parameters by yourself! Shocking!

Parameters:

← ***n*** The `mpz_t` integer to factor.

→ ***params*** A pointer to the `ecm_params_t` structure to fill.

5.9 exit_codes.h File Reference

Exit codes used by/in some of the TIFA functions.

Defines

- #define _TIFA_EXIT_CODES_H_
- #define PRINT_ECODE(ECODE) printf("%s\n", ecode_to_str[ECODE]);

Typedefs

- typedef enum ecode_enum ecode_t

  Equivalent to `enum ecode_enum`.

Enumerations

- enum ecode_enum {

  UNKNOWN_FACTORING_MODE, SOME_FACTORS_FOUND, SOME_PRIME_FACTORS_-FOUND, SOME_COPRIME_FACTORS_FOUND,

  PARTIAL_FACTORIZATION_FOUND, COMPLETE_FACTORIZATION_FOUND, NO_-FACTOR_FOUND, FATAL_INTERNAL_ERROR,

  QUEUE_OVERFLOW, NO_PROPER_FORM_FOUND, GIVING_UP, INTEGER_TOO_LARGE,

  SUCCESS, FAILURE }

Variables

- static const char *const ecode_to_str [14]

5.9.1 Detailed Description

Exit codes used by/in some of the TIFA functions.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Defines several exit codes used by/in some of the TIFA functions together with their string representations.

Definition in file exit_codes.h.

5.9.2 Define Documentation

5.9.2.1 #define _TIFA_EXIT_CODES_H_

Standard include guard.

Definition at line 36 of file exit_codes.h.

5.9.2.2 #define PRINT_ECODE(ECODE) printf("%s\n", ecode_to_str[ECODE]);

Macro printing the string representation of the exit code `ECODE` on the standard output, followed by a newline.

Definition at line 156 of file exit_codes.h.

5.9.3 Enumeration Type Documentation

5.9.3.1 enum ecode_enum

An enumeration of the possible exit codes used by/in some TIFA functions.

Enumerator:

UNKNOWN_FACTORING_MODE Used by a `factoring_machine_t` to indicate that the `factoring_mode_t` passed as parameter is not valid.

SOME_FACTORS_FOUND Used by the factorization algorithm to indicate that *some* factors were found. In that case, the factors' multiplicities are not computed.

SOME_PRIME_FACTORS_FOUND Used by the factorization algorithm to indicate that *some* prime factors were found. In that case, the factors' multiplicities are not computed.

SOME_COPRIME_FACTORS_FOUND Used by the factorization algorithm to indicate that *some* coprime factors were found. In that case, the factors' multiplicities are not computed.

PARTIAL_FACTORIZATION_FOUND Used by the factorization algorithm to indicate that a partial factorization (in terms of a set of coprimes and multiplicities) was found. The term "partial" refers to the fact that some found factors may not be prime. However the product of the found factors (taking into account their associated multiplicities) does yield the original number to factor.

COMPLETE_FACTORIZATION_FOUND Used by the factorization algorithm to indicate that a complete factorization (in terms of primes and multiplicities) was found.

NO_FACTOR_FOUND Used by the factorization algorithm to indicate that no factor was found.

FATAL_INTERNAL_ERROR Generic exit code used to indicate a serious internal error, possibly leading to an unpredictable behavior.

QUEUE_OVERFLOW Used by the SQUFOF algorithm to indicate that the queue overflowed, thus leading to give up the factorization process.

NO_PROPER_FORM_FOUND Used by the SQUFOF algorithm to indicate that no proper form was found, thus leading to give up the factorization process.

GIVING_UP Used to indicate that an abort limit has been reached leading to give up the current operation.

INTEGER_TOO_LARGE Used by the SQUFOF and Fermat/McKee implementations to indicate that the integer to factor is too large and cannot be processed.

SUCCESS Generic exit code used to indicate that an operation succeeded.

FAILURE Generic exit code used to indicate that an operation failed.

Definition at line 47 of file exit_codes.h.

5.9.4 Variable Documentation

5.9.4.1 const char* const ecode_to_str[14] `[static]`

Initial value:

```
{
    "unknown factoring mode",
    "some factors found",
    "some prime factors found",
    "some coprime factors found",
    "partial factorization found",
    "complete factorization found",
    "no factor found",
    "fatal internal error",
    "queue overflow",
    "no proper form found",
    "giving up",
    "number to factor is too large",
    "success",
    "failure"
}
```

Global constant array mapping exit codes to their string representations.

Definition at line 134 of file exit_codes.h.

5.10 factoring_machine.h File Reference

Abstraction of an integer factorization algorithm.

`#include <stdbool.h>`

`#include <gmp.h>`

`#include "array.h"`

`#include "exit_codes.h"`

Data Structures

- struct struct_factoring_machine

    Defines a structure to represent the logic behind all factorization algorithms.

Defines

- #define _TIFA_FACTORING_MACHINE_H_

Typedefs

- typedef enum factoring_mode_enum factoring_mode_t

    Equivalent to `struct factoring_mode_enum`.

- typedef struct struct_factoring_machine factoring_machine_t

    Equivalent to `struct_factoring_machine`.

Enumerations

- enum factoring_mode_enum {

SINGLE_RUN, FIND_SOME_FACTORS, FIND_SOME_COPRIME_FACTORS, FIND_SOME_-PRIME_FACTORS,
FIND_COMPLETE_FACTORIZATION }

Functions

- ecode_t run_machine (factoring_machine_t *machine)

    Attempt to factor an integer.

Variables

- static const int mode_to_outcome [5]

5.10.1 Detailed Description

Abstraction of an integer factorization algorithm.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Implements a machine-like abstraction of an integer factorization algorithm.

Definition in file factoring_machine.h.

5.10.2 Define Documentation

5.10.2.1 #define _TIFA_FACTORING_MACHINE_H_

Standard include guard.

Definition at line 35 of file factoring_machine.h.

5.10.3 Enumeration Type Documentation

5.10.3.1 enum factoring_mode_enum

An enumeration of the factoring mode available to the implemented factorization algorithm.

Enumerator:

SINGLE_RUN Perform only a single run of the factorization algorithm.

FIND_SOME_FACTORS Run the factorization algorithm until either some factors are found or the abort limit (defined on a per-algorithm basis) is reached.

FIND_SOME_COPRIME_FACTORS Run the factorization algorithm until either some coprime factors are found or the abort limit (defined on a per-algorithm basis) is reached.

FIND_SOME_PRIME_FACTORS Run the factorization algorithm until either some prime factors are found or the abort limit (defined on a per-algorithm basis) is reached.

Note:

This is probably not useful at all. Why would we discard found factors even if they are not prime? This should better be left to the client application.

FIND_COMPLETE_FACTORIZATION Run the factorization algorithm until either the complete factorization (as a product of prime numbers) is found or the abort limit (defined on a per-algorithm basis) is reached.

Definition at line 53 of file factoring_machine.h.

5.10.4 Function Documentation

5.10.4.1 ecode_t run_machine (factoring_machine_t ∗ *machine*)

Attempt to factor an integer.

Attempt to factor an integer with all parameters given by `machine`.

Note:

This function is meant to be a starting point for implementations of factorization algorithms and is obviously not intended to be directly used as a factoring function all by itself.

Parameters:

machine A pointer to the `factoring_machine_t` to use.

5.10.5 Variable Documentation

5.10.5.1 const int mode_to_outcome[5] `[static]`

Initial value:

```
 {
    SOME_FACTORS_FOUND,
    SOME_FACTORS_FOUND,
    SOME_COPRIME_FACTORS_FOUND,
    SOME_PRIME_FACTORS_FOUND,
    COMPLETE_FACTORIZATION_FOUND
}
```

Global constant array mapping factoring modes to their respective best outcome.

Definition at line 96 of file factoring_machine.h.

5.11 factoring_program.h File Reference

The logic common to all TIFA's factorization executable programs.

`#include <gmp.h>`

`#include "array.h"`

```
#include "exit_codes.h"
#include "factoring_machine.h"
```

Data Structures

- struct struct_factoring_program
  Defines a structure to represent the logic behind all factorization programs.

Defines

- #define _TIFA_FACTORING_PROGRAM_H_

Typedefs

- typedef struct struct_factoring_program factoring_program_t
  Equivalent to `struct struct_factoring_program`.

Functions

- ecode_t run_program (factoring_program_t *const program)
  Run a factoring program.

5.11.1 Detailed Description

The logic common to all TIFA's factorization executable programs.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Definition in file factoring_program.h.

5.11.2 Define Documentation

5.11.2.1 #define _TIFA_FACTORING_PROGRAM_H_

Standard include guard.

Definition at line 33 of file factoring_program.h.

5.11.3 Function Documentation

5.11.3.1 ecode_t run_program (factoring_program_t *const *program*)

Run a factoring program.

Run an actual factoring program from the command line.

Parameters:

program The `factoring_program_t` to run.

5.12 fermat.h File Reference

McKee's variant of the Fermat factorization algorithm.

```
#include <stdlib.h>
#include <gmp.h>
#include "array.h"
#include "factoring_machine.h"
#include "exit_codes.h"
```

Data Structures

- struct struct_fermat_params_t

  Defines the variable parameters used in Fermat's algorithm (dummy structure).

Defines

- #define _TIFA_FERMAT_H_

Typedefs

- typedef struct struct_fermat_params_t fermat_params_t

  Equivalent to `struct struct_fermat_params_t`.

Functions

- void set_fermat_params_to_default (fermat_params_t *const params)

  Fills a `fermat_params_t` *with default values (dummy function).*

- ecode_t fermat (mpz_array_t *const factors, uint32_array_t *const multis, const mpz_t n, const fermat_params_t *const params, const factoring_mode_t mode)

  Integer factorization via McKee's speedup of Fermat's factorization algorithm.

5.12.1 Detailed Description

McKee's variant of the Fermat factorization algorithm.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

This is the TIFA library's implementation of James McKee's proposed speedup of the Fermat factorization algorithm (SQUFOF), based on the description given by McKee in its paper "Speeding Fermat's Factoring Method".

Note:

This implementation can only factor numbers whose size is less than twice the size of an `unsigned long int`.

See also:

"Speeding Fermat's Factoring Method", James McKee. *Mathematics of Computation*, Volume 68, Number 228, pages 1729-1737.

Definition in file fermat.h.

5.12.2 Define Documentation

5.12.2.1 #define _TIFA_FERMAT_H_

Standard include guard.

Definition at line 44 of file fermat.h.

5.12.3 Function Documentation

5.12.3.1 ecode_t fermat (mpz_array_t ∗const *factors*, uint32_array_t ∗const *multis*, const mpz_t *n*, const fermat_params_t ∗const *params*, const factoring_mode_t *mode*)

Integer factorization via McKee's speedup of Fermat's factorization algorithm.

Attempts to factor the non perfect square integer `n` using James McKee's proposed enhancement of Fermat's algorithm, using the factoring mode given by `mode`. Found factors are then stored in `factors`. Additionally, if the factoring mode used is set to FIND_COMPLETE_FACTORIZATION, factors' multiplicities are stored in the array `multis`.

Warning:

This implementation can only factor numbers whose sizes in bits are strictly less than twice the size of an `unsigned long int`. This choice was made to maximize the use of single precision operations. Such a limitation should not be much of a problem since Fermat's algorithm is mostly used to factor very small integers (up to, say, 20 decimal digits).

Note:

If the factoring mode used is different from FIND_COMPLETE_FACTORIZATION, `multis` is allowed to be a NULL pointer. Otherwise, using a NULL pointer will lead to a fatal error.

Warning:

If the `factors` and `multis` arrays have not enough room to store the found factors (and the multiplicities, if any), they will be automatically resized to accommodate the data. This has to be kept in mind when trying to do ingenious stuff with memory management (hint: don't try to be clever here).

Parameters:

← ***factors*** Pointer to the found factors of `n`.

← ***multis*** Pointer to the multiplicities of the found factors (only computed if `mode` is set to FIND_-COMPLETE_FACTORIZATION).

→ ***n*** The non perfect square integer to factor.

→ ***params*** Fermat's algorithm parameters (currently unused).

→ ***mode*** The factoring mode to use.

Returns:

An exit code.

5.12.3.2 void set_fermat_params_to_default (fermat_params_t *const *params*)

Fills a `fermat_params_t` with default values (dummy function).

This function is intended to fill a `fermat_params_t` with default values.

Warning:

For the time being, this is a dummy function which does absolutely nothing at all, but is kept only as a placeholder should the need for user parameters arise in future code revisions.

Parameters:

params A pointer to the `fermat_params_t` structure to fill.

5.13 first_primes.h File Reference

Precomputed small primes.

```
#include <inttypes.h>
#include "array.h"
#include "tifa_config.h"
```

Defines

- #define _TIFA_FIRST_PRIMES_H_
- #define NFIRST_PRIMES 65536

Variables

- const uint32_t first_primes[NFIRST_PRIMES] MAYBE_UNUSED

5.13.1 Detailed Description

Precomputed small primes.

Author:

Automatically generated by genprimes.pl

Date:

Fri Jun 10 2011

Version:

2011-06-10

This is a list of the precomputed small primes together with a `uint32_array_t` wrapper.

Definition in file first_primes.h.

5.13.2 Define Documentation

5.13.2.1 #define _TIFA_FIRST_PRIMES_H_

Standard include guard.

Definition at line 36 of file first_primes.h.

5.13.2.2 #define NFIRST_PRIMES 65536

Number of precomputed primes in the `first_primes` array.

Definition at line 47 of file first_primes.h.

5.13.3 Variable Documentation

5.13.3.1 const uint32_array_t first_primes_array MAYBE_UNUSED

The `first_primes` array is a global array of `uint32_t` elements containing the first `NFIRST_PRIMES` prime numbers (from 2 and beyond).

The largest prime in the `first_primes` array.

`first_primes_array` is a `uint32_array_t` wrapper to the array `first_primes`.

Note:

`first_primes_array` 's `alloced` field is set to zero. Indeed, `first_primes_array` is merely a `uint32_array_t` wrapper for `first_primes`, and as such, it has no real "alloced" memory. Setting `first_primes_array.alloced` to 0 will prevent errors if `free_mpz_array` is inadvertently called on `first_primes_array`.

Definition at line 58 of file first_primes.h.

5.14 funcs.h File Reference

Number theoretical, hash and comparison functions.

`#include <limits.h>`

`#include <inttypes.h>`

`#include <math.h>`

`#include <gmp.h>`

`#include "macros.h"`

`#include "array.h"`

`#include "tifa_config.h"`

Data Structures

- struct struct_mult_data_t

    Ad hoc structure used in the computation of the multiplier to use.

Defines

- #define _TIFA_FUNCS_H_
- #define LARGEST_MULTIPLIER 97
- #define BITSIZE_LARGEST_MULTIPLIER 7
- #define MAX_IPRIME_IN_MULT_CALC 31
- #define NO_SQRT_MOD_P (UINT32_MAX)
- #define NO_SQRT_MOD_P2 (ULONG_MAX)

Typedefs

- typedef struct struct_mult_data_t mult_data_t

    Equivalent to `struct struct_mult_data_t`.

Functions

- uint32_t most_significant_bit (uint32_t n)

    Most significant bit of a positive integer.

- static uint32_t floor_log2 (uint32_t n)

    Floor of logarithm in base 2 of a positive integer.

- static uint32_t ceil_log2 (uint32_t n)

    Ceil of logarithm in base 2 of a positive integer.

- static uint32_t ceil_log2_mp_limb (mp_limb_t limb)

    Ceil of logarithm in base 2 of a `mp_limb_t`.

- void find_coprime_base (mpz_array_t *const base, const mpz_t n, const mpz_array_t *const factors)

  Find a coprime base from a list of factors.

- int8_t kronecker_ui (uint32_t a, uint32_t b)

  Kronecker symbol restricted to positive simple precision integers.

- uint32_t powm (uint32_t base, uint32_t power, uint32_t modulus)

  Modular exponentiation restricted to positive simple precision integers.

- uint32_t sqrtm (uint32_t a, uint32_t p)

  Shanks' algorithm for modular square roots computation.

- static unsigned long int is_square (unsigned long int x)

  Perfect square detection test.

- bool is_prime (uint32_t n)

  Composition test for `uint32_t` integers.

- unsigned long int gcd_ulint (unsigned long int a, unsigned long int b)

  Greatest common divisor for unsigned long int.

- unsigned long int modinv_ui (unsigned long int n, unsigned long int p)

  Modular inverse for unsigned long int.

- unsigned long int sqrtm_p2 (uint32_t a, uint32_t p)

  Modular square root modulo the square of a prime.

- uint32_t ks_multiplier (const mpz_t n, const uint32_t size_base)

  Find best multiplier using the Knuth-Schroeppel function.

- uint32_t hash_rj_32 (const void *const keyptr)

  Robert Jenkins' 32 bit mix function.

- uint32_t hash_pjw (const void *const keyptr)

  An hash function for strings.

- uint32_t hash_sfh_ph (const void *const keyptr)

  The "Super Fast Hash" function By Paul Hsieh.

- int mpz_cmp_func (const void *const mpza, const void *const mpzb)

  Comparison function between two `mpz_t`.

- int uint32_cmp_func (const void *const uinta, const void *const uintb)

  Comparison function between two `uint32_t`.

- int string_cmp_func (const void *const stra, const void *const strb)

  Comparison function between two strings.

- int cmp_mult_data (const void *mda, const void *mdb)

Comparison function between two `mult_data_t`.

- uint32_t n_choose_k (uint8_t n, uint8_t k)

  Binomial coefficient C(n, k) (n choose k).

- void next_subset_lex (uint32_t n, uint32_t k, uint32_t ∗subset, bool ∗end)

  Generate the successor of a fixed cardinal subset from a base set, in lexicographic order.

- void unrank_subset_lex (uint32_t n, uint32_t k, uint32_t r, uint32_t ∗subset)

  Generate a fixed cardinal subset from a base set, according to a given rank.

- void unrank_subset_revdoor (uint32_t n, uint32_t k, uint32_t r, uint32_t ∗subset)

  Generate a fixed cardinal subset from a base set, according to a given rank.

- void tifa_srand (uint32_t seed)

  Initializes TIFA's basic pseudo-random generator.

- uint32_t tifa_rand ()

  Returns a pseudo-random integer.

Variables

- const unsigned short qres_mod_221[221] MAYBE_UNUSED

  Quadratic residues mod 221.

5.14.1 Detailed Description

Number theoretical, hash and comparison functions.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Defines several number theoretical functions, hash functions and comparison functions.

Definition in file funcs.h.

5.14.2 Define Documentation

5.14.2.1 #define _TIFA_FUNCS_H_

Standard include guard.

Definition at line 36 of file funcs.h.

5.14.2.2 #define BITSIZE_LARGEST_MULTIPLIER 7

Size in bits of the largest multiplier allowed.

Definition at line 61 of file funcs.h.

5.14.2.3 #define LARGEST_MULTIPLIER 97

Largest multiplier allowed.

Definition at line 55 of file funcs.h.

5.14.2.4 #define MAX_IPRIME_IN_MULT_CALC 31

The `MAX_IPRIME_IN_MULT_CALC`-th smallest prime number is the largest prime used in the determination of the best multiplier.

Definition at line 68 of file funcs.h.

5.14.2.5 #define NO_SQRT_MOD_P (UINT32_MAX)

Value returned by the `sqrtm(n, p)` function if no modular square root of `n` mod `p` exits.

Definition at line 248 of file funcs.h.

5.14.2.6 #define NO_SQRT_MOD_P2 (ULONG_MAX)

Value returned by the `sqrtm_p2(n, p)` function if no modular square root of `n` mod `p*p` exits.

Definition at line 255 of file funcs.h.

5.14.3 Function Documentation

5.14.3.1 static uint32_t ceil_log2 (uint32_t *n*) `[inline, static]`

Ceil of logarithm in base 2 of a positive integer.

Returns the value of the ceil of the logarithm (in base 2) of a positive integer in essentially constant time. In other words, it returns the smallest natural `i` such that `2^i >= n`.

Parameters:

← ***n*** A positive integer.

Returns:

Ceil of log(`n`) in base 2.

Definition at line 162 of file funcs.h.

References IS_POWER_OF_2_UI, and most_significant_bit().

Referenced by ceil_log2_mp_limb().

5.14.3.2 static uint32_t ceil_log2_mp_limb (mp_limb_t *limb*) `[inline, static]`

Ceil of logarithm in base 2 of a `mp_limb_t`.

Returns the value of the ceil of the logarithm (in base 2) of a `mp_limb_t` in essentially constant time. In other words, it returns the smallest natural `i` such that `2^i >= n`.

Parameters:

← ***n*** A positive integer as an `mp_limb_t`.

Returns:

Ceil of log(`n`) in base 2.

Definition at line 180 of file funcs.h.

References ceil_log2().

5.14.3.3 int cmp_mult_data (const void ∗ *mda*, const void ∗ *mdb*)

Comparison function between two `mult_data_t`.

This is a comparison function between two `mult_data_t` structures passed as pointers to `void`, according to the criteria set forth in Morrison and Brillhart's paper "A Method of Factoring and the Factorization of F_7" (Mathematics of Computation, vol 29, #129, Jan 1975, pages 183-205).

If `a` and `b` are the two underlying `mult_data_t` structures to compare, it returns:

- 1 if `a.count` > `b.count`
- -1 if `a.count` < `b.count`
- If `a.count` == `b.count`, returns:
  - 1 if `a.sum_inv_pi` > `b.sum_inv_pi`
  - -1 if `a.sum_inv_pi` > `b.sum_inv_pi`
  - If `a.sum_inv_pi` == `b.sum_inv_pi`, returns:
  -
    * 1 if `a.multiplier` < `b.multiplier` (Indeed, we prefer smaller multipliers)
    * -1 if `a.multiplier` > `b.multiplier`
    * 0 if if `a.multiplier` > `b.multiplier`

Parameters:

← ***mda*** A pointer to the first `mult_data_t` to compare.

← ***mdb*** A pointer to the second `mult_data_t` to compare.

Returns:

The comparison between the two `mult_data_t`.

5.14.3.4 void find_coprime_base (mpz_array_t ∗const *base*, const mpz_t *n*, const mpz_array_t ∗const *factors*)

Find a coprime base from a list of factors.

Finds a coprime base for the list of factors of `n` given by the array `*factors` and stores it in the allocated but *uninitialized* array `base`. After invocation, we know that `n` is smooth on the returned computed base and that all elements of the base are coprime to each other.

The resulting base is obtained:

1. by completing the list of original factors with their cofactors,

2. by keeping only factors (or non-trivial divisors of factors) coprime to all others.

Warning:

There is absolutely no guarantee that the returned base elements are prime. If, by chance, the base only contains primes then it means that we have found the complete factorization of `n` (up to the prime multiplicities).

Note:

If the `base` array has not enough room to hold all the coprimes found, it will be resized via a call to `resize_mpz_array` with `ELONGATION` extra `mpz_t` slots to avoid too frequent resizes.
Consecutive invocation of this function with the same `base` and `n` but for different `factors` arrays will build a coprime base for all elements in all the aforementioned `factors` arrays.

Parameters:

↔ ***base*** The found coprime base.
← ***n*** A positive integer.
← ***factors*** A pointer to an array holding some factors of `n`.
↔ ***A*** pointer to the *unintialized* `mpz_array_t` to hold the coprime base.

5.14.3.5 static uint32_t floor_log2 (uint32_t *n*) `[inline, static]`

Floor of logarithm in base 2 of a positive integer.

Returns the value of the floor of the logarithm (in base 2) of a positive integer in essentially constant time. In other words, it returns the greatest natural `i` such that `2^i <= n`.

Note:

This is actually just a call to `most_significant_bit`.

Parameters:

← ***n*** A positive integer.

Returns:

Floor of log(`n`) in base 2.

Definition at line 148 of file funcs.h.

References most_significant_bit().

5.14.3.6 unsigned long int gcd_ulint (unsigned long int *a*, unsigned long int *b*)

Greatest common divisor for unsigned long int.

Returns the greatest common divisor of `a` and `b` as an unsigned long int.

Parameters:

← ***a*** An unsigned long int.
← ***b*** An unsigned long int.

Returns:

The greatest common divisor of `a` and `b`.

5.14.3.7 uint32_t hash_pjw (const void $*$const *keyptr*)

An hash function for strings.

Returns the hash of a C-style character string (passed as a pointer to `void`) using an hash function attributed to P.J. Weinberger.

Note:

This hash function and its implementation is extracted from the famous Dragon book: "Compilers: Principles, Techniques and Tools", Aho, Sethi, & Ullman.

Parameters:

← ***keyptr*** A pointer to the character string to hash.

Returns:

The value of the hash function.

5.14.3.8 uint32_t hash_rj_32 (const void $*$const *keyptr*)

Robert Jenkins' 32 bit mix function.

Returns the hash of a uint32_t integer (passed as a pointer to `void`) using Robert Jenkins' 32 bit mix function.

See also:

http://www.concentric.net/~Ttwang/tech/inthash.htm

Parameters:

← ***keyptr*** A pointer to the `uint32_t` to hash.

Returns:

The value of the hash function.

5.14.3.9 uint32_t hash_sfh_ph (const void $*$const *keyptr*)

The "Super Fast Hash" function By Paul Hsieh.

Returns the hash of a C-style character string (passed as a pointer to `void`) using the so-called "SuperFastHash" function By Paul Hsieh.

See also:

http://www.azillionmonkeys.com/qed/hash.html

Parameters:

← ***keyptr*** A pointer to the character string to hash.

Returns:

The value of the hash function.

5.14.3.10 bool is_prime (uint32_t *n*)

Composition test for `uint32_t` integers.

Returns `false` if `n` is definitely composite. Returns `true` if `n` is *probably* prime.

Note:

This is actually a basic Miller-Rabin composition test with `NMILLER_RABIN` iterations preceded with some trial divisions if `n` is sufficiently small.

Parameters:

← ***n*** The `uint32_t` to be checked for composition.

Returns:

Returns `false` if `n` is found to be definitely composite. `true` otherwise.

5.14.3.11 static unsigned long int is_square (unsigned long int *x*) `[inline, static]`

Perfect square detection test.

Returns sqrt(`x`) if and only if `x` is a perfect square. Returns 0 otherwise.

Parameters:

← ***x*** The integer to test.

Returns:

sqrt(`x`) if `x` is a perfect square. 0 otherwise.

Definition at line 335 of file funcs.h.

5.14.3.12 int8_t kronecker_ui (uint32_t *a*, uint32_t *b*)

Kronecker symbol restricted to positive simple precision integers.

Returns the value of the Kronecker symbol (`a`/`b`) where `a` and `b` are positive integers.

Parameters:

← ***a*** A positive integer.

← ***b*** A positive integer.

Returns:

The value of the kronecker symbol (`a`/`b`).

5.14.3.13 uint32_t ks_multiplier (const mpz_t *n*, const uint32_t *size_base*)

Find best multiplier using the Knuth-Schroeppel function.

Given the size of factor base `size_base`, returns the "best" multiplier to factor `n`, using the modified version of the Knuth-Schroeppel function described by Silverman in: "The Multiple Quadratic Sieve".

Note:

The greatest multiplier considered is given by LARGEST_MULTIPLIER.

See also:

"The Multiple Quadratic Sieve", Robert D. Silverman, *Mathematics of Computation*, Volume 48, Number 177, January 1987, pages 329-339.

Parameters:

← ***n*** The number to factor

← ***size_base*** The desired size of the factor base

Returns:

The "best" multiplier to factor n.

5.14.3.14 unsigned long int modinv_ui (unsigned long int *n*, unsigned long int *p*)

Modular inverse for unsigned long int.

Returns the modular inverse of n modulo the odd prime p as an unsigned long int.

Warning:

p must be a positive odd prime, strictly less than LONG_MAX (yes, LONG_MAX and not ULONG_MAX!) and, of course, n % p must be non-null.

Parameters:

← ***n*** An unsigned long int.

← ***p*** An odd prime unsigned long int.

Returns:

The modular inverse of n mod p.

5.14.3.15 uint32_t most_significant_bit (uint32_t *n*)

Most significant bit of a positive integer.

Returns the value of the most significant bit of the integer n in essentially constant time, or in other words, its logarithm in base 2. The returned result is an integer from 0 (the least significant bit) to 31 included (the most significant bit).

Note:

This function is adapted from public domain code from the Bit Twiddling Hacks web page: http://graphics.stanford.edu/~seander/bithacks.html

Parameters:

← ***n*** A positive integer.

Returns:

log(n) in base 2.

Referenced by ceil_log2(), and floor_log2().

5.14.3.16 int mpz_cmp_func (const void ∗const *mpza*, const void ∗const *mpzb*)

Comparison function between two `mpz_t`.

This is a natural order comparison function between two `mpz_t` elements passed as pointers to `void`. It returns:

- 1 if the first `mpz_t` is greater than the second one.
- 0 if the first `mpz_t` is equal to the second one.
- -1 if the first `mpz_t` is less than the second one.

Note:

This function is actually nothing more than a wrapper for `mpz_cmp`.

Parameters:

← ***mpza*** A pointer to a `mpz_t`.
← ***mpzb*** A pointer to another `mpz_t`.

Returns:

The comparison between the two `mpz_t`.

5.14.3.17 uint32_t n_choose_k (uint8_t *n*, uint8_t *k*)

Binomial coefficient C(n, k) (n choose k).

Returns the binomial coefficient C(`n`, `k`) (i.e `n` choose `k`).

Note that this single precision function only returns correct results if the actual value of the binomial coefficient fits in 32 bits.

Parameters:

← ***n***
← ***k***

Returns:

The binomial coefficient C(n, k).

5.14.3.18 void next_subset_lex (uint32_t *n*, uint32_t *k*, uint32_t ∗ *subset*, bool ∗ *end*)

Generate the successor of a fixed cardinal subset from a base set, in lexicographic order.

Starting with a subset of cardinal `k` of a base set of cardinal `n`, generates the subset's successor in the lexicographic order. The new subset is stored in `subset` and thus overrides the previous one.

Subsets are decribed by an array of length `k` holding indexes in the interval [1, `n`].

The first `k`-subset in the lexicographic order is given by {1, 2, 3, ... `k`}. After a call to `next_subset_lex` `end` is `true` if and only if the last `k`-subset has been reached (i.e. the next one will be {1, 2, 3, ... `k`}).

This is actually algorithm 2.6 from the book *"Combinatorial Algorithms - Generation, Enumeration, and Search"* by Donald L. Kreher and Douglas Stinson.

Parameters:

← ***n*** Cardinal of the base set.

← ***k*** Cardinal of the subset.

in/out] subset Current subset to be replaced by its successor.

← ***end*** Have we reached the end of the cycle?

5.14.3.19 uint32_t powm (uint32_t *base*, uint32_t *power*, uint32_t *modulus*)

Modular exponentiation restricted to positive simple precision integers.

Returns (`base^power`) mod `modulus` as an unsigned integer.

Parameters:

← ***base*** The base of the modular exponential.

← ***power*** The power of the modular exponential.

← ***modulus*** The modulus of the modular exponential.

Returns:

The modular exponential (`base^power`) mod `modulus`.

5.14.3.20 uint32_t sqrtm (uint32_t *a*, uint32_t *p*)

Shanks' algorithm for modular square roots computation.

Returns the modular square root of `a` (mod `p`) (where `p` is an *odd prime*) using Shanks' algorithm, that is, returns the positive integer `s` such that `s^2` = `a` (mod `p`). If no such integer exists, returns `NO_SQRT_MOD_P`.

Warning:

The primality of `p` is not checked by `sqrtm`. It is the responsability of the caller to check whether `p` is indeed prime. Failure to assure such a precondition will lead to an infinite loop.

Parameters:

← ***a*** The modular square.

← ***p*** The modulus.

Returns:

The modular square root of `a` (mod `p`) if it exists. `NO_SQRT_MOD_P` otherwise.

5.14.3.21 unsigned long int sqrtm_p2 (uint32_t *a*, uint32_t *p*)

Modular square root modulo the square of a prime.

Provided that `a` verifies 1 <= `a` < `p*p`, returns the modular square root of `a` (mod `p*p`) (where `p` is an *odd prime*) that is, returns a positive integer `s` such that `s^2` = `a` (mod `p*p`). If no such integer exists, returns `NO_SQRT_MOD_P2`.

Warning:

In order to use only single precision computation, the product `p*p` should be strictly less than `LONG_-MAX`.
The primality of `p` is not checked by `sqrtm_p2`. It is the responsability of the caller to check whether `p` is indeed prime. Failure to assure such a precondition will lead to an infinite loop.

Parameters:

← ***a*** The modular square.

← ***p*** The square root of the modulus.

Returns:

The modular square root of `a` (mod `p*p`) if it exists. `NO_SQRT_MOD_P2` otherwise.

5.14.3.22 int string_cmp_func (const void ∗const *stra*, const void ∗const *strb*)

Comparison function between two strings.

This is a lexicographical order comparison function between two C-style character strings passed as pointers to `void`. It returns:

- 1 if the first string is greater than the second one.
- 0 if the first string is equal to the second one.
- -1 if the first string is less than the second one.

Note:

This function is actually nothing more than a wrapper for `strcmp`.

Parameters:

← ***stra*** A pointer to a C-style character string.

← ***strb*** A pointer to another C-style character string.

Returns:

The lexicographical comparison between the two strings.

5.14.3.23 uint32_t tifa_rand ()

Returns a pseudo-random integer.

Returns a pseudo-random integer using TIFA's basic random number generator.

Parameters:

← ***seed*** The seed as a `uint32_t`.

5.14.3.24 void tifa_srand (uint32_t *seed*)

Initializes TIFA's basic pseudo-random generator.

Initializes TIFA's basic random number generator with a user defined seed.

Parameters:

← ***seed*** The seed as a `uint32_t`.

5.14.3.25 int uint32_cmp_func (const void ∗const *uinta*, const void ∗const *uintb*)

Comparison function between two `uint32_t`.

This is a natural order comparison function between two `uint32_t` elements passed as pointers to `void`. It returns:

- 1 if the first `uint32_t` is greater than the second one.
- 0 if the first `uint32_t` is equal to the second one.
- -1 if the first `uint32_t` is less than the second one.

Parameters:

← ***uinta*** A pointer to a `uint32_t`.

← ***uintb*** A pointer to another `uint32_t`.

Returns:

The comparison between the two `uint32_t`.

5.14.3.26 void unrank_subset_lex (uint32_t *n*, uint32_t *k*, uint32_t *r*, uint32_t ∗ *subset*)

Generate a fixed cardinal subset from a base set, according to a given rank.

Starting with a base set of cardinal `n`, constructs a subset of cardinal `k` and rank `r` (in [0, c(`n`, `k`)]) where the rank is given by the lexicographic order. The constructed subset is stored in `subset` and thus overrides the previous data.

Subsets are decribed by an array of length `k` holding indexes in the interval [1, `n`].

This is actually algorithm 2.8 from the book *"Combinatorial Algorithms - Generation, Enumeration, and Search"* by Donald L. Kreher and Douglas Stinson.

Parameters:

← ***n*** Cardinal of the base set.

← ***k*** Cardinal of the subset.

← ***r*** Rank of the subset (assuming lexicographic order).

→ ***subset*** Subset to be returned.

5.14.3.27 void unrank_subset_revdoor (uint32_t *n*, uint32_t *k*, uint32_t *r*, uint32_t * *subset*)

Generate a fixed cardinal subset from a base set, according to a given rank.

Starting with a base set of cardinal `n`, constructs a subset of cardinal `k` and rank `r` (in [0, c(`n`, `k`)]) where the rank is given by the minimal change order. The constructed subset is stored in `subset` and thus overrides the previous data.

Subsets are decribed by an array of length `k` holding indexes in the interval [1, `n`].

This is actually algorithm 2.12 from the book *"Combinatorial Algorithms - Generation, Enumeration, and Search"* by Donald L. Kreher and Douglas Stinson.

Parameters:

← ***n*** Cardinal of the base set.

← ***k*** Cardinal of the subset.

← ***r*** Rank of the subset (assuming minimal change order).

→ ***subset*** Subset to be returned.

5.14.4 Variable Documentation

5.14.4.1 const uint32_array_t first_primes_array MAYBE_UNUSED

Quadratic residues mod 221.

Quadratic residues mod 315.

Quadratic residues mod 256.

`x` is a square mod 221 if `qres_mod_221[x % 221] == 1`.

`x` is a square mod 256 if `qres_mod_256[x % 256] == 1`.

`x` is a square mod 315 if `qres_mod_315[x % 315] == 1`.

The largest prime in the `first_primes` array.

`first_primes_array` is a `uint32_array_t` wrapper to the array `first_primes`.

Note:

`first_primes_array` 's `alloced` field is set to zero. Indeed, `first_primes_array` is merely a `uint32_array_t` wrapper for `first_primes`, and as such, it has no real "alloced" memory. Setting `first_primes_array.alloced` to 0 will prevent errors if `free_mpz_-array` is inadvertently called on `first_primes_array`.

Definition at line 317 of file funcs.h.

5.15 gauss_elim.h File Reference

Gaussian elimination over GF(2) (from a paper by D. Parkinson and M. Wunderlich).

```
#include <inttypes.h>
#include "matrix.h"
#include "x_array_list.h"
```

Defines

- #define _TIFA_GAUSS_ELIM_H_

Functions

- void gaussian_elim (uint32_array_list_t *relations, binary_matrix_t *const matrix)
  Gaussian elimination on a `binary_matrix_t`.

5.15.1 Detailed Description

Gaussian elimination over GF(2) (from a paper by D. Parkinson and M. Wunderlich).

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Gaussian elimination over GF(2) as presented in the paper "A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers" written by Dennis Parkinson and Marvin Wunderlich (Parallel Computing 1, 1984).

See also:

"A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers", D. Parkinson and M. Wunderlich, *Parallel Computing 1*, 1984, pages 65-73.

Definition in file gauss_elim.h.

5.15.2 Define Documentation

5.15.2.1 #define _TIFA_GAUSS_ELIM_H_

Standard include guard.

Definition at line 43 of file gauss_elim.h.

5.15.3 Function Documentation

5.15.3.1 void gaussian_elim (uint32_array_list_t * *relations*, binary_matrix_t *const *matrix*)

Gaussian elimination on a `binary_matrix_t`.

Performs a gaussian elimination on a `binary_matrix_t` as described in the paper "A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers", by D. Parkinson and M. Wunderlich (Parallel Computing 1 (1984) 65-73).

Solutions (if any) of this linear system are stored in `relations` where each entry is a `uint32_array_t` containing the indexes of the rows (from the *original* matrix) composing a solution. In other words, for each entry, the sum of the indexed rows (from the original matrix) is a nul binary vector.

Parameters:

← ***matrix*** A pointer to the `binary_matrix_t` giving the linear system to solve.

→ ***relations*** A pointer to a `uint32_array_list_t` holding the solutions of the system, if any.

5.16 gmp_utils.h File Reference

Various GMP small utilities.

`#include <gmp.h>`

`#include "hashtable.h"`

Data Structures

- struct struct_mpz_pair_t

  A pair of `mpz_t` *integers.*

Defines

- #define _TIFA_GMP_UTILS_H_

Typedefs

- typedef struct struct_mpz_pair_t mpz_pair_t

  Equivalent to `struct struct_mpz_pair_t`.

Functions

- static void init_mpz_pair (mpz_pair_t ∗pair)

  inits a `mpz_pair_t`.

- static void clear_mpz_pair (mpz_pair_t ∗pair)

  Clears a `mpz_pair_t`.

- void empty_mpzpair_htable (hashtable_t ∗const htable)

  Empties a `hashtable_t` *holding* `mpz_pair_t`*'s.*

- static void free_mpzpair_htable (hashtable_t ∗htable)

  Clears a `hashtable_t` *holding* `mpz_pair_t`*'s.*

- float mpz_log10 (mpz_t n)

  Logarithm in base 10 of a multi-precision integer.

5.16.1 Detailed Description

Various GMP small utilities.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

GMP small utilities' definitions should go here.

Definition in file gmp_utils.h.

5.16.2 Define Documentation

5.16.2.1 #define _TIFA_GMP_UTILS_H_

Standard include guard.

Definition at line 35 of file gmp_utils.h.

5.16.3 Function Documentation

5.16.3.1 static void clear_mpz_pair (mpz_pair_t * *pair*) `[inline, static]`

Clears a `mpz_pair_t`.

Clears a `mpz_pair_t`.

Parameters:

← ***pair*** A pointer to the `mpz_pair_t` to clear.

Definition at line 87 of file gmp_utils.h.

References struct_mpz_pair_t::x, and struct_mpz_pair_t::y.

5.16.3.2 void empty_mpzpair_htable (hashtable_t *const *htable*)

Empties a `hashtable_t` holding `mpz_pair_t`'s.

Empties a `hashtable_t` holding `mpz_pair_t`'s and clears the memory associated to the keys and their associated data.

Parameters:

← ***htable*** A pointer to the `hashtable_t` to empty.

Referenced by free_mpzpair_htable().

5.16.3.3 static void free_mpzpair_htable (hashtable_t * *htable*) [inline, static]

Clears a hashtable_t holding mpz_pair_t's.

Clears a hashtable_t holding mpz_pair_t's. It clears the memory associated to the keys, their associated data and the hashtable itself.

Parameters:

← ***htable*** A pointer to the hashtable_t to clear.

Definition at line 111 of file gmp_utils.h.

References empty_mpzpair_htable(), and free_hashtable().

5.16.3.4 static void init_mpz_pair (mpz_pair_t * *pair*) [inline, static]

inits a mpz_pair_t.

Inits a mpz_pair_t by initializing each of its mpz_t element.

Parameters:

← ***pair*** A pointer to the mpz_pair_t to init.

Definition at line 75 of file gmp_utils.h.

References struct_mpz_pair_t::x, and struct_mpz_pair_t::y.

5.16.3.5 float mpz_log10 (mpz_t *n*)

Logarithm in base 10 of a multi-precision integer.

Returns an *crude approximation* of the logarithm (in base 10) of a positive multi-precision integer n. The approximation is usually valid up to the 4th decimal.

Parameters:

← ***n*** A positive multi-precision integer.

Returns:

An approximation of log(n) in base 10.

5.17 hashtable.h File Reference

Generic hashtable.

```
#include <inttypes.h>
#include "linked_list.h"
```

Data Structures

- struct struct_hashtable_t

  A basic implementation of a hashtable.

- struct struct_hashtable_entry_t

  The structure of a hashtable's entry.

Defines

- #define _TIFA_HASHTABLE_H_

Typedefs

- typedef struct struct_hashtable_t hashtable_t

  Equivalent to `struct struct_hashtable_t`.

- typedef struct struct_hashtable_entry_t hashtable_entry_t

  Equivalent to `struct struct_hashtable_entry_t`.

Functions

- hashtable_t * alloc_init_hashtable (uint32_t size, int(*cmp_func)(const void *const key_a, const void *const key_b), uint32_t(*hash_func)(const void *const key))

  Allocates and returns a new `hashtable_t`.

- void free_hashtable (hashtable_t *htable)

  Clears a `hashtable_t`.

- void add_entry_in_hashtable (hashtable_t *const htable, const void *const key, const void *const data)

  Adds an entry in a `hashtable_t`.

- void * get_entry_in_hashtable (hashtable_t *const htable, const void *const key)

  Gets an entry's data from a `hashtable_t`.

- void * remove_entry_in_hashtable (hashtable_t *const htable, const void *const key)

  Removes an entry from a `hashtable_t`.

5.17.1 Detailed Description

Generic hashtable.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Yet another implementation of a generic hashtable.

Definition in file hashtable.h.

5.17.2 Define Documentation

5.17.2.1 #define _TIFA_HASHTABLE_H_

Standard include guard.

Definition at line 35 of file hashtable.h.

5.17.3 Function Documentation

5.17.3.1 void add_entry_in_hashtable (hashtable_t ∗const *htable*, const void ∗const *key*, const void ∗const *data*)

Adds an entry in a `hashtable_t`.

Creates an entry with its `data` field given by the pointer `data` and its `key` field given by the pointer `key` and adds it in a `hashtable_t`.

Warning:

This function does not copy the actual content of the variable referenced by `key` and `data` but merely copies these references in the `hashtable_entry_t` structure.

Parameters:

← ***htable*** A pointer to the `hashtable_t`.

← ***key*** A pointer to the key of the new entry.

← ***data*** A pointer to the data of the new entry.

5.17.3.2 hashtable_t∗ alloc_init_hashtable (uint32_t *size*, int(∗)(const void ∗const key_a, const void ∗const key_b) *cmp_func*, uint32_t(∗)(const void ∗const key) *hash_func*)

Allocates and returns a new `hashtable_t`.

Allocates and returns a pointer to a new `hashtable_t` with `size` allocated buckets, using the comparison function pointed by `cmp_func` and the hash function given by `hash_func`.

Note:

If `size` is not a power of two, the lowest power of two greater than `size` is used instead.

Parameters:

← ***size*** The number of allocated buckets.

← ***cmp_func*** A pointer to the comparison function.

← ***hash_func*** A pointer to the hash function.

5.17.3.3 void free_hashtable (hashtable_t ∗ *htable*)

Clears a `hashtable_t`.

Clears the `hashtable_t` pointed by `htable`.

Warning:

This function merely clears the memory used by the linked lists but *does not* frees the memory space used by the `key` and `data` pointers of the `hashtable_entry_t`.
Do not call `free(htable)` in client code after a call to `free_hashtable(htable)`: it would result in an error.

Parameters:

← ***htable*** A pointer to the `hashtable_t` to clear.

Referenced by free_mpzpair_htable().

5.17.3.4 void∗ get_entry_in_hashtable (hashtable_t ∗const *htable*, const void ∗const *key*)

Gets an entry's data from a `hashtable_t`.

Gets the `data` field of the entry from the hashtable `htable` whose key is given by `key`.

Parameters:

← ***htable*** A pointer to the `hashtable_t`.

← ***key*** A pointer to the key of the entry to read.

Returns:

The `data` field of the entry whose key is given by `key`, if any.
NULL if no such entry is found in the hashtable.

5.17.3.5 void∗ remove_entry_in_hashtable (hashtable_t ∗const *htable*, const void ∗const *key*)

Removes an entry from a `hashtable_t`.

Removes the entry from the hashtable `htable` whose key is given by `key` and returns its `data` field.

Warning:

The `key` field of the removed entry is freed if a matching entry is found. The means that if `key` is a pointer to a structure containing some other pointers, all the memory may not be freed since the real type of `key` is not known by the hashtable. This is why it is strongly recommended to use *only* pointers to integers or strings as keys.

Parameters:

← ***htable*** A pointer to the `hashtable_t`.

← ***key*** A pointer to the key of the entry to remove.

Returns:

The `data` field of the removed entry.
NULL if no matching entry is found in the hashtable.

5.18 lindep.h File Reference

Functions used in the resolution of the linear systems.

```
#include <inttypes.h>
#include "array.h"
#include "x_array_list.h"
#include "matrix.h"
#include "exit_codes.h"
```

Defines

- #define _TIFA_LINDEP_H_

Typedefs

- typedef enum linalg_method_enum linalg_method_t

    Equivalent to `enum linalg_method_enum`.

Enumerations

- enum linalg_method_enum { SMART_GAUSS_ELIM = 0 }

Functions

- void fill_matrix_trial_div (binary_matrix_t ∗const matrix, mpz_array_t ∗const partially_factored, const mpz_array_t ∗const to_factor, const uint32_array_t ∗const factor_base)

    Fills a binary matrix via trial divisions.

- void fill_trial_div_decomp (binary_matrix_t ∗const matrix, byte_matrix_t ∗const decomp_matrix, mpz_array_t ∗const partially_factored, const mpz_array_t ∗const to_factor, const uint32_array_t ∗const factor_base)

    Similar to `fill_matrix_trial_div` *but also stores valuations.*

- void fill_matrix_from_list (binary_matrix_t ∗const matrix, const mpz_array_t ∗const smooth_array, const uint32_array_list_t ∗const list, const uint32_array_t ∗const factor_base)

    Fills a binary matrix from a list of factors.

- void fill_matrix_from_list_decomp (binary_matrix_t ∗const matrix, byte_matrix_t ∗const decomp_matrix, const mpz_array_t ∗const smooth_array, const uint32_array_list_t ∗const list, const uint32_array_t ∗const factor_base)

    Similar to `fill_matrix_from_list` *but also stores valuations.*

- uint32_array_list_t ∗ find_dependencies (binary_matrix_t ∗const matrix, linalg_method_t method)

    Solves a linear system over GF(2).

- ecode_t find_factors (mpz_array_t ∗const factors, const mpz_t n, const mpz_array_t ∗const xi_array, const mpz_array_t ∗const yi_array, const uint32_array_list_t ∗const dependencies)

Find factors of an integer from congruence relations.

- ecode_t find_factors_decomp (mpz_array_t ∗const factors, const mpz_t n, const mpz_array_t ∗const xi_array, const byte_matrix_t ∗const yi_decomp_matrix, const uint32_array_list_t ∗const dependencies, const uint32_array_t ∗const factor_base)

  Similar to `find_factors` *but uses the factorization of each* `y_i`.

Variables

- static const char ∗const linalg_method_to_str [1]

5.18.1 Detailed Description

Functions used in the resolution of the linear systems.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Functions used in the resolution of the linear systems over GF(2) found in factorization problems and in the very last stage of the factorization process (determination of factors after linear algebra phase).

Definition in file lindep.h.

5.18.2 Define Documentation

5.18.2.1 #define _TIFA_LINDEP_H_

Standard include guard.

Definition at line 37 of file lindep.h.

5.18.3 Enumeration Type Documentation

5.18.3.1 enum linalg_method_enum

Enumeration listing the different linear system resolution method implemented.

For the time being, only one method is available.

Enumerator:

SMART_GAUSS_ELIM "Smart" gaussian elimination described in: "A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers", by D. Parkinson and M. Wunderlich (Parallel Computing 1 (1984) 65-73).

Definition at line 58 of file lindep.h.

5.18.4 Function Documentation

5.18.4.1 void fill_matrix_from_list (binary_matrix_t ∗const *matrix*, const mpz_array_t ∗const *smooth_array*, const uint32_array_list_t ∗const *list*, const uint32_array_t ∗const *factor_base*)

Fills a binary matrix from a list of factors.

Fills the binary matrix `matrix` from a previously computed list giving all known factors.

Note:

`list` countains the previously computed factors of each integers in `smooth_array`, in other words, we know that `smooth_array->data[i]` is divisible by all the integers of the `uint32_array_t` given by `list->data[i]`.

The binary matrix is filled so that:

- There is a '1' in the (i-th row, 1st col) position in the matrix if `smooth_factor->data[i]` is negative.
- There is a '1' in the (i-th row, j-th col) position in the matrix if `smooth_factor->data[i]` is divisible by an odd power of `list->data[i]->data[k]`. Here j is found so that `factor_base->data[j] = list->data[i]->data[k]`.
- In all other cases, the (i-th row, j-th col) position in the matrix contains a 0.

Parameters:

→ ***matrix*** A pointer to the binary matrix to fill.

→ ***smooth_array*** A pointer to the array giving the integers to factor.

← ***list*** A pointer to the factor list for each integer to factor.

← ***factor_base*** A pointer to the array listing the integers to trial divide by.

5.18.4.2 void fill_matrix_from_list_decomp (binary_matrix_t ∗const *matrix*, byte_matrix_t ∗const *decomp_matrix*, const mpz_array_t ∗const *smooth_array*, const uint32_array_list_t ∗const *list*, const uint32_array_t ∗const *factor_base*)

Similar to `fill_matrix_from_list` but also stores valuations.

Fills the binary matrix `matrix` from a previously computed list giving all known factors.

Also stores in `decomp_matrix` the valuation of each integer from `smooth_array` for each prime in the factor base. For example the valuation of `smooth_array->data[i]` for the prime given by `factor_base->data[j]` will be stored in `decomp_matrix->data[i][j]`.

Note:

`list` countains the previously computed factors of each integers in `smooth_array`, in other words, we know that `smooth_array->data[i]` is divisible by all the integers of the `uint32_array_t` given by `list->data[i]`.

The binary matrix is filled so that:

- There is a '1' in the (i-th row, 1st col) position in the matrix if `smooth_factor->data[i]` is negative.

- There is a '1' in the (i-th row, j-th col) position in the matrix if `smooth_factor->data[i]` is divisible by an odd power of `list->data[i]->data[k]`. Here j is found so that `factor_base->data[j] = list->data[i]->data[k]`.
- In all other cases, the (i-th row, j-th col) position in the matrix contains a 0.

Parameters:

→ ***matrix*** A pointer to the binary matrix to fill.

→ ***decomp_matrix*** A pointer to the byte matrix to fill.

→ ***smooth_array*** A pointer to the array giving the integers to factor.

← ***list*** A pointer to the factor list for each integer to factor.

← ***factor_base*** A pointer to the array listing the integers to trial divide by.

5.18.4.3 void fill_matrix_trial_div (binary_matrix_t ∗const *matrix*, mpz_array_t ∗const *partially_factored*, const mpz_array_t ∗const *to_factor*, const uint32_array_t ∗const *factor_base*)

Fills a binary matrix via trial divisions.

Fills the binary matrix `matrix` by trial divisions of the integers listed in `to_factor` by the integers listed in `factor_base`. After these trial divisions, each partially factored integer from `to_factor` are stored in `partially_factored`.

Note:

The binary matrix is filled so that:

- There is a '1' in the (i-th row, 1st col) position in the matrix if `to_factor->data[i]` is negative.
- There is a '1' in the (i-th row, j-th col) position in the matrix if `to_factor->data[i]` is divisible by an odd power of `factor_base->data[j]`.
- In all other cases, the (i-th row, j-th col) position in the matrix contains a 0.

Parameters:

→ ***matrix*** A pointer to the binary matrix to fill.

→ ***partially_factored*** A pointer to the partially factored integers.

← ***to_factor*** A pointer to the array listing the integers to factor.

← ***factor_base*** A pointer to the array listing the integers to trial divide by.

5.18.4.4 void fill_trial_div_decomp (binary_matrix_t ∗const *matrix*, byte_matrix_t ∗const *decomp_matrix*, mpz_array_t ∗const *partially_factored*, const mpz_array_t ∗const *to_factor*, const uint32_array_t ∗const *factor_base*)

Similar to `fill_matrix_trial_div` but also stores valuations.

Fills the binary matrix `matrix` by trial divisions of the integers listed in `to_factor` by the integers listed in `factor_base`. After these trial divisions, each partially factored integer from `to_factor` are stored in `partially_factored`.

Also stores in `decomp_matrix` the valuation of each integer from `to_factor` for each prime in the factor base. For example the valuation of `to_factor->data[i]` for the prime given by `factor_base->data[j]` will be stored in `decomp_matrix->data[i][j]`.

Note:

The binary matrix is filled so that:

- There is a '1' in the (i-th row, 1st col) position in the matrix if `to_factor->data[i]` is negative.
- There is a '1' in the (i-th row, j-th col) position in the matrix if `to_factor->data[i]` is divisible by an odd power of `factor_base->data[j]`.
- In all other cases, the (i-th row, j-th col) position in the matrix contains a 0.

Parameters:

→ ***matrix*** A pointer to the binary matrix to fill.

→ ***decomp_matrix*** A pointer to the byte matrix to fill.

→ ***partially_factored*** A pointer to the partially factored integers.

← ***to_factor*** A pointer to the array listing the integers to factor.

← ***factor_base*** A pointer to the array listing the integers to trial divide by.

5.18.4.5 uint32_array_list_t* find_dependencies (binary_matrix_t *const *matrix*, linalg_method_t *method*)

Solves a linear system over GF(2).

Solves the linear system over GF(2) given by the binary matrix `matrix`, using the resolution method `method`.

Note:

For the time being, the only implemented method is `SMART_GAUSS_ELIM`.

Parameters:

↔ ***matrix*** A pointer to the binary matrix giving the system to solve.

← ***method*** The linear system resolution method to use.

5.18.4.6 ecode_t find_factors (mpz_array_t *const *factors*, const mpz_t *n*, const mpz_array_t *const *xi_array*, const mpz_array_t *const *yi_array*, const uint32_array_list_t *const *dependencies*)

Find factors of an integer from congruence relations.

Find factors of the integer `n` from congruence relations of the form `{(x_i)^2}_i = {y_i}_i (mod n)` where `{y_i}_i` is a perfect square.

Each entry in `dependencies` gives the list of the aforementioned indexes i so that such a previous relation will hold.

Upon termination, returns the following `ecode_t`:

- `SOME_FACTORS_FOUND` if some factors were found
- `NO_FACTOR` FOUND is no factor was found
- `FATAL_INTERNAL_ERROR` is case of... a really ugly error!

Parameters:

→ ***factors*** The factors of `n` found.

← ***n*** The integer to factor.

← ***xi_array*** A pointer to an array giving the avalaible `x_i` values.

← ***yi_array*** A pointer to an array giving the avalaible `y_i` values.

← ***dependencies*** A pointer to an array list giving the sets of indexes from which a congruence relation can be computed.

Returns:

An exit code.

5.18.4.7 ecode_t find_factors_decomp (mpz_array_t ∗const *factors*, const mpz_t *n*, const mpz_array_t ∗const *xi_array*, const byte_matrix_t ∗const *yi_decomp_matrix*, const uint32_array_list_t ∗const *dependencies*, const uint32_array_t ∗const *factor_base*)

Similar to `find_factors` but uses the factorization of each `y_i`.

Find factors of the integer `n` from congruence relations of the form `{(x_i)^2}_i = {y_i}_i (mod n)` where `{y_i}_i` is a perfect square.

Each entry in `dependencies` gives the list of the aforementioned indexes i so that such a previous relation will hold.

The difference with the `find_factors` function is that here the `y_i` are not given directly but rather by their factorizations on the factor base by `yi_decomp_matrix`. For example the valuation of `y_i` for the prime given by `factor_base->data[j]` is stored in `yi_decomp_matrix->data[i][j]`.

Upon termination, returns the following `ecode_t`:

- `SOME_FACTORS_FOUND` if some factors were found
- `NO_FACTOR` FOUND is no factor was found
- `FATAL_INTERNAL_ERROR` is case of... a really ugly error!

Parameters:

→ ***factors*** The factors of `n` found.

← ***n*** The integer to factor.

← ***xi_array*** A pointer to an array giving the avalaible `x_i` values.

← ***yi_decomp_matrix*** A pointer to a byte matrix giving the factorization of the `y_i`.

← ***dependencies*** A pointer to an array list giving the sets of indexes from which a congruence relation can be computed.

← ***factor_base*** A pointer to the array listing the primes in the factor base.

Returns:

An exit code.

5.18.5 Variable Documentation

5.18.5.1 const char* const linalg_method_to_str[1] [static]

Initial value:

```
 {
    "smart gaussian elimination"
}
```

Global constant array mapping linalg methods to their string representations.

Definition at line 78 of file lindep.h.

5.19 linked_list.h File Reference

Standard singly-linked list.

`#include <inttypes.h>`

Data Structures

- struct struct_linked_list_node_t

  A basic implementation of a linked list node.

- struct struct_linked_list_t

  A basic implementation of a linked list.

Defines

- #define _TIFA_LINKED_LIST_H_

Typedefs

- typedef struct struct_linked_list_t linked_list_t

  Equivalent to `struct struct_linked_list_t`.

- typedef struct struct_linked_list_node_t linked_list_node_t

  Equivalent to `struct struct_linked_list_node_t`.

Functions

- void init_linked_list (linked_list_t *const list, int(*cmp_func)(const void *const data_a, const void *const data_b))

  Initializes a `linked_list_t`.

- void clear_linked_list (linked_list_t *const list)

  Clears a `linked_list_t`.

- void append_to_linked_list (linked_list_t *const list, void *const data)

  Appends a node in a `linked_list_t`.

- void prepend_to_linked_list (linked_list_t *const list, void *const data)

  Prepends a node in a `linked_list_t`.

- void * pop_linked_list (linked_list_t *const list)

  Deletes the last node of a `linked_list_t`.

- void * push_linked_list (linked_list_t *const list)

  Deletes the first node of a `linked_list_t`.

- void insert_in_linked_list (linked_list_t *const list, void *const data)

  Inserts a node in a `linked_list_t`.

- linked_list_node_t * get_node_in_linked_list (linked_list_t *const list, void *const data)

  Gets a node in a `linked_list_t`.

- linked_list_node_t * remove_from_linked_list (linked_list_t *const list, void *const data)

  Gets a node in a `linked_list_t` *and removes it from the list.*

- void remove_node_from_linked_list (linked_list_t *const list, linked_list_node_t *const node)

  Removes a given node from a `linked_list_t`.

- void delete_in_linked_list (linked_list_t *const list, void *const data)

  Finds and deletes a node from a `linked_list_t`.

5.19.1 Detailed Description

Standard singly-linked list.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Defines generic singly-linked lists and their associated functions.

Definition in file linked_list.h.

5.19.2 Define Documentation

5.19.2.1 #define _TIFA_LINKED_LIST_H_

Standard include guard.

Definition at line 35 of file linked_list.h.

5.19.3 Function Documentation

5.19.3.1 void append_to_linked_list (linked_list_t *const *list*, void *const *data*)

Appends a node in a `linked_list_t`.

Appends a node (whose data is given by the pointer `data`) in the linked list `list`.

Parameters:

← ***list*** A pointer to the `linked_list_t`.

← ***data*** A pointer to the new node's data.

5.19.3.2 void clear_linked_list (linked_list_t *const *list*)

Clears a `linked_list_t`.

Clears a linked list `list` by freeing its nodes.

Warning:

Each node's `data` field is freed. However, if `data` is a pointer to a structure containing some other pointers, all the memory may not be freed.

Parameters:

← ***list*** A pointer to the `linked_list_t` to clear.

5.19.3.3 void delete_in_linked_list (linked_list_t *const *list*, void *const *data*)

Finds and deletes a node from a `linked_list_t`.

Deletes the node whose data is given by the pointer `data` from the linked list `list`. If the node is not found, the linked list is left unchanged.

Warning:

If a matching node is found, its `data` field is freed. However, if `data` is a pointer to a structure containing some other pointers, all the memory may not be freed.

Parameters:

← ***list*** A pointer to the `linked_list_t`.

← ***data*** A pointer to the data of the node to delete.

5.19.3.4 linked_list_node_t* get_node_in_linked_list (linked_list_t *const *list*, void *const *data*)

Gets a node in a `linked_list_t`.

Returns a pointer to a node (whose data is given by the pointer `data`) from the linked list `list`.

Parameters:

← ***list*** A pointer to the `linked_list_t`.

← ***data*** A pointer to the seeked node data.

Returns:

A pointer to the node with data pointed by `data`, if such a node exists.
NULL if no matching node is found in the linked list.

5.19.3.5 void init_linked_list (linked_list_t *const *list*, int(*)(const void *const data_a, const void *const data_b) *cmp_func*)

Initializes a `linked_list_t`.

Initializes a linked list `list`:

- Sets its `head` to `NULL`.
- Sets its `tail` to `NULL`.
- Sets its `cmp_func` to the `cmp_func` argument.
- Sets its `length` to 0.

Parameters:

← ***list*** A pointer to the `linked_list_t` to initialize.
← ***cmp_func*** A pointer to the comparison function.

5.19.3.6 void insert_in_linked_list (linked_list_t *const *list*, void *const *data*)

Inserts a node in a `linked_list_t`.

Inserts a node (whose data is given by the pointer `data`) in the linked list `list`, so that all the previous nodes have `data` fields pointing to datas less than the new node's data.

Parameters:

← ***list*** A pointer to the `linked_list_t`.
← ***data*** A pointer to the new node's data.

5.19.3.7 void* pop_linked_list (linked_list_t *const *list*)

Deletes the last node of a `linked_list_t`.

Returns the data of the last node of the linked list `list` and deletes this node, similar to Perl's pop function.

Parameters:

← ***list*** A pointer to the `linked_list_t`.

5.19.3.8 void prepend_to_linked_list (linked_list_t *const *list*, void *const *data*)

Prepends a node in a `linked_list_t`.

Prepends a node (whose data is given by the pointer `data`) in the linked list `list`.

Parameters:

← ***list*** A pointer to the `linked_list_t`.
← ***data*** A pointer to the new node's data.

5.19.3.9 void* push_linked_list (linked_list_t *const *list*)

Deletes the first node of a `linked_list_t`.

Returns the data of the first node of the linked list `list` and deletes this node, similar to Perl's push function.

Parameters:

← ***list*** A pointer to the `linked_list_t`.

5.19.3.10 linked_list_node_t* remove_from_linked_list (linked_list_t *const *list*, void *const *data*)

Gets a node in a `linked_list_t` and removes it from the list.

Returns a pointer to a node (whose data is given by the pointer `data`) from the linked list `list` and removes this node from the list.

Parameters:

← ***list*** A pointer to the `linked_list_t`.

← ***data*** A pointer to the seeked node data.

Returns:

A pointer to the node with data pointed by `data`, if such a node exists.
NULL if no matching node is found in the linked list.

5.19.3.11 void remove_node_from_linked_list (linked_list_t *const *list*, linked_list_node_t *const *node*)

Removes a given node from a `linked_list_t`.

Removes the node whose data is given by the pointer `data` from the linked list `list`. If the node is not found, the linked list is left unchanged.

Parameters:

← ***list*** A pointer to the `linked_list_t`.

← ***node*** A pointer to the node to remove from the list.

5.20 macros.h File Reference

Various CPP macros.

Defines

- #define _TIFA_MACROS_H_
- #define MPN_NORMALIZE(dest, nlimbs)
- #define SIZ(x) ((x) → _mp_size)
- #define ABSIZ(x) (ABS(SIZ(x)))
- #define MPZ_TO_ABS(x) (SIZ(x) = ABSIZ(x))

- #define PTR(x) ((x) → _mp_d)
- #define ALLOC(x) ((x) → _mp_alloc)
- #define MPZ_LIMB_VALUE(x, i) (PTR(x)[(i)] & GMP_NUMB_MASK)
- #define MPZ_LAST_LIMB_VALUE(x) (PTR(x)[SIZ(x) - 1] & GMP_NUMB_MASK)
- #define MAX(a, b) (((a) > (b)) ? (a) : (b))
- #define MIN(a, b) (((a) < (b)) ? (a) : (b))
- #define ABS(a) (((a) < 0) ? (-(a)) : (a))
- #define IS_POWER_OF_2_UI(ui) (((ui) & ((ui) - 1)) == 0)
- #define IS_EVEN(ui) (((ui) & 1) == 0)
- #define IS_ODD(ui) (((ui) & 1) != 0)
- #define ARE_EVEN(uia, uib) ((((uia) | (uib)) & 1) == 0)
- #define ARE_ODD(uia, uib) ((((uia) | (uib)) & 1) != 0)
- #define BIT(N, i) (((N) & (1<<(i))) ? 1 : 0)
- #define DUFF_DEVICE(COUNT, STATEMENT,...)
- #define MPZ_IS_SQUARE(X) (0 != mpz_perfect_square_p(X))
- #define NMILLER_RABIN 32
- #define MPZ_IS_PRIME(X) (0 != mpz_probab_prime_p((X), NMILLER_RABIN))
- #define MPN_ADD(A, B, C)
- #define MPN_ADD_CS(A, B, C)
- #define MPN_SUB(A, B, C)
- #define MPN_SUB_N(A, B, C)
- #define MPN_TDIV_QR(Q, R, N, D)
- #define MPN_MUL(A, B, C)
- #define MPN_MUL_N(A, B, C)
- #define MPN_MUL_CS(A, B, C)
- #define MPN_MUL_CS_S(A, B, C)
- #define DECLARE_MPZ_SWAP_VARS
- #define MPZ_SWAP(A, B)
- #define TIFA_DEBUG_MSG(...)

5.20.1 Detailed Description

Various CPP macros.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Defines some C preprocessor macros that should be kept internal to the TIFA library to avoid poluting client code.

Definition in file macros.h.

5.20.2 Define Documentation

5.20.2.1 #define _TIFA_MACROS_H_

Standard include guard.

Definition at line 36 of file macros.h.

5.20.2.2 #define ABS(a) (((a) < 0) ? (-(a)) : (a))

Standard macro returning the absolute value of a.

Note:

As usual, be careful of possible side effects when using this kind of macro. The standard disclaimers apply.

Definition at line 204 of file macros.h.

5.20.2.3 #define ABSIZ(x) (ABS(SIZ(x)))

Macro from the GMP library: Returns the absolute value of `SIZ(x)`.

Returns the absolute value of `SIZ(x)` that is to say the number of `mp_limbs_t` integers needed to represent the value of `x`.

Note:

This macro is the ABSIZ macro from the GMP library. It is redistributed under the GNU LGPL license.

Definition at line 115 of file macros.h.

5.20.2.4 #define ALLOC(x) ((x) → _mp_alloc)

Macro from the GMP library: Returns the `_mp_alloc` field of an `mpz_t` integer.

Returns the `_mp_alloc` field of an `mpz_t` integer, that is to say the size (in units of `mp_limb_t`) of the `x->_mp_d` array.

Note:

This macro is the ALLOC macro from the GMP library. It is redistributed under the GNU LGPL license.

Definition at line 154 of file macros.h.

5.20.2.5 #define ARE_EVEN(uia, uib) ((((uia) | (uib)) & 1) == 0)

Macro returning True if both of the unsigned integers `uia` and `uib` are even, False otherwise.

Definition at line 237 of file macros.h.

5.20.2.6 #define ARE_ODD(uia, uib) ((((uia) | (uib)) & 1) != 0)

Macro returning True if both of the unsigned integers `uia` and `uib` are odd, False otherwise.

Definition at line 244 of file macros.h.

5.20.2.7 #define BIT(N, i) (((N) & (1<<(i))) ? 1 : 0)

Macro returning the value of the i-th least significant bit of N. BIT(N, 0) returns the least significant bit of N.

Definition at line 251 of file macros.h.

5.20.2.8 #define DECLARE_MPZ_SWAP_VARS

Value:

```
mp_ptr __TMPPTR__MACROS_H__a9b3c01__;                           \
    mp_size_t __TMPSIZ__MACROS_H__a9b3c01__;
```

Macro declaring local variables needed by the `MPZ_SWAP` macro. Should be called *once* prior to any use of the `MPZ_SWAP` macro.

Warning:

Declares the variables `__TMPPTR__MACROS_H__a9b3c01__` and `__TMPSIZ__MACROS_H__a9b3c01__`. Hopefully their names are fancy enough to avoid any local conflict.

Definition at line 562 of file macros.h.

5.20.2.9 #define DUFF_DEVICE(COUNT, STATEMENT, ...)

Value:

```
do {                                            \
        long int __count__ = (COUNT);               \
        long int __niter__ = (__count__ + 7) >> 3;  \
        switch (__count__ & 7) {                    \
            case 0: do { STATEMENT; __VA_ARGS__;    \
            case 7:      STATEMENT; __VA_ARGS__;    \
            case 6:      STATEMENT; __VA_ARGS__;    \
            case 5:      STATEMENT; __VA_ARGS__;    \
            case 4:      STATEMENT; __VA_ARGS__;    \
            case 3:      STATEMENT; __VA_ARGS__;    \
            case 2:      STATEMENT; __VA_ARGS__;    \
            case 1:      STATEMENT; __VA_ARGS__;    \
                         __niter__--;               \
                     } while (__niter__ > 0);       \
        }                                           \
    } while (0)
```

Implements the so-called "Duff device" which is a fairly well-known (and so ugly looking!) loop unrolling technique. `COUNT` is the number of times to perform the operations given by STATEMENTS.

Warning:

`COUNT` should be strictly positive. Using `COUNT` equals to zero will yield wrong results.

Note:

This macro was actually inspired (borrowed? stolen?) from the example given in the article "A Reusable Duff Device" written by Ralf Holly.

See also:

Tom Duff's comments about this technique at `http://www.lysator.liu.se/c/duffs-device.html` (URL last accessed on Thu 17 Feb 2011)
"A Reusable Duff Device", Ralf Holly, *Dr. Dobb's Journal*, August 2005. Available online at: `http://www.drdobbs.com/high-performance-computing/184406208` (URL last accessed on Thu 17 Feb 2011)

Definition at line 277 of file macros.h.

5.20.2.10 #define IS_EVEN(ui) (((ui) & 1) == 0)

Macro returning True if the unsigned integer `ui` is even, False otherwise.

Definition at line 223 of file macros.h.

5.20.2.11 #define IS_ODD(ui) (((ui) & 1) != 0)

Macro returning True if the unsigned integer `ui` is odd, False otherwise.

Definition at line 230 of file macros.h.

5.20.2.12 #define IS_POWER_OF_2_UI(ui) (((ui) & ((ui) - 1)) == 0)

Macro returning a non-zero value if the unsigned integer `ui` is a power of 2.

Note:

As usual, be careful of possible side effects when using this kind of macro. The standard disclaimers apply.

Definition at line 216 of file macros.h.

Referenced by ceil_log2().

5.20.2.13 #define MAX(a, b) (((a) > (b)) ? (a) : (b))

Standard macro returning the maximum of a and b.

Note:

As usual, be careful of possible side effects when using this kind of macro. The standard disclaimers apply.

Definition at line 180 of file macros.h.

5.20.2.14 #define MIN(a, b) (((a) < (b)) ? (a) : (b))

Standard macro returning the minimum of a and b.

Note:

As usual, be careful of possible side effects when using this kind of macro. The standard disclaimers apply.

Definition at line 192 of file macros.h.

5.20.2.15 #define MPN_ADD(A, B, C)

Value:

```
do {                                                                  \
        if (mpn_add(PTR(A), PTR(B), SIZ(B), PTR(C), SIZ(C))) {       \
            SIZ(A) = SIZ(B);                                          \
            MPN_NORMALIZE(PTR(A), SIZ(A));                            \
            PTR(A)[SIZ(A)] = 1;                                       \
            SIZ(A)++;                                                 \
        } else {                                                      \
            SIZ(A) = SIZ(B);                                          \
            MPN_NORMALIZE(PTR(A), SIZ(A));                            \
        }                                                             \
    } while (0)
```

Syntaxic sugar macro wrapping a call to mpn_add. Performs size normalization on the result and takes care of the possible carry out. Does not perform any reallocation: the user should make sure the result has enough space to accomodate the possible carry out.

Takes as parameters three mpz_t (and not arrays of mp_limb_t). SIZ(B) should be greater than or equal to SIZ(C).

Warning:

B and C should be both positive or the result will be unpredictable.

See also:

The GMP documentation for more information on the mpn_add function.

Definition at line 339 of file macros.h.

5.20.2.16 #define MPN_ADD_CS(A, B, C)

Value:

```
do {                                                                  \
        if (SIZ(B) > SIZ(C)) {                                        \
            MPN_ADD(A, B, C);                                         \
        } else {                                                      \
            MPN_ADD(A, C, B);                                         \
        }                                                             \
    } while (0)
```

Syntaxic sugar macro wrapping a call to mpn_add. Performs size normalization on the result, takes care of the possible carry out, and Checks the Sizes of the operands to call mpn_add with the proper parameters' order. However, does not perform any reallocation: the user should make sure the result has enough space to accomodate the possible carry out.

Takes as parameters three mpz_t (and not arrays of mp_limb_t). B and C can be used interchangeably.

Warning:

B and C should be both positive or the result will be unpredictable.

See also:

The GMP documentation for more information on the mpn_add function.

Definition at line 371 of file macros.h.

5.20.2.17 #define MPN_MUL(A, B, C)

Value:

```
do {                                                                    \
        mpn_mul(PTR(A), PTR(B), SIZ(B), PTR(C), SIZ(C));                    \
        SIZ(A) = SIZ(B) + SIZ(C);                                           \
        MPN_NORMALIZE(PTR(A), SIZ(A));                                      \
    } while (0)
```

Syntaxic sugar macro wrapping a call to mpn_mul. Performs size normalization on the result.

Takes as parameters three mpz_t (and not arrays of mp_limb_t). SIZ(B) should be greater than or equal to SIZ(C).

Warning:

B and C should be both positive or the result will be unpredictable.

See also:

The GMP documentation for more information on the mpn_mul function.

Definition at line 466 of file macros.h.

5.20.2.18 #define MPN_MUL_CS(A, B, C)

Value:

```
do {                                                                    \
        if (SIZ(B) > SIZ(C)) {                                              \
            mpn_mul(PTR(A), PTR(B), SIZ(B), PTR(C), SIZ(C));                \
        } else {                                                            \
            mpn_mul(PTR(A), PTR(C), SIZ(C), PTR(B), SIZ(B));                \
        }                                                                   \
        SIZ(A) = SIZ(B) + SIZ(C);                                           \
        MPN_NORMALIZE(PTR(A), SIZ(A));                                      \
    } while (0)
```

Syntaxic sugar macro wrapping a call to mpn_mul. Performs size normalization on the result, and Checks the Sizes of the operands to call mpn_mul with the proper parameters' order.

Takes as parameters three mpz_t (and not arrays of mp_limb_t). B and C can be used interchangeably.

Warning:

B and C should be both positive or the result will be unpredictable.

See also:

The GMP documentation for more information on the mpn_mul function.

Definition at line 511 of file macros.h.

5.20.2.19 #define MPN_MUL_CS_S(A, B, C)

Value:

```
do {                                                          \
       if (ABSIZ(B) > ABSIZ(C)) {                              \
           mpn_mul(PTR(A), PTR(B), ABSIZ(B), PTR(C), ABSIZ(C)); \
       } else {                                                \
           mpn_mul(PTR(A), PTR(C), ABSIZ(C), PTR(B), ABSIZ(B)); \
       }                                                       \
       SIZ(A) = ABSIZ(B) + ABSIZ(C);                           \
       MPN_NORMALIZE(PTR(A), SIZ(A));                          \
       if ((SIZ(B) ^ SIZ(C)) < 0) {                            \
            SIZ(A) = -SIZ(A);                                  \
       }                                                       \
    } while (0)
```

Syntaxic sugar macro wrapping a call to `mpn_mul`. Performs size normalization on the result, and Checks the Sizes and the Signs of the operands to call `mpn_mul` with the proper parameters' order.

Takes as parameters three `mpz_t` (and not arrays of `mp_limb_t`). `B` and `C` can be used interchangeably.

Note:

`B` and `C` are allowed to be negative.

See also:

The GMP documentation for more information on the `mpn_mul` function.

Definition at line 537 of file macros.h.

5.20.2.20 #define MPN_MUL_N(A, B, C)

Value:

```
do {                                                          \
       mpn_mul_n(PTR(A), PTR(B), PTR(C), SIZ(B));              \
       SIZ(A) = SIZ(B) << 1;                                   \
       MPN_NORMALIZE(PTR(A), SIZ(A));                          \
    } while (0)
```

Syntaxic sugar macro wrapping a call to `mpn_mul_n`. Performs size normalization on the result.

Takes as parameters three `mpz_t` (and not arrays of `mp_limb_t`). `SIZ(B)` and `SIZ(C)` should be the same.

Warning:

`B` and `C` should be both positive or the result will be unpredictable.

See also:

The GMP documentation for more information on the `mpn_mul_n` function.

Definition at line 488 of file macros.h.

5.20.2.21 #define MPN_NORMALIZE(dest, nlimbs)

Value:

```
do {                                                          \
        while (((nlimbs) > 0) && ((dest)[(nlimbs) - 1] == 0)) {   \
            (nlimbs)--;                                           \
        }                                                         \
    } while (0)
```

Macro from the GMP library: Computes the effective size of an MPN number.

Given `dest`, a pointer to an array of `nlimbs` `mp_limbs_t` integers giving the representation of a multi-precision integer n, computes the absolute value of the effective size of n, i.e the number of significant `mp_size_t` integers needed to represent n and modifies the value of `nlimbs` accordingly.

Note:

This macro is originally the MPN_NORMALIZE macro from the GMP library. It has been slightly modified.

Definition at line 79 of file macros.h.

5.20.2.22 #define MPN_SUB(A, B, C)

Value:

```
do {                                                             \
       mpn_sub(PTR(A), PTR(B), SIZ(B), PTR(C), SIZ(C));             \
       SIZ(A) = SIZ(B);                                             \
       MPN_NORMALIZE(PTR(A), SIZ(A));                               \
    } while (0)
```

Syntaxic sugar macro wrapping a call to `mpn_sub`. Performs size normalization on the result but does not take care of the possible borrow out.

Takes as parameters three `mpz_t` (and not arrays of `mp_limb_t`). `B` should be greater than or equal to `C`.

Warning:

`B` and `C` should be both positive or the result will be unpredictable.

See also:

The GMP documentation for more information on the `mpn_sub` function.

Definition at line 396 of file macros.h.

5.20.2.23 #define MPN_SUB_N(A, B, C)

Value:

```
do {                                                             \
       mpn_sub_n(PTR(A), PTR(B), PTR(C), SIZ(B));                   \
       SIZ(A) = SIZ(B);                                             \
       MPN_NORMALIZE(PTR(A), SIZ(A));                               \
    } while (0)
```

Syntaxic sugar macro wrapping a call to `mpn_sub_n`. Performs size normalization on the result but does not take care of the possible borrow out.

Takes as parameters three `mpz_t` (and not arrays of `mp_limb_t`). `B` should be greater than or equal to `C`.

Warning:

B and C should be both positive or the result will be unpredictable.

See also:

The GMP documentation for more information on the mpn_sub_n function.

Definition at line 419 of file macros.h.

5.20.2.24 #define MPN_TDIV_QR(Q, R, N, D)

Value:

```
do {                                                                    \
        if (SIZ(N) >= SIZ(D)) {                                             \
            mpn_tdiv_qr(PTR(Q), PTR(R), 0,  PTR(N), SIZ(N), PTR(D), SIZ(D)); \
            SIZ(Q) = SIZ(N) - SIZ(D) + 1;                                   \
            MPN_NORMALIZE(PTR(Q), SIZ(Q));                                  \
            SIZ(R) = SIZ(D);                                                \
            MPN_NORMALIZE(PTR(R), SIZ(R));                                  \
        }                                                                   \
    } while (0)
```

Syntaxic sugar macro wrapping a call to mpn_tdiv_qr. Performs size normalization on both the quotient and remainder.

Takes as parameters four mpz_t (and not arrays of mp_limb_t).

Warning:

N and D should be both positive or the result will be unpredictable.

See also:

The GMP documentation for more information on the mpn_tdiv_qr function.

Definition at line 440 of file macros.h.

5.20.2.25 #define MPZ_IS_PRIME(X) (0 != mpz_probab_prime_p((X), NMILLER_RABIN))

Syntaxic sugar macro wrapping a call to mpz_probab_prime_p. "Returns" true if and only if the mpz_t X is (probably) prime.

Takes as parameter an mpz_t.

Definition at line 320 of file macros.h.

5.20.2.26 #define MPZ_IS_SQUARE(X) (0 != mpz_perfect_square_p(X))

Syntaxic sugar macro wrapping a call to mpz_perfect_square_p . "Returns" true if and only if the mpz_t X is a perfect square.

Takes as parameter an mpz_t.

Definition at line 303 of file macros.h.

5.20.2.27 #define MPZ_LAST_LIMB_VALUE(x) (PTR(x)[SIZ(x) - 1] & GMP_NUMB_MASK)

Returns the value of the most significant limb of the mpz_t integer x. Is equivalent to MPZ_LIMB_-VALUE(x, SIZ(x) - 1).

Definition at line 169 of file macros.h.

5.20.2.28 #define MPZ_LIMB_VALUE(x, i) (PTR(x)[(i)] & GMP_NUMB_MASK)

Returns the value of the i-th limb of the mpz_t integer x. The least significant limb is given by i = 0.

Definition at line 162 of file macros.h.

5.20.2.29 #define MPZ_SWAP(A, B)

Value:

```
do {                                                         \
        __TMPPTR__MACROS_H__a9b3c01__ = PTR(A);                  \
        __TMPSIZ__MACROS_H__a9b3c01__ = SIZ(A);                  \
        PTR(A) = PTR(B);                                         \
        SIZ(A) = SIZ(B);                                         \
        PTR(B) = __TMPPTR__MACROS_H__a9b3c01__;                  \
        SIZ(B) = __TMPSIZ__MACROS_H__a9b3c01__;                  \
    } while (0)
```

Macro swapping the values of the two mpz_t A and B.

Warning:

The macro DECLARE_MPZ_SWAP_VARS should be called *once* before using MPZ_SWAP.

Definition at line 575 of file macros.h.

5.20.2.30 #define MPZ_TO_ABS(x) (SIZ(x) = ABSIZ(x))

Sets the mpz_t x to its absolute value.

Definition at line 123 of file macros.h.

5.20.2.31 #define NMILLER_RABIN 32

Number of Miller-Rabin iterations to perform for each compositeness test.

Definition at line 310 of file macros.h.

5.20.2.32 #define PTR(x) ((x) → _mp_d)

Macro from the GMP library: Returns the _mp_d field of an mpz_t integer.

Returns the _mp_d field of an mpz_t integer, that is to say a pointer to an array of mp_limbs_t integers giving the representation of the value of x.

Note:

This macro is the PTR macro from the GMP library. It is redistributed under the GNU LGPL license.

Definition at line 139 of file macros.h.

5.20.2.33 #define SIZ(x) ((x) → _mp_size)

Macro from the GMP library: Returns the `_mp_size` field of an `mpz_t` integer.

Returns the `_mp_size` field of the variable `x` of type `mpz_t`, that is to say the number of `mp_limbs_t` integers needed to represent the value of `x`. The sign of the returned value is given by the sign of `x`'s value.

Note:

This macro is the SIZ macro from the GMP library. It is redistributed under the GNU LGPL license.

Definition at line 100 of file macros.h.

5.20.2.34 #define TIFA_DEBUG_MSG(...)

Macro printing debug message with filename and line number.

Warning:

The symbol `ENABLE_TIFA_DEBUG_MSG` should be defined to non zero *before* including this file.

Definition at line 601 of file macros.h.

5.21 mainpage.h File Reference

Defines

- #define _TIFA_MAINPAGE_H_

5.21.1 Detailed Description

Author:

Jerome Milan

Definition in file mainpage.h.

5.21.2 Define Documentation

5.21.2.1 #define _TIFA_MAINPAGE_H_

Standard include guard.

Definition at line 6 of file mainpage.h.

5.22 matrix.h File Reference

Byte and binary matrices and associated functions.

```
#include "tifa_config.h"
#include <inttypes.h>
#include "bitstring_t.h"
```

Data Structures

- struct struct_binary_matrix_t

    Defines a matrix of bits.

- struct struct_byte_matrix_t

    Defines a matrix of bytes.

Defines

- #define _TIFA_MATRIX_H_
- #define NO_SUCH_ROW UINT32_MAX

Typedefs

- typedef struct struct_binary_matrix_t binary_matrix_t

    Equivalent to `struct struct_binary_matrix_t`.

- typedef struct struct_byte_matrix_t byte_matrix_t

    Equivalent to `struct struct_byte_matrix_t`.

Functions

- binary_matrix_t * alloc_binary_matrix (uint32_t nrows, uint32_t ncols)

    Allocates and returns a new `binary_matrix_t`.

- binary_matrix_t * clone_binary_matrix (const binary_matrix_t *const matrix)

    Allocates and returns a cloned `binary_matrix_t`.

- void reset_binary_matrix (binary_matrix_t *const matrix)

    Sets a `binary_matrix_t` *to zero.*

- void free_binary_matrix (binary_matrix_t *const matrix)

    Clears a `binary_matrix_t`.

- void print_binary_matrix (const binary_matrix_t *const matrix)

    Prints a `binary_matrix_t`.

- static uint8_t get_matrix_bit (uint32_t row, uint32_t col, const binary_matrix_t *const matrix)

    Returns the value of a given bit in a `binary_matrix_t`.

- static void set_matrix_bit_to_one (uint32_t row, uint32_t col, binary_matrix_t *const matrix)

    Sets to one the value of a given bit in a `binary_matrix_t`.

- static void set_matrix_bit_to_zero (uint32_t row, uint32_t col, binary_matrix_t *const matrix)

    Sets to zero the value of a given bit in a `binary_matrix_t`.

- static void flip_matrix_bit (uint32_t row, uint32_t col, binary_matrix_t ∗const matrix)

  Flips the value of a given bit in a `binary_matrix_t`.

- static uint32_t first_row_with_one_on_col (uint32_t col, const binary_matrix_t ∗const matrix)

  Returns the index of the first row of a `binary_matrix_t` *with a one in a given column.*

- byte_matrix_t ∗ alloc_byte_matrix (uint32_t nrows, uint32_t ncols)

  Allocates and returns a new `byte_matrix_t`.

- byte_matrix_t ∗ clone_byte_matrix (const byte_matrix_t ∗const matrix)

  Allocates and returns a cloned `byte_matrix_t`.

- void reset_byte_matrix (byte_matrix_t ∗const matrix)

  Sets a `byte_matrix_t` *to zero.*

- void free_byte_matrix (byte_matrix_t ∗const matrix)

  Clears a `byte_matrix_t`.

- void print_byte_matrix (const byte_matrix_t ∗const matrix)

  Prints a `byte_matrix_t`.

5.22.1 Detailed Description

Byte and binary matrices and associated functions.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Defines binary matrices (i.e. matrices over GF(2)), byte matrices, and their associated functions.

Definition in file matrix.h.

5.22.2 Define Documentation

5.22.2.1 #define _TIFA_MATRIX_H_

Standard include guard.

Definition at line 36 of file matrix.h.

5.22.2.2 #define NO_SUCH_ROW UINT32_MAX

Value returned by the `first_row_with_one_on_col(col, matrix)` function if no row of the matrix has a bit 1 in its `col-th` column.

Definition at line 53 of file matrix.h.

Referenced by first_row_with_one_on_col().

5.22.3 Function Documentation

5.22.3.1 binary_matrix_t∗ alloc_binary_matrix (uint32_t *nrows*, uint32_t *ncols*)

Allocates and returns a new `binary_matrix_t`.

Allocates and returns a new `binary_matrix_t` such that:

- its `nrows_alloced` field is set to nrows.
- its `ncols_alloced` field is set to the minimum number of `TIFA_BITSTRING_T` integers needed to store ncols bits.
- its `nrows` field is set to nrows.
- its `ncols` field is set to ncols.
- its `data` array of arrays is completely filled with zeroes.

Note:

The behaviour of this alloc function differs from the ones in array.h . This discrepancy will be corrected in later versions.

Parameters:

← ***nrows*** The maximum number of rows of the `binary_matrix_t` to allocate.

← ***ncols*** The maximum number of bits per row of the `binary_matrix_t` to allocate.

Returns:

A pointer to the newly allocated `binary_matrix_t` structure. Note that this matrix may hold more that `ncols` bits per column if `ncols` is not a multiple of 8 ∗ `sizeof(TIFA_BITSTRING_T)`.

5.22.3.2 byte_matrix_t∗ alloc_byte_matrix (uint32_t *nrows*, uint32_t *ncols*)

Allocates and returns a new `byte_matrix_t`.

Allocates and returns a new `byte_matrix_t` such that:

- its `nrows_alloced` field is set to nrows.
- its `ncols_alloced` field is set to ncols.
- its `nrows` field is set to nrows.
- its `ncols` field is set to ncols.
- its `data` array of arrays is completely filled with zeroes.

Note:

The behaviour of this alloc function differs from the ones in array.h. This discrepancy will be corrected in later versions.

Parameters:

← ***nrows*** The maximum number of rows of the `byte_matrix_t` to allocate.

← ***ncols*** The maximum number of columns `byte_matrix_t` to allocate.

Returns:

A pointer to the newly allocated `byte_matrix_t` structure.

5.22.3.3 binary_matrix_t* clone_binary_matrix (const binary_matrix_t *const *matrix*)

Allocates and returns a cloned `binary_matrix_t`.

Allocates and returns a clone of the `binary_matrix_t` pointed by `matrix`.

Parameters:

← ***matrix*** A pointer to the binary matrix to clone.

Returns:

A pointer to the newly allocated `binary_matrix_t` clone.

5.22.3.4 byte_matrix_t* clone_byte_matrix (const byte_matrix_t *const *matrix*)

Allocates and returns a cloned `byte_matrix_t`.

Allocates and returns a clone of the `byte_matrix_t` pointed by `matrix`.

Parameters:

← ***matrix*** A pointer to the byte matrix to clone.

Returns:

A pointer to the newly allocated `byte_matrix_t` clone.

5.22.3.5 static uint32_t first_row_with_one_on_col (uint32_t *col*, const binary_matrix_t *const *matrix*) `[inline, static]`

Returns the index of the first row of a `binary_matrix_t` with a one in a given column.

Returns the index of the first row of a `binary_matrix_t` which has a one on its `col`-th column. It returns NO_SUCH_ROW if no such row is found.

Parameters:

← ***col*** The column of the matrix.

← ***matrix*** A pointer to the `binary_matrix_t`.

Returns:

- An unsigned integer `row` between 0 and `matrix->nrows`-1 such that `(matrix->data[row][col] == 1)`
- NO_SUCH_ROW if, for all valid `i`, `(matrix->data[i][col] != 1)`.

Note:

This function is needed in the gaussian elimination algorithm described in the paper "A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers", by D. Parkinson and M. Wunderlich (Parallel Computing 1 (1984) 65-73).

It could be argued that such a function should then be declared and implemented in the files relevant to the aforementionned algorithm. However, this would lead to a very inefficient implementation of this function since proprer programming pratices would lead to consider the matrix as some kind of opaque structure. Granted, nothing could have prevented us to implement `first_row_with_one_on_col` exactly as in `matrix.c`, but the future maintainer would have the burden to check and update code scattered around several files should the inner structure of `binary_matrix_t` be modified.

This can be seen as a moot point: after all, the TIFA code does not strictly enforce type encapsulation. Indeed, some parts of the code do assume (a minimal!) knowledge of the internal structures of some types (have a look at siqs.c for instance). That does not make it the right thing to do though. Unless when facing a real bottleneck, let's try to keep the "programmer's omniscience" to a manageable level...

Definition at line 316 of file matrix.h.

References struct_binary_matrix_t::data, NO_SUCH_ROW, and struct_binary_matrix_t::nrows.

5.22.3.6 static void flip_matrix_bit (uint32_t *row*, uint32_t *col*, binary_matrix_t ∗const *matrix*) `[inline, static]`

Flips the value of a given bit in a `binary_matrix_t`.

Flips the value of the bit at the `row`-th row and the `col`-th column of the `binary_array_t` pointed by `array`.

Parameters:

← ***row*** The row of the bit to flip.

← ***col*** The column of the bit to flip.

← ***matrix*** A pointer to the `binary_matrix_t`.

Definition at line 265 of file matrix.h.

References struct_binary_matrix_t::data.

5.22.3.7 void free_binary_matrix (binary_matrix_t ∗const *matrix*)

Clears a `binary_matrix_t`.

Clears a `binary_matrix_t`, or, more precisely, clears the memory space used by the arrays pointed by the `data` field of a `binary_matrix_t`. Also set its `nrows_alloced` , `ncols_alloced` , `nrows` and `ncols` fields to zero.

Parameters:

← ***matrix*** A pointer to the `binary_matrix_t` to clear.

5.22.3.8 void free_byte_matrix (byte_matrix_t *const *matrix*)

Clears a `byte_matrix_t`.

Clears a `byte_matrix_t`, or, more precisely, clears the memory space used by the arrays pointed by the `data` field of a `byte_matrix_t`. Also set its `nrows_alloced`, `ncols_alloced`, `nrows` and `ncols` fields to zero.

Parameters:

← ***matrix*** A pointer to the `byte_matrix_t` to clear.

5.22.3.9 static uint8_t get_matrix_bit (uint32_t *row*, uint32_t *col*, const binary_matrix_t *const *matrix*) `[inline, static]`

Returns the value of a given bit in a `binary_matrix_t`.

Returns the value of the bit at the `row`-th row and the `col`-th column of the `binary_array_t` pointed by `array`, as either 0 or 1.

Parameters:

← ***row*** The row of the bit to read.

← ***col*** The column of the bit to read.

← ***matrix*** A pointer to the `binary_matrix_t`.

Returns:

The value of the bit at the (`row`,`col`) position: either 0 or 1.

Definition at line 186 of file matrix.h.

References struct_binary_matrix_t::data.

5.22.3.10 void print_binary_matrix (const binary_matrix_t *const *matrix*)

Prints a `binary_matrix_t`.

Prints a `binary_matrix_t`'s on the standard output.

Parameters:

← ***matrix*** A pointer to the `binary_matrix_t` to print.

5.22.3.11 void print_byte_matrix (const byte_matrix_t *const *matrix*)

Prints a `byte_matrix_t`.

Prints a `byte_matrix_t`'s on the standard output.

Parameters:

← ***matrix*** A pointer to the `byte_matrix_t` to print.

5.22.3.12 void reset_binary_matrix (binary_matrix_t *const *matrix*)

Sets a `binary_matrix_t` to zero.

Sets the `binary_matrix_t matrix` to the zero matrix.

Parameters:

← ***matrix*** A pointer to the `binary_matrix_t` to reset.

5.22.3.13 void reset_byte_matrix (byte_matrix_t *const *matrix*)

Sets a `byte_matrix_t` to zero.

Sets the `byte_matrix_t matrix` to the zero matrix.

Parameters:

← ***matrix*** A pointer to the `byte_matrix_t` to reset.

5.22.3.14 static void set_matrix_bit_to_one (uint32_t *row*, uint32_t *col*, binary_matrix_t *const *matrix*) `[inline, static]`

Sets to one the value of a given bit in a `binary_matrix_t`.

Sets to one the value of the bit at the `row`-th row and the `col`-th column of the `binary_array_t` pointed by `array`.

Parameters:

← ***row*** The row of the bit to set.

← ***col*** The column of the bit to set.

← ***matrix*** A pointer to the `binary_matrix_t`.

Definition at line 214 of file matrix.h.

References struct_binary_matrix_t::data.

5.22.3.15 static void set_matrix_bit_to_zero (uint32_t *row*, uint32_t *col*, binary_matrix_t *const *matrix*) `[inline, static]`

Sets to zero the value of a given bit in a `binary_matrix_t`.

Sets to zero the value of the bit at the `row`-th row and the `col`-th column of the `binary_array_t` pointed by `array`.

Parameters:

← ***row*** The row of the bit to set.

← ***col*** The column of the bit to set.

← ***matrix*** A pointer to the `binary_matrix_t`.

Definition at line 240 of file matrix.h.

References struct_binary_matrix_t::data.

5.23 messages.h File Reference

Status / error messages output macros.

```
#include "timer.h"
#include "tifa_config.h"
#include <stdio.h>
#include "exit_codes.h"
```

Defines

- #define _TIFA_MESSAGES_H_

5.23.1 Detailed Description

Status / error messages output macros.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

This file defines some macros used to output status or error messages if some of the `*_VERBOSE` and/or `*_TIMING` symbols are set to non-zero.

Warning:

The `__VERBOSE__`, `__TIMING__` and `__PREFIX__` symbols should be defined in the file including this header. It is under the responsability of the including file to check for these symbol definitions or to define them, if needed.

Definition in file messages.h.

5.23.2 Define Documentation

5.23.2.1 #define _TIFA_MESSAGES_H_

Standard include guard.

Definition at line 41 of file messages.h.

5.24 print_error.h File Reference

Error printing macro.

```
#include "tifa_config.h"
```

Defines

- #define _TIFA_PRINT_ERROR_H_

5.24.1 Detailed Description

Error printing macro.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

This file defines a macro used to output critical error messages on stderr if the global symbol `TIFA_-PRINT_ERROR` is set to non-zero.

Definition in file print_error.h.

5.24.2 Define Documentation

5.24.2.1 #define _TIFA_PRINT_ERROR_H_

Standard include guard.

Definition at line 37 of file print_error.h.

5.25 res_tdiv.h File Reference

Trial division of residues with optional early abort.

```
#include <inttypes.h>
#include "smooth_filter.h"
```

Defines

- #define _TIFA_RES_TDIV_H_

Functions

- uint32_t res_tdiv (smooth_filter_t *const filter, unsigned long int step)
  Trial divide residues using data from a `smooth_filter_t`*.*

5.25.1 Detailed Description

Trial division of residues with optional early abort.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

This file defines functions used to trial divide residues on a factor base using optional multi-step early abort.

C. Pomerance, *Analysis and Comparison of Some Integer Factoring Algorithm*, in Mathematical Centre Tracts 154.

Definition in file res_tdiv.h.

5.25.2 Define Documentation

5.25.2.1 #define _TIFA_RES_TDIV_H_

Standard include guard.

Definition at line 39 of file res_tdiv.h.

5.25.3 Function Documentation

5.25.3.1 uint32_t res_tdiv (smooth_filter_t *const *filter*, unsigned long int *step*)

Trial divide residues using data from a `smooth_filter_t`.

Filters the relations given by `filter->candidate_*` via trial division at the `step`-th early abort step and stores the 'good' relations in `filter->accepted_*`. The `step` parameter has no effect if `filter->method == TDIV` (i.e. no early abort variation).

Warning:

This function is only meant to be used if `filter->method == TDIV` is either `TDIV` or `TDIV_-EARLY_ABORT`.

Parameters:

← ***filter*** a pointer to the `smooth_filter_t` to use.

← ***step*** the step in the early abort strategy to perform.

5.26 siqs.h File Reference

The Self-Initializing Quadratic Sieve factorization algorithm.

`#include <stdbool.h>`

```
#include <inttypes.h>
#include <gmp.h>
#include "array.h"
#include "lindep.h"
#include "factoring_machine.h"
#include "exit_codes.h"
```

Data Structures

- struct struct_siqs_params_t

  Defines the variable parameters used in the SIQS algorithm.

Defines

- #define _TIFA_SIQS_H_
- #define SIQS_DFLT_SIEVE_HALF_WIDTH 500000
- #define SIQS_DFLT_NPRIMES_IN_BASE NFIRST_PRIMES/16
- #define SIQS_DFLT_NPRIMES_TDIV NFIRST_PRIMES/16
- #define SIQS_DFLT_NRELATIONS 24
- #define SIQS_DFLT_LINALG_METHOD SMART_GAUSS_ELIM
- #define SIQS_DFLT_USE_LARGE_PRIMES true

Typedefs

- typedef struct struct_siqs_params_t siqs_params_t

  Equivalent to `struct struct_siqs_params_t`.

Functions

- void set_siqs_params_to_default (const mpz_t n, siqs_params_t ∗const params)

  Fills a `siqs_params_t` *with default values.*

- ecode_t siqs (mpz_array_t ∗const factors, uint32_array_t ∗const multis, const mpz_t n, const siqs_params_t ∗const params, const factoring_mode_t mode)

  Integer factorization via the self-initializing quadratic sieve (SIQS) algorithm.

5.26.1 Detailed Description

The Self-Initializing Quadratic Sieve factorization algorithm.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Definition in file siqs.h.

5.26.2 Define Documentation

5.26.2.1 #define _TIFA_SIQS_H_

Standard include guard.

Definition at line 33 of file siqs.h.

5.26.2.2 #define SIQS_DFLT_LINALG_METHOD SMART_GAUSS_ELIM

Default linear system resolution method to use.

Definition at line 75 of file siqs.h.

5.26.2.3 #define SIQS_DFLT_NPRIMES_IN_BASE NFIRST_PRIMES/16

Default number of prime numbers composing the factor base on which to factor the residues.

Definition at line 58 of file siqs.h.

5.26.2.4 #define SIQS_DFLT_NPRIMES_TDIV NFIRST_PRIMES/16

Default number of the first primes to use in the trial division of the residues.

Definition at line 64 of file siqs.h.

5.26.2.5 #define SIQS_DFLT_NRELATIONS 24

Default number of congruence relations to find before attempting the factorization of the large integer.

Definition at line 70 of file siqs.h.

5.26.2.6 #define SIQS_DFLT_SIEVE_HALF_WIDTH 500000

Default sieving half-width.

Definition at line 52 of file siqs.h.

5.26.2.7 #define SIQS_DFLT_USE_LARGE_PRIMES true

Use the large prime variation by default.

Definition at line 80 of file siqs.h.

5.26.3 Function Documentation

5.26.3.1 void set_siqs_params_to_default (const mpz_t *n*, siqs_params_t ∗const *params*)

Fills a `siqs_params_t` with default values.

Fills a `siqs_params_t` with default values.

Parameters:

← ***n*** The number to factor.

→ ***params*** A pointer to the `siqs_params_t` structure to fill.

5.26.3.2 ecode_t siqs (mpz_array_t ∗const *factors*, uint32_array_t ∗const *multis*, const mpz_t *n*, const siqs_params_t ∗const *params*, const factoring_mode_t *mode*)

Integer factorization via the self-initializing quadratic sieve (SIQS) algorithm.

Attempts to factor the non perfect square integer `n` with the SIQS algorithm, using the set of parameters given by `params` and the factoring mode given by `mode`. Found factors are then stored in `factors`. Additionally, if the factoring mode used is set to FIND_COMPLETE_FACTORIZATION, factors' multiplicities are stored in the array `multis`.

Note:

If the factoring mode used is different from FIND_COMPLETE_FACTORIZATION, `multis` is allowed to be a NULL pointer. Otherwise, using a NULL pointer will lead to a fatal error.

Warning:

If the `factors` and `multis` arrays have not enough room to store the found factors (and the multiplicities, if any), they will be automatically resized to accommodate the data. This has to be kept in mind when trying to do ingenious stuff with memory management (hint: don't try to be clever here).

Parameters:

→ ***factors*** Pointer to the found factors of `n`.

→ ***multis*** Pointer to the multiplicities of the found factors (only computed if `mode` is set to FIND_-COMPLETE_FACTORIZATION).

← ***n*** The non perfect square integer to factor.

← ***params*** Pointer to the values of the parameters used in the SIQS algorithm.

← ***mode*** The factoring mode to use.

Returns:

An exit code.

5.27 siqs_poly.h File Reference

Structure and functions related to the polynomials used in the SIQS algorithm.

```
#include <stdint.h>
#include <stdbool.h>
#include <gmp.h>
```

```
#include "exit_codes.h"
#include "array.h"
#include "approx.h"
```

Data Structures

- struct struct_siqs_poly_t
  Defines polynomials used by SIQS.

Typedefs

- typedef struct struct_siqs_poly_t siqs_poly_t
  Equivalent to `struct_siqs_poly_t`.

Functions

- siqs_poly_t ∗ alloc_siqs_poly (mpz_t target_a, mpz_t n, uint32_array_t ∗const factor_base, uint32_-array_t ∗const sqrtm_pi)
  Allocates and returns a new `siqs_poly_t`.
- void free_siqs_poly (siqs_poly_t ∗poly)
  Frees a previously allocated `siqs_poly_t`.
- ecode_t update_polynomial (siqs_poly_t ∗const poly)
  Updates a polynomial.
- int na_used (siqs_poly_t ∗const poly)
  Returns the number of "full" initialization performed.

5.27.1 Detailed Description

Structure and functions related to the polynomials used in the SIQS algorithm.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Definition in file siqs_poly.h.

5.27.2 Function Documentation

5.27.2.1 siqs_poly_t* alloc_siqs_poly (mpz_t *target_a*, mpz_t *n*, uint32_array_t *const *factor_base*, uint32_array_t *const *sqrtm_pi*)

Allocates and returns a new `siqs_poly_t`.

Parameters:

target_a the target leading coefficient to approximate.

n the number to factor (or a small multiple).

factor_base the factor base.

sqrtm_pi the modular square roots of n.

Returns:

A pointer to the newly allocated `siqs_poly_t`.

5.27.2.2 void free_siqs_poly (siqs_poly_t * *poly*)

Frees a previously allocated `siqs_poly_t`.

Frees all memory used by the pointed `siqs_poly_t` and then frees the `poly` pointer.

Warning:

Do not call `free(poly)` in client code after a call to `free_siqs_poly(poly)`: it would result in an error.

Parameters:

poly the `siqs_poly_t` to free.

5.27.2.3 int na_used (siqs_poly_t *const *poly*)

Returns the number of "full" initialization performed.

This is also the number of distinct `a` used.

Parameters:

poly the polynomial used.

Returns:

The number of "full" initialization performed.

5.27.2.4 ecode_t update_polynomial (siqs_poly_t *const *poly*)

Updates a polynomial.

Updates the polynomial `poly` by either, deriving a new `b` value (the so-called "fast" initialization) or by computing a new leading coefficient (the "full" or "slow" initilization).

Parameters:

poly the polynomial to update.

Returns:

An error code (either `SUCCESS` or `FATAL_INTERNAL_ERROR`)

5.28 siqs_sieve.h File Reference

Structure and functions related to the sieve used in the SIQS algorithm.

```
#include <stdint.h>
#include <stdbool.h>
#include <gmp.h>
#include "exit_codes.h"
#include "array.h"
#include "approx.h"
#include "buckets.h"
#include "siqs_poly.h"
#include "stopwatch.h"
```

Data Structures

- struct struct_siqs_sieve_t

  Defines the sieve used by SIQS.

Typedefs

- typedef struct struct_siqs_sieve_t siqs_sieve_t

  Equivalent to `struct_siqs_sieve_t`.

Functions

- siqs_sieve_t ∗ alloc_siqs_sieve (mpz_t n, uint32_array_t ∗const factor_base, byte_array_t ∗const log_primes, uint32_array_t ∗const sqrtm_pi, uint32_t half_width)

  Allocates and returns a new `siqs_sieve_t`.

- void free_siqs_sieve (siqs_sieve_t ∗sieve)

  Frees a previously allocated `siqs_sieve_t`.

- ecode_t fill_sieve (siqs_sieve_t ∗const sieve)

  Fills the next chunk of an `siqs_sieve_t`.

- ecode_t scan_sieve (siqs_sieve_t ∗const sieve, int32_array_t ∗const survivors, uint32_t nsurvivors)

  Scans a chunk of an `siqs_sieve_t`.

- void set_siqs_sieve_threshold (siqs_sieve_t ∗const sieve, uint32_t threshold)

  Sets the `siqs_sieve_t`*'s threshold.*

- void print_init_poly_timing (siqs_sieve_t ∗const sieve)

  Prints an `siqs_sieve_t`*'s poly init timing.*

- void print_fill_timing (siqs_sieve_t ∗const sieve)

  Prints an `siqs_sieve_t`*'s fill timing.*

- void print_scan_timing (siqs_sieve_t ∗const sieve)

  Prints an `siqs_sieve_t`*'s scan timing.*

5.28.1 Detailed Description

Structure and functions related to the sieve used in the SIQS algorithm.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Definition in file siqs_sieve.h.

5.28.2 Function Documentation

5.28.2.1 siqs_sieve_t∗ alloc_siqs_sieve (mpz_t *n*, uint32_array_t ∗const *factor_base*, byte_array_t ∗const *log_primes*, uint32_array_t ∗const *sqrtm_pi*, uint32_t *half_width*)

Allocates and returns a new `siqs_sieve_t`.

Parameters:

n the number to factor (or a small multiple)

factor_base the factor base

log_primes logarithms (in base 2) of the primes in the base

sqrtm_pi modular square roots of `n` for each prime in the base

half_width the (approximate) half_width of the sieving interval (the real half_width will be adjusted to be a multiple of `chunk_size` if `ROUND_HALF_WIDTH` is defined as non zero)

Returns:

A pointer to the newly allocated `siqs_sieve_t`.

5.28.2.2 ecode_t fill_sieve (siqs_sieve_t ∗const *sieve*)

Fills the next chunk of an `siqs_sieve_t`.

Fills the next chunk of `sieve`, transparently updating (if needed) the polynomial used.

Parameters:

sieve the `siqs_sieve_t` to fill.

Returns:

An exit code.

5.28.2.3 void free_siqs_sieve (siqs_sieve_t ∗ *sieve*)

Frees a previously allocated `siqs_sieve_t`.

Frees all memory used by the pointed `siqs_sieve_t` and then frees the `sieve` pointer.

Warning:

Do not call `free(sieve)` in client code after a call to `free_siqs_sieve(sieve)`: it would result in an error.

Parameters:

sieve the `siqs_sieve_t` to free.

5.28.2.4 void print_fill_timing (siqs_sieve_t ∗const *sieve*)

Prints an `siqs_sieve_t`'s fill timing.

Prints the time taken by `sieve` to fill its sieve chunks.

Parameters:

sieve the `siqs_sieve_t` to read.

5.28.2.5 void print_init_poly_timing (siqs_sieve_t ∗const *sieve*)

Prints an `siqs_sieve_t`'s poly init timing.

Prints the time taken by `sieve` to initialize its polynomials.

Parameters:

sieve the `siqs_sieve_t` to read.

5.28.2.6 void print_scan_timing (siqs_sieve_t ∗const *sieve*)

Prints an `siqs_sieve_t`'s scan timing.

Prints the time taken by `sieve` to scan its sieve chunks.

Parameters:

sieve the `siqs_sieve_t` to read.

5.28.2.7 ecode_t scan_sieve (siqs_sieve_t *const *sieve*, int32_array_t *const *survivors*, uint32_t *nsurvivors*)

Scans a chunk of an `siqs_sieve_t`.

Scans the last filled chunk of `sieve`.

Parameters:

sieve the `siqs_sieve_t` to scan.

Returns:

An exit code.

5.28.2.8 void set_siqs_sieve_threshold (siqs_sieve_t *const *sieve*, uint32_t *threshold*)

Sets the `siqs_sieve_t`'s threshold.

Sets `sieve`'s threshold (all positions **xi** with `sieve[`**xi**`]` < threshold will be tested for smoothness).

Parameters:

sieve the `siqs_sieve_t` to update.

threshold the new threshold's value.

5.29 smooth_filter.h File Reference

Smooth integer filter.

`#include <inttypes.h>`

`#include <array.h>`

`#include <hashtable.h>`

`#include <gmp.h>`

Data Structures

- struct struct_smooth_filter_t

  Structure grouping variables needed for multi-step early abort strategy.

Defines

- #define _TIFA_SMOOTH_FILTER_H_
- #define MAX_NSTEPS 4

Typedefs

- typedef struct struct_smooth_filter_t smooth_filter_t

  Equivalent to `struct struct_smooth_filter_t`.

- typedef enum smooth_filter_method_enum smooth_filter_method_t

  Equivalent to `enum smooth_filter_method_enum`.

Enumerations

- enum smooth_filter_method_enum { TDIV = 0, TDIV_EARLY_ABORT, DJB_BATCH }

Functions

- void complete_filter_init (smooth_filter_t ∗const filter, uint32_array_t ∗const base)

  Complete initialization of a `smooth_filter_t`.

- void clear_smooth_filter (smooth_filter_t ∗const filter)

  Clears a `smooth_filter_t`.

- void filter_new_relations (smooth_filter_t ∗const filter)

  Filters new relations to keep 'good' ones.

- void print_filter_status (smooth_filter_t ∗const filter)

  Prints the status of the buffers of a `smooth_filter_t`.

Variables

- static const char ∗const filter_method_to_str [3]

5.29.1 Detailed Description

Smooth integer filter.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

The `smooth_filter_t` structure and its associated functions implement the multi-step early abort strategy in a way reminiscent of Pomerance's suggestion in "Analysis and Comparison of Some Integer Factoring Algorithm" with the exception that the smoothness tests are performed by batch (see bernsteinisms.h) instead of trial division.

How to use a `smooth_filter_t` structure? The following code snippet, while incomplete, illustrates the way a `smooth_filter_t` should be used.

```
//
// Fill the with the smooth_filter_t with our parameters...
//
smooth_filter_t filter;
filter.n           = n;    // number to factor
filter.kn          = kn;   // number to factor x multiplier
```

```
filter.batch_size = 1024; // number of relations to perform a batch
filter.methid     = TDIV_EARLY_ABORT; // use the early abort strategy
filter.nsteps     = 2;    // use a 2-step early abort strategy

filter.htable                 = htable;
filter.use_large_primes       = true;
filter.use_siqs_batch_variant = false;

filter.base_size    = factor_base->length; // size of factor base
filter.candidate_xi = candidate_xi; // candidate relations stored
filter.candidate_yi = candidate_yi; // in candidate_* arrays
filter.accepted_xi  = accepted_xi;  // 'good' relations stored
filter.accepted_yi  = accepted_yi;  // in accepted_* arrays
//
// Complete the filter initialization by allocating its internal
// buffers, computing the early abort bounds and the intermediate
// factor bases...
//
complete_filter_init(&filter, factor_base);

while (accepted_yi->length != nrels_to_collect) {
    //
    // While we don't have enough relations, create new ones and
    // stores them in the candidate_* arrays such that
    // yi = xi^2 (mod kn). (The generate_relations function here
    // is completely fictitious)
    //
    generate_relations(candidate_xi, candidate_yi);

    //
    // Select 'good' relations such that yi = xi^2 (mod kn) with
    // yi smooth. Note that pointers to the candidate_* and
    // accepted_* arrays were given in the filter structure.
    //
    filter_new_relations(&filter);
}
//
// This clears _only_ the memory allocated by complete_init_filter.
//
clear_smooth_filter(&filter);
```

Definition in file smooth_filter.h.

5.29.2 Define Documentation

5.29.2.1 #define _TIFA_SMOOTH_FILTER_H_

Standard include guard.

Definition at line 95 of file smooth_filter.h.

5.29.2.2 #define MAX_NSTEPS 4

Maximum number of steps used in the multi-step early abort strategy.

Definition at line 110 of file smooth_filter.h.

5.29.3 Enumeration Type Documentation

5.29.3.1 enum smooth_filter_method_enum

An enumeration of the possible methods used to test residue smoothness.

Enumerator:

TDIV Simple trial division.

TDIV_EARLY_ABORT Simple trial division with (multi-step) early abort.

DJB_BATCH D. Bernstein's batch method described in "How to find smooth parts of integers", `http://cr.yp.to/factorization/smoothparts-20040510.pdf.`

Definition at line 117 of file smooth_filter.h.

5.29.4 Function Documentation

5.29.4.1 void clear_smooth_filter (smooth_filter_t *const *filter*)

Clears a `smooth_filter_t`.

Clears the memory of a `smooth_filter_t` that was allocated by `complete_filter_init()`.

Warning:

This clears *only* the internal buffers allocated by by complete_filter_init(), and not the whole structure. For example, it is still the responsability of the client code to properly clears the `candidate_*` arrays or the `htable` hashtable.

5.29.4.2 void complete_filter_init (smooth_filter_t *const *filter*, uint32_array_t *const *base*)

Complete initialization of a `smooth_filter_t`.

Complete the initialization of a `smooth_filter_t` by allocating needed memory space.

Warning:

It is the responsability of the client code to set the following structure variables *before* calling this function:

- `n`
- `kn`
- `nsteps`
- `batch_size`
- `base_size`
- `htable`
- `candidate_xi`
- `candidate_yi`
- `accepted_xi`
- `accepted_yi`
- `use_large_primes`

- use_siqs_batch_variant

No pointer ownership is transfered. For example, it is still the responsability of the client code to properly clears the candidate_* arrays since the structure just *refers* to them, but does not *own* them.

Note:

If nsteps > MAX_NSTEPS then complete_filter_init will set it to MAX_NSTEPS.

Warning:

If method == DJB_BATCH then nsteps will be set to 0 and no early abort will be performed.

Parameters:

filter a pointer to the smooth_filter_t to initialize
base a pointer to the complete factor base used

5.29.4.3 void filter_new_relations (smooth_filter_t *const *filter*)

Filters new relations to keep 'good' ones.

Filters the relations given by filter->candidate_* via a smoothness detecting batch using a filter->nsteps steps early abort strategy and stores the 'good' relations in filter->accepted_*. Has no effect if the filter->candidate_* are not full since we need filter->batch_size relations to perform a batch.

Parameters:

filter a pointer to the smooth_filter_t used

5.29.4.4 void print_filter_status (smooth_filter_t *const *filter*)

Prints the status of the buffers of a smooth_filter_t.

Prints a status summary of the internal buffers of a smooth_filter_t on the standard output.

Note:

This function is mostly intended for debugging purposes as the output is not particularly well structured.

Parameters:

filter a pointer to the smooth_filter_t to inspect

5.29.5 Variable Documentation

5.29.5.1 const char* const filter_method_to_str[3] [static]

Initial value:

```
 {
    "trial division",
    "trial division + early abort",
    "batch",
}
```

Global constant array mapping filter methods to their string representations.

Definition at line 143 of file smooth_filter.h.

5.30 sqrt_cont_frac.h File Reference

Continued fraction expansion for square root of integers.

`#include <inttypes.h>`

`#include <gmp.h>`

Data Structures

- struct struct_cont_frac_state_t

  An ad-hoc structure for the computation of the continued fraction of a square root.

Defines

- #define _TIFA_SQRT_CONT_FRAC_H_

Typedefs

- typedef struct struct_cont_frac_state_t cont_frac_state_t

  Equivalent to `struct struct_cont_frac_state_t`.

Functions

- void init_cont_frac_state (cont_frac_state_t ∗const state, const mpz_t n)

  Initializes a `cont_frac_state_t`.

- void clear_cont_frac_state (cont_frac_state_t ∗const state)

  Clears a `cont_frac_state_t`.

- static void step_cont_frac_state (cont_frac_state_t ∗const state, uint32_t nsteps)

  Computes another term of a continued fraction.

5.30.1 Detailed Description

Continued fraction expansion for square root of integers.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Defines the continued fraction expansion for the square root of non-perfect square integers.

The expansion is computed via an iterative process, each step giving the value of a new numerator. All the variables needed to perform this computation is stored in an ad-hoc structure called `struct_cont_frac_state_t`.

Note:

Since the denominator of the continued fraction is not used in the CFRAC algorithm, it is not computed here. Also, the numerator of the fraction is only given modulo n. These restrictions are completely trivial to fix should one need the complete approximation a/b of a square root.

Definition in file sqrt_cont_frac.h.

5.30.2 Define Documentation

5.30.2.1 #define _TIFA_SQRT_CONT_FRAC_H_

Standard include guard.

Definition at line 47 of file sqrt_cont_frac.h.

5.30.3 Function Documentation

5.30.3.1 void clear_cont_frac_state (cont_frac_state_t ∗const *state*)

Clears a `cont_frac_state_t`.

Clears a `cont_frac_state_t`.

Parameters:

← ***state*** A pointer to the `cont_frac_state_t` to clear.

5.30.3.2 void init_cont_frac_state (cont_frac_state_t ∗const *state*, const mpz_t *n*)

Initializes a `cont_frac_state_t`.

Initializes a `cont_frac_state_t` to begin the computation of a continued fraction. After invocation of this function, the fields of `state` corresponds to the calculation of the second term of the computed fraction, the first term beeing of course `ceil(sqrt(n))`.

Parameters:

← ***state*** A pointer to the `cont_frac_state_t` to initialize.

← ***n*** The non perfect square integer whose square root will be approximated by the computation of a continued fraction.

5.30.3.3 static void step_cont_frac_state (cont_frac_state_t *const *state*, uint32_t *nsteps*) [inline, static]

Computes another term of a continued fraction.

Computes another coefficient in the expansion of a continued fraction and updates the structure `state`. The parameter `nsteps` gives the number of iteration to perform. A new term is computed at each iteration.

Note:

This function is actually given by `state->step_function`.

Parameters:

← ***state*** A pointer to the `cont_frac_state_t`.

← ***nsteps*** Number of steps to perfom.

Definition at line 192 of file sqrt_cont_frac.h.

References struct_cont_frac_state_t::step_function.

5.31 squfof.h File Reference

The SQUFOF factorization algorithm.

```
#include <stdlib.h>
#include <gmp.h>
#include "array.h"
#include "factoring_machine.h"
#include "exit_codes.h"
```

Data Structures

- struct struct_squfof_params_t

  Defines the variable parameters used in the SQUFOF algorithm (dummy structure).

Defines

- #define _TIFA_SQUFOF_H_

Typedefs

- typedef struct struct_squfof_params_t squfof_params_t

  Equivalent to `struct struct_squfof_params_t`.

Functions

- void set_squfof_params_to_default (squfof_params_t *const params)

Fills a `squfof_params_t` *with default values (dummy function).*

- ecode_t squfof (mpz_array_t ∗const factors, uint32_array_t ∗const multis, const mpz_t n, const squfof_params_t ∗const params, const factoring_mode_t mode)

  Integer factorization via the square form factorization (SQUFOF) algorithm.

5.31.1 Detailed Description

The SQUFOF factorization algorithm.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

This is the TIFA library's implementation of Shanks' square form factorization algorithm (SQUFOF), based on the description given by Jason Gower and Samuel Wagstaff in their paper "Square Form Factorization" to be published in Mathematics of Computation.

Note:

This implementation can only factor numbers whose size is less than twice the size of an `unsigned long int`.

See also:

"Square Form Factorization", Jason E. Gower & Samuel S. Wagstaff Jr. *Mathematics of Computation*, S 0025-5718(07)02010-8, Article electronically published on May 14, 2007.
"Square Form Factorization", Jason E. Gower, PhD thesis, Purdue University, December 2004.
For a description of the "large step algorithm" used to quickly jump over several forms, see: "On the Parallel Generation of the Residues for the Continued Fraction Factoring Algorithm", Hugh C. Williams, Marvin C. Wunderlich, *Mathematics of Computation*, Volume 48, Number 177, January 1987, pages 405-423

Definition in file squfof.h.

5.31.2 Define Documentation

5.31.2.1 #define _TIFA_SQUFOF_H_

Standard include guard.

Definition at line 54 of file squfof.h.

5.31.3 Function Documentation

5.31.3.1 void set_squfof_params_to_default (squfof_params_t *const *params*)

Fills a `squfof_params_t` with default values (dummy function).

This function is intended to fill a `squfof_params_t` with default values.

Warning:

For the time being, this is a dummy function which does absolutely nothing at all, but is kept only as a placeholder should the need for user parameters arise in future code revisions.

Parameters:

params A pointer to the `squfof_params_t` structure to fill.

5.31.3.2 ecode_t squfof (mpz_array_t *const *factors*, uint32_array_t *const *multis*, const mpz_t *n*, const squfof_params_t *const *params*, const factoring_mode_t *mode*)

Integer factorization via the square form factorization (SQUFOF) algorithm.

Attempts to factor the non perfect square integer `n` with the SQUFOF algorithm, using the factoring mode given by `mode`. Found factors are then stored in `factors`. Additionally, if the factoring mode used is set to FIND_COMPLETE_FACTORIZATION, factors' multiplicities are stored in the array `multis`.

Warning:

This implementation can only factor numbers whose sizes in bits are strictly less than twice the size of an `unsigned long int` (the exact limit depending on the number to factor and the multiplier used). This choice was made because most of the computations are then performed using only single precision operations. Such a limitation should not be much of a problem since SQUFOF is mostly used to factor very small integers (up to, say, 20 decimal digits).

Note:

If the factoring mode used is different from FIND_COMPLETE_FACTORIZATION, `multis` is allowed to be a NULL pointer. Otherwise, using a NULL pointer will lead to a fatal error.

Warning:

If the `factors` and `multis` arrays have not enough room to store the found factors (and the multiplicities, if any), they will be automatically resized to accommodate the data. This has to be kept in mind when trying to do ingenious stuff with memory management (hint: don't try to be clever here).

Parameters:

→ ***factors*** Pointer to the found factors of `n`.

→ ***multis*** Pointer to the multiplicities of the found factors (only computed if `mode` is set to FIND_-COMPLETE_FACTORIZATION).

← ***n*** The non perfect square integer to factor.

← ***params*** SQUFOF's parameters (currently unused).

← ***mode*** The factoring mode to use.

Returns:

An exit code.

5.32 stopwatch.h File Reference

A very basic stopwatch-like timer.

```
#include <sys/resource.h>
#include <sys/time.h>
#include <stdint.h>
#include <stdbool.h>
```

Data Structures

- struct struct_stopwatch_t
  Defines a very basic stopwatch-like timer.

Defines

- #define _TIFA_STOPWATCH_H_

Typedefs

- typedef struct struct_stopwatch_t stopwatch_t
  Equivalent to `struct struct_stopwatch_t`.

Functions

- void init_stopwatch (stopwatch_t *const watch)
  Inits a `stopwatch_t`.
- void start_stopwatch (stopwatch_t *const watch)
  Starts a `stopwatch_t`.
- void stop_stopwatch (stopwatch_t *const watch)
  Stop a `stopwatch_t`.
- void reset_stopwatch (stopwatch_t *const watch)
  Reset a `stopwatch_t`.
- double get_stopwatch_elapsed (stopwatch_t *const watch)
  Returns the elapsed time measured.

5.32.1 Detailed Description

A very basic stopwatch-like timer.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

This file implements a very basic stopwatch-like timer (with microsecond precision) based on the `rusage` structure and using the `getrusage` function.

Definition in file stopwatch.h.

5.32.2 Define Documentation

5.32.2.1 #define _TIFA_STOPWATCH_H_

Standard include guard.

Definition at line 37 of file stopwatch.h.

5.32.3 Function Documentation

5.32.3.1 double get_stopwatch_elapsed (stopwatch_t *const *watch*)

Returns the elapsed time measured.

Returns the elapsed time measured by `watch` in seconds.

Warning:

The returned result is only meaningful if the stopwatch is not running (i.e. it has been stopped with the `stop_stopwatch` function).

Parameters:

← ***watch*** The `stopwatch_t` used for timing.

5.32.3.2 void init_stopwatch (stopwatch_t *const *watch*)

Inits a `stopwatch_t`.

Initializes the `stopwatch_t` pointed to by `watch`.

Parameters:

watch The `stopwatch_t` to init.

5.32.3.3 void reset_stopwatch (stopwatch_t *const *watch*)

Reset a `stopwatch_t`.

Reset the `stopwatch_t` pointed to by `watch`. The stopwatch is *not* stopped and will continue to run unless it was already stopped.

Parameters:

watch The `stopwatch_t` to reset.

5.32.3.4 void start_stopwatch (stopwatch_t *const *watch*)

Starts a `stopwatch_t`.

Starts the `stopwatch_t` pointed to by `watch`.

Note:

Consecutive calls to `start_stopwatch` are without effect.

Parameters:

watch The `stopwatch_t` to start.

5.32.3.5 void stop_stopwatch (stopwatch_t *const *watch*)

Stop a `stopwatch_t`.

Stop the `stopwatch_t` pointed to by `watch`. Successive calls to `start_stopwatch` and `stop_stopwatch` are cumulative. In other words, the stopwatch's state holds the time elapsed during all previous time intervals defined by a call to `start_stopwatch` followed by a call to `stop_stopwatch` (provided that the stopwatch was not reset with `reset_stopwatch`).

Note:

Consecutive calls to `stop_stopwatch` are without effect.

Parameters:

watch The `stopwatch_t` to stop.

5.33 tdiv.h File Reference

The trial division factorization algorithm.

```
#include <inttypes.h>
#include <gmp.h>
#include "array.h"
#include "factoring_machine.h"
#include "exit_codes.h"
```

Defines

- #define _TIFA_TDIV_H_
- #define TDIV_DFLT_NPRIMES_TDIV (NFIRST_PRIMES/32)

Functions

- ecode_t tdiv (mpz_array_t ∗const factors, uint32_array_t ∗const multis, const mpz_t n, const uint32_t nprimes)

  Integer factorization via trial division (TDIV).

5.33.1 Detailed Description

The trial division factorization algorithm.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Naive (partial) factorization via trial divisions by a few small primes.

Definition in file tdiv.h.

5.33.2 Define Documentation

5.33.2.1 #define _TIFA_TDIV_H_

Standard include guard.

Definition at line 35 of file tdiv.h.

5.33.2.2 #define TDIV_DFLT_NPRIMES_TDIV (NFIRST_PRIMES/32)

Default number of the first primes to use for trial division.

Definition at line 52 of file tdiv.h.

5.33.3 Function Documentation

5.33.3.1 ecode_t tdiv (mpz_array_t ∗const *factors*, uint32_array_t ∗const *multis*, const mpz_t *n*, const uint32_t *nprimes*)

Integer factorization via trial division (TDIV).

Attempts to factor the integer `n` via trial division by the first `nprimes` primes. Found factors are then stored in the array `factors` and multiplicities are stored in `multis`.

Returns:

- `COMPLETE_FACTORIZATION_FOUND` if the complete factorization of `n` was found.

- SOME_FACTORS_FOUND if some factors were found but could not account for the complete factorization of n. In that case, the unfactored part of n is stored in factors->data[factors->lenth - 1].

- NO_FACTOR_FOUND if no factor were found.

Warning:

The prime numbers are not computed but read from a table. Consequently the number of primes nprimes should be less than or equal to NFIRST_PRIMES (defined in array.h). If nprimes is zero, then the default value DFLT_TDIV_NPRIMES will be used instead.

Parameters:

→ ***factors*** Pointer to the found factors of n.

→ ***multis*** Pointer to the multiplicities of the found factors.

← ***n*** The integer to factor.

← ***nprimes*** The number of primes to trial divide n by.

Returns:

An exit code.

5.34 tifa.h File Reference

Library wide public include file.

```
#include "tifa_config.h"
#include "cfrac.h"
#include "ecm.h"
#include "fermat.h"
#include "qs.h"
#include "siqs.h"
#include "squfof.h"
#include "tdiv.h"
#include "tifa_factor.h"
#include "first_primes.h"
#include "array.h"
#include "exit_codes.h"
#include "factoring_machine.h"
```

Defines

- #define _TIFA_TIFA_H_

5.34.1 Detailed Description

Library wide public include file.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Includes only TIFA's structures and functions needed from client code perspective.

Definition in file tifa.h.

5.34.2 Define Documentation

5.34.2.1 #define _TIFA_TIFA_H_

Standard include guard.

Definition at line 36 of file tifa.h.

5.35 tifa_factor.h File Reference

TIFA's generic factorization function.

```
#include <gmp.h>
#include "array.h"
#include "factoring_machine.h"
#include "exit_codes.h"
```

Defines

- #define _TIFA_TIFA_FACTOR_H_

Functions

- ecode_t tifa_factor (mpz_array_t *const factors, uint32_array_t *const multis, const mpz_t n, const factoring_mode_t mode)

  Generic Integer factorization.

5.35.1 Detailed Description

TIFA's generic factorization function.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

This is the TIFA library's generic factorization function: it picks the most suitable factoring algorithm depending on the size of the number to factor.

Definition in file tifa_factor.h.

5.35.2 Define Documentation

5.35.2.1 #define _TIFA_TIFA_FACTOR_H_

Standard include guard.

Definition at line 36 of file tifa_factor.h.

5.35.3 Function Documentation

5.35.3.1 ecode_t tifa_factor (mpz_array_t ∗const *factors*, uint32_array_t ∗const *multis*, const mpz_t *n*, const factoring_mode_t *mode*)

Generic Integer factorization.

Attempts to factor the non perfect square integer `n` with the most suitable algorithm (chosen according to the size of `n`) and with the factoring mode given by `mode`. Found factors are then stored in `factors`. Additionally, if the factoring mode used is set to `FIND_COMPLETE_FACTORIZATION`, factors' multiplicities are stored in the array `multis`. For the time being, no trial divisions are performed so depending on the situation, it could be worthwhile to carry out such a step *before* calling `tifa_factor`.

Note:

If the factoring mode used is different from `FIND_COMPLETE_FACTORIZATION`, `multis` is allowed to be a NULL pointer. Otherwise, using a NULL pointer will lead to a fatal error.

Warning:

If the `factors` and `multis` arrays have not enough room to store the found factors (and the multiplicities, if any), they will be automatically resized to accommodate the data. This has to be kept in mind when trying to do ingenious stuff with memory management (hint: don't try to be clever here).

Parameters:

→ ***factors*** Pointer to the found factors of `n`.

→ ***multis*** Pointer to the multiplicities of the found factors (only computed if `mode` is set to `FIND_COMPLETE_FACTORIZATION`).

← ***n*** The non perfect square integer to factor.

← ***mode*** The factoring mode to use.

Returns:

An exit code.

5.36 tifa_internals.h File Reference

Library wide include file (complete with internal structures / functions).

```
#include "tifa_config.h"
#include "cfrac.h"
#include "ecm.h"
#include "fermat.h"
#include "qs.h"
#include "siqs.h"
#include "squfof.h"
#include "tdiv.h"
#include "tifa_factor.h"
#include "first_primes.h"
#include "array.h"
#include "bernsteinisms.h"
#include "bitstring_t.h"
#include "exit_codes.h"
#include "factoring_machine.h"
#include "funcs.h"
#include "gauss_elim.h"
#include "gmp_utils.h"
#include "hashtable.h"
#include "lindep.h"
#include "linked_list.h"
#include "macros.h"
#include "matrix.h"
#include "messages.h"
#include "print_error.h"
#include "res_tdiv.h"
#include "smooth_filter.h"
#include "sqrt_cont_frac.h"
#include "timer.h"
#include "x_array_list.h"
#include "x_tree.h"
```

Defines

- #define _TIFA_TIFA_INTERNALS_H_

5.36.1 Detailed Description

Library wide include file (complete with internal structures / functions).

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Includes all TIFA's structures and functions.

Warning:

Usually, only the tifa.h include file is needed. tifa_internals.h should only be included to access some internal structures or functions. Be warned that conflicts with client code or external libraries are then more likely to occur.

Definition in file tifa_internals.h.

5.36.2 Define Documentation

5.36.2.1 #define _TIFA_TIFA_INTERNALS_H_

Standard include guard.

Definition at line 41 of file tifa_internals.h.

5.37 timer.h File Reference

This file defines some macros used to perform timing measurements.

```
#include "stopwatch.h"
```

Defines

- #define _TIFA_TIMER_H_
- #define TIMING_FORMAT "%8.3f"
- #define INIT_NAMED_TIMER(NAME)
- #define RESET_NAMED_TIMER(NAME)
- #define START_NAMED_TIMER(NAME)
- #define STOP_NAMED_TIMER(NAME)
- #define GET_NAMED_TIMING(NAME) get_stopwatch_elapsed(&__TIFA_STOPWATCH_##NAME)
- #define INIT_TIMER INIT_NAMED_TIMER()
- #define RESET_TIMER RESET_NAMED_TIMER()
- #define START_TIMER START_NAMED_TIMER()
- #define STOP_TIMER STOP_NAMED_TIMER()
- #define GET_TIMING GET_NAMED_TIMING()

5.37.1 Detailed Description

This file defines some macros used to perform timing measurements.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Warning:

The `__TIMING__` symbol should be defined in the file including this header. It is under the responsability of the including file to check for its definition or to define it, if needed.

If `__TIMING__` is set to 0, then the macros defined in this file do nothing.

Definition in file timer.h.

5.37.2 Define Documentation

5.37.2.1 #define _TIFA_TIMER_H_

Standard include guard.

Definition at line 40 of file timer.h.

5.37.2.2 #define GET_NAMED_TIMING(NAME) get_stopwatch_elapsed(&__TIFA_-STOPWATCH_ ##NAME)

Return the timing of the timer named NAME as seconds.

Warning:

The returned result is only meaningful if the timer is not running (i.e. it has been stopped via STOP_-NAMED_TIMER).

Definition at line 118 of file timer.h.

5.37.2.3 #define GET_TIMING GET_NAMED_TIMING()

Get timing from an unamed timer.

Definition at line 149 of file timer.h.

5.37.2.4 #define INIT_NAMED_TIMER(NAME)

Value:

```
stopwatch_t __TIFA_STOPWATCH_ ##NAME;        \
        init_stopwatch(&__TIFA_STOPWATCH_ ##NAME);
```

Initialize a timer named NAME.

Warning:

The timer name should not be enclosed in quotes, e.g. `INIT_NAMED_TIMER(my_timer)` is correct, but `INIT_NAMED_TIMER("my_timer")` is wrong and will result in a compilation error.

This warning holds for all of the `*_NAMED_TIMER` macros.

Definition at line 70 of file timer.h.

5.37.2.5 #define INIT_TIMER INIT_NAMED_TIMER()

Initialize an unamed timer.

Definition at line 125 of file timer.h.

5.37.2.6 #define RESET_NAMED_TIMER(NAME)

Value:

```
do {                                         \
            reset_stopwatch(&__TIFA_STOPWATCH_ ##NAME); \
        } while (0)
```

Reset the timer named NAME to zero.

Definition at line 78 of file timer.h.

5.37.2.7 #define RESET_TIMER RESET_NAMED_TIMER()

Reset an unamed timer.

Definition at line 131 of file timer.h.

5.37.2.8 #define START_NAMED_TIMER(NAME)

Value:

```
do {                                         \
            start_stopwatch(&__TIFA_STOPWATCH_ ##NAME); \
        } while (0)
```

Start the timer named NAME.

Note:

Consecutive "calls" to START_NAMED_TIMER are without effect.

Definition at line 89 of file timer.h.

5.37.2.9 #define START_TIMER START_NAMED_TIMER()

Start an unamed timer.

Definition at line 137 of file timer.h.

5.37.2.10 #define STOP_NAMED_TIMER(NAME)

Value:

```
do {                                         \
            stop_stopwatch(&__TIFA_STOPWATCH_ ##NAME);   \
        } while (0)
```

Stop the timer named NAME. Successive "calls" to START_NAMED_TIMER and STOP_NAMED_-TIMER are cumulative. In other words, the timer's state holds the time elapsed during all previous time intervals defined by a "call" to START_NAMED_TIMER followed by a "call" to STOP_NAMED_TIMER (provided that the timer was not reset via RESET_NAMED_TIMER).

Note:

Consecutive "calls" to STOP_NAMED_TIMER are without side effects.

Definition at line 106 of file timer.h.

5.37.2.11 #define STOP_TIMER STOP_NAMED_TIMER()

Stop an unamed timer.

Definition at line 143 of file timer.h.

5.37.2.12 #define TIMING_FORMAT "%8.3f"

Format used to print timing measurements.

Definition at line 57 of file timer.h.

5.38 tool_utils.h File Reference

Miscellaneous helpful functions.

`#include <inttypes.h>`

`#include <stdbool.h>`

Defines

- #define _TIFA_TOOL_UTILS_H_

Functions

- bool is_a_number (const char *const str_n, uint32_t length)

  Does a string `str_n` *read as a number?*

- void chomp (char *const str_n, uint32_t length)

  `NULL` *terminates a string.*

5.38.1 Detailed Description

Miscellaneous helpful functions.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Miscellaneous functions used by the "tool programs".

Definition in file tool_utils.h.

5.38.2 Define Documentation

5.38.2.1 #define _TIFA_TOOL_UTILS_H_

Standard include guard.

Definition at line 35 of file tool_utils.h.

5.38.3 Function Documentation

5.38.3.1 void chomp (char *const *str_n*, uint32_t *length*)

`NULL` terminates a string.

`NULL` terminates the string `str_n` at the first occurence of a newline encountered. If no newline is found `str_n` is left unchanged.

Note:

This function is actually quite different from the Perl builtin `chomp` function. It's name should probably be changed to avoid possible confusion.

Parameters:

← ***str_n*** The string to `NULL` terminate.

← ***length*** The maximum length of the string to check.

5.38.3.2 bool is_a_number (const char *const *str_n*, uint32_t *length*)

Does a string `str_n` read as a number?

Returns `true` if the string `str_n` represents a number in the decimal base, `false` otherwise.

Parameters:

← ***str_n*** The string to check.

← ***length*** The maximum length of the string to check.

5.39 x_array_list.h File Reference

Higher level lists of arrays and associated functions.

`#include <inttypes.h>`

`#include "array.h"`

Data Structures

- struct struct_uint32_array_list_t

  Defines a list of `uint32_array_t`.

- struct struct_mpz_array_list_t

  Defines a list of `mpz_array_t`.

Defines

- #define _TIFA_X_ARRAY_LIST_H_

Typedefs

- typedef struct struct_uint32_array_list_t uint32_array_list_t

  Equivalent to `struct struct_uint32_array_list_t`.

- typedef struct struct_mpz_array_list_t mpz_array_list_t

  Equivalent to `struct struct_mpz_array_list_t`.

Functions

- uint32_array_list_t ∗ alloc_uint32_array_list (uint32_t alloced)

  Allocates and returns a new `uint32_array_list_t`.

- static void add_entry_in_uint32_array_list (uint32_array_t ∗const entry, uint32_array_list_t ∗const list)

  Adds an entry to a `uint32_array_list_t`.

- void free_uint32_array_list (uint32_array_list_t ∗const list)

  Clears a `uint32_array_list_t`.

- void print_uint32_array_list (const uint32_array_list_t ∗const list)

  Prints a `uint32_array_list_t`.

- mpz_array_list_t ∗ alloc_mpz_array_list (uint32_t alloced)

  Allocates and returns a new `mpz_array_list_t`.

- static void add_entry_in_mpz_array_list (mpz_array_t ∗const entry, mpz_array_list_t ∗const list)

  Adds an entry to a `mpz_array_list_t`.

- void free_mpz_array_list (mpz_array_list_t *const list)

  Clears a `mpz_array_list_t`.

- void print_mpz_array_list (const mpz_array_list_t *const list)

  Prints a `mpz_array_list_t`.

5.39.1 Detailed Description

Higher level lists of arrays and associated functions.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Defines higher level lists of arrays and their associated functions. This terminology is actually a bit confusing since they are actually more arrays of arrays rather than lists strictly speaking.

Definition in file x_array_list.h.

5.39.2 Define Documentation

5.39.2.1 #define _TIFA_X_ARRAY_LIST_H_

Standard include guard.

Definition at line 37 of file x_array_list.h.

5.39.3 Function Documentation

5.39.3.1 static void add_entry_in_mpz_array_list (mpz_array_t *const *entry*, mpz_array_list_t *const *list*) `[inline, static]`

Adds an entry to a `mpz_array_list_t`.

Adds the `entry` pointer to `list` and increments its `length` field.

Warning:

This function tranfers the ownership of the `mpz_array_t` pointed to by `entry` to `list`. This means that any client code *should not* clear any `mpz_array_t` that has been added to a `mpz_array_list_t` since this is the exclusive responsability of the `mpz_array_list_t`.

Parameters:

← ***entry*** A pointer to the `mpz_array_t` to add in the list.

← ***list*** A pointer to the `mpz_array_list_t`.

Returns:

A pointer to the newly allocated `mpz_array_list_t` structure.

Definition at line 225 of file x_array_list.h.

References struct_mpz_array_list_t::data, and struct_mpz_array_list_t::length.

5.39.3.2 static void add_entry_in_uint32_array_list (uint32_array_t *const *entry*, uint32_array_-list_t *const *list*) [inline, static]

Adds an entry to a `uint32_array_list_t`.

Adds the `entry` pointer to `list` and increments its `length` field.

Warning:

This function tranfers the ownership of the `uint32_array_t` pointed to by `entry` to `list`. This means that any client code *should not* clear any `uint32_array_t` that has been added to a `uint32_array_list_t` since this is the exclusive responsability of the `uint32_array_list_t`.

Parameters:

← ***entry*** A pointer to the array to add.

← ***list*** A pointer to the `uint32_array_list_t`.

Returns:

A pointer to the newly allocated `uint32_array_list_t` structure.

Definition at line 117 of file x_array_list.h.

References struct_uint32_array_list_t::alloced, struct_uint32_array_list_t::data, and struct_uint32_array_-list_t::length.

5.39.3.3 mpz_array_list_t* alloc_mpz_array_list (uint32_t *alloced*)

Allocates and returns a new `mpz_array_list_t`.

Allocates and returns a new `mpz_array_list_t` such that:

- its `alloced` field is set to the parameter alloced.
- its `length` field set to zero.
- its `data` array is left *uninitialized*.

Parameters:

← ***alloced*** The allocated length of the `mpz_array_list_t` to allocate.

Returns:

A pointer to the newly allocated `mpz_array_list_t` structure.

5.39.3.4 uint32_array_list_t∗ alloc_uint32_array_list (uint32_t *alloced*)

Allocates and returns a new `uint32_array_list_t`.

Allocates and returns a new `uint32_array_list_t` such that:

- its `alloced` field is set to the parameter alloced.
- its `length` field set to zero.
- its `data` array is left *uninitialized*.

Parameters:

← ***alloced*** The allocated length of the `uint32_array_list_t` to allocate.

Returns:

A pointer to the newly allocated `uint32_array_list_t` structure.

5.39.3.5 void free_mpz_array_list (mpz_array_list_t ∗const *list*)

Clears a `mpz_array_list_t`.

Clears a `mpz_array_list_t`, or, more precisely, clears the memory space used by the array pointed by the `data` field of a `mpz_array_list_t`. Also set its `alloced` and `length` fields to zero.

Parameters:

← ***list*** A pointer to the `mpz_array_list_t` to clear.

5.39.3.6 void free_uint32_array_list (uint32_array_list_t ∗const *list*)

Clears a `uint32_array_list_t`.

Clears a `uint32_array_list_t`, or, more precisely, clears the memory space used by the array pointed by the `data` field of a `uint32_array_list_t`. Also set its `alloced` and `length` fields to zero.

Parameters:

← ***list*** A pointer to the `uint32_array_list_t` to clear.

5.39.3.7 void print_mpz_array_list (const mpz_array_list_t ∗const *list*)

Prints a `mpz_array_list_t`.

Prints a `mpz_array_list_t` on the standard output.

Note:

This function is mostly intended for debugging purposes as the output is not particularly well structured

Parameters:

← ***list*** A pointer to the `mpz_array_list_t` to print.

5.39.3.8 void print_uint32_array_list (const uint32_array_list_t *const *list*)

Prints a `uint32_array_list_t`.

Prints a `uint32_array_list_t` on the standard output.

Note:

This function is mostly intended for debugging purposes as the output is not particularly well structured

Parameters:

← ***list*** A pointer to the `uint32_array_list_t` to print.

5.40 x_tree.h File Reference

Product and remainder trees.

`#include <gmp.h>`

`#include "array.h"`

Defines

- #define _TIFA_X_TREE_H_

Typedefs

- typedef mpz_array_t mpz_tree_t

  Equivalent to `mpz_array_t`.

Functions

- mpz_tree_t * prod_tree (const mpz_array_t *const array)

  Computes the product tree of some `mpz_t` *integers.*

- mpz_tree_t * prod_tree_mod (const mpz_array_t *const array, const mpz_t n)

  Computes the product tree of some `mpz_t` *integers modulo a positive integer.*

- mpz_tree_t * prod_tree_ui (const uint32_array_t *const array)

  Computes the product tree of some `uint32_t` *integers.*

- mpz_tree_t * rem_tree (const mpz_t z, const mpz_tree_t *const ptree)

  Computes a remainder tree.

- void free_mpz_tree (mpz_tree_t *tree)

  Clears a tree of `mpz_t` *integers.*

- void print_mpz_tree (const mpz_tree_t *const tree)

  Prints a tree of `mpz_t` *integers.*

5.40.1 Detailed Description

Product and remainder trees.

Author:

Jerome Milan

Date:

Fri Jun 10 2011

Version:

2011-06-10

Implementation of the product and remainder trees used in D. J. Bernstein's algorithms.

Definition in file x_tree.h.

5.40.2 Define Documentation

5.40.2.1 #define _TIFA_X_TREE_H_

Standard include guard.

Definition at line 36 of file x_tree.h.

5.40.3 Typedef Documentation

5.40.3.1 mpz_tree_t

Equivalent to `mpz_array_t`.

While an `mpz_tree_t` is just an `mpz_array_t`, its memory is allocated in a different manner than in the `mpz_array_t` case. Indeed, the elements of an `mpz_tree_t` array should NOT be modified later on since the memory used is allocated in one huge block to prevent overhead from multiple malloc calls. So the allocated memory of the mpz_t's in the tree can NOT be increased.

The `mpz_tree_t` typedef is introduced only as a reminder of this different underlying memory allocation. `free_mpz_tree` should be used to clear the memory space occupied by an `mpz_tree_t`. Do NOT call `free_mpz_array` on an `mpz_tree_t`!

Definition at line 61 of file x_tree.h.

5.40.4 Function Documentation

5.40.4.1 void free_mpz_tree (mpz_tree_t * *tree*)

Clears a tree of `mpz_t` integers.

Clears a tree of `mpz_t` integers returned by `prod_tree`, `prod_tree_ui` or `rem_tree`.

Note:

This function is actually different from `free_mpz_array`. Indeed, even if the `mpz_tree_t` type is merely a typedef of `mpz_array_t`, the memory used by the `mpz_t` elements is allocated in a completely different manner, hence the need for a different function.

Parameters:

← ***tree*** Pointer to the `mpz_tree_t` to clear.

5.40.4.2 void print_mpz_tree (const mpz_tree_t ∗const *tree*)

Prints a tree of `mpz_t` integers.

Prints a tree of `mpz_t` integers on the standard output. Useful only for debugging purposes and for relatively small trees.

Parameters:

← ***tree*** Pointer to the `mpz_array_t` containing the tree to print.

5.40.4.3 mpz_tree_t∗ prod_tree (const mpz_array_t ∗const *array*)

Computes the product tree of some `mpz_t` integers.

Computes the product tree of the `mpz_t` integers given by `array` and returns it as a newly allocated `mpz_tree_t`.

Note:

The product tree is implemented as a single `mpz_array_t tree` with the usual compact representation: `tree->data`[2i+1] and `tree->data`[2i+2] are the children of the node `tree->data`[i].

Hence, in order to avoid useless nodes (i.e nodes with value 1), it is recommended to have `array->length` equals to a power of 2. If this is not the case, the product tree will be computed as if array was completed by as many useless nodes as necessary until a power of 2 is reached.

This choice was made to keep a space efficient representation and to avoid dynamic allocations of nodes.

Warning:

Although the product tree returned is actually a pointer to an `mpz_tree_t` structure (i.e. an `mpz_array_t`), the elements of the array should NOT be modified later on since the memory used is allocated in one huge block to prevent overhead from multiple malloc calls. So the allocated memory of the mpz_t's in the array can NOT be increased...

Parameters:

← ***array*** Pointer to the `mpz_array_t` containing the `mpz_t` integers to multiply.

Returns:

A pointer to a newly allocated `mpz_tree_t` holding the computed product tree.

5.40.4.4 mpz_tree_t∗ prod_tree_mod (const mpz_array_t ∗const *array*, const mpz_t *n*)

Computes the product tree of some `mpz_t` integers modulo a positive integer.

Similar to `prod_tree` but each node is reduced mod `n`.

Warning:

n should be strictly positive or results will be unpredictable.

See also:

The function prod_tree(const mpz_array_t* const array).

Parameters:

← ***array*** Pointer to the mpz_array_t containing the mpz_t integers to multiply.

← ***n*** The positive modulo.

Returns:

A pointer to a newly allocated mpz_tree_t holding the computed product tree.

5.40.4.5 mpz_tree_t∗ prod_tree_ui (const uint32_array_t ∗const *array*)

Computes the product tree of some uint32_t integers.

Computes the product tree of the uint32_t integers given by array and returns it as a newly allocated mpz_tree_t.

Note:

The product tree is implemented as a single mpz_array_t tree with the usual compact representation: tree->data[2i+1] and tree->data[2i+2] are the children of the node tree->data[i].

Hence, in order to avoid useless nodes (i.e nodes with value 1), it is recommended to have array->length equals to a power of 2. If this is not the case, the product tree will be computed as if array was completed by as many useless nodes as necessary until a power of 2 is reached.

This choice was made to keep a space efficient representation and to avoid dynamic allocations of nodes.

Warning:

Although the product tree returned is actually a pointer to an mpz_tree_t structure (i.e. an mpz_array_t), the elements of the array should NOT be modified later on since the memory used is allocated in one huge block to prevent overhead from multiple malloc calls. So the allocated memory of the mpz_t's in the array can NOT be increased...

Parameters:

← ***array*** Pointer to the uint32_array_t containing the mpz_t integers to multiply.

Returns:

A pointer to a newly allocated mpz_tree_t holding the computed product tree.

5.40.4.6 mpz_tree_t∗ rem_tree (const mpz_t *z*, const mpz_tree_t ∗const *ptree*)

Computes a remainder tree.

Computes the remainder tree of z by the mpz_t integers whose product tree is given by ptree and returns it as a newly allocated mpz_tree_t. If rtree is the returned remainder tree, then one has: rtree->data[i] = z mod ptree->data[i].

Note:

The remainder tree is implemented as a single mpz_array_t tree with the usual compact representation: tree->data[2i+1] and tree->data[2i+2] are the children of the node tree->data[i].

Warning:

Although the remainder tree returned is actually a pointer to an `mpz_tree_t` structure (i.e. an `mpz_array_t`), the elements of the array should NOT be modified later on since the memory used is allocated in one huge block to prevent overhead from multiple malloc calls. So the allocated memory of the mpz_t's in the array can NOT be increased...

Parameters:

← ***z*** The integer to divide.

← ***ptree*** Pointer to the `mpz_tree_t` containing the product tree of the `mpz_t` integers to divide `z` by.

Returns:

A pointer to a newly allocated `mpz_tree_t` holding the computed remainder tree.

Index