

iZ-Library  
0.0.1

Generated by Doxygen 1.16.1



1	iZprime User Manual	1
1.1	What this manual covers	1
1.2	Related project docs	1
2	Benchmarks	3
2.1	1. Run Benchmarks	3
2.2	2. Result Files Used In This Snapshot	3
2.3	3. Sieve Benchmark Table and Plot	4
2.4	4. Prime Generation Benchmark Table	4
2.5	5. Plot Artifacts	4
2.5.1	Prime generation plots	4
2.6	6. SiZ_count Benchmark Table	4
2.7	7. Plot From Terminal (Interactive Prompt)	5
3	izprime CLI	7
3.1	Build	7
3.2	Numeric input syntax	7
3.3	Commands	7
3.4	stream_primes	7
3.5	count_primes	8
3.6	next_prime	8
3.7	is_prime	8
3.8	Existing maintenance commands	8
4	Contributing to iZprime	9
4.1	Scope	9
4.2	Development Setup	9
4.3	Code Style	10
4.4	Validation Requirements	10
4.5	Benchmarking Expectations	10
4.6	Pull Request Checklist	10
4.7	Algorithm Contributions	11
4.8	Review Standard	11
5	Makefile documentation (iZprime)	13
5.1	Quick start	13
5.2	Dependencies	13
5.3	Targets	14
5.3.1	Build / run	14
5.3.2	Tests	14
5.3.3	Benchmarks	14
5.3.4	Examples	15
5.4	Configuration variables	15
5.4.1	Local configuration file	15
5.4.2	Dependency discovery	15

---

5.4.3 Build toggles . . . . .	15
5.4.4 Test/benchmark options . . . . .	16
5.5 Common recipes . . . . .	16
5.6 Notes on outputs . . . . .	16
6 Releasing iZprime . . . . .	17
6.1 Versioning . . . . .	17
6.2 Pre-release Gate . . . . .	17
6.3 Changelog and Release Notes . . . . .	18
6.4 Tag and Publish . . . . .	18
6.5 Post-release . . . . .	18
7 Tests . . . . .	19
7.1 1. Test Targets . . . . .	19
7.2 2. What Each Target Runs . . . . .	19
7.3 3. Current Snapshot Results . . . . .	19
7.4 4. Show Results In Terminal . . . . .	20
7.5 5. Notes . . . . .	20
8 Topic Index . . . . .	21
8.1 Topics . . . . .	21
9 Data Structure Index . . . . .	23
9.1 Data Structures . . . . .	23
10 File Index . . . . .	25
10.1 File List . . . . .	25
11 Topic Documentation . . . . .	27
11.1 Bitmap Module . . . . .	27
11.1.1 Detailed Description . . . . .	28
11.1.2 Function Documentation . . . . .	29
11.1.2.1 bitmap_clear_all() . . . . .	29
11.1.2.2 bitmap_clear_bit() . . . . .	29
11.1.2.3 bitmap_clear_steps() . . . . .	30
11.1.2.4 bitmap_clear_steps_simd() . . . . .	30
11.1.2.5 bitmap_clone() . . . . .	31
11.1.2.6 bitmap_compute_hash() . . . . .	32
11.1.2.7 bitmap_flip_bit() . . . . .	32
11.1.2.8 bitmap_fread() . . . . .	33
11.1.2.9 bitmap_free() . . . . .	34
11.1.2.10 bitmap_fwrite() . . . . .	34
11.1.2.11 bitmap_get_bit() . . . . .	35
11.1.2.12 bitmap_init() . . . . .	36
11.1.2.13 bitmap_set_all() . . . . .	37

---

11.1.2.14	<code>bitmap_set_bit()</code>	37
11.1.2.15	<code>bitmap_validate_hash()</code>	38
11.1.2.16	<code>TEST_BITMAP()</code>	39
11.2	Integer Arrays	39
11.2.1	Detailed Description	41
11.2.2	Function Documentation	41
11.2.2.1	<code>TEST_GENERIC_INT_ARRAYS()</code>	41
11.3	iZ Public API	42
11.3.1	Detailed Description	43
11.3.2	Typedef Documentation	43
11.3.2.1	<code>INPUT_SIEVE_RANGE</code>	43
11.3.3	Function Documentation	44
11.3.3.1	<code>iZ_next_prime()</code>	44
11.3.3.2	<code>SiZ()</code>	44
11.3.3.3	<code>SiZ_count()</code>	45
11.3.3.4	<code>SiZ_stream()</code>	47
11.3.3.5	<code>SiZm()</code>	48
11.3.3.6	<code>SiZm_vy()</code>	49
11.3.3.7	<code>SoA()</code>	50
11.3.3.8	<code>SoE()</code>	50
11.3.3.9	<code>SoEu()</code>	51
11.3.3.10	<code>SoS()</code>	52
11.3.3.11	<code>SSoE()</code>	53
11.3.3.12	<code>vx_random_prime()</code>	54
11.3.3.13	<code>vy_random_prime()</code>	54
11.4	Toolkit (iZ/iZm)	55
11.4.1	Detailed Description	57
11.4.2	Macro Definition Documentation	58
11.4.2.1	<code>MR_ROUNDS</code>	58
11.4.2.2	<code>VX2</code>	58
11.4.2.3	<code>VX3</code>	58
11.4.2.4	<code>VX4</code>	58
11.4.2.5	<code>VX5</code>	58
11.4.2.6	<code>VX6</code>	59
11.4.2.7	<code>VX7</code>	59
11.4.2.8	<code>VX8</code>	59
11.4.3	Function Documentation	59
11.4.3.1	<code>check_primalty()</code>	59
11.4.3.2	<code>compute_l2_vx()</code>	60
11.4.3.3	<code>compute_max_vx()</code>	60
11.4.3.4	<code>compute_vx_k()</code>	61
11.4.3.5	<code>get_root_primes()</code>	61
11.4.3.6	<code>iZ()</code>	62

---

11.4.3.7 iZ_mpz()	62
11.4.3.8 iZm_clone()	63
11.4.3.9 iZm_construct_vx_base()	64
11.4.3.10 iZm_free()	64
11.4.3.11 iZm_init()	65
11.4.3.12 iZm_solve_for_x0()	65
11.4.3.13 iZm_solve_for_x0_mpz()	66
11.4.3.14 iZm_solve_for_y0()	68
11.4.3.15 process_iZ_bitmaps()	69
11.4.3.16 range_info_free()	69
11.4.3.17 range_info_init()	70
11.4.3.18 vx_collect_p_gaps()	70
11.4.3.19 vx_free()	71
11.4.3.20 vx_full_sieve()	71
11.4.3.21 vx_init()	72
11.4.3.22 vx_search_prime()	73
11.4.3.23 vx_stream()	74
11.4.3.24 vy_search_prime()	75
11.5 Logging	76
11.5.1 Detailed Description	77
11.5.2 Macro Definition Documentation	77
11.5.2.1 LOG_DIR	77
11.5.2.2 LOG_FILE	77
11.5.2.3 LOG_MAX_SIZE	77
11.5.2.4 LOGGER_FORMAT_PRINTF	77
11.5.3 Enumeration Type Documentation	77
11.5.3.1 LogLevel	77
11.5.4 Function Documentation	78
11.5.4.1 log_console()	78
11.5.4.2 log_debug()	78
11.5.4.3 log_error()	79
11.5.4.4 log_fatal()	79
11.5.4.5 log_info()	79
11.5.4.6 log_init()	80
11.5.4.7 log_level_to_string()	80
11.5.4.8 log_message()	80
11.5.4.9 log_message_extended()	81
11.5.4.10 log_set_log_level()	82
11.5.4.11 log_warn()	82
11.6 Tests and Benchmarks	82
11.6.1 Detailed Description	83
11.6.2 Typedef Documentation	83
11.6.2.1 SIEVE_FN	83

11.7 Utilities	84
11.7.1 Detailed Description	84
11.7.2 Macro Definition Documentation	85
11.7.2.1 DIR_output	85
11.7.2.2 MAX	85
11.7.2.3 MAX_CORES	85
11.7.2.4 MIN	85
11.7.3 Function Documentation	86
11.7.3.1 create_dir()	86
11.7.3.2 gcd()	86
11.7.3.3 get_cpu_cores_count()	87
11.7.3.4 get_cpu_L2_cache_size_bits()	87
11.7.3.5 gmp_seed_randstate()	88
11.7.3.6 is_numeric_str()	88
11.7.3.7 modular_inverse()	89
11.7.3.8 parse_inclusive_range_mpz()	89
11.7.3.9 parse_numeric_expr_mpz()	90
11.7.3.10 parse_numeric_expr_u64()	90
11.8 Printer, implemented in toolkit/print_utils.c	91
11.8.1 Detailed Description	91
11.8.2 Function Documentation	92
11.8.2.1 print_centered_text()	92
11.8.2.2 print_line()	93
11.8.2.3 print_sha256_hash()	93
11.8.2.4 print_test_fn_header()	93
11.8.2.5 print_test_module_header()	94
11.8.2.6 print_test_module_result()	94
11.8.2.7 print_test_summary()	95
11.8.2.8 print_test_table_header()	96
11.9 Stopwatch	96
11.9.1 Detailed Description	97
11.9.2 Function Documentation	97
11.9.2.1 sw_elapsed_now_seconds()	97
11.9.2.2 sw_elapsed_seconds()	97
11.9.2.3 sw_start()	97
11.9.2.4 sw_stop()	98
12 Data Structure Documentation	99
12.1 BITMAP Struct Reference	99
12.1.1 Detailed Description	99
12.1.2 Field Documentation	99
12.1.2.1 byte_size	99
12.1.2.2 data	100

---

12.1.2.3 sha256 . . . . .	100
12.1.2.4 size . . . . .	100
12.2 INPUT_SIEVE_RANGE Struct Reference . . . . .	101
12.2.1 Detailed Description . . . . .	101
12.2.2 Field Documentation . . . . .	101
12.2.2.1 filepath . . . . .	101
12.2.2.2 mr_rounds . . . . .	101
12.2.2.3 range . . . . .	102
12.2.2.4 start . . . . .	102
12.3 IZM Struct Reference . . . . .	102
12.3.1 Detailed Description . . . . .	102
12.3.2 Field Documentation . . . . .	103
12.3.2.1 base_x5 . . . . .	103
12.3.2.2 base_x7 . . . . .	103
12.3.2.3 k_vx . . . . .	103
12.3.2.4 root_primes . . . . .	103
12.3.2.5 vx . . . . .	103
12.4 IZM_RANGE_INFO Struct Reference . . . . .	104
12.4.1 Detailed Description . . . . .	104
12.4.2 Field Documentation . . . . .	104
12.4.2.1 vx . . . . .	104
12.4.2.2 Xe . . . . .	105
12.4.2.3 Xs . . . . .	105
12.4.2.4 y_range . . . . .	105
12.4.2.5 Ye . . . . .	105
12.4.2.6 Ys . . . . .	105
12.4.2.7 Ze . . . . .	106
12.4.2.8 Zs . . . . .	106
12.5 SIEVE_LIMIT Struct Reference . . . . .	106
12.5.1 Detailed Description . . . . .	106
12.5.2 Field Documentation . . . . .	106
12.5.2.1 base . . . . .	106
12.5.2.2 exp . . . . .	107
12.6 SIEVE_MODEL Struct Reference . . . . .	107
12.6.1 Detailed Description . . . . .	107
12.6.2 Field Documentation . . . . .	107
12.6.2.1 function . . . . .	107
12.6.2.2 name . . . . .	107
12.7 STOPWATCH Struct Reference . . . . .	108
12.7.1 Detailed Description . . . . .	108
12.7.2 Field Documentation . . . . .	108
12.7.2.1 elapsed_sec . . . . .	108
12.7.2.2 end_time . . . . .	108



12.7.2.3 running	108
12.7.2.4 start_time	109
12.8 UI16_ARRAY Struct Reference	109
12.8.1 Detailed Description	109
12.8.2 Field Documentation	109
12.8.2.1 array	109
12.8.2.2 capacity	110
12.8.2.3 count	110
12.8.2.4 ordered	110
12.8.2.5 sha256	110
12.9 UI32_ARRAY Struct Reference	110
12.9.1 Detailed Description	111
12.9.2 Field Documentation	111
12.9.2.1 array	111
12.9.2.2 capacity	111
12.9.2.3 count	111
12.9.2.4 ordered	111
12.9.2.5 sha256	111
12.10 UI64_ARRAY Struct Reference	112
12.10.1 Detailed Description	112
12.10.2 Field Documentation	112
12.10.2.1 array	112
12.10.2.2 capacity	112
12.10.2.3 count	112
12.10.2.4 ordered	113
12.10.2.5 sha256	113
12.11 VX_SEG Struct Reference	113
12.11.1 Detailed Description	113
12.11.2 Field Documentation	114
12.11.2.1 bit_ops	114
12.11.2.2 end_x	114
12.11.2.3 is_large_limit	114
12.11.2.4 mr_rounds	114
12.11.2.5 p_count	114
12.11.2.6 p_gaps	115
12.11.2.7 p_test_ops	115
12.11.2.8 root_limit	115
12.11.2.9 start_x	115
12.11.2.10 vx	115
12.11.2.11 x5	116
12.11.2.12 x7	116
12.11.2.13 y	116
12.11.2.14 yvx	116

13 File Documentation	117
13.1 include/bitmap.h File Reference	117
13.1.1 Detailed Description	118
13.2 bitmap.h	118
13.3 include/int_arrays.h File Reference	119
13.3.1 Detailed Description	121
13.4 int_arrays.h	121
13.5 include/iZ_api.h File Reference	122
13.5.1 Detailed Description	124
13.6 iZ_api.h	124
13.7 include/iZ_toolkit.h File Reference	124
13.7.1 Detailed Description	126
13.8 iZ_toolkit.h	127
13.9 include/logger.h File Reference	128
13.9.1 Detailed Description	129
13.10 logger.h	130
13.11 include/test_api.h File Reference	131
13.11.1 Detailed Description	131
13.12 test_api.h	132
13.13 include/utils.h File Reference	132
13.13.1 Detailed Description	134
13.14 utils.h	134
13.15 src/iZ_apps.c File Reference	135
13.15.1 Detailed Description	136
13.16 iZ_apps.c	136
13.17 src/playground.c File Reference	146
13.17.1 Detailed Description	146
13.18 playground.c	146
13.19 src/prime_sieve.c File Reference	147
13.19.1 Detailed Description	147
13.19.2 Macro Definition Documentation	148
13.19.2.1 ASSERT_LIMIT	148
13.19.2.2 N_LIMIT	148
13.19.2.3 Pi	148
13.19.3 Function Documentation	148
13.19.3.1 process_N_bitmap()	148
13.20 prime_sieve.c	149
13.21 src/toolkit/bitmap.c File Reference	155
13.21.1 Detailed Description	156
13.21.1.1 Implementation Notes	156
13.21.1.2 Performance Characteristics	157
13.22 bitmap.c	157
13.23 src/toolkit/int_arrays.c File Reference	161

---

13.23.1 Detailed Description . . . . .	162
13.23.1.1 Implementation Notes . . . . .	162
13.23.1.2 Performance Characteristics . . . . .	162
13.24 int_arrays.c . . . . .	163
13.25 src/toolkit/iZ_toolkit.c File Reference . . . . .	163
13.25.1 Detailed Description . . . . .	165
13.25.2 Function Documentation . . . . .	165
13.25.2.1 compute_k_vx() . . . . .	165
13.25.2.2 vx_det_sieve() . . . . .	165
13.25.2.3 vx_prob_sieve() . . . . .	166
13.25.2.4 vx_set_base_values() . . . . .	166
13.25.3 Variable Documentation . . . . .	167
13.25.3.1 s_primes . . . . .	167
13.25.3.2 s_primes_count . . . . .	167
13.26 iZ_toolkit.c . . . . .	167
13.27 src/toolkit/logger.c File Reference . . . . .	178
13.27.1 Detailed Description . . . . .	178
13.28 logger.c . . . . .	178
13.29 src/toolkit/print_utils.c File Reference . . . . .	181
13.29.1 Detailed Description . . . . .	181
13.30 print_utils.c . . . . .	182
13.31 src/toolkit/stopwatch.c File Reference . . . . .	183
13.31.1 Detailed Description . . . . .	183
13.32 stopwatch.c . . . . .	184
13.33 src/toolkit/utils.c File Reference . . . . .	184
13.33.1 Detailed Description . . . . .	185
13.34 utils.c . . . . .	185
14 Examples . . . . .	193
14.1 /Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c . . . . .	193
Index . . . . .	199



# Chapter 1

## iZprime User Manual

iZprime is a C library for prime sieving, range counting/streaming, and random-prime generation.

### 1.1 What this manual covers

- Public API ([include/iZ\\_api.h](#))
- Core toolkit types and internals ([include/iZ\\_toolkit.h](#))
- Utility modules and shared data structures

### 1.2 Related project docs

- Algorithm pseudocode: docs/pseudocode.pdf
- CLI usage: docs/cli.md
- Testing guide: docs/tests.md
- Benchmark guide and results: docs/benchmarks.md
- Build system reference: docs/Makefile.md



## Chapter 2

# Benchmarks

This document captures benchmark methodology, current result snapshots, and plotting usage for the iZprime project.

### 2.1 1. Run Benchmarks

From repository root:

```
make benchmark-p_sieve
make benchmark-p_gen
make benchmark-SiZ_count
```

Save results into docs/test\_results/:

```
make benchmark-p_sieve save-results
make benchmark-p_gen save-results
make benchmark-SiZ_count save-results
```

Save results and generate plot artifacts automatically:

```
make benchmark-p_sieve plot
make benchmark-p_gen plot
```

Notes:

- plot implies save-results.
- Dash-style flags also work with GNU make passthrough:

```
make -- benchmark-p_sieve --save-results --plot
make -- benchmark-p_gen --save-results --plot
make -- benchmark-SiZ_count --save-results
```

Note: plotting is currently wired for p\_sieve and p\_gen result formats.

### 2.2 2. Result Files Used In This Snapshot

Sieve benchmark snapshot:

- docs/test\_results/psieve\_10093442.txt

Prime generation benchmark snapshots:

- docs/test\_results/p\_gen\_12094122.txt (1024-bit)
- docs/test\_results/p\_gen\_12094126.txt (2048-bit)
- docs/test\_results/p\_gen\_12094224.txt (4096-bit)

SiZ\_count benchmark snapshot:

- docs/test\_results/SiZ\_count\_19205134.txt

## 2.3 3. Sieve Benchmark Table and Plot

Times are in microseconds (us) from docs/test\_results/psieve\_10093442.txt.

Algorithm	10 <sup>4</sup>	10 <sup>5</sup>	10 <sup>6</sup>	10 <sup>7</sup>	10 <sup>8</sup>	10 <sup>9</sup>	10 <sup>10</sup>
SoE	14	130	1136	28874	136817	2510303	29944939
SSoE	21	230	1826	16351	148737	1393887	13312626
SoEu	18	198	1567	14153	139020	1793877	18707288
SoS	14	159	1395	14393	142700	2414564	28599960
SoA	24	197	1645	14909	145648	3600075	46331079
SiZ	9	119	912	8924	87100	1436860	18074650
SiZm	12	139	931	8825	73872	709836	6917661
SiZm_vy	25	178	1061	7997	68305	684938	6385593

## 2.4 4. Prime Generation Benchmark Table

Average times are in seconds for each bit size from the corresponding ./test\_results/p\_gen\_\*.txt files.

Algorithm	1024	2048	4096
vy_random_prime	0.021182	0.063515	3.181408
vx_random_prime	0.008107	0.082872	1.220519
iZ_random_next_prime	0.023231	0.150618	0.578747
gmp_random_next_prime	0.009421	0.071797	1.666787
BN_generate_prime_ex	0.026731	0.422709	4.959384

## 2.5 5. Plot Artifacts

Current plot files:

- ./test\_results/p\_gen\_12094122\_avg.svg
- ./test\_results/p\_gen\_12094126\_avg.svg
- ./test\_results/p\_gen\_12094224\_avg.svg

### 2.5.1 Prime generation plots

## 2.6 6. SiZ\_count Benchmark Table

Times are in seconds from docs/test\_results/SiZ\_count\_19205134.txt.

Window size: 10<sup>9</sup> integers for each starting point.

Start Point	Prime Count	Time (s)
10 <sup>10</sup>	43336106	0.157103



Start Point	Prime Count	Time (s)
10^20	21710426	45.271972
10^30	14473890	50.446129
10^40	10860250	75.721432
10^50	8684378	83.009872
10^60	7237481	107.691231
10^70	6205976	119.342140
10^80	5428021	167.147788
10^90	4825182	178.556225
10^100	4342289	218.545428

## 2.7 7. Plot From Terminal (Interactive Prompt)

You can generate/view plots directly from terminal. If no filepath is passed, scripts prompt for one:

```
python3 py_tools/plot_results.py
python3 py_tools/plot_p_sieve_results.py
python3 py_tools/plot_p_gen_results.py
```

Or pass a file explicitly:

```
python3 py_tools/plot_results.py output/psieve_10093442.txt --save
python3 py_tools/plot_results.py output/p_gen_12094224.txt --avg --save
```



## Chapter 3

# izprime CLI

The CLI binary is `izprime` and exposes task-oriented subcommands over the library API.

### 3.1 Build

```
make cli
./build/bin/izprime help
```

### 3.2 Numeric input syntax

Commands that accept numbers support:

- decimal integers: 1000000
- grouped decimals: 1,000,000
- powers: 10^6
- scientific shorthand: 1e6, 10e100
- additive expressions: 10e100 + 10e9

### 3.3 Commands

#### 3.4 stream\_primes

Streams primes in an inclusive range using [SiZ\\_stream](#).

```
izprime stream_primes --range "[LOWER, UPPER]" [--print | --stream-to FILE] [--mr-rounds N]
```

Examples:

```
izprime stream_primes --range "[0, 10^5]" --print
izprime stream_primes --range "[1,000,000, 1,001,000]" --stream-to output/range.txt
```

Alias: `sieve`.

### 3.5 count\_primes

Counts primes in an inclusive range using [SiZ\\_count](#).

```
izprime count_primes --range "[LOWER, UPPER]" [--cores-number N] [--mr-rounds N]
```

Examples:

```
izprime count_primes --range "[0, 10^9]" --cores-number 8
izprime count_primes --range "10^100 + 10^9, 10^100 + 10^9 + 10^9" --cores-number 4
```

Alias: count.

### 3.6 next\_prime

Finds the next prime after n using [iZ\\_next\\_prime](#).

```
izprime next_prime --n VALUE
```

Example:

```
izprime next_prime --n "10^12 + 39"
```

### 3.7 is\_prime

Checks primality of n using [check\\_primality](#).

```
izprime is_prime --n VALUE [--rounds N]
```

Example:

```
izprime is_prime --n "10^61 + 1" --rounds 40
```

### 3.8 Existing maintenance commands

```
izprime test [--limit N]
izprime benchmark [--limit N] [--repeat N] [--algo NAME|all] [--save-results FILE]
izprime doctor
```

Use `izprime help <command>` for command-specific help.

## Chapter 4

# Contributing to iZprime

Thanks for contributing. This project is prepared for research-grade publication, so we hold changes to a strict quality bar.

### 4.1 Scope

Contributions are welcome for:

- correctness fixes,
- performance improvements,
- documentation quality,
- tests/benchmarks and tooling,
- new algorithms that align with the iZprime design.

### 4.2 Development Setup

Required dependencies:

- C compiler (GCC/Clang),
- make,
- GMP,
- OpenSSL.

Optional:

- pkg-config,
- Doxygen,
- Python packages in `py_tools/requirements.txt` for plots.

## 4.3 Code Style

- Keep modules focused and composable.
- Prefer explicit ownership and cleanup paths.
- Keep public API declarations in `include/`, implementations in `src/`.
- Document new public structs/functions with Doxygen comments.
- Avoid introducing hidden global state.

## 4.4 Validation Requirements

Before opening a pull request:

```
make clean
make test-unit
make test-integration
make -- test-all --verbose
```

If you touched performance-sensitive code, also run:

```
make benchmark-p_sieve save-results
make benchmark-p_gen save-results
make benchmark-SiZ_count save-results
```

If you changed API/docs:

```
make userManual
```

## 4.5 Benchmarking Expectations

- Include the exact command(s) used.
- Include hardware/OS/compiler context.
- Share result files from output/ when relevant.
- Prefer reporting both absolute timing and relative speedup/regression.

## 4.6 Pull Request Checklist

1. Explain the problem and the technical approach.
2. List behavioral changes and compatibility impact.
3. Link tests that validate the change.
4. Add or update benchmark evidence for performance claims.
5. Update docs (README.md, docs/\*.md, Doxygen comments) when needed.

## 4.7 Algorithm Contributions

For new sieve/generation techniques:

- map design to existing toolkit abstractions ([IZM](#), [VX\\_SEG](#), bitmaps, arrays),
- provide pseudocode-level explanation in docs/pseudocode.pdf source pipeline,
- add integration tests versus trusted baselines,
- add benchmark hooks so users can compare against existing models.

## 4.8 Review Standard

PRs are reviewed for:

- correctness and edge-case behavior,
- memory/process safety,
- API clarity and consistency,
- test coverage,
- reproducibility of benchmark claims.





## Chapter 5

# Makefile documentation (iZprime)

This document describes the modular make workflow for this repository.

### 5.1 Quick start

- Build static library + CLI and run:
  - make
- Build library only:
  - make lib
- Check dependencies:
  - make doctor
- Create release source archive:
  - make dist VERSION=1.0.0
- Run unit + integration tests:
  - make test-all
- Run sieve benchmarks:
  - make benchmark-p\_sieve
- Run prime-generation benchmarks:
  - make benchmark-p\_gen

### 5.2 Dependencies

- C toolchain: gcc or clang
- Build tool: make
- Libraries:
  - GMP (-lgmp)
  - OpenSSL 3 (-lssl -lcrypto)

The Makefile prefers pkg-config for portability; on macOS it also supports Homebrew path fallbacks.

## 5.3 Targets

### 5.3.1 Build / run

- all (default): Builds the CLI and runs it
- lib: Builds build/lib/libizprime.a and shared-library artifacts (default)
- cli: Builds build/bin/izprime (or PROGRAM override)
- run: Runs the CLI binary (builds first if needed)
- debug: Builds with debug flags (-O0 -g) and logging enabled
- release: Builds an optimized binary and disables logging
- clean: Removes build artifacts and generated output/log directories
- help: Prints a concise help summary of targets and options
- install: Installs headers, static/shared libraries, CLI, and pkg-config file
- install-lib: Installs headers, static/shared libraries, and pkg-config file
- uninstall: Removes files installed by the targets above
- doctor: Verifies build dependencies with a compile/link smoke test
- dist: Builds source tarball and SHA256 checksum under dist/

### 5.3.2 Tests

The primary test runner is built at build/test/test\_runner. It links against the static library target (build/lib/libizprime.a).

- test-all: Runs unit + integration tests
- test-unit: Runs unit tests only
- test-integration: Runs integration tests only

Notes:

- Test selection defaults to --unit --integration when no flags are provided.
- Test runner arguments can be extended via TEST\_ARGS.

### 5.3.3 Benchmarks

Benchmarks are intentionally separate from tests, so they can be invoked explicitly and kept out of typical CI workflows.

- benchmark-p\_sieve: Runs prime sieve model benchmarks
  - Invokes: ./build/test/test\_runner --benchmark-p-sieve ...
- benchmark-p\_gen: Runs random prime generation benchmarks
  - Invokes: ./build/test/test\_runner --benchmark-p-gen ...
- benchmark-SiZ\_count: Runs large-window SiZ\_count benchmark
  - Invokes: ./build/test/test\_runner --benchmark-siz-count ...

### 5.3.4 Examples

- examples: Builds all example programs into build/examples/
- run-example: Runs one example
  - Required variables:
    - \* EX=<example\_name> (name of the example source without extension)
    - \* ARGS="..." (optional arguments passed to the example)

Example:

- make run-example EX=sieve\_primes ARGS="SiZm 10000000 10"

## 5.4 Configuration variables

### 5.4.1 Local configuration file

- config.mk (optional, local)
  - If present, it overrides defaults in Makefile.
  - Start from:
    - \* cp config.mk.example config.mk

### 5.4.2 Dependency discovery

- USE\_PKG\_CONFIG (default: 1)
  - When 1, attempts to use pkg-config to discover GMP and OpenSSL.
  - When 0, skips pkg-config and uses the Homebrew/generic fallbacks.
- PKG\_CONFIG (default: pkg-config)
  - Override if you need a specific pkg-config binary.

### 5.4.3 Build toggles

- LOGGING (default: 1)
  - When 1, compiles with -DENABLE\_LOGGING.
  - The release target forces LOGGING=0.
- PROGRAM (default: izprime)
  - Output name for the CLI binary under build/bin/.
- CLI\_ENTRY (default: src/main.c)
  - Source file used as CLI entrypoint (main()).
- VERSION (default: 1.0.0)
  - Release/version metadata for the CLI and shared library.
- SOVERSION (default: first component of VERSION)
  - Shared-library ABI major version.
- BUILD\_SHARED (default: 1)
  - Build shared library artifacts in addition to static library.

#### 5.4.4 Test/benchmark options

- VERBOSE (default: 0)
  - When 1, adds `--verbose` to the test runner.
- SAVE\_RESULTS (default: 0)
  - When 1, adds `--save-results` to benchmark runs (where supported).
- TEST\_OUTPUT (default: empty)
  - When set, redirects test runner output to the given file.
- TEST\_ARGS (default: empty)
  - Extra arguments appended to the test runner invocation.

### 5.5 Common recipes

- Run integration tests with verbose output:
  - `make test-integration VERBOSE=1`
- Save benchmark results:
  - `make benchmark-p_sieve SAVE_RESULTS=1`
  - `make benchmark-p_gen SAVE_RESULTS=1`
  - `make benchmark-SiZ_count SAVE_RESULTS=1`
- Redirect test output to a file:
  - `make test-all TEST_OUTPUT=output/test_all.txt`
- Pass extra args to the test runner:
  - `make test-all TEST_ARGS="--help"`
- Install under a local prefix:
  - `make install PREFIX=$HOME/.local`
- Build static-only artifacts:
  - `make lib BUILD_SHARED=0`
- Create a release tarball:
  - `make dist VERSION=1.0.0`

### 5.6 Notes on outputs

- Build outputs live under `build/`.
- Benchmark/test logs can be redirected via `TEST_OUTPUT`.
- Some benchmarks may be long-running; use `SAVE_RESULTS=1` to persist results when supported by that benchmark.

## Chapter 6

# Releasing iZprime

This document defines the release workflow for v1.x and later.

### 6.1 Versioning

- Use SemVer tags: vMAJOR.MINOR.PATCH (example: v1.0.0).
- VERSION in Makefile should match the release tag.
- SOVERSION defaults to the major version and should change only on ABI breaks.

### 6.2 Pre-release Gate

Run this checklist from a clean working tree:

```
make clean
make test-unit
make test-integration
make benchmark-p_sieve save-results
make benchmark-p_gen save-results
make benchmark-SiZ_count save-results
make userManual
make dist VERSION=1.0.0
```

Confirm:

- generated docs/userManual.pdf opens correctly,
- benchmark outputs are saved in output/,
- dist/ contains tarball + checksum,
- release tarball does not include transient artifacts (build/, logs/, output/, .git/).

## 6.3 Changelog and Release Notes

ChangeLog should summarize user-visible changes.

Suggested generation workflow:

```
LAST_TAG=$(git describe --tags --abbrev=0 2>/dev/null || echo "")
if [ -n "$LAST_TAG" ]; then
  git log --pretty=format:"- %s (%h)" "$LAST_TAG"..HEAD
else
  git log --pretty=format:"- %s (%h)"
fi
```

Use the output as a draft, then curate into sections:

- Added
- Changed
- Fixed
- Performance
- Documentation

Release notes in GitHub/GitLab should include:

- highlights,
- migration notes (if any),
- validation commands used,
- benchmark artifacts summary.

## 6.4 Tag and Publish

```
git tag -a v1.0.0 -m "iZprime v1.0.0"
git push origin v1.0.0
```

Attach dist/izprime-<version>.tar.gz and checksum file to the release.

## 6.5 Post-release

- bump VERSION to next development version (for example 1.0.1-dev),
- update docs if commands/artifacts changed,
- track follow-up fixes as issues/milestone items.

# Chapter 7

## Tests

This document describes test targets, execution commands, and current pass snapshots.

### 7.1 1. Test Targets

From repository root:

```
make test-all
make test-unit
make test-integration
```

Verbose output:

```
make test-all verbose
make test-unit verbose
make test-integration verbose
```

GNU make passthrough form (dash flags):

```
make -- test-all --verbose
make -- test-unit --verbose
make -- test-integration --verbose
```

### 7.2 2. What Each Target Runs

- test-unit: bitmap/int-array/iZm/vx-seg module-level tests.
- test-integration: sieve hash integrity, range APIs, and prime-generation integration checks.
- test-all: unit + integration suites through the shared test runner.

### 7.3 3. Current Snapshot Results

Snapshot command set:

```
make test-unit
make test-integration
make test-all
```

Observed exit codes and summaries:

Target	Exit code	Summary
make test-unit	0	6/6 module groups passed (100.0%)

Target	Exit code	Summary
make test-integration	0	6/6 integration groups passed (100.0%)
make test-all	0	full test runner completed successfully

Expected success markers in output:

- Unit suite: [SUCCESS] ALL MODULE TESTS PASSED!
- Integration suite: [SUCCESS] ALL INTEGRATION TESTS PASSED!

## 7.4 4. Show Results In Terminal

Quick pass/fail runs:

```
make test-unit
make test-integration
```

Detailed run (prints per-test information and hashes):

```
make test-integration verbose
```

Capture logs for review:

```
make test-unit > /tmp/test_unit.log 2>&1
make test-integration > /tmp/test_integration.log 2>&1
```

Tail the summaries:

```
tail -n 40 /tmp/test_unit.log
tail -n 50 /tmp/test_integration.log
```

## 7.5 5. Notes

- The integrity suite intentionally skips [SiZm\\_vy](#) in hash equality checks because it emits unordered prime lists.
- The test runner exits non-zero when any group fails; make surfaces that as a target failure.



# Chapter 8

## Topic Index

### 8.1 Topics

Here is a list of all topics with brief descriptions:

Bitmap Module . . . . .	27
Integer Arrays . . . . .	39
iZ Public API . . . . .	42
Toolkit (iZ/iZm) . . . . .	55
Logging . . . . .	76
Tests and Benchmarks . . . . .	82
Utilities . . . . .	84
Printer, implemented in toolkit/print_utils.c . . . . .	91
Stopwatch . . . . .	96



## Chapter 9

# Data Structure Index

### 9.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">BITMAP</a>	
Packed bit-array with optional SHA-256 checksum . . . . .	99
<a href="#">INPUT_SIEVE_RANGE</a>	
Input parameters for range sieving/counting . . . . .	101
<a href="#">IZM</a>	
Precomputed iZm assets for repeated VX-segment sieving . . . . .	102
<a href="#">IZM_RANGE_INFO</a>	
Precomputed iZm coordinates for an inclusive numeric interval . . . . .	104
<a href="#">SIEVE_LIMIT</a> . . . . .	106
<a href="#">SIEVE_MODEL</a> . . . . .	107
<a href="#">STOPWATCH</a>	
Stopwatch state for elapsed wall-clock measurements . . . . .	108
<a href="#">UI16_ARRAY</a>	
Dynamic array for uint16_t values . . . . .	109
<a href="#">UI32_ARRAY</a>	
Dynamic array for uint32_t values . . . . .	110
<a href="#">UI64_ARRAY</a>	
Dynamic array for uint64_t values . . . . .	112
<a href="#">VX_SEG</a>	
Runtime state for one VX segment at a specific y . . . . .	113



# Chapter 10

## File Index

### 10.1 File List

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

include/ <a href="#">bitmap.h</a>	Bitmap container and bit operations used by sieve implementations . . . . .	117
include/ <a href="#">int_arrays.h</a>	Dynamic arrays for <code>uint16_t</code> , <code>uint32_t</code> , and <code>uint64_t</code> values . . . . .	119
include/ <a href="#">iZ_api.h</a>	Public API for prime sieving, range scans, and prime generation . . . . .	122
include/ <a href="#">iZ_toolkit.h</a>	Core <code>iZ</code> / <code>iZm</code> primitives used by <code>SiZ</code> -family algorithms . . . . .	124
include/ <a href="#">logger.h</a>	Header file for the logging system . . . . .	128
include/ <a href="#">test_api.h</a>	Public test/benchmark entry points for the <code>iZ</code> library . . . . .	131
include/ <a href="#">utils.h</a>	Shared utilities and common includes for the <code>iZprime</code> library . . . . .	132
src/ <a href="#">iZ_apps.c</a>	High-level application routines built on top of the <code>iZ_toolkit</code> . . . . .	135
src/ <a href="#">playground.c</a>	Scratch space for testing new ideas . . . . .	146
src/ <a href="#">prime_sieve.c</a>	Implementations of classical and <code>SiZ</code> -family sieve algorithms . . . . .	147
src/toolkit/ <a href="#">bitmap.c</a>	Implementation of the bitmap module for efficient bit array operations . . . . .	155
src/toolkit/ <a href="#">int_arrays.c</a>	Implementation of dynamic integer array module for 16-bit, 32-bit, and 64-bit unsigned integers . . . . .	161
src/toolkit/ <a href="#">iZ_toolkit.c</a>	Implementation of <code>iZ</code> index space helpers and <code>iZm</code> / <code>VX</code> segment machinery . . . . .	163
src/toolkit/ <a href="#">logger.c</a>	Logging module . . . . .	178
src/toolkit/ <a href="#">print_utils.c</a>	Print formatting helpers for tests, benchmarks, and CLI output . . . . .	181
src/toolkit/ <a href="#">stopwatch.c</a>	Stopwatch helpers based on monotonic wall-clock time . . . . .	183
src/toolkit/ <a href="#">utils.c</a>	Implementations for the shared utility layer . . . . .	184



# Chapter 11

## Topic Documentation

### 11.1 Bitmap Module

Packed bit-array primitives for sieve and toolkit modules.

#### Files

- file [bitmap.c](#)  
Implementation of the bitmap module for efficient bit array operations.

#### Data Structures

- struct [BITMAP](#)  
Packed bit-array with optional SHA-256 checksum.

#### Functions

- int [TEST\\_BITMAP](#) (int verbose)  
Run bitmap module tests.

#### Lifecycle

- [BITMAP](#) \* [bitmap\\_init](#) (size\_t size, int set\_bits)  
Allocate a bitmap with size bits.
- void [bitmap\\_free](#) ([BITMAP](#) \*\*bitmap)  
Free a bitmap and set the caller pointer to NULL.
- [BITMAP](#) \* [bitmap\\_clone](#) ([BITMAP](#) \*src)  
Deep-copy a bitmap, including data and checksum.

## Single-bit Operations

- int `bitmap_get_bit` (`BITMAP *bitmap`, `size_t idx`)  
Read bit value at `idx`.
- void `bitmap_set_bit` (`BITMAP *bitmap`, `size_t idx`)  
Set bit at `idx` to 1.
- void `bitmap_flip_bit` (`BITMAP *bitmap`, `size_t idx`)  
Toggle bit at `idx`.
- void `bitmap_clear_bit` (`BITMAP *bitmap`, `size_t idx`)  
Clear bit at `idx` (set to 0).

## Bulk Operations

- void `bitmap_set_all` (`BITMAP *bitmap`)  
Set all bits to 1.
- void `bitmap_clear_all` (`BITMAP *bitmap`)  
Clear all bits to 0.
- void `bitmap_clear_steps` (`BITMAP *bitmap`, `uint64_t step`, `uint64_t start_idx`, `uint64_t limit`)  
Clear every `step`-th bit from `start_idx` up to `limit`.
- void `bitmap_clear_steps_simd` (`BITMAP *bitmap`, `uint64_t step`, `uint64_t start_idx`, `uint64_t limit`)  
SIMD-accelerated variant of `bitmap_clear_steps()`.

## Integrity and I/O

- void `bitmap_compute_hash` (`BITMAP *bitmap`)  
Compute SHA-256 over bitmap data and store it in `BITMAP::sha256`.
- int `bitmap_validate_hash` (`BITMAP *bitmap`)  
Verify `BITMAP::sha256` against current bitmap data.
- int `bitmap_fwrite` (`BITMAP *bitmap`, `FILE *file`)  
Write bitmap payload and checksum to a binary stream.
- `BITMAP * bitmap_fread` (`FILE *file`)  
Read bitmap payload and checksum from a binary stream.

### 11.1.1 Detailed Description

Packed bit-array primitives for sieve and toolkit modules.



## 11.1.2 Function Documentation

### 11.1.2.1 `bitmap_clear_all()`

```
void bitmap_clear_all (
    BITMAP * bitmap)
```

Clear all bits to 0.

Parameters

bitmap	Bitmap to modify.
--------	-------------------

Clear all bits to 0.

Parameters

bitmap	A pointer to the <a href="#">BITMAP</a> structure.
--------	--

Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line [180](#) of file [bitmap.c](#).

References [BITMAP::data](#), and [BITMAP::size](#).

### 11.1.2.2 `bitmap_clear_bit()`

```
void bitmap_clear_bit (
    BITMAP * bitmap,
    size_t idx)
```

Clear bit at idx (set to 0).

Parameters

bitmap	Bitmap to modify.
idx	Zero-based bit index.

Clear bit at idx (set to 0).

Parameters

bitmap	The <a href="#">BITMAP</a> to modify.
idx	The index of the bit to clear.

Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line [159](#) of file [bitmap.c](#).

References [BITMAP::data](#).

Referenced by [iZm\\_construct\\_vx\\_base\(\)](#), [SoEu\(\)](#), [vx\\_prob\\_sieve\(\)](#), and [vx\\_stream\(\)](#).

### 11.1.2.3 `bitmap_clear_steps()`

```
void bitmap_clear_steps (
    BITMAP * bitmap,
    uint64_t step,
    uint64_t start_idx,
    uint64_t limit)
```

Clear every step-th bit from `start_idx` up to `limit`.

This is the core primitive for marking composite progressions in sieve code.

#### Parameters

<code>bitmap</code>	Bitmap to modify.
<code>step</code>	Index increment between cleared bits.
<code>start_idx</code>	First index to clear.
<code>limit</code>	Inclusive upper index bound.

#### Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line 210 of file `bitmap.c`.

References `BITMAP::data`, `MIN`, and `BITMAP::size`.

Referenced by `vx_search_prime()`.

### 11.1.2.4 `bitmap_clear_steps_simd()`

```
void bitmap_clear_steps_simd (
    BITMAP * bitmap,
    uint64_t step,
    uint64_t start_idx,
    uint64_t limit)
```

SIMD-accelerated variant of `bitmap_clear_steps()`.

#### Parameters

<code>bitmap</code>	Bitmap to modify.
<code>step</code>	Index increment between cleared bits.
<code>start_idx</code>	First index to clear.
<code>limit</code>	Inclusive upper index bound.

SIMD-accelerated variant of `bitmap_clear_steps()`.

This function uses SIMD instructions (AVX2, SSE2, or NEON) to clear bits at regular intervals in the bitmap. It processes multiple bits in parallel for improved performance on large bitmaps.

#### Parameters

<code>bitmap</code>	Pointer to the bitmap to modify
---------------------	---------------------------------

step	Interval between bits to clear (must be > 0)
start_idx	Starting bit index (inclusive)
limit	Upper bound for clearing (inclusive)

Examples

[/Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c](#).

Definition at line 233 of file [bitmap.c](#).

References [BITMAP::data](#), [MIN](#), and [BITMAP::size](#).

Referenced by [iZm\\_construct\\_vx\\_base\(\)](#), [process\\_iZ\\_bitmaps\(\)](#), [process\\_N\\_bitmap\(\)](#), [SiZm\(\)](#), [SiZm\\_vy\(\)](#), [SoA\(\)](#), [SoS\(\)](#), [SSoE\(\)](#), and [vx\\_det\\_sieve\(\)](#).

#### 11.1.2.5 bitmap\_clone()

```
BITMAP * bitmap_clone (
    BITMAP * src)
```

Deep-copy a bitmap, including data and checksum.

Parameters

src	Source bitmap.
-----	----------------

Returns

Cloned bitmap, or NULL on failure.

Deep-copy a bitmap, including data and checksum.

Allocates a new bitmap with the same size as the source and copies all data including the bit array and SHA-256 hash. The returned bitmap is completely independent of the source and must be freed separately.

Parameters

src	Pointer to the source bitmap to clone
-----	---------------------------------------

Returns

Pointer to newly allocated cloned [BITMAP](#), or NULL on failure

Return values

NULL	if src is NULL, src->size is 0, or memory allocation fails
------	--

## Note

The clone includes a copy of the SHA-256 hash from the source  
 Caller must free the returned bitmap using [bitmap\\_free\(\)](#)  
 Changes to the clone do not affect the source and vice versa

## Examples

[/Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c.](#)

Definition at line 382 of file [bitmap.c](#).

References [bitmap\\_init\(\)](#), [BITMAP::byte\\_size](#), [BITMAP::data](#), [BITMAP::sha256](#), and [BITMAP::size](#).

Referenced by [iZm\\_clone\(\)](#), [SiZm\(\)](#), and [vx\\_init\(\)](#).

11.1.2.6 [bitmap\\_compute\\_hash\(\)](#)

```
void bitmap_compute_hash (
    BITMAP * bitmap)
```

Compute SHA-256 over bitmap data and store it in [BITMAP::sha256](#).

## Parameters

bitmap	Bitmap to hash.
--------	-----------------

Compute SHA-256 over bitmap data and store it in [BITMAP::sha256](#).

## Parameters

bitmap	The <a href="#">BITMAP</a> for which the SHA-256 hash is generated.
--------	---

## Examples

[/Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c.](#)

Definition at line 398 of file [bitmap.c](#).

References [BITMAP::byte\\_size](#), [BITMAP::data](#), and [BITMAP::sha256](#).

Referenced by [bitmap\\_fwrite\(\)](#).

11.1.2.7 [bitmap\\_flip\\_bit\(\)](#)

```
void bitmap_flip_bit (
    BITMAP * bitmap,
    size_t idx)
```

Toggle bit at idx.

## Parameters

bitmap	Bitmap to modify.
--------	-------------------

idx	Zero-based bit index.
-----	-----------------------

Toggle bit at idx.

Parameters

bitmap	The <a href="#">BITMAP</a> to modify.
idx	The index of the bit to flip.

Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line 148 of file [bitmap.c](#).

References [BITMAP::data](#).

Referenced by [SoA\(\)](#).

#### 11.1.2.8 bitmap\_fread()

[BITMAP](#) \* bitmap\_fread (  
FILE \* file)

Read bitmap payload and checksum from a binary stream.

Parameters

file	Readable stream.
------	------------------

Returns

Newly allocated bitmap, or NULL on parse/verification failure.

Read bitmap payload and checksum from a binary stream.

Parameters

file	The file pointer to read from.
------	--------------------------------

Returns

[BITMAP](#)\* A pointer to the newly created [BITMAP](#), or NULL on failure.

Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line 492 of file [bitmap.c](#).

References [bitmap\\_free\(\)](#), [bitmap\\_init\(\)](#), [bitmap\\_validate\\_hash\(\)](#), [BITMAP::byte\\_size](#), [BITMAP::data](#), [log\\_error\(\)](#), and [BITMAP::sha256](#).

11.1.2.9 `bitmap_free()`

```
void bitmap_free (
    BITMAP ** bitmap)
```

Free a bitmap and set the caller pointer to NULL.

Parameters

bitmap	Address of a <a href="#">BITMAP</a> pointer.
--------	--

Free a bitmap and set the caller pointer to NULL.

This function safely deallocates the bitmap's data array and the structure itself, then sets the pointer to NULL to prevent use-after-free bugs and double-free errors. Safe to call with NULL pointers.

Parameters

bitmap	Double pointer to the <a href="#">BITMAP</a> to free
--------	--

Postcondition

\*bitmap is set to NULL after successful deallocation

Safe to call multiple times on the same pointer

Note

No operation performed if `bitmap == NULL` or `*bitmap == NULL`

This is the only correct way to free a [BITMAP](#); never use `free()` directly

Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line [103](#) of file [bitmap.c](#).

Referenced by [bitmap\\_fread\(\)](#), [get\\_root\\_primes\(\)](#), [iZm\\_free\(\)](#), [SiZ\(\)](#), [SiZm\(\)](#), [SiZm\\_vy\(\)](#), [SoA\(\)](#), [SoE\(\)](#), [SoEu\(\)](#), [SoS\(\)](#), [SSoE\(\)](#), [vx\\_free\(\)](#), and [vx\\_search\\_prime\(\)](#).

11.1.2.10 `bitmap_fwrite()`

```
int bitmap_fwrite (
    BITMAP * bitmap,
    FILE * file)
```

Write bitmap payload and checksum to a binary stream.

Parameters

bitmap	Bitmap to serialize.
--------	----------------------

file	Writable stream.
------	------------------

Returns

1 on success, otherwise 0.

Write bitmap payload and checksum to a binary stream.

Parameters

bitmap	The <a href="#">BITMAP</a> to write.
file	The file pointer to write to.

Returns

int 1 on success, 0 on failure.

Examples

[/Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c](#).

Definition at line [440](#) of file [bitmap.c](#).

References [bitmap\\_compute\\_hash\(\)](#), [BITMAP::byte\\_size](#), [BITMAP::data](#), [log\\_error\(\)](#), [BITMAP::sha256](#), and [BITMAP::size](#).

#### 11.1.2.11 bitmap\_get\_bit()

```
int bitmap_get_bit (
    BITMAP * bitmap,
    size_t idx)
```

Read bit value at idx.

Parameters

bitmap	Bitmap to inspect.
idx	Zero-based bit index.

Returns

1 if set, otherwise 0.

Read bit value at idx.

Parameters

bitmap	The <a href="#">BITMAP</a> to read from.
--------	--

idx	The index of the bit to read.
-----	-------------------------------

#### Returns

int 1 if the bit is set, 0 if unset, -1 on error.

#### Examples

[/Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c](#).

Definition at line 137 of file [bitmap.c](#).

References [BITMAP::data](#).

Referenced by [iZ\\_next\\_prime\(\)](#), [process\\_iZ\\_bitmaps\(\)](#), [process\\_N\\_bitmap\(\)](#), [SiZm\(\)](#), [SiZm\\_vy\(\)](#), [SoA\(\)](#), [SoEu\(\)](#), [SoS\(\)](#), [SSoE\(\)](#), [vx\\_collect\\_p\\_gaps\(\)](#), [vx\\_det\\_sieve\(\)](#), [vx\\_prob\\_sieve\(\)](#), [vx\\_search\\_prime\(\)](#), and [vx\\_stream\(\)](#).

#### 11.1.2.12 bitmap\_init()

```
BITMAP * bitmap_init (
    size_t size,
    int set_bits)
```

Allocate a bitmap with size bits.

#### Parameters

size	Number of bits to allocate (must be > 0).
set_bits	Non-zero initializes all bits to 1, otherwise to 0.

#### Returns

Newly allocated bitmap, or NULL on allocation failure.

Allocate a bitmap with size bits.

Allocates memory for both the [BITMAP](#) structure and its underlying data array. The data array is allocated using `calloc()`, ensuring all bits start at 0. The SHA-256 hash field is initialized to all zeros.

#### Parameters

size	The number of bits in the bitmap (must be > 0)
set_bits	If non-zero, initializes all bits to 1; otherwise, all bits are 0.

#### Returns

Pointer to newly allocated [BITMAP](#) on success, NULL on failure

#### Return values

NULL	if size is 0
------	--------------



NULL	if memory allocation fails
------	----------------------------

## Note

Memory allocated: `sizeof(BITMAP) + size/8` bytes

On failure, any partially allocated memory is freed before returning

## Warning

Caller must free returned bitmap using `bitmap_free()`, not `free()`

## Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line 57 of file `bitmap.c`.

References `BITMAP::byte_size`, `BITMAP::data`, `log_error()`, `BITMAP::sha256`, and `BITMAP::size`.

Referenced by `bitmap_clone()`, `bitmap_fread()`, `get_root_primes()`, `iZm_init()`, `SiZ()`, `SiZm()`, `SiZm_vy()`, `SoA()`, `SoE()`, `SoEu()`, `SoS()`, `SSoE()`, and `vx_search_prime()`.

11.1.2.13 `bitmap_set_all()`

```
void bitmap_set_all (
    BITMAP * bitmap)
```

Set all bits to 1.

## Parameters

bitmap	Bitmap to modify.
--------	-------------------

Set all bits to 1.

## Parameters

bitmap	The <code>BITMAP</code> to modify.
--------	------------------------------------

## Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line 170 of file `bitmap.c`.

References `BITMAP::data`, and `BITMAP::size`.

Referenced by `iZm_construct_vx_base()`, `SiZm_vy()`, and `SSoE()`.

11.1.2.14 `bitmap_set_bit()`

```
void bitmap_set_bit (
    BITMAP * bitmap,
    size_t idx)
```

Set bit at `idx` to 1.

## Parameters

bitmap	Bitmap to modify.
--------	-------------------

idx	Zero-based bit index.
-----	-----------------------

Set bit at idx to 1.

Parameters

bitmap	The <a href="#">BITMAP</a> to modify.
idx	The index of the bit to set.

Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line [125](#) of file [bitmap.c](#).

References [BITMAP::data](#).

#### 11.1.2.15 bitmap\_validate\_hash()

```
int bitmap_validate_hash (
    BITMAP * bitmap)
```

Verify [BITMAP::sha256](#) against current bitmap data.

Parameters

bitmap	Bitmap to verify.
--------	-------------------

Returns

1 if checksum matches, otherwise 0.

Verify [BITMAP::sha256](#) against current bitmap data.

Parameters

bitmap	The <a href="#">BITMAP</a> whose hash is validated.
--------	---

Returns

int 1 if the hash matches, 0 otherwise.

Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line [412](#) of file [bitmap.c](#).

References [BITMAP::byte\\_size](#), [BITMAP::data](#), [log\\_error\(\)](#), and [BITMAP::sha256](#).

Referenced by [bitmap\\_fread\(\)](#).

### 11.1.2.16 TEST\_BITMAP()

```
int TEST_BITMAP (  
    int verbose)
```

Run bitmap module tests.

Parameters

verbose	Non-zero enables detailed logging.
---------	------------------------------------

Returns

1 when all tests pass, otherwise 0.

## 11.2 Integer Arrays

Dynamic integer containers used by sieve and toolkit layers.

Files

- file [int\\_arrays.c](#)  
Implementation of dynamic integer array module for 16-bit, 32-bit, and 64-bit unsigned integers.

Data Structures

- struct [UI16\\_ARRAY](#)  
Dynamic array for uint16\_t values.
- struct [UI32\\_ARRAY](#)  
Dynamic array for uint32\_t values.
- struct [UI64\\_ARRAY](#)  
Dynamic array for uint64\_t values.

Functions

- int [TEST\\_GENERIC\\_INT\\_ARRAYS](#) (int verbose)  
Run generic dispatch tests for C11 \_Generic helper macros.

## UI16 API

- [UI16\\_ARRAY](#) \* ui16\_init (int capacity)  
Allocate a UI16 array with an initial capacity.
- void ui16\_free ([UI16\\_ARRAY](#) \*\*array)  
Free a UI16 array and null the caller pointer.
- void ui16\_resize\_to ([UI16\\_ARRAY](#) \*array, int new\_capacity)  
Resize UI16 storage to new\_capacity (must be >= count).
- void ui16\_resize\_to\_fit ([UI16\\_ARRAY](#) \*array)  
Shrink UI16 storage so capacity equals count.
- void ui16\_push ([UI16\\_ARRAY](#) \*array, uint16\_t element)  
Append a uint16 value, growing storage if needed.
- void ui16\_sort ([UI16\\_ARRAY](#) \*array)  
Sort values in ascending order.
- void ui16\_pop ([UI16\\_ARRAY](#) \*array)  
Remove the last element if the array is non-empty.
- void ui16\_compute\_hash ([UI16\\_ARRAY](#) \*array)  
Compute SHA-256 checksum over active payload.
- int ui16\_verify\_hash ([UI16\\_ARRAY](#) \*array)  
Verify the stored checksum against current payload.
- int ui16\_fwrite ([UI16\\_ARRAY](#) \*array, FILE \*file)  
Serialize count, payload, and checksum to a binary stream.
- [UI16\\_ARRAY](#) \* ui16\_fread (FILE \*file)  
Deserialize a UI16 array from a binary stream.
- int TEST\_UI16\_ARRAY (int verbose)  
Execute UI16 test suite.

## UI32 API

- [UI32\\_ARRAY](#) \* ui32\_init (int capacity)  
Allocate a UI32 array with an initial capacity.
- void ui32\_free ([UI32\\_ARRAY](#) \*\*array)  
Free a UI32 array and null the caller pointer.
- void ui32\_resize\_to ([UI32\\_ARRAY](#) \*array, int new\_capacity)  
Resize UI32 storage to new\_capacity (must be >= count).
- void ui32\_resize\_to\_fit ([UI32\\_ARRAY](#) \*array)  
Shrink UI32 storage so capacity equals count.
- void ui32\_push ([UI32\\_ARRAY](#) \*array, uint32\_t element)  
Append a uint32 value, growing storage if needed.
- void ui32\_sort ([UI32\\_ARRAY](#) \*array)  
Sort values in ascending order.
- void ui32\_pop ([UI32\\_ARRAY](#) \*array)  
Remove the last element if the array is non-empty.
- void ui32\_compute\_hash ([UI32\\_ARRAY](#) \*array)  
Compute SHA-256 checksum over active payload.
- int ui32\_verify\_hash ([UI32\\_ARRAY](#) \*array)  
Verify the stored checksum against current payload.
- int ui32\_fwrite ([UI32\\_ARRAY](#) \*array, FILE \*file)  
Serialize count, payload, and checksum to a binary stream.
- [UI32\\_ARRAY](#) \* ui32\_fread (FILE \*file)  
Deserialize a UI32 array from a binary stream.
- int TEST\_UI32\_ARRAY (int verbose)  
Execute UI32 test suite.

## UI64 API

- `UI64_ARRAY * ui64_init (int capacity)`  
Allocate a UI64 array with an initial capacity.
- `void ui64_free (UI64_ARRAY **array)`  
Free a UI64 array and null the caller pointer.
- `void ui64_resize_to (UI64_ARRAY *array, int new_capacity)`  
Resize UI64 storage to new\_capacity (must be >= count).
- `void ui64_resize_to_fit (UI64_ARRAY *array)`  
Shrink UI64 storage so capacity equals count.
- `void ui64_push (UI64_ARRAY *array, uint64_t element)`  
Append a uint64 value, growing storage if needed.
- `void ui64_sort (UI64_ARRAY *array)`  
Sort values in ascending order.
- `void ui64_pop (UI64_ARRAY *array)`  
Remove the last element if the array is non-empty.
- `void ui64_compute_hash (UI64_ARRAY *array)`  
Compute SHA-256 checksum over active payload.
- `int ui64_verify_hash (UI64_ARRAY *array)`  
Verify the stored checksum against current payload.
- `int ui64_fwrite (UI64_ARRAY *array, FILE *file)`  
Serialize count, payload, and checksum to a binary stream.
- `UI64_ARRAY * ui64_fread (FILE *file)`  
Deserialize a UI64 array from a binary stream.
- `int TEST_UI64_ARRAY (int verbose)`  
Execute UI64 test suite.

## 11.2.1 Detailed Description

Dynamic integer containers used by sieve and toolkit layers.

## 11.2.2 Function Documentation

## 11.2.2.1 TEST\_GENERIC\_INT\_ARRAYS()

```
int TEST_GENERIC_INT_ARRAYS (
    int verbose)
```

Run generic dispatch tests for C11 \_Generic helper macros.

## Parameters

verbose	Non-zero enables detailed logging.
---------	------------------------------------

## Returns

1 when all tests pass, otherwise 0.

## 11.3 iZ Public API

< iZ/iZm toolkit structures and helpers.

### Files

- file [iZ\\_apps.c](#)  
High-level application routines built on top of the iZ\_toolkit.
- file [prime\\_sieve.c](#)  
Implementations of classical and SiZ-family sieve algorithms.

### Data Structures

- struct [INPUT\\_SIEVE\\_RANGE](#)  
Input parameters for range sieving/counting.

### SiZ Range Variants

Count/stream primes over a numeric interval.

- typedef struct INPUT\_SIEVE\_RANGE [INPUT\\_SIEVE\\_RANGE](#)  
Input parameters for range sieving/counting.
- uint64\_t [SiZ\\_stream](#) ([INPUT\\_SIEVE\\_RANGE](#) \*range)  
Stream primes in a range to filepath (and return the count).
- uint64\_t [SiZ\\_count](#) ([INPUT\\_SIEVE\\_RANGE](#) \*input\_range, int cores\_num)  
Count primes in a range using multiple worker processes.

### Classic Prime Sieve Algorithms

Baseline sieves up to a numeric limit.

All classic sieve functions take a limit n and return an ascending list of primes  $\leq n$ .

- [UI64\\_ARRAY](#) \* [SoE](#) (uint64\_t n)  
Optimized Sieve of Eratosthenes.
- [UI64\\_ARRAY](#) \* [SSoE](#) (uint64\_t n)  
Segmented Sieve of Eratosthenes.
- [UI64\\_ARRAY](#) \* [SoEu](#) (uint64\_t n)  
Euler (linear) sieve.
- [UI64\\_ARRAY](#) \* [SoS](#) (uint64\_t n)  
Sieve of Sundaram.
- [UI64\\_ARRAY](#) \* [SoA](#) (uint64\_t n)  
Sieve of Atkin.

## iZ-based Sieve Algorithms

Sieve family operating in the  $6x-1$  /  $6x+1$  index space.

- [UI64\\_ARRAY](#) \* [SiZ](#) (uint64\_t n)  
Solid Sieve-iZ (wheel 6, iZ index space).
- [UI64\\_ARRAY](#) \* [SiZm](#) (uint64\_t n)  
Segmented Sieve-iZm (VX segmented, horizontal processing).
- [UI64\\_ARRAY](#) \* [SiZm\\_vy](#) (uint64\_t n)  
Segmented Sieve-iZm (vertical processing; faster, unordered output).

## Prime Generators

Probabilistic prime searches using iZm/VX machinery.

- int [vy\\_random\\_prime](#) (mpz\_t p, int bit\_size, int cores\_num)  
Search for a random prime using vertical (vy) sieving.
- int [vx\\_random\\_prime](#) (mpz\_t p, int bit\_size, int cores\_num)  
Search for a random prime using horizontal (vx) sieving.
- int [iZ\\_next\\_prime](#) (mpz\_t p, mpz\_t base, int forward)  
Advance to the next (or previous) prime from a base value.

### 11.3.1 Detailed Description

< iZ/iZm toolkit structures and helpers.

< Common utilities, types, and dependencies.

High-level entry points for sieves and prime generation.

The API is organized into:

- Classic sieves: operate directly on the integer domain.
- SiZ / SiZm: operate in iZ index space ( $6x-1$  and  $6x+1$ ), optionally segmented.
- Range variants: count/stream primes over large intervals.
- Prime generators: probabilistic searches using the iZm/VX machinery.

### 11.3.2 Typedef Documentation

#### 11.3.2.1 INPUT\_SIEVE\_RANGE

```
typedef struct INPUT_SIEVE_RANGE INPUT_SIEVE_RANGE
```

Input parameters for range sieving/counting.

The interval is interpreted as:

- Start  $Z_s$  from the decimal string start.
- End  $Z_e = Z_s + \text{range} - 1$ .

Forward declaration; concrete definition lives in [iZ\\_api.h](#).

### 11.3.3 Function Documentation

#### 11.3.3.1 iZ\_next\_prime()

```
int iZ_next_prime (
    mpz_t p,
    mpz_t base,
    int forward)
```

Advance to the next (or previous) prime from a base value.

Find the next prime number after a given base.

##### Parameters

p	Output prime (initialized by caller).
base	Starting value.
forward	Non-zero to search forward, 0 to search backward.

##### Returns

1 on success, 0 on failure.

Description: This function searches for the next/previous prime number after a given base using the iZ framework.

##### Parameters

p	The mpz_t variable to store the found prime number.
base	The base number to start the search from.
forward	If true, search for the next prime; if false, search for the previous prime.

##### Returns

1 if a prime is found, 0 otherwise.

Definition at line 819 of file [iZ\\_apps.c](#).

References [IZM::base\\_x5](#), [IZM::base\\_x7](#), [bitmap\\_get\\_bit\(\)](#), [iZ\\_mpz\(\)](#), [iZm\\_free\(\)](#), [iZm\\_init\(\)](#), [log\\_debug\(\)](#), [log\\_error\(\)](#), [MR\\_ROUNDS](#), [VX5](#), and [VX6](#).

#### 11.3.3.2 SiZ()

```
UI64_ARRAY * SiZ (
    uint64_t n)
```

Solid Sieve-iZ (wheel 6, iZ index space).

Classic Sieve-iZ algorithm for prime generation up to n.

##### Parameters

n	Upper bound (inclusive).
---	--------------------------



## Returns

Heap-allocated prime list, or NULL on allocation failure.

## Precondition

$n$  is in  $(10, 10^{12}]$ .

The Sieve-iZ algorithm operates in the iZ index space of numbers of the form  $6x \pm 1$ , excluding all multiples of 2 and 3. It maintains two bitmaps, `x5` and `x7`, representing candidates of the form  $6x - 1$  (iZ-) and  $6x + 1$  (iZ+), respectively. For each index  $x$  whose bit remains set, the corresponding candidate is treated as prime and its composites are marked using the  $Xp$  identity in the iZ space.

This provides a wheel factorization of modulo 6, reducing the number of candidates by approximately a factor of 3 compared to a classical sieve over all integers. After sieving, any remaining set bits correspond to primes of the form  $6x \pm 1$ , which are mapped back to their integer values and appended to the output array alongside the primes 2 and 3.

## Parameters

<code>n</code>	Upper bound (inclusive) for prime generation. Must satisfy $10 < n \leq \text{pow}(10, 12)$ .
----------------	---

## Returns

Pointer to a [UI64\\_ARRAY](#) containing all primes  $\leq n$  on success, or NULL if allocation fails.

Definition at line 444 of file [prime\\_sieve.c](#).

References [UI64\\_ARRAY::array](#), [ASSERT\\_LIMIT](#), [bitmap\\_free\(\)](#), [bitmap\\_init\(\)](#), [UI64\\_ARRAY::count](#), [Pi](#), [process\\_iZ\\_bitmaps\(\)](#), [ui64\\_free\(\)](#), [ui64\\_init\(\)](#), [ui64\\_pop\(\)](#), [ui64\\_push\(\)](#), and [ui64\\_resize\\_to\\_fit\(\)](#).

Referenced by [compute\\_max\\_vx\(\)](#), [iZm\\_init\(\)](#), [SiZm\(\)](#), and [SiZm\\_vy\(\)](#).

## 11.3.3.3 SiZ\_count()

```
uint64_t SiZ_count (
    INPUT_SIEVE_RANGE * input_range,
    int cores_num)
```

Count primes in a range using multiple worker processes.

Multi-process prime counting over an arbitrary numeric range using iZ toolkit.

## Parameters

<code>input_range</code>	Range configuration (output disabled).
<code>cores_num</code>	Requested worker process count.

## Returns

Prime count in the interval, or 0 on error.

This function parallelizes prime counting over the interval  $[Zs, Ze]$  (interpreted from `input_range`) by partitioning the corresponding `iZ` index space into `VX` segments and distributing contiguous blocks of segments across multiple processes. Each child process returns a local prime count to the parent via a pipe.

The parent process aggregates all child counts, applies boundary corrections for endpoints that do not align exactly with  $6x \pm 1$ , and returns the total number of primes in  $[Zs, Ze]$ . Any failure to create pipes, fork children, or initialize per-process resources causes the function to log an error and return 0.

## Parameters

<code>input_range</code>	Pointer to an <a href="#">INPUT_SIEVE_RANGE</a> structure describing the numeric interval and Miller–Rabin configuration.
--------------------------	---

cores_num	Requested number of worker processes. The actual number used is clamped to the available CPU cores and the number of VX segments in the range.
-----------	--

#### Returns

The total number of primes found in  $[Zs, Ze]$  on success, or 0 on any error or child-process failure.

Definition at line 184 of file `iZ_apps.c`.

References `UI64_ARRAY::array`, `compute_l2_vx()`, `UI64_ARRAY::count`, `get_cpu_cores_count()`, `iZ_mpz()`, `iZm_clone()`, `iZm_free()`, `iZm_init()`, `log_error()`, `MIN`, `INPUT_SIEVE_RANGE::mr_rounds`, `VX_SEG::p_count`, `INPUT_SIEVE_RANGE::range`, `range_info_free()`, `range_info_init()`, `SiZm()`, `INPUT_SIEVE_RANGE::start`, `ui64_free()`, `vx_free()`, `vx_full_sieve()`, `vx_init()`, `IZM_RANGE_INFO::Xe`, `IZM_RANGE_INFO::Xs`, `IZM_RANGE_INFO::y_range`, `IZM_RANGE_INFO::Ye`, `IZM_RANGE_INFO::Ys`, `IZM_RANGE_INFO::Ze`, and `IZM_RANGE_INFO::Zs`.

#### 11.3.3.4 SiZ\_stream()

```
uint64_t SiZ_stream (
    INPUT_SIEVE_RANGE * input_range)
```

Stream primes in a range to filepath (and return the count).

Stream primes in an arbitrary numeric range using iZ toolkit.

#### Parameters

range	Range configuration (must include filepath).
-------	--

#### Returns

Prime count in the interval, or 0 on error.

This function counts and streams primes in the interval  $[Zs, Ze]$  defined by the input range. It maps the numeric bounds into the iZ index space ( $6x \pm 1$ ), partitions that space into  $y$  segments of length  $vx$ , and streams primes in each segment using the iZ toolkit.

When an output file path is provided in `input_range->filepath`, the function additionally streams all primes in the requested interval to that file in ascending order, reconstructing them from per-segment prime gaps. Otherwise, it operates in counting mode only.

#### Parameters

input_range	Pointer to an <code>INPUT_SIEVE_RANGE</code> structure describing the start value (as a decimal string), the range length, the optional output filepath, and Miller–Rabin configuration.
-------------	--

## Returns

The total number of primes found in  $[Zs, Ze]$  on success, or 0 on invalid input, allocation failure, or I/O error.

Definition at line 35 of file [iZ\\_apps.c](#).

References [UI64\\_ARRAY::array](#), [UI64\\_ARRAY::count](#), [INPUT\\_SIEVE\\_RANGE::filepath](#), [iZm\\_free\(\)](#), [iZm\\_init\(\)](#), [log\\_error\(\)](#), [MAX](#), [MIN](#), [INPUT\\_SIEVE\\_RANGE::mr\\_rounds](#), [VX\\_SEG::p\\_count](#), [range\\_info\\_free\(\)](#), [range\\_info\\_init\(\)](#), [SiZm\(\)](#), [INPUT\\_SIEVE\\_RANGE::start](#), [ui64\\_free\(\)](#), [IZM\\_RANGE\\_INFO::vx](#), [VX6](#), [vx\\_free\(\)](#), [vx\\_init\(\)](#), [vx\\_stream\(\)](#), [IZM\\_RANGE\\_INFO::Xe](#), [IZM\\_RANGE\\_INFO::Xs](#), [IZM\\_RANGE\\_INFO::y\\_range](#), [IZM\\_RANGE\\_INFO::Ye](#), [IZM\\_RANGE\\_INFO::Ys](#), [IZM\\_RANGE\\_INFO::Ze](#), and [IZM\\_RANGE\\_INFO::Zs](#).

## 11.3.3.5 SiZm()

```
UI64_ARRAY * SiZm (
    uint64_t n)
```

Segmented Sieve-iZm (VX segmented, horizontal processing).

Segmented Sieve-iZm algorithm for prime generation up to  $n$ .

## Parameters

n	Upper bound (inclusive).
---	--------------------------

## Returns

Heap-allocated prime list, or NULL on allocation failure.

## Precondition

$n$  is in  $(10, 10^{12}]$ .

Description: The Sieve-iZm algorithm is a segmented, cache-aware variant of the Sieve-iZ that operates in the iZ index space of numbers of the form  $6x \pm 1$ . It partitions the search interval into segments of length  $vx$ , where  $vx$  is a product of small primes, and constructs a pre-sieved base segment using those small primes once at initialization time.

For each segment, Sieve-iZm clones the base bitmaps and only marks composites of the remaining root primes that do not divide  $vx$ . This amortizes the cost of small-prime sieving across all segments and keeps the working set small and highly cache-resident. After marking, any remaining set bits correspond to primes of the form  $6x \pm 1$  within that segment, which are then mapped back to their integer values and appended to the output array.

Compared to the basic Sieve-iZ, Sieve-iZm achieves the same  $O(n \log \log n)$  time complexity with effectively constant auxiliary memory (aside from the output) and significantly reduced constant factors, making it well-suited for enumerating primes up to large bounds.

## Parameters

n	Upper bound (inclusive) for prime generation. Must satisfy $10 < n \leq \text{pow}(10, 12)$ .
---	---

## Returns

Pointer to a [UI64\\_ARRAY](#) containing all primes  $\leq n$  on success, or NULL on allocation or initialization failure.

Definition at line 516 of file [prime\\_sieve.c](#).

References [UI64\\_ARRAY::array](#), [ASSERT\\_LIMIT](#), [bitmap\\_clear\\_steps\\_simd\(\)](#), [bitmap\\_clone\(\)](#), [bitmap\\_free\(\)](#), [bitmap\\_get\\_bit\(\)](#), [bitmap\\_init\(\)](#), [BITMAP::byte\\_size](#), [compute\\_l2\\_vx\(\)](#), [UI64\\_ARRAY::count](#), [BITMAP::data](#), [iZ\(\)](#), [iZm\\_construct\\_vx\\_base\(\)](#), [iZm\\_solve\\_for\\_x0\(\)](#), [Pi](#), [process\\_iZ\\_bitmaps\(\)](#), [SiZ\(\)](#), [ui64\\_free\(\)](#), [ui64\\_init\(\)](#), [ui64\\_pop\(\)](#), [ui64\\_push\(\)](#), and [ui64\\_resize\\_to\\_fit\(\)](#).

Referenced by [SiZ\\_count\(\)](#), [SiZ\\_stream\(\)](#), and [vx\\_search\\_prime\(\)](#).

## 11.3.3.6 SiZm\_vy()

```
UI64\_ARRAY * SiZm_vy (
    uint64_t n)
```

Segmented Sieve-iZm (vertical processing; faster, unordered output).

Segmented Sieve-iZm with vertical (vy) traversal order.

This variant emphasizes throughput and may not preserve ascending order of returned primes.

## Parameters

n	Upper bound (inclusive).
---	--------------------------

## Returns

Heap-allocated prime list, or NULL on allocation failure.

## Precondition

$n$  is in  $(10, 10^{12}]$ .

This variant prioritizes throughput and returns primes in non-sorted order.

## Parameters

n	Inclusive upper bound for prime generation.
---	---

## Returns

Heap-allocated list of primes  $\leq n$ , or NULL on failure.

Definition at line 623 of file [prime\\_sieve.c](#).

References [UI64\\_ARRAY::array](#), [ASSERT\\_LIMIT](#), [bitmap\\_clear\\_steps\\_simd\(\)](#), [bitmap\\_free\(\)](#), [bitmap\\_get\\_bit\(\)](#), [bitmap\\_init\(\)](#), [bitmap\\_set\\_all\(\)](#), [UI64\\_ARRAY::count](#), [gcd\(\)](#), [get\\_root\\_primes\(\)](#), [iZ\(\)](#), [iZm\\_solve\\_for\\_y0\(\)](#), [UI64\\_ARRAY::ordered](#), [Pi](#), [SiZ\(\)](#), [ui64\\_init\(\)](#), [ui64\\_push\(\)](#), and [ui64\\_resize\\_to\\_fit\(\)](#).

### 11.3.3.7 SoA()

[UI64\\_ARRAY](#) \* SoA (  
     uint64\_t n)

Sieve of Atkin.

Sieve of Atkin: Generates a list of prime numbers up to a given limit using the Sieve of Atkin algorithm.

Parameters

n	Upper bound (inclusive).
---	--------------------------

Returns

Heap-allocated prime list, or NULL on allocation failure.

Precondition

n is in (10, 10<sup>12</sup>].

Description: This function implements the Sieve of Atkin, a modern algorithm to find all prime numbers up to a specified integer n. It initializes a bitmap to mark potential primes and applies the Atkin conditions to identify primes. The function also marks odd multiples of squares of primes as non-prime, then collects the remaining unmarked numbers as primes.

Parameters

n	The upper limit (inclusive) up to which prime numbers are to be found.
---	--

Returns

- [UI64\\_ARRAY](#)\* A pointer to the [UI64\\_ARRAY](#) structure containing the list of primes up to n.
- NULL if memory allocation fails or if n is less than 10.

Definition at line 328 of file [prime\\_sieve.c](#).

References [ASSERT\\_LIMIT](#), [bitmap\\_clear\\_steps\\_simd\(\)](#), [bitmap\\_flip\\_bit\(\)](#), [bitmap\\_free\(\)](#), [bitmap\\_get\\_bit\(\)](#), [bitmap\\_init\(\)](#), [Pi](#), [ui64\\_free\(\)](#), [ui64\\_init\(\)](#), [ui64\\_push\(\)](#), and [ui64\\_resize\\_to\\_fit\(\)](#).

### 11.3.3.8 SoE()

[UI64\\_ARRAY](#) \* SoE (  
     uint64\_t n)

Optimized Sieve of Eratosthenes.

Optimized Sieve of Eratosthenes algorithm to find all primes up to n.

Parameters

n	Upper bound (inclusive).
---	--------------------------

## Returns

Heap-allocated prime list, or NULL on allocation failure.

## Precondition

$n$  is in  $(10, 10^{12}]$ .

Description: This function applies common speedups to the Sieve of Eratosthenes algorithm. It works the same way as the classic sieve but skips checking or marking even numbers and starts with 3, incrementing by 2, it also starts the sieve from  $p*p$ .

## Parameters

<code>n</code>	The upper limit to find primes.
----------------	---------------------------------

## Returns

- A pointer to a [UI64\\_ARRAY](#) structure containing the list of prime numbers up to  $n$ ,
- NULL if memory allocation fails or if  $n$  is less than 10.

Definition at line 72 of file [prime\\_sieve.c](#).

References [ASSERT\\_LIMIT](#), [bitmap\\_free\(\)](#), [bitmap\\_init\(\)](#), [Pi](#), [process\\_N\\_bitmap\(\)](#), [ui64\\_free\(\)](#), [ui64\\_init\(\)](#), and [ui64\\_resize\\_to\\_fit\(\)](#).

## 11.3.3.9 SoEu()

```
UI64_ARRAY * SoEu (
    uint64_t n)
```

Euler (linear) sieve.

Sieve of Euler: Generates a list of prime numbers up to a given limit using the Euler Sieve algorithm.

## Parameters

<code>n</code>	Upper bound (inclusive).
----------------	--------------------------

## Returns

Heap-allocated prime list, or NULL on allocation failure.

## Precondition

$n$  is in  $(10, 10^{12}]$ .

Description: This function uses the Euler Sieve algorithm to find all prime numbers up to a specified limit  $n$ . It marks each composite only once, allowing for a more efficient sieve process. It initializes a bitmap to track prime numbers and iterates through the numbers, marking distinct multiples of each prime found. The function also handles memory allocation and resizing of the primes array as needed.

## Parameters

$n$	The upper limit up to which prime numbers are to be found.
-----	--

## Returns

- `UI64_ARRAY*` A pointer to the `UI64_ARRAY` structure containing the list of primes up to  $n$ .
- `NULL` if memory allocation fails or if  $n$  is less than 10.

Definition at line 200 of file `prime_sieve.c`.

References `UI64_ARRAY::array`, `ASSERT_LIMIT`, `bitmap_clear_bit()`, `bitmap_free()`, `bitmap_get_bit()`, `bitmap_init()`, `UI64_ARRAY::count`, `Pi`, `ui64_free()`, `ui64_init()`, `ui64_push()`, and `ui64_resize_to_fit()`.

## 11.3.3.10 SoS()

`UI64_ARRAY * SoS (`  
     `uint64_t n)`

Sieve of Sundaram.

Sieve of Sundaram: Generates a list of prime numbers up to a given limit using the Sieve of Sundaram algorithm.

## Parameters

$n$	Upper bound (inclusive).
-----	--------------------------

## Returns

Heap-allocated prime list, or `NULL` on allocation failure.

## Precondition

$n$  is in  $(10, 10^{12}]$ .

Description: This function implements the Sieve of Sundaram algorithm to find all prime numbers up to a specified limit  $n$ . The Sieve of Sundaram works by eliminating numbers of the form  $i + j + 2ij$  from a list of integers from 1 to  $k = (n-1)/2$ . If an integer  $x$  in this range is not eliminated, then  $2x+1$  is a prime number. The prime number 2 is handled as a special case, as the sieve only generates odd primes.

## Parameters

$n$	The upper limit (inclusive) up to which prime numbers are to be found.
-----	--



## Returns

- `UI64_ARRAY*` A pointer to the `UI64_ARRAY` structure containing the list of primes up to `n`.
- `NULL` if memory allocation fails or if `n` is less than 10.

Definition at line 264 of file `prime_sieve.c`.

References `ASSERT_LIMIT`, `bitmap_clear_steps_simd()`, `bitmap_free()`, `bitmap_get_bit()`, `bitmap_init()`, `Pi`, `ui64_free()`, `ui64_init()`, `ui64_push()`, and `ui64_resize_to_fit()`.

## 11.3.3.11 SSoE()

`UI64_ARRAY *` SSoE (  
`uint64_t n`)

Segmented Sieve of Eratosthenes.

Segmented Sieve of Eratosthenes algorithm to find all primes up to `n`.

## Parameters

n	Upper bound (inclusive).
---	--------------------------

## Returns

Heap-allocated prime list, or `NULL` on allocation failure.

## Precondition

`n` is in  $(10, 10^{12}]$ .

Description: This function implements the Segmented Sieve of Eratosthenes algorithm to find all prime numbers up to a given limit `n`. It divides the range into segments and uses a bitmap to mark prime numbers in each segment. The function returns a pointer to a `UI64_ARRAY` structure containing the list of prime numbers found. The function also handles memory allocation and resizing of the primes array as needed.

## Parameters

n	The upper limit to find primes.
---	---------------------------------

## Returns

- A pointer to the `UI64_ARRAY` structure containing the list of primes up to `n`,
- `NULL` if memory allocation fails or if `n` is less than 1000.

Definition at line 114 of file `prime_sieve.c`.

References `UI64_ARRAY::array`, `ASSERT_LIMIT`, `bitmap_clear_steps_simd()`, `bitmap_free()`, `bitmap_get_bit()`, `bitmap_init()`, `bitmap_set_all()`, `UI64_ARRAY::count`, `MAX`, `Pi`, `process_N_bitmap()`, `ui64_free()`, `ui64_init()`, `ui64_push()`, and `ui64_resize_to_fit()`.

### 11.3.3.12 vx\_random\_prime()

```
int vx_random_prime (
    mpz_t p,
    int bit_size,
    int cores_num)
```

Search for a random prime using horizontal (vx) sieving.

Generates a random prime candidate using the vx\_search\_prime routine.

#### Parameters

p	Output prime (initialized by caller).
bit_size	Target size of p in bits.
cores_num	Requested worker processes.

#### Returns

1 on success, 0 on failure.

Description: This function generates a random prime of a given bit size using the vx\_search\_prime routine. It initializes the random base and sets up the search parameters. The function also allows for parallel processing of the search using multiple child processes. The generated prime is stored in the provided mpz\_t p variable.

#### Parameters

p	The mpz_t variable to store the generated prime number.
bit_size	The target bit size of the prime.
cores_num	The number of cores to use for parallel processing.

#### Returns

1 if a prime is found, 0 otherwise.

Definition at line 692 of file [iZ\\_apps.c](#).

References [log\\_error\(\)](#), [MAX](#), [VX5](#), [VX6](#), and [vx\\_search\\_prime\(\)](#).

### 11.3.3.13 vy\_random\_prime()

```
int vy_random_prime (
    mpz_t p,
    int bit_size,
    int cores_num)
```

Search for a random prime using vertical (vy) sieving.

Generates a random prime candidate using the vy\_search\_prime routine.

#### Parameters

p	Output prime (initialized by caller).
---	---------------------------------------

bit_size	Target size of p in bits.
cores_num	Requested worker processes.

#### Returns

1 on success, 0 on failure.

Description: This function generates a random prime of a given bit size using the `vy_search_prime` routine. It initializes the random base and sets up the search parameters. The function also allows for parallel processing of the search using multiple child processes. The generated prime is stored in the provided `mpz_t p` variable.

#### Parameters

p	The <code>mpz_t</code> variable to store the generated prime number.
bit_size	The target bit size of the prime.
cores_num	The number of cores to use for parallel processing.

#### Returns

1 if a prime is found, 0 otherwise.

Definition at line 545 of file [iZ\\_apps.c](#).

References [compute\\_max\\_vx\(\)](#), [log\\_error\(\)](#), [MAX](#), and [vy\\_search\\_prime\(\)](#).

## 11.4 Toolkit (iZ/iZm)

Internal building blocks for SiZ and SiZm implementations.

#### Files

- file [iZ\\_toolkit.c](#)  
Implementation of iZ index space helpers and iZm/VX segment machinery.

#### Data Structures

- struct [IZM](#)  
Precomputed iZm assets for repeated VX-segment sieving.
- struct [VX\\_SEG](#)  
Runtime state for one VX segment at a specific y.
- struct [IZM\\_RANGE\\_INFO](#)  
Precomputed iZm coordinates for an inclusive numeric interval.

## Macros

- `#define MR_ROUNDS 25`

## Functions

- `void iZm_construct_vx_base (uint64_t vx, BITMAP *base_x5, BITMAP *base_x7)`  
Build pre-sieved base bitmaps for a VX segment.
- `IZM_RANGE_INFO range_info_init (INPUT_SIEVE_RANGE *input_range, int vx)`  
Map a decimal range input into iZm coordinates and segment bounds.
- `void range_info_free (IZM_RANGE_INFO *info)`  
Clear all GMP fields owned by info.

## iZ Mapping Helpers

- `uint64_t iZ (uint64_t x, int i)`  
Map iZ coordinates to an integer:  $6*x + i$ .
- `void iZ_mpz (mpz_t z, mpz_t x, int i)`  
GMP variant of `iZ()`.
- `void process_iZ_bitmaps (UI64_ARRAY *primes, BITMAP *x5, BITMAP *x7, uint64_t x_limit)`  
Traverse iZ bitmaps, emit surviving primes, and mark composites.
- `void get_root_primes (UI64_ARRAY *primes, uint64_t limit)`  
Generate primes up to limit for deterministic sieving.
- `int check_primality (mpz_t n, int rounds)`  
Check the primality of a number using GMP's probabilistic test.

## IZM Lifecycle

- `IZM * iZm_init (size_t vx)`  
Allocate and initialize an IZM object for a given VX.
- `IZM * iZm_clone (IZM *src)`  
Deep-copy an IZM object for per-worker ownership.
- `void iZm_free (IZM **iZm)`  
Release an IZM object and set the caller pointer to NULL.

## VX Selection Helpers

- `uint64_t compute_vx_k (int k)`  
Calculate  $VX_{\{k\}}$ .
- `uint64_t compute_l2_vx (uint64_t n)`  
Choose VX using an L2-cache-aware heuristic.
- `void compute_max_vx (mpz_t vx, int bit_size)`  
Compute largest VX below  $2^{\text{bit\_size}}$ .

## Modular Hit Solvers

- `uint64_t iZm_solve_for_x0` (int m\_id, uint64\_t p, uint64\_t vx, uint64\_t y)  
Solve first x-hit of prime p for line m\_id in segment y.
- `uint64_t iZm_solve_for_x0_mpz` (int m\_id, uint64\_t p, uint64\_t vx, mpz\_t y)  
GMP variant of `iZm_solve_for_x0()` for very large y.
- `int64_t iZm_solve_for_y0` (int m\_id, uint64\_t p, uint64\_t vx, uint64\_t x)  
Solve first y-hit for fixed x in vertical (vy) scanning.

## VX Segment Lifecycle and Execution

- `VX_SEG * vx_init` (IZM \*iZm, int start\_x, int end\_x, char \*y\_str, int mr\_rounds)  
Initialize and deterministically sieve one VX segment.
- `void vx_free` (VX\_SEG \*\*vx\_obj)  
Free a VX segment and all owned resources.
- `void vx_collect_p_gaps` (VX\_SEG \*vx\_obj)  
Extract prime-gap encoding from a fully sieved segment.
- `void vx_full_sieve` (VX\_SEG \*vx\_obj, int collect\_p\_gaps)  
Complete segment processing (probabilistic stage and optional gaps).
- `void vx_stream` (VX\_SEG \*vx\_obj, FILE \*output)  
Stream segment primes to an output stream.

## Random Prime Search Routines

- `int vx_search_prime` (mpz\_t p, int m\_id, int vx, int bit\_size)  
Horizontal iZm/VX random-prime search.
- `int vy_search_prime` (mpz\_t p, int m\_id, mpz\_t vx)  
Vertical iZm/VY random-prime search.

## Toolkit Tests

- `int TEST_IZM` (int verbose)  
Run IZM construction and solver tests.
- `int TEST_VX_SEG` (int verbose)  
Run VX segment tests.

## Standard VX Sizes (primorial products excluding 2,3)

- `#define VX2` (5 \* 7ULL)
- `#define VX3` (VX2 \* 11ULL)
- `#define VX4` (VX3 \* 13ULL)
- `#define VX5` (VX4 \* 17ULL)
- `#define VX6` (VX5 \* 19ULL)
- `#define VX7` (VX6 \* 23ULL)
- `#define VX8` (VX7 \* 29ULL)

### 11.4.1 Detailed Description

Internal building blocks for SiZ and SiZm implementations.

## 11.4.2 Macro Definition Documentation

### 11.4.2.1 MR\_ROUNDS

```
#define MR_ROUNDS 25
```

Default Miller-Rabin rounds used by toolkit search/sieve helpers.

Definition at line 79 of file [iZ\\_toolkit.h](#).

Referenced by [iZ\\_next\\_prime\(\)](#), [vx\\_init\(\)](#), [vx\\_search\\_prime\(\)](#), and [vy\\_search\\_prime\(\)](#).

### 11.4.2.2 VX2

```
#define VX2 (5 * 7ULL)
```

35

Definition at line 69 of file [iZ\\_toolkit.h](#).

### 11.4.2.3 VX3

```
#define VX3 (VX2 * 11ULL)
```

385

Definition at line 70 of file [iZ\\_toolkit.h](#).

### 11.4.2.4 VX4

```
#define VX4 (VX3 * 13ULL)
```

5005

Definition at line 71 of file [iZ\\_toolkit.h](#).

### 11.4.2.5 VX5

```
#define VX5 (VX4 * 17ULL)
```

85085

Definition at line 72 of file [iZ\\_toolkit.h](#).

Referenced by [iZ\\_next\\_prime\(\)](#), and [vx\\_random\\_prime\(\)](#).

#### 11.4.2.6 VX6

```
#define VX6 (VX5 * 19ULL)
```

1616615

Definition at line 73 of file [iZ\\_toolkit.h](#).

Referenced by [iZ\\_next\\_prime\(\)](#), [SiZ\\_stream\(\)](#), and [vx\\_random\\_prime\(\)](#).

#### 11.4.2.7 VX7

```
#define VX7 (VX6 * 23ULL)
```

37260615

Definition at line 74 of file [iZ\\_toolkit.h](#).

#### 11.4.2.8 VX8

```
#define VX8 (VX7 * 29ULL)
```

1080558835

Definition at line 75 of file [iZ\\_toolkit.h](#).

### 11.4.3 Function Documentation

#### 11.4.3.1 [check\\_primality\(\)](#)

```
int check_primality (
    mpz_t n,
    int rounds)
```

Check the primality of a number using GMP's probabilistic test.

Parameters

n	Number to check.
rounds	Number of Miller-Rabin rounds.

Returns

Non-zero if probably prime, 0 if composite.

Parameters

n	Number to check.
---	------------------

rounds	Number of Miller-Rabin rounds.
--------	--------------------------------

Returns

Non-zero if probably prime, 0 if composite.

This function serves as a single source of truth for primality testing, wrapping GMP's `mpz_probab_prime_p` function, allowing for changes to the underlying primality testing method in the future without affecting the API.

Definition at line 153 of file `iz_toolkit.c`.

Referenced by `vx_prob_sieve()`, `vx_search_prime()`, `vx_stream()`, and `vy_search_prime()`.

#### 11.4.3.2 `compute_l2_vx()`

```
uint64_t compute_l2_vx (
    uint64_t n)
```

Choose VX using an L2-cache-aware heuristic.

Choose a cache-aware VX size based on detected L2 capacity.

Parameters

n	Target numeric limit.
---	-----------------------

Returns

Selected VX size.

Parameters

n	Target numeric sieve bound.
---	-----------------------------

Returns

L2 compatible VX size.

Definition at line 306 of file `iz_toolkit.c`.

References `get_cpu_L2_cache_size_bits()`, `MIN`, and `s_primes`.

Referenced by `SiZ_count()`, and `SiZm()`.

#### 11.4.3.3 `compute_max_vx()`

```
void compute_max_vx (
    mpz_t vx,
    int bit_size)
```

Compute largest VX below  $2^{\text{bit\_size}}$ .

Compute the maximum vx such that  $vx < 2^{\text{bit\_size}}$ .

Parameters

vx	Output mpz_t containing VX.
----	-----------------------------



bit_size	Bit-size ceiling.
----------	-------------------

Parameters:

Parameters

vx	mpz_t variable to store the computed maximum vx value.
bit_size	Integer representing the bit size limit for vx.

Definition at line 331 of file [iZ\\_toolkit.c](#).

References [UI64\\_ARRAY::array](#), and [SiZ\(\)](#).

Referenced by [vy\\_random\\_prime\(\)](#).

#### 11.4.3.4 compute\_vx\_k()

```
uint64_t compute_vx_k (
    int k)
```

Calculate  $VX_{\{k\}}$ .

Calculate a  $VX_{\{k\}}$  as the product of the first k primes  $> 3$ .

Parameters

k	Number of multiplicative prime factors ( $> 3$ ).
---	---

Returns

Selected VX size.

Parameters

k	number of consecutive primes ( $> 3$ ) to include in the product.
---	---

Returns

Computed  $VX_k$  value.

Definition at line 286 of file [iZ\\_toolkit.c](#).

References [s\\_primes](#).

#### 11.4.3.5 get\_root\_primes()

```
void get_root_primes (
    UI64_ARRAY * primes,
    uint64_t limit)
```

Generate primes up to limit for deterministic sieving.

Generate root primes up to limit using iZ bitmap traversal.

Parameters

primes	Output array, appended in ascending order.
--------	--

limit	Numeric upper bound.
primes	Destination array.
limit	Numeric upper bound.

Definition at line 112 of file [iZ\\_toolkit.c](#).

References [bitmap\\_free\(\)](#), [bitmap\\_init\(\)](#), [process\\_iZ\\_bitmaps\(\)](#), [ui64\\_free\(\)](#), and [ui64\\_push\(\)](#).

Referenced by [SiZm\\_vy\(\)](#).

#### 11.4.3.6 iZ()

```
uint64_t iZ (
    uint64_t x,
    int i)    [inline]
```

Map iZ coordinates to an integer:  $6x + i$ .

Parameters

x	iZ x-coordinate.
i	Matrix identifier, typically -1 or +1.

Returns

Mapped integer value.

Map iZ coordinates to an integer:  $6x + i$ .

Parameters:

Parameters

x	(uint64_t) The value of x in $6x + i$ .
i	(int) The value of i in $6x + i$ .

Returns

The computed value  $6x + i$  as a 64-bit unsigned integer.

Definition at line 24 of file [iZ\\_toolkit.c](#).

Referenced by [process\\_iZ\\_bitmaps\(\)](#), [SiZm\(\)](#), and [SiZm\\_vy\(\)](#).

#### 11.4.3.7 iZ\_mpz()

```
void iZ_mpz (
    mpz_t z,
    mpz_t x,
    int i)
```

GMP variant of [iZ\(\)](#).

Parameters

z	Output value.
---	---------------

x	Input x-coordinate.
i	Matrix identifier, typically -1 or +1.

GMP variant of [iZ\(\)](#).

Parameters:

Parameters

z	(mpz_t) The result of the calculation $6x + i$ .
x	(mpz_t) The input value of x in $6x + i$ .
i	(int) The value of i in $6x + i$ .

Definition at line 37 of file [iZ\\_toolkit.c](#).

Referenced by [iZ\\_next\\_prime\(\)](#), [SiZ\\_count\(\)](#), [vx\\_prob\\_sieve\(\)](#), [vx\\_search\\_prime\(\)](#), [vx\\_set\\_base\\_values\(\)](#), [vx\\_stream\(\)](#), and [vy\\_search\\_prime\(\)](#).

#### 11.4.3.8 iZm\_clone()

```
IZM * iZm_clone (
    IZM * src)
```

Deep-copy an [IZM](#) object for per-worker ownership.

Deep-copy an [IZM](#) object for independent worker usage.

Parameters

src	Source <a href="#">IZM</a> instance.
-----	--------------------------------------

Returns

Newly allocated clone, or NULL on failure.

Parameters

src	Source <a href="#">IZM</a> object.
-----	------------------------------------

Returns

Cloned [IZM](#) object, or NULL on failure.

Definition at line 232 of file [iZ\\_toolkit.c](#).

References [UI64\\_ARRAY::array](#), [IZM::base\\_x5](#), [IZM::base\\_x7](#), [bitmap\\_clone\(\)](#), [UI64\\_ARRAY::capacity](#), [UI64\\_ARRAY::count](#), [iZm\\_free\(\)](#), [IZM::k\\_vx](#), [log\\_error\(\)](#), [IZM::root\\_primes](#), [ui64\\_init\(\)](#), and [IZM::vx](#).

Referenced by [SiZ\\_count\(\)](#).

11.4.3.9 `iZm_construct_vx_base()`

```
void iZm_construct_vx_base (
    uint64_t vx,
    BITMAP * base_x5,
    BITMAP * base_x7)
```

Build pre-sieved base bitmaps for a VX segment.

Constructs a pre-sieved iZm base segment of size vx.

Parameters

vx	Segment width.
base_x5	Output bitmap for 6x-1 candidates.
base_x7	Output bitmap for 6x+1 candidates.

Description: This function constructs a pre-sieved iZm base segment of size vx. It marks all composites of small primes that divide vx in the bitmaps base\_x5 (iZ-) and base\_x7 (iZ+). This pre-sieved base can then be used as a template for sieving all vx segments.

Parameters:

Parameters

iZm	Pointer to the <a href="#">IZM</a> structure containing the base segment bitmaps.
-----	---

Definition at line 363 of file [iZ\\_toolkit.c](#).

References [bitmap\\_clear\\_bit\(\)](#), [bitmap\\_clear\\_steps\\_simd\(\)](#), [bitmap\\_set\\_all\(\)](#), [s\\_primes](#), and [s\\_primes\\_count](#).

Referenced by [iZm\\_init\(\)](#), and [SiZm\(\)](#).

11.4.3.10 `iZm_free()`

```
void iZm_free (
    IZM ** iZm)
```

Release an [IZM](#) object and set the caller pointer to NULL.

Free the memory allocated for an [IZM](#) structure.

Parameters

iZm	Address of an <a href="#">IZM</a> pointer.
-----	--

Parameters:

Parameters

iZm	A pointer to the <a href="#">IZM</a> structure to be freed.
-----	---

Definition at line 266 of file [iZ\\_toolkit.c](#).

References [bitmap\\_free\(\)](#), and [ui64\\_free\(\)](#).

Referenced by [iZ\\_next\\_prime\(\)](#), [iZm\\_clone\(\)](#), [iZm\\_init\(\)](#), [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

## 11.4.3.11 iZm\_init()

```
IZM * iZm_init (
    size_t vx)
```

Allocate and initialize an [IZM](#) object for a given VX.

Initialize an iZm structure for a given vx size.

## Parameters

vx	Segment width; must be odd, not divisible by 3, and $\geq 35$ .
----	---

## Returns

Initialized [IZM](#) object, or NULL on failure.

This function initializes an iZm structure for a specified vx size. It allocates memory for the structure, generates root primes for deterministic sieving, initializes base bitmaps, and constructs pre-sieved base segments for iZm5 and iZm7.

## Parameters:

## Parameters

vx	The size of the segment for the iZm structure.
----	--

## Returns

A pointer to the initialized [IZM](#) structure.

Definition at line 184 of file [iZ\\_toolkit.c](#).

References [IZM::base\\_x5](#), [IZM::base\\_x7](#), [bitmap\\_init\(\)](#), [compute\\_k\\_vx\(\)](#), [iZm\\_construct\\_vx\\_base\(\)](#), [iZm\\_free\(\)](#), [IZM::k\\_vx](#), [log\\_error\(\)](#), [IZM::root\\_primes](#), [SiZ\(\)](#), [ui64\\_free\(\)](#), and [IZM::vx](#).

Referenced by [iZ\\_next\\_prime\(\)](#), [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

## 11.4.3.12 iZm\_solve\_for\_x0()

```
uint64_t iZm_solve_for_x0 (
    int m_id,
    uint64_t p,
    uint64_t vx,
    uint64_t y)
```

Solve first x-hit of prime p for line m\_id in segment y.

Compute the first composite hit of p in the yth vx segment in iZm index space.

## Parameters

m_id	Line id (-1 for x5, +1 for x7).
------	---------------------------------

p	Prime used for marking.
vx	Segment width.
y	Segment index.

Returns

First x index to clear for this prime/line.

Description: This function solves for the smallest x that satisfies:  $(x + vx * y) \equiv x_p \pmod{p}$ , where  $x_p$  is normalized based on the `m_id` and `p` to either  $x_p$  or  $p - x_p$ , then computes x and returns it.

Parameters:

Parameters

m_id	int matrix identifier (-1 for iZm5, 1 for iZm7).
p	Unsigned 64-bit integer parameter.
vx	size_t parameter representing the vx value.
y	Unsigned 64-bit integer parameter.

Returns

The computed x value as a 64-bit unsigned integer.

Definition at line 415 of file [iZ\\_toolkit.c](#).

Referenced by [SiZm\(\)](#), and [vx\\_det\\_sieve\(\)](#).

#### 11.4.3.13 iZm\_solve\_for\_x0\_mpz()

```
uint64_t iZm_solve_for_x0_mpz (
    int m_id,
    uint64_t p,
    uint64_t vx,
    mpz_t y)
```

GMP variant of [iZm\\_solve\\_for\\_x0\(\)](#) for very large y.

Solve for x given `m_id`, `p`, `vx`, and `y` using GMP. Same as above but using GMP. Suitable for arbitrary y values.

Parameters

m_id	Line id (-1 for x5, +1 for x7).
p	Prime used for marking.
vx	Segment width.
y	Segment index as mpz.

Returns

First x index to clear for this prime/line.

Parameters:

Parameters

m_id	int matrix identifier (-1 for iZm5, 1 for iZm7).
------	--

p	Unsigned 64-bit integer parameter.
vx	Size_t parameter representing the vx value.
y	mpz_t parameter representing the y value.

Returns

The computed x value as a 64-bit unsigned integer.

Definition at line 449 of file [iZ\\_toolkit.c](#).

Referenced by [vx\\_det\\_sieve\(\)](#), and [vx\\_search\\_prime\(\)](#).

#### 11.4.3.14 iZm\_solve\_for\_y0()

```
int64_t iZm_solve_for_y0 (
    int m_id,
    uint64_t p,
    uint64_t vx,
    uint64_t x)
```

Solve first y-hit for fixed x in vertical (vy) scanning.

Compute the first composite hit of p in the xth vy segment in iZm of width vx.

Parameters

m_id	Line id (-1 for x5, +1 for x7).
p	Prime used for marking.
vx	Segment width.
x	Fixed x-coordinate.

Returns

y index on success, -1 when no modular solution exists.

Description: This function solves for the smallest y that satisfies the equation  $(x + vx * y) \equiv x\_p \pmod{p}$ , where  $x\_p$  is normalized based on the  $m\_id$  and  $p$ . The function checks if  $p$  and  $vx$  are co-primes, and if not, it returns -1 indicating no solution can be found. If they are co-primes, it calculates the multiplicative inverse of  $vx$  modulo  $p$  and computes  $y$  using the formula  $y = (\text{delta} * vx\_inv) \% p$ , where  $\text{delta}$  is the difference between  $x\_p$  and  $x$  modulo  $p$ .

Parameters:

Parameters

m_id	int matrix identifier (-1 for iZm5, 1 for iZm7).
p	Unsigned 64-bit integer parameter.



vx	Size_t parameter representing the vx value.
x	Unsigned 64-bit integer parameter.

Returns

The computed y value as a 64-bit unsigned integer.

Definition at line 490 of file [iZ\\_toolkit.c](#).

References [gcd\(\)](#), and [modular\\_inverse\(\)](#).

Referenced by [SiZm\\_vy\(\)](#).

#### 11.4.3.15 process\_iZ\_bitmaps()

```
void process_iZ_bitmaps (
    UI64_ARRAY * primes,
    BITMAP * x5,
    BITMAP * x7,
    uint64_t x_limit)
```

Traverse iZ bitmaps, emit surviving primes, and mark composites.

Consume iZ candidate bitmaps, emit primes, and mark root composites.

Parameters

primes	Output prime array.
x5	Bitmap for 6x-1 candidates.
x7	Bitmap for 6x+1 candidates.
x_limit	Exclusive x upper bound.
primes	Destination array for discovered primes.
x5	Bitmap for 6x-1 candidates.
x7	Bitmap for 6x+1 candidates.
x_limit	Exclusive x upper bound.

Definition at line 70 of file [iZ\\_toolkit.c](#).

References [bitmap\\_clear\\_steps\\_simd\(\)](#), [bitmap\\_get\\_bit\(\)](#), [iZ\(\)](#), and [ui64\\_push\(\)](#).

Referenced by [get\\_root\\_primes\(\)](#), [SiZ\(\)](#), and [SiZm\(\)](#).

#### 11.4.3.16 range\_info\_free()

```
void range_info_free (
    IZM_RANGE_INFO * info)
```

Clear all GMP fields owned by info.

Definition at line 989 of file [iZ\\_toolkit.c](#).

References [IZM\\_RANGE\\_INFO::Xe](#), [IZM\\_RANGE\\_INFO::Xs](#), [IZM\\_RANGE\\_INFO::Ye](#), [IZM\\_RANGE\\_INFO::Ys](#), [IZM\\_RANGE\\_INFO::Ze](#), and [IZM\\_RANGE\\_INFO::Zs](#).

Referenced by [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

11.4.3.17 `range_info_init()`

```
IZM_RANGE_INFO range_info_init (
    INPUT_SIEVE_RANGE * input_range,
    int vx)
```

Map a decimal range input into iZm coordinates and segment bounds.

## Parameters

<code>input_range</code>	Range input (start, range).
<code>vx</code>	iZm segment width.

## Returns

Initialized range-info object; `y_range < 0` indicates invalid input or an unsupported y-span for the current implementation.

Definition at line 944 of file [iZ\\_toolkit.c](#).

References [log\\_error\(\)](#), [INPUT\\_SIEVE\\_RANGE::range](#), [INPUT\\_SIEVE\\_RANGE::start](#), [IZM\\_RANGE\\_INFO::vx](#), [IZM\\_RANGE\\_INFO::Xe](#), [IZM\\_RANGE\\_INFO::Xs](#), [IZM\\_RANGE\\_INFO::y\\_range](#), [IZM\\_RANGE\\_INFO::Ye](#), [IZM\\_RANGE\\_INFO::Ys](#), [IZM\\_RANGE\\_INFO::Ze](#), and [IZM\\_RANGE\\_INFO::Zs](#).

Referenced by [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

11.4.3.18 `vx_collect_p_gaps()`

```
void vx_collect_p_gaps (
    VX_SEG * vx_obj)
```

Extract prime-gap encoding from a fully sieved segment.

Convert survivor bits into compact prime-gap encoding.

## Parameters

<code>vx_obj</code>	Segment object.
---------------------	-----------------

This routine must run after deterministic/probabilistic sieving is complete. It populates [VX\\_SEG::p\\_gaps](#) for downstream streaming and gap traversal.

## Parameters

<code>vx_obj</code>	Segment object with validated prime survivors.
---------------------	--

Definition at line 787 of file [iZ\\_toolkit.c](#).

References [bitmap\\_get\\_bit\(\)](#), [VX\\_SEG::is\\_large\\_limit](#), [VX\\_SEG::p\\_count](#), [VX\\_SEG::p\\_gaps](#), [VX\\_SEG::start\\_x](#), [ui16\\_init\(\)](#), [ui16\\_push\(\)](#), [vx\\_full\\_sieve\(\)](#), [VX\\_SEG::x5](#), [VX\\_SEG::x7](#), and [VX\\_SEG::y](#).

Referenced by [vx\\_full\\_sieve\(\)](#).

## 11.4.3.19 vx\_free()

```
void vx_free (
    VX_SEG ** vx_obj)
```

Free a VX segment and all owned resources.

Free all memory owned by a VX segment object.

## Parameters

vx_obj	Address of a <a href="#">VX_SEG</a> pointer.
vx_obj	Address of the <a href="#">VX_SEG</a> pointer to release.

Definition at line [759](#) of file [iZ\\_toolkit.c](#).

References [bitmap\\_free\(\)](#), and [ui16\\_free\(\)](#).

Referenced by [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

## 11.4.3.20 vx\_full\_sieve()

```
void vx_full_sieve (
    VX_SEG * vx_obj,
    int collect_p_gaps)
```

Complete segment processing (probabilistic stage and optional gaps).

This function performs the sieve process on a given vx and y defined in the [VX\\_SEG](#) structure, and stores the primes gaps in the vx\_obj->p\_gaps array.

## Parameters

vx_obj	Segment object.
collect_p_gaps	Non-zero to populate <a href="#">VX_SEG::p_gaps</a> .

Description: This function combines deterministic sieving and probabilistic primality tests to identify prime candidates in a standard VX segment of a specific y in the iZ-Matrix. It could be used to only count primes in the vx segment, and optionally populates the vx\_obj->p\_gaps array with prime gaps between consecutive primes detected in the segment.

## Parameters

vx_obj	The <a href="#">VX_SEG</a> to be processed.
collect_p_gaps	0 only counts primes and doesn't store prime gaps, other non-zero int values populates vx_obj->p_gaps with detected prime gaps.

Definition at line [847](#) of file [iZ\\_toolkit.c](#).

References [VX\\_SEG::is\\_large\\_limit](#), [vx\\_collect\\_p\\_gaps\(\)](#), and [vx\\_prob\\_sieve\(\)](#).

Referenced by [SiZ\\_count\(\)](#), and [vx\\_collect\\_p\\_gaps\(\)](#).

## 11.4.3.21 vx\_init()

```

VX_SEG * vx_init (
    IZM * iZm,
    int start_x,
    int end_x,
    char * y_str,
    int mr_rounds)

```

Initialize and deterministically sieve one VX segment.

Initialize the members of the **VX\_SEG** structure with the given parameters and defaults.

## Parameters

iZm	Initialized toolkit context.
start_x	Inclusive x start index.
end_x	Inclusive x end index.
y_str	Segment index y as a decimal string.
mr_rounds	Miller-Rabin rounds (0 uses default).

## Returns

Initialized and partially processed segment, or NULL on failure.

Description: This function allocates memory for a **VX\_SEG** structure and initializes its members. The count is initialized to 0, and the p\_gaps array is allocated with an initial size of (vx/2).

## Parameters:

## Parameters

y_str	A character pointer representing a numeric string.
-------	--

## Returns

**VX\_SEG\*** A pointer to the initialized **VX\_SEG** structure. NULL if memory allocation fails or y\_str is not a numeric string.

Definition at line 713 of file [iz\\_toolkit.c](#).

References [IZM::base\\_x5](#), [IZM::base\\_x7](#), [VX\\_SEG::bit\\_ops](#), [bitmap\\_clone\(\)](#), [VX\\_SEG::end\\_x](#), [log\\_error\(\)](#), [MAX](#), [MIN](#), [MR\\_ROUNDS](#), [VX\\_SEG::mr\\_rounds](#), [VX\\_SEG::p\\_count](#), [VX\\_SEG::p\\_gaps](#), [VX\\_SEG::p\\_test\\_ops](#), [VX\\_SEG::start\\_x](#), [IZM::vx](#), [VX\\_SEG::vx](#), [vx\\_det\\_sieve\(\)](#), [vx\\_set\\_base\\_values\(\)](#), [VX\\_SEG::x5](#), and [VX\\_SEG::x7](#).

Referenced by [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

## 11.4.3.22 vx\_search\_prime()

```
int vx_search_prime (
    mpz_t p,
    int m_id,
    int vx,
    int bit_size)
```

Horizontal iZm/VX random-prime search.

horizontal search routine for generating a random prime.

Parameters

p	Output prime.
m_id	Requested line id (-1, +1, or random when other value).
vx	Segment width.
bit_size	Target bit size.

Returns

1 on success, otherwise 0.

Description: This function searches for a prime number using the given parameters. It combines the iZ-Matrix filtering techniques with the Miller-Rabin primality test. The search is performed by finding suitable x values that do not correspond to composites of primes that divide vx. Then, it iterates over y values in the equation  $p = iZ(x + vx * y)$  until a prime is found.

Parameters

p	The prime number found in the search.
m_id	The identifier (1 or -1) for the iZ matrix.
vx	The horizontal vector of the iZ matrix.
bit_size	The target bit size of the prime.

Returns

1 if a prime is found, 0 otherwise.

Definition at line 1017 of file [iZ\\_toolkit.c](#).

References [UI64\\_ARRAY::array](#), [bitmap\\_clear\\_steps\(\)](#), [bitmap\\_free\(\)](#), [bitmap\\_get\\_bit\(\)](#), [bitmap\\_init\(\)](#), [check\\_primality\(\)](#), [UI64\\_ARRAY::count](#), [gmp\\_seed\\_randstate\(\)](#), [iZ\\_mpz\(\)](#), [iZm\\_solve\\_for\\_x0\\_mpz\(\)](#), [log\\_error\(\)](#), [MAX](#), [MR\\_ROUNDS](#), [SiZm\(\)](#), and [ui64\\_free\(\)](#).

Referenced by [vx\\_random\\_prime\(\)](#).

#### 11.4.3.23 vx\_stream()

```
void vx_stream (  
    VX\_SEG * vx_obj,  
    FILE * output)
```

Stream segment primes to an output stream.

Stream segment primes to an output stream in traversal order.

Parameters

vx_obj	Segment object.
--------	-----------------

output	Writable output stream (e.g. stdout or a file).
vx_obj	Segment object.
output	Destination stream (stdout or file).

Definition at line 866 of file [iZ\\_toolkit.c](#).

References [bitmap\\_clear\\_bit\(\)](#), [bitmap\\_get\\_bit\(\)](#), [check\\_primalty\(\)](#), [VX\\_SEG::is\\_large\\_limit](#), [iZ\\_mpz\(\)](#), [VX\\_SEG::mr\\_rounds](#), [VX\\_SEG::p\\_count](#), [VX\\_SEG::p\\_test\\_ops](#), [VX\\_SEG::start\\_x](#), [VX\\_SEG::x5](#), [VX\\_SEG::x7](#), and [VX\\_SEG::yvx](#).

Referenced by [SiZ\\_stream\(\)](#).

#### 11.4.3.24 vy\_search\_prime()

```
int vy_search_prime (
    mpz_t p,
    int m_id,
    mpz_t vx)
```

Vertical iZm/VY random-prime search.

vertical search routine for generating a random prime.

Parameters

p	Output prime.
m_id	Requested line id (-1, +1, or random when other value).
vx	Segment width.

Returns

1 on success, otherwise 0.

Description: This function searches for a prime number using the given parameters. It combines the iZ-Matrix filtering techniques with the Miller-Rabin primality test. The search is performed by finding a suitable x value that does not correspond to a composite of a prime that divides vx. Then, it iterates over y value in the equation  $p = iZ(x + vx * y, m\_id)$  until a prime is found.

Parameters

p	The prime number found in the search.
m_id	The identifier (1 or -1) for the iZ matrix.
vx	The horizontal vector of the iZ matrix.

Returns

1 if a prime is found, 0 otherwise.

Definition at line 1114 of file [iZ\\_toolkit.c](#).

References [check\\_primalty\(\)](#), [gmp\\_seed\\_randstate\(\)](#), [iZ\\_mpz\(\)](#), and [MR\\_ROUNDS](#).

Referenced by [vy\\_random\\_prime\(\)](#).

## 11.5 Logging

Thread-safe runtime logging helpers.

### Files

- file [logger.c](#)  
Logging module.

### Macros

- `#define LOGGER\_FORMAT\_PRINTF(fmt_idx, arg_idx)`  
Compiler attribute wrapper for printf-style format checking.
- `#define LOG\_DIR "logs/"`  
Directory where logs are stored.
- `#define LOG\_FILE LOG\_DIR "log.txt"`  
Default log file.
- `#define LOG\_MAX\_SIZE 1024 * 1024 * 5`  
Maximum log file size (5 MB).

### Enumerations

- enum [LogLevel](#) {  
[LOG\\_DEBUG](#) , [LOG\\_INFO](#) , [LOG\\_WARNING](#) , [LOG\\_ERROR](#) ,  
[LOG\\_FATAL](#) }  
Enumeration of log levels.

### Functions

- const char \* [log\\_level\\_to\\_string](#) ([LogLevel](#) level)  
Returns a string representation of the log level.
- void [log\\_init](#) (const char \*log\_file)  
Initializes the logging system.
- void [log\\_shutdown](#) (void)  
Shuts down the logging system and cleans up resources.
- void [log\\_set\\_log\\_level](#) ([LogLevel](#) level)  
Sets the current log level. Messages below this level will not be logged.
- void [log\\_message](#) ([LogLevel](#) level, const char \*format,...) [LOGGER\\_FORMAT\\_PRINTF](#)(2)  
Logs a formatted message at the given log level.
- void [log\\_message\\_extended](#) ([LogLevel](#) level, const char \*file\_name, int line\_number, const char \*format,...) [LOGGER\\_FORMAT\\_PRINTF](#)(4)  
Logs a formatted message with extended information (file name, line number).
- void [log\\_console](#) (const char \*format,...) [LOGGER\\_FORMAT\\_PRINTF](#)(1)  
Logs a timestamped message to the console only, without requiring a log level.
- void [log\\_debug](#) (const char \*format,...) [LOGGER\\_FORMAT\\_PRINTF](#)(1)  
Logs a debug message.
- void [log\\_info](#) (const char \*format,...) [LOGGER\\_FORMAT\\_PRINTF](#)(1)  
Logs an info message.
- void [log\\_warn](#) (const char \*format,...) [LOGGER\\_FORMAT\\_PRINTF](#)(1)  
Logs a warning message.
- void [log\\_error](#) (const char \*format,...) [LOGGER\\_FORMAT\\_PRINTF](#)(1)  
Logs an error message.
- void [log\\_fatal](#) (const char \*format,...) [LOGGER\\_FORMAT\\_PRINTF](#)(1)  
Logs a fatal message.



### 11.5.1 Detailed Description

Thread-safe runtime logging helpers.

### 11.5.2 Macro Definition Documentation

#### 11.5.2.1 LOG\_DIR

```
#define LOG_DIR "logs/"
```

Directory where logs are stored.

Definition at line 54 of file [logger.h](#).

#### 11.5.2.2 LOG\_FILE

```
#define LOG_FILE LOG_DIR "log.txt"
```

Default log file.

Definition at line 55 of file [logger.h](#).

#### 11.5.2.3 LOG\_MAX\_SIZE

```
#define LOG_MAX_SIZE 1024 * 1024 * 5
```

Maximum log file size (5 MB).

Definition at line 56 of file [logger.h](#).

#### 11.5.2.4 LOGGER\_FORMAT\_PRINTF

```
#define LOGGER_FORMAT_PRINTF(  
    fmt_idx,  
    arg_idx)
```

Compiler attribute wrapper for printf-style format checking.

Definition at line 50 of file [logger.h](#).

Referenced by [log\\_console\(\)](#), [log\\_debug\(\)](#), [log\\_error\(\)](#), [log\\_fatal\(\)](#), [log\\_info\(\)](#), [log\\_message\(\)](#), [log\\_message\\_extended\(\)](#), and [log\\_warn\(\)](#).

### 11.5.3 Enumeration Type Documentation

#### 11.5.3.1 LogLevel

enum [LogLevel](#)

Enumeration of log levels.

This enumeration defines the different log levels that can be used in the logging system. Each level corresponds to a severity of importance.

Enumerator

LOG_DEBUG	Debug level.
-----------	--------------

LOG_INFO	Info level.
LOG_WARNING	Warning level.
LOG_ERROR	Error level.
LOG_FATAL	Fatal level.

Definition at line 65 of file [logger.h](#).

## 11.5.4 Function Documentation

### 11.5.4.1 log\_console()

```
void void void log_console (
    const char * format,
    ...)
```

Logs a timestamped message to the console only, without requiring a log level.

This can be used for debugging or tracking specific points in the code.

#### Parameters

format	The format string for the log message.
--------	--

References [log\\_console\(\)](#), and [LOGGER\\_FORMAT\\_PRINTF](#).

Referenced by [log\\_console\(\)](#).

### 11.5.4.2 log\_debug()

```
void void void void log_debug (
    const char * format,
    ...)
```

Logs a debug message.

#### Parameters

format	The format string for the log message.
--------	--

References [log\\_debug\(\)](#), and [LOGGER\\_FORMAT\\_PRINTF](#).

Referenced by [iZ\\_next\\_prime\(\)](#), and [log\\_debug\(\)](#).

11.5.4.3 `log_error()`

```
void void void void void void void log_error (
    const char * format,
    ...)
```

Logs an error message.

Parameters

format	The format string for the log message.
--------	--

Examples

[/Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c](#).

References [log\\_error\(\)](#), and [LOGGER\\_FORMAT\\_PRINTF](#).

Referenced by [bitmap\\_fread\(\)](#), [bitmap\\_fwrite\(\)](#), [bitmap\\_init\(\)](#), [bitmap\\_validate\\_hash\(\)](#), [create\\_dir\(\)](#), [iZ\\_next\\_prime\(\)](#), [iZm\\_clone\(\)](#), [iZm\\_init\(\)](#), [log\\_error\(\)](#), [range\\_info\\_init\(\)](#), [SiZ\\_count\(\)](#), [SiZ\\_stream\(\)](#), [vx\\_init\(\)](#), [vx\\_random\\_prime\(\)](#), [vx\\_search\\_prime\(\)](#), [vx\\_set\\_base\\_values\(\)](#), and [vy\\_random\\_prime\(\)](#).

11.5.4.4 `log_fatal()`

```
void void void void void void void void log_fatal (
    const char * format,
    ...)
```

Logs a fatal message.

Parameters

format	The format string for the log message.
--------	--

References [log\\_fatal\(\)](#), and [LOGGER\\_FORMAT\\_PRINTF](#).

Referenced by [log\\_fatal\(\)](#).

11.5.4.5 `log_info()`

```
void void void void void void log_info (
    const char * format,
    ...)
```

Logs an info message.

Parameters

format	The format string for the log message.
--------	--

References [log\\_info\(\)](#), and [LOGGER\\_FORMAT\\_PRINTF](#).

Referenced by [log\\_info\(\)](#), and [vx\\_prob\\_sieve\(\)](#).

#### 11.5.4.6 log\_init()

```
void log_init (  
    const char * log_file)
```

Initializes the logging system.

Creates the log directory if it doesn't exist and rotates logs if necessary.

Parameters

log_file	The path to the log file.
----------	---------------------------

#### 11.5.4.7 log\_level\_to\_string()

```
const char * log_level_to_string (  
    LogLevel level)
```

Returns a string representation of the log level.

Parameters

level	The log level.
-------	----------------

Returns

A string corresponding to the log level.

#### 11.5.4.8 log\_message()

```
void log_message (  
    LogLevel level,  
    const char * format,  
    ...)
```

Logs a formatted message at the given log level.

This function logs messages to the log file in a thread-safe manner.

Parameters

level	The log level for the message.
format	The format string for the log message.

References [LOGGER\\_FORMAT\\_PRINTF](#).

#### 11.5.4.9 log\_message\_extended()

```
void void log_message_extended (  
    LogLevel level,  
    const char * file_name,  
    int line_number,  
    const char * format,  
    ...)
```

Logs a formatted message with extended information (file name, line number).

This function logs messages to the log file in a thread-safe manner.

##### Parameters

level	The log level for the message.
-------	--------------------------------

file_name	The source file where the log was generated.
line_number	The line number in the source file.
format	The format string for the log message.

References [log\\_message\\_extended\(\)](#), and [LOGGER\\_FORMAT\\_PRINTF](#).

Referenced by [log\\_message\\_extended\(\)](#).

#### 11.5.4.10 log\_set\_log\_level()

```
void log_set_log_level (
    LogLevel level)
```

Sets the current log level. Messages below this level will not be logged.

Parameters

level	The log level to set.
-------	-----------------------

#### 11.5.4.11 log\_warn()

```
void void void void void log_warn (
    const char * format,
    ...)
```

Logs a warning message.

Parameters

format	The format string for the log message.
--------	--

References [log\\_warn\(\)](#), and [LOGGER\\_FORMAT\\_PRINTF](#).

Referenced by [log\\_warn\(\)](#).

## 11.6 Tests and Benchmarks

Test harness entry points.

Data Structures

- struct [SIEVE\\_LIMIT](#)
- struct [SIEVE\\_MODEL](#)

## Typedefs

- typedef [UI64\\_ARRAY](#) [\\*\(\\* SIEVE\\_FN\)](#) (uint64\_t)

## Sieve model tests/benchmarks

- int TEST\_UTILS (int verbose)  
Validate utility helpers (numeric/range parsers and shared utilities).
- int TEST\_SIEVE\_MODELS\_INTEGRITY (int verbose)  
Run correctness checks across all registered sieve models.
- void BENCHMARK\_SIEVE\_MODELS (int save\_results)  
Benchmark registered sieve models and optionally persist results.

## Range sieve tests/benchmarks

- int TEST\_SiZ\_stream (int verbose)  
Validate [SiZ\\_stream](#) behavior and output formatting.
- int TEST\_SiZ\_count (int verbose)  
Validate [SiZ\\_count](#) correctness across worker counts.
- void BENCHMARK\_SiZ\_count (int save\_results)  
Benchmark [SiZ\\_count](#) over increasing ranges starting from  $10^{10}$ ,  $10^{20}$ , ...,  $10^{100}$  using max cores.

## Prime generation tests/benchmarks

- int TEST\_iZ\_next\_prime (int verbose)  
Validate [iZ\\_next\\_prime](#) for forward/backward traversal cases.
- int TEST\_vy\_random\_prime (int verbose)  
Validate [vy\\_random\\_prime](#) for primality and bit-length constraints.
- int TEST\_vx\_random\_prime (int verbose)  
Validate [vx\\_random\\_prime](#) for primality and bit-length constraints.
- int BENCHMARK\_P\_GEN\_ALGORITHMS (int bit\_size, int test\_rounds, int save\_results)  
Benchmark random-prime generation routines over repeated trials.

### 11.6.1 Detailed Description

Test harness entry points.

### 11.6.2 Typedef Documentation

#### 11.6.2.1 SIEVE\_FN

typedef [UI64\\_ARRAY](#) [\\*\(\\* SIEVE\\_FN\)](#) (uint64\_t)

Sieve function type: takes a limit and returns a prime list.

Definition at line 20 of file [test\\_api.h](#).

## 11.7 Utilities

Cross-cutting helpers used across modules.

### Files

- file [utils.c](#)  
Implementations for the shared utility layer.

### Macros

- `#define MIN(a, b)`  
Return the smaller of two expressions.
- `#define MAX(a, b)`  
Return the larger of two expressions.
- `#define DIR_output "/output"`
- `#define MAX_CORES get_cpu_cores_count()`

### Functions

- `int create_dir (const char *dir)`  
Create a directory if it does not exist.
- `int is_numeric_str (const char *str)`  
Return non-zero if str contains only ASCII digits.
- `int parse_numeric_expr_mpz (mpz_t out, const char *expr)`  
Parse an integer expression into an mpz value.
- `int parse_numeric_expr_u64 (const char *expr, uint64_t *out)`  
Parse an integer expression into uint64\_t.
- `int parse_inclusive_range_mpz (const char *range_expr, mpz_t lower, mpz_t upper)`  
Parse an inclusive range expression into lower/upper bounds.
- `uint64_t gcd (uint64_t a, uint64_t b)`  
Compute the greatest common divisor of a and b.
- `uint64_t modular_inverse (uint64_t a, uint64_t m)`  
Compute the modular inverse of a modulo m.
- `void gmp_seed_randstate (gmp_randstate_t state)`  
Seed a GMP random state.
- `int get_cpu_cores_count (void)`  
Get the number of online CPU cores ( $\geq 1$ ).
- `int get_cpu_L2_cache_size_bits (void)`  
Return the CPU L2 cache size in bits (best effort).

### 11.7.1 Detailed Description

Cross-cutting helpers used across modules.



## 11.7.2 Macro Definition Documentation

### 11.7.2.1 DIR\_output

```
#define DIR_output "./output"
```

Default directory for output artifacts produced by examples/tests.

Definition at line 56 of file [utils.h](#).

### 11.7.2.2 MAX

```
#define MAX(  
    a,  
    b)
```

Value:

$((a) > (b)) ? (a) : (b)$

Return the larger of two expressions.

Definition at line 52 of file [utils.h](#).

Referenced by [SiZ\\_stream\(\)](#), [SSoE\(\)](#), [vx\\_init\(\)](#), [vx\\_random\\_prime\(\)](#), [vx\\_search\\_prime\(\)](#), and [vy\\_random\\_prime\(\)](#).

### 11.7.2.3 MAX\_CORES

```
#define MAX_CORES get\_cpu\_cores\_count\(\)
```

Convenience macro: number of online CPU cores.

Definition at line 58 of file [utils.h](#).

### 11.7.2.4 MIN

```
#define MIN(  
    a,  
    b)
```

Value:

$((a) < (b)) ? (a) : (b)$

Return the smaller of two expressions.

Examples

[/Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c](#).

Definition at line 48 of file [utils.h](#).

Referenced by [bitmap\\_clear\\_steps\(\)](#), [bitmap\\_clear\\_steps\\_simd\(\)](#), [compute\\_l2\\_vx\(\)](#), [SiZ\\_count\(\)](#), [SiZ\\_stream\(\)](#), and [vx\\_init\(\)](#).

### 11.7.3 Function Documentation

#### 11.7.3.1 `create_dir()`

```
int create_dir (
    const char * dir)
```

Create a directory if it does not exist.

Parameters

dir	Directory path.
-----	-----------------

Returns

0 on success (including already exists), -1 on failure.

Parameters

dir	The directory path.
-----	---------------------

Returns

int 0 if the directory already exists or was successfully created, -1 otherwise.

Definition at line [402](#) of file [utils.c](#).

References [log\\_error\(\)](#).

#### 11.7.3.2 `gcd()`

```
uint64_t gcd (
    uint64_t a,
    uint64_t b)
```

Compute the greatest common divisor of a and b.

Compute the greatest common divisor of a and b.

Parameters

a	The first integer.
b	The second integer.

Returns

int The GCD of a and b.

Definition at line [425](#) of file [utils.c](#).

Referenced by [iZm\\_solve\\_for\\_\\_y0\(\)](#), and [SiZm\\_vy\(\)](#).

### 11.7.3.3 `get_cpu_cores_count()`

```
int get_cpu_cores_count (  
    void )
```

Get the number of online CPU cores ( $\geq 1$ ).

Get the number of online CPU cores ( $\geq 1$ ).

Returns

int The number of CPU cores.

Definition at line 509 of file [utils.c](#).

Referenced by [SiZ\\_count\(\)](#).

### 11.7.3.4 `get_cpu_L2_cache_size_bits()`

```
int get_cpu_L2_cache_size_bits (  
    void )
```

Return the CPU L2 cache size in bits (best effort).

Returns

Cache size in bits, or a conservative default.

Return the CPU L2 cache size in bits (best effort).

Description: This function retrieves the L2 cache size using platform-specific methods:

- Linux: Reads from sysfs (`/sys/devices/system/cpu/cpu0/cache/index2/size`)
- macOS/BSD: Uses `sysctl` to query `hw.l2cachesize`
- Fallback: Returns 256 Kbits as a conservative default

The function attempts multiple detection methods to ensure robustness across different architectures and operating systems.

Returns

int The L2 cache size in bits, or 256 Kbits if unable to determine.

Definition at line 531 of file [utils.c](#).

Referenced by [compute\\_l2\\_vx\(\)](#).

### 11.7.3.5 gmp\_seed\_randstate()

```
void gmp_seed_randstate (  
    gmp_randstate_t state)
```

Seed a GMP random state.

Attempts to seed from `/dev/urandom`, falling back to time-based seeding.

Seed a GMP random state.

Description: This function seeds the GMP random state using `/dev/urandom`.

Parameters

state	The GMP random state.
-------	-----------------------

Definition at line [486](#) of file [utils.c](#).

Referenced by [vx\\_search\\_prime\(\)](#), and [vy\\_search\\_prime\(\)](#).

### 11.7.3.6 is\_numeric\_str()

```
int is_numeric_str (  
    const char * str)
```

Return non-zero if str contains only ASCII digits.

Parameters

str	Input string.
-----	---------------

Returns

1 if numeric, 0 otherwise.

Return non-zero if str contains only ASCII digits.

Parameters

str	Pointer to the string.
-----	------------------------

Returns

int 1 if the string is numeric, 0 otherwise.

Definition at line [258](#) of file [utils.c](#).

## 11.7.3.7 modular\_inverse()

```
uint64_t modular_inverse (
    uint64_t a,
    uint64_t m)
```

Compute the modular inverse of a modulo m.

Returns

The inverse in  $[0, m-1]$  if it exists; otherwise an implementation-defined value.

Compute the modular inverse of a modulo m.

Description: This function computes the modular inverse of a modulo m using the Extended Euclidean Algorithm. The function takes two integers a and m as input and returns the modular inverse of a modulo m. If a and m are not co-prime, the function returns -1 indicating no modular inverse exists. The function also handles the case where m is 1, returning 0 as the modular inverse.

Parameters:

Parameters

a	The integer for which the modular inverse is to be computed.
m	The modulus.

Returns

The modular inverse of a modulo m, or -1 if no inverse exists.

Definition at line [453](#) of file [utils.c](#).

Referenced by [iZm\\_solve\\_for\\_y0\(\)](#).

## 11.7.3.8 parse\_inclusive\_range\_mpz()

```
int parse_inclusive_range_mpz (
    const char * range_expr,
    mpz_t lower,
    mpz_t upper)
```

Parse an inclusive range expression into lower/upper bounds.

Accepted forms:

- L,R
- [L, R]
- range[L, R]
- L..R
- L:R

Each bound accepts the same numeric formats as [parse\\_numeric\\_expr\\_mpz](#).

Parameters

range_expr	Range expression string.
------------	--------------------------

lower	Output lower bound.
upper	Output upper bound.

Returns

1 on success, 0 on parse failure.

Definition at line 335 of file [utils.c](#).

#### 11.7.3.9 `parse_numeric_expr_mpz()`

```
int parse_numeric_expr_mpz (
    mpz_t out,
    const char * expr)
```

Parse an integer expression into an mpz value.

Supported term formats:

- plain decimal integer (1000000, 1,000,000)
- power notation ( $10^6$ )
- scientific shorthand (1e6, 10e100)
- additive expressions ( $10e100 + 10e9$ )

Parameters

out	Output mpz value.
expr	Input expression string.

Returns

1 on success, 0 on parse failure.

Definition at line 272 of file [utils.c](#).

Referenced by [parse\\_numeric\\_expr\\_u64\(\)](#).

#### 11.7.3.10 `parse_numeric_expr_u64()`

```
int parse_numeric_expr_u64 (
    const char * expr,
    uint64_t * out)
```

Parse an integer expression into uint64\_t.

Parameters

expr	Input expression string.
------	--------------------------

out	Output value.
-----	---------------

#### Returns

1 on success, 0 on parse failure or overflow.

Definition at line 314 of file [utils.c](#).

References [parse\\_numeric\\_expr\\_mpz\(\)](#).

## 11.8 Printer, implemented in toolkit/print\_utils.c

Console formatting helpers used by tests and benchmark runners.

#### Files

- file [print\\_utils.c](#)  
Print formatting helpers for tests, benchmarks, and CLI output.

#### Functions

- void [print\\_sha256\\_hash](#) (const unsigned char \*hash)  
Print a SHA-256 digest as hex to stdout.
- void [print\\_line](#) (int length, char fill\_char)  
Print a repeated-character horizontal line.
- void [print\\_centered\\_text](#) (const char \*text, int line\_length, char fill\_char)  
Print text centered inside a padded line.
- void [print\\_test\\_table\\_header](#) (void)  
Print the generic test runner header.
- void [print\\_test\\_module\\_header](#) (char \*module\_name)  
Print a test-suite header banner for a module.
- void [print\\_test\\_fn\\_header](#) (char \*fn\_name)  
Print a formatted header for a test function.
- void [print\\_test\\_module\\_result](#) (int result, int test\_id, const char \*unit\_name, const char \*format,...)  
Print a single test-row result.
- void [print\\_test\\_summary](#) (char \*module\_name, int passed, int failed, int verbose)  
Print module-level test summary.

### 11.8.1 Detailed Description

Console formatting helpers used by tests and benchmark runners.

## 11.8.2 Function Documentation

### 11.8.2.1 `print_centered_text()`

```
void print_centered_text (  
    const char * text,  
    int line_length,  
    char fill_char)
```

Print text centered inside a padded line.

Print centered text within a line of specified length.

Parameters

text	Text payload.
------	---------------



line_length	Target total line width.
fill_char	Padding character.
text	Text to center.
line_length	Total length of the line.
fill_char	Character to use for padding (' ' is used when zero).

Definition at line 30 of file [print\\_utils.c](#).

Referenced by [print\\_test\\_fn\\_header\(\)](#).

#### 11.8.2.2 print\_line()

```
void print_line (
    int length,
    char fill_char)
```

Print a repeated-character horizontal line.

Parameters

length	Number of characters to print.
fill_char	Character to repeat ('-' is used when zero).

Definition at line 14 of file [print\\_utils.c](#).

Referenced by [print\\_test\\_fn\\_header\(\)](#), [print\\_test\\_module\\_header\(\)](#), [print\\_test\\_summary\(\)](#), and [print\\_test\\_table\\_header\(\)](#).

#### 11.8.2.3 print\_sha256\_hash()

```
void print_sha256_hash (
    const unsigned char * hash)
```

Print a SHA-256 digest as hex to stdout.

Print the SHA-256 hash in hexadecimal format.

Parameters

hash	Pointer to the SHA-256 hash array.
------	------------------------------------

Definition at line 58 of file [print\\_utils.c](#).

#### 11.8.2.4 print\_test\_fn\_header()

```
void print_test_fn_header (
    char * fn_name)
```

Print a formatted header for a test function.

Definition at line 148 of file [print\\_utils.c](#).

References [print\\_centered\\_text\(\)](#), and [print\\_line\(\)](#).

### 11.8.2.5 `print_test_module_header()`

```
void print_test_module_header (
    char * module_name)
```

Print a test-suite header banner for a module.

Print module-level banner for test output.

Parameters

<code>module_name</code>	Display name of the tested module.
--------------------------	------------------------------------

Definition at line 70 of file [print\\_utils.c](#).

References [print\\_line\(\)](#).

### 11.8.2.6 `print_test_module_result()`

```
void print_test_module_result (
    int result,
    int test_id,
    const char * unit_name,
    const char * format,
    ...)
```

Print a single test-row result.

Print a single test unit result row with formatted details.

Parameters

<code>result</code>	1 for pass, 0 for fail.
<code>test_id</code>	Monotonic case index.
<code>unit_name</code>	Short test unit label.
<code>format</code>	printf-style detail format.
<code>result</code>	1 for pass, 0 for fail.
<code>test_id</code>	Monotonic case index.
<code>unit_name</code>	Short test unit label.
<code>format</code>	printf-style detail format string.
<code>...</code>	Arguments for the detail format string.

Definition at line 96 of file [print\\_utils.c](#).

### 11.8.2.7 print\_test\_summary()

```
void print_test_summary (  
    char * module_name,  
    int passed,  
    int failed,  
    int verbose)
```

Print module-level test summary.

Print a summary of test results for a module.

Parameters

module_name	Test module name.
-------------	-------------------

passed	Number of passing tests.
failed	Number of failing tests.
verbose	Non-zero enables richer output hints.
module_name	Name of the tested module.
passed	Number of passing tests.
failed	Number of failing tests.
verbose	If non-zero, prints additional hints about the results.

Definition at line 126 of file [print\\_utils.c](#).

References [print\\_line\(\)](#).

#### 11.8.2.8 print\_test\_table\_header()

```
void print_test_table_header (
    void )
```

Print the generic test runner header.

Print standard test-table column headers.

Definition at line 80 of file [print\\_utils.c](#).

References [print\\_line\(\)](#).

## 11.9 Stopwatch

Monotonic wall-clock timing helpers for tests and benchmarks.

Files

- file [stopwatch.c](#)  
Stopwatch helpers based on monotonic wall-clock time.

Data Structures

- struct [STOPWATCH](#)  
Stopwatch state for elapsed wall-clock measurements.

Functions

- void [sw\\_start](#) ([STOPWATCH](#) \*sw)  
Start or restart a stopwatch.
- void [sw\\_stop](#) ([STOPWATCH](#) \*sw)  
Stop a running stopwatch, capturing the elapsed time in seconds.
- double [sw\\_elapsed\\_seconds](#) (const [STOPWATCH](#) \*sw)  
Return elapsed seconds for the stopwatch.
- double [sw\\_elapsed\\_now\\_seconds](#) (void)  
Capture the current monotonic time in seconds.

## 11.9.1 Detailed Description

Monotonic wall-clock timing helpers for tests and benchmarks.

## 11.9.2 Function Documentation

### 11.9.2.1 `sw_elapsed_now_seconds()`

```
double sw_elapsed_now_seconds (  
    void )
```

Capture the current monotonic time in seconds.

Returns

Current monotonic timestamp in seconds.

Definition at line 59 of file [stopwatch.c](#).

### 11.9.2.2 `sw_elapsed_seconds()`

```
double sw_elapsed_seconds (  
    const STOPWATCH * sw)
```

Return elapsed seconds for the stopwatch.

If the stopwatch is still running, elapsed time is computed against the current monotonic timestamp.

Parameters

sw	Stopwatch object.
----	-------------------

Returns

Elapsed seconds.

Definition at line 48 of file [stopwatch.c](#).

References [STOPWATCH::elapsed\\_sec](#), [STOPWATCH::running](#), and [STOPWATCH::start\\_time](#).

### 11.9.2.3 `sw_start()`

```
void sw_start (  
    STOPWATCH * sw)
```

Start or restart a stopwatch.

Parameters

sw	Stopwatch object.
----	-------------------

Definition at line 30 of file [stopwatch.c](#).

References [STOPWATCH::end\\_time](#), [STOPWATCH::running](#), and [STOPWATCH::start\\_time](#).

#### 11.9.2.4 `sw_stop()`

```
void sw_stop (  
    STOPWATCH * sw)
```

Stop a running stopwatch, capturing the elapsed time in seconds.

Parameters

sw	Stopwatch object.
----	-------------------

Definition at line 38 of file [stopwatch.c](#).

References [STOPWATCH::elapsed\\_sec](#), [STOPWATCH::end\\_time](#), [STOPWATCH::running](#), and [STOPWATCH::start\\_time](#).

# Chapter 12

## Data Structure Documentation

### 12.1 BITMAP Struct Reference

Packed bit-array with optional SHA-256 checksum.

```
#include <bitmap.h>
```

Data Fields

- `size_t` [size](#)
- `size_t` [byte\\_size](#)
- `unsigned char *` [data](#)
- `unsigned char` [sha256](#) [SHA256\_DIGEST\_LENGTH]

#### 12.1.1 Detailed Description

Packed bit-array with optional SHA-256 checksum.

[size](#) is measured in bits and [byte\\_size](#) is the backing storage in bytes. The checksum is maintained explicitly via [bitmap\\_compute\\_hash\(\)](#) and verified via [bitmap\\_validate\\_hash\(\)](#).

Examples

```
/Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c.
```

Definition at line [26](#) of file [bitmap.h](#).

#### 12.1.2 Field Documentation

##### 12.1.2.1 `byte_size`

`size_t` BITMAP::byte\_size

Number of bytes in [data](#).

Examples

```
/Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c.
```

Definition at line [29](#) of file [bitmap.h](#).

Referenced by [bitmap\\_clone\(\)](#), [bitmap\\_compute\\_hash\(\)](#), [bitmap\\_fread\(\)](#), [bitmap\\_fwrite\(\)](#), [bitmap\\_init\(\)](#), [bitmap\\_validate\\_hash\(\)](#), and [SiZm\(\)](#).

### 12.1.2.2 data

unsigned char\* BITMAP::data

Packed bits (LSB-first per byte).

Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line 30 of file [bitmap.h](#).

Referenced by [bitmap\\_clear\\_all\(\)](#), [bitmap\\_clear\\_bit\(\)](#), [bitmap\\_clear\\_steps\(\)](#), [bitmap\\_clear\\_steps\\_simd\(\)](#), [bitmap\\_clone\(\)](#), [bitmap\\_compute\\_hash\(\)](#), [bitmap\\_flip\\_bit\(\)](#), [bitmap\\_fread\(\)](#), [bitmap\\_fwrite\(\)](#), [bitmap\\_get\\_bit\(\)](#), [bitmap\\_init\(\)](#), [bitmap\\_set\\_all\(\)](#), [bitmap\\_set\\_bit\(\)](#), [bitmap\\_validate\\_hash\(\)](#), and [SiZm\(\)](#).

### 12.1.2.3 sha256

unsigned char BITMAP::sha256[SHA256\_DIGEST\_LENGTH]

Cached SHA-256 checksum.

Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line 31 of file [bitmap.h](#).

Referenced by [bitmap\\_clone\(\)](#), [bitmap\\_compute\\_hash\(\)](#), [bitmap\\_fread\(\)](#), [bitmap\\_fwrite\(\)](#), [bitmap\\_init\(\)](#), and [bitmap\\_validate\\_hash\(\)](#).

### 12.1.2.4 size

size\_t BITMAP::size

Number of addressable bits.

Examples

</Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/bitmap.c>.

Definition at line 28 of file [bitmap.h](#).

Referenced by [bitmap\\_clear\\_all\(\)](#), [bitmap\\_clear\\_steps\(\)](#), [bitmap\\_clear\\_steps\\_simd\(\)](#), [bitmap\\_clone\(\)](#), [bitmap\\_fwrite\(\)](#), [bitmap\\_init\(\)](#), and [bitmap\\_set\\_all\(\)](#).

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

- [include/bitmap.h](#)



## 12.2 INPUT\_SIEVE\_RANGE Struct Reference

Input parameters for range sieving/counting.

```
#include <iZ_api.h>
```

Data Fields

- `char * start`  
Start of range as a base-10 numeric string.
- `uint64_t range`  
Interval size (number of integers to cover).
- `int mr\_rounds`  
Miller–Rabin rounds for large primality checks.
- `char * filepath`  
Output path for streaming primes (NULL to disable output).

### 12.2.1 Detailed Description

Input parameters for range sieving/counting.

The interval is interpreted as:

- Start  $Z_s$  from the decimal string [start](#).
- End  $Z_e = Z_s + \text{range} - 1$ .

Definition at line [124](#) of file [iZ\\_api.h](#).

### 12.2.2 Field Documentation

#### 12.2.2.1 filepath

```
char* INPUT_SIEVE_RANGE::filepath
```

Output path for streaming primes (NULL to disable output).

Definition at line [129](#) of file [iZ\\_api.h](#).

Referenced by [SiZ\\_stream\(\)](#).

#### 12.2.2.2 mr\_rounds

```
int INPUT_SIEVE_RANGE::mr_rounds
```

Miller–Rabin rounds for large primality checks.

Definition at line [128](#) of file [iZ\\_api.h](#).

Referenced by [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

### 12.2.2.3 range

uint64\_t INPUT\_SIEVE\_RANGE::range

Interval size (number of integers to cover).

Definition at line 127 of file [iZ\\_api.h](#).

Referenced by [range\\_info\\_init\(\)](#), and [SiZ\\_count\(\)](#).

### 12.2.2.4 start

char\* INPUT\_SIEVE\_RANGE::start

Start of range as a base-10 numeric string.

Definition at line 126 of file [iZ\\_api.h](#).

Referenced by [range\\_info\\_init\(\)](#), [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

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

- [include/iZ\\_api.h](#)

## 12.3 IZM Struct Reference

Precomputed iZm assets for repeated VX-segment sieving.

```
#include <iZ_toolkit.h>
```

Data Fields

- int [vx](#)
- int [k\\_vx](#)
- BITMAP \* [base\\_x5](#)
- BITMAP \* [base\\_x7](#)
- UI64\_ARRAY \* [root\\_primes](#)

### 12.3.1 Detailed Description

Precomputed iZm assets for repeated VX-segment sieving.

Definition at line 84 of file [iZ\\_toolkit.h](#).

## 12.3.2 Field Documentation

### 12.3.2.1 base\_x5

[BITMAP](#)\* IZM::base\_x5

Pre-sieved base bitmap for 6x-1 line.

Definition at line 88 of file [iZ\\_toolkit.h](#).

Referenced by [iZ\\_next\\_prime\(\)](#), [iZm\\_clone\(\)](#), [iZm\\_init\(\)](#), and [vx\\_init\(\)](#).

### 12.3.2.2 base\_x7

[BITMAP](#)\* IZM::base\_x7

Pre-sieved base bitmap for 6x+1 line.

Definition at line 89 of file [iZ\\_toolkit.h](#).

Referenced by [iZ\\_next\\_prime\(\)](#), [iZm\\_clone\(\)](#), [iZm\\_init\(\)](#), and [vx\\_init\(\)](#).

### 12.3.2.3 k\_vx

int IZM::k\_vx

Count of small primes dividing vx (excluding 2,3).

Definition at line 87 of file [iZ\\_toolkit.h](#).

Referenced by [iZm\\_clone\(\)](#), [iZm\\_init\(\)](#), and [vx\\_det\\_sieve\(\)](#).

### 12.3.2.4 root\_primes

[UI64\\_ARRAY](#)\* IZM::root\_primes

Root primes used for deterministic marking.

Definition at line 90 of file [iZ\\_toolkit.h](#).

Referenced by [iZm\\_clone\(\)](#), [iZm\\_init\(\)](#), and [vx\\_det\\_sieve\(\)](#).

### 12.3.2.5 vx

int IZM::vx

Segment width in iZ x-units.

Definition at line 86 of file [iZ\\_toolkit.h](#).

Referenced by [compute\\_k\\_vx\(\)](#), [iZm\\_clone\(\)](#), [iZm\\_init\(\)](#), and [vx\\_init\(\)](#).

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

- [include/iZ\\_toolkit.h](#)

## 12.4 IZM\_RANGE\_INFO Struct Reference

Precomputed iZm coordinates for an inclusive numeric interval.

```
#include <iZ_toolkit.h>
```

Data Fields

- int [vx](#)
- mpz\_t [Zs](#)
- mpz\_t [Ze](#)
- mpz\_t [Xs](#)
- mpz\_t [Xe](#)
- mpz\_t [Ys](#)
- mpz\_t [Ye](#)
- int [y\\_range](#)

### 12.4.1 Detailed Description

Precomputed iZm coordinates for an inclusive numeric interval.

This structure maps an input range [[Zs](#), [Ze](#)] into the iZ index space used by segmented SiZ routines:

- $X^*$  are integer-domain indices such that  $Z \sim 6X \pm 1$ .
- $Y^*$  are segment indices ( $X / vx$ ).
- [y\\_range](#) = [Ye](#) - [Ys](#) and is used to drive segment iteration counts.

A negative [y\\_range](#) marks an invalid/unusable mapping (bad input or range outside current implementation bounds).

Definition at line [254](#) of file [iZ\\_toolkit.h](#).

### 12.4.2 Field Documentation

#### 12.4.2.1 vx

```
int IZM_RANGE_INFO::vx
```

iZm segment width.

Definition at line [256](#) of file [iZ\\_toolkit.h](#).

Referenced by [range\\_info\\_init\(\)](#), and [SiZ\\_stream\(\)](#).

## 12.4.2.2 Xe

mpz\_t IZM\_RANGE\_INFO::Xe

$x\_Ze = Ze / 6$ .

Definition at line 260 of file [iZ\\_toolkit.h](#).

Referenced by [range\\_info\\_free\(\)](#), [range\\_info\\_init\(\)](#), [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

## 12.4.2.3 Xs

mpz\_t IZM\_RANGE\_INFO::Xs

$x\_Zs = Zs / 6$ .

Definition at line 259 of file [iZ\\_toolkit.h](#).

Referenced by [range\\_info\\_free\(\)](#), [range\\_info\\_init\(\)](#), [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

## 12.4.2.4 y\_range

int IZM\_RANGE\_INFO::y\_range

Number of y-steps minus one ([Ye](#) - [Ys](#)), or -1 when invalid.

Definition at line 263 of file [iZ\\_toolkit.h](#).

Referenced by [range\\_info\\_init\(\)](#), [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

## 12.4.2.5 Ye

mpz\_t IZM\_RANGE\_INFO::Ye

$y\_Ze = Xe / vx$ .

Definition at line 262 of file [iZ\\_toolkit.h](#).

Referenced by [range\\_info\\_free\(\)](#), [range\\_info\\_init\(\)](#), [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

## 12.4.2.6 Ys

mpz\_t IZM\_RANGE\_INFO::Ys

$y\_Zs = Xs / vx$ .

Definition at line 261 of file [iZ\\_toolkit.h](#).

Referenced by [range\\_info\\_free\(\)](#), [range\\_info\\_init\(\)](#), [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

#### 12.4.2.7 Ze

mpz\_t IZM\_RANGE\_INFO::Ze

Inclusive upper bound of the search interval.

Definition at line 258 of file [iZ\\_toolkit.h](#).

Referenced by [range\\_info\\_free\(\)](#), [range\\_info\\_init\(\)](#), [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

#### 12.4.2.8 Zs

mpz\_t IZM\_RANGE\_INFO::Zs

Inclusive lower bound of the search interval.

Definition at line 257 of file [iZ\\_toolkit.h](#).

Referenced by [range\\_info\\_free\(\)](#), [range\\_info\\_init\(\)](#), [SiZ\\_count\(\)](#), and [SiZ\\_stream\(\)](#).

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

- [include/iZ\\_toolkit.h](#)

## 12.5 SIEVE\_LIMIT Struct Reference

```
#include <test_api.h>
```

Data Fields

- int [base](#)
- int [exp](#)

### 12.5.1 Detailed Description

Limit specification as  $\text{base}^{\text{exp}}$ .

Definition at line 23 of file [test\\_api.h](#).

### 12.5.2 Field Documentation

#### 12.5.2.1 base

int SIEVE\_LIMIT::base

Base value in  $\text{base}^{\text{exp}}$  limit notation.

Definition at line 25 of file [test\\_api.h](#).

### 12.5.2.2 exp

int SIEVE\_LIMIT::exp

Exponent value in base^exp limit notation.

Definition at line 26 of file [test\\_api.h](#).

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

- include/[test\\_api.h](#)

## 12.6 SIEVE\_MODEL Struct Reference

```
#include <test_api.h>
```

Data Fields

- [SIEVE\\_FN](#) function
- const char [name](#) [32]

### 12.6.1 Detailed Description

Pair a sieve function with a human-readable name (for reports/benchmarks).

Definition at line 30 of file [test\\_api.h](#).

### 12.6.2 Field Documentation

#### 12.6.2.1 function

[SIEVE\\_FN](#) SIEVE\_MODEL::function

Implementation entry point.

Definition at line 32 of file [test\\_api.h](#).

#### 12.6.2.2 name

const char SIEVE\_MODEL::name[32]

Display name used in test/benchmark output.

Definition at line 33 of file [test\\_api.h](#).

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

- include/[test\\_api.h](#)

## 12.7 STOPWATCH Struct Reference

Stopwatch state for elapsed wall-clock measurements.

```
#include <utils.h>
```

Data Fields

- struct timespec [start\\_time](#)
- struct timespec [end\\_time](#)
- int [running](#)
- double [elapsed\\_sec](#)

### 12.7.1 Detailed Description

Stopwatch state for elapsed wall-clock measurements.

Definition at line [209](#) of file [utils.h](#).

### 12.7.2 Field Documentation

#### 12.7.2.1 elapsed\_sec

double STOPWATCH::elapsed\_sec

Elapsed seconds, captured at stop.

Definition at line [214](#) of file [utils.h](#).

Referenced by [sw\\_elapsed\\_seconds\(\)](#), and [sw\\_stop\(\)](#).

#### 12.7.2.2 end\_time

struct timespec STOPWATCH::end\_time

Captured stop timestamp.

Definition at line [212](#) of file [utils.h](#).

Referenced by [sw\\_start\(\)](#), and [sw\\_stop\(\)](#).

#### 12.7.2.3 running

int STOPWATCH::running

Non-zero when timing is active.

Definition at line [213](#) of file [utils.h](#).

Referenced by [sw\\_elapsed\\_seconds\(\)](#), [sw\\_start\(\)](#), and [sw\\_stop\(\)](#).



#### 12.7.2.4 start\_time

struct timespec STOPWATCH::start\_time

Captured start timestamp.

Definition at line 211 of file [utils.h](#).

Referenced by [sw\\_elapsed\\_seconds\(\)](#), [sw\\_start\(\)](#), and [sw\\_stop\(\)](#).

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

- include/[utils.h](#)

## 12.8 UI16\_ARRAY Struct Reference

Dynamic array for uint16\_t values.

```
#include <int_arrays.h>
```

Data Fields

- int [capacity](#)
- int [count](#)
- uint16\_t \* [array](#)
- int [ordered](#)
- unsigned char [sha256](#) [SHA256\_DIGEST\_LENGTH]

### 12.8.1 Detailed Description

Dynamic array for uint16\_t values.

Definition at line 19 of file [int\\_arrays.h](#).

### 12.8.2 Field Documentation

#### 12.8.2.1 array

uint16\_t\* UI16\_ARRAY::array

Contiguous element storage.

Definition at line 23 of file [int\\_arrays.h](#).

#### 12.8.2.2 capacity

`int UI16_ARRAY::capacity`

Allocated element capacity.

Definition at line 21 of file [int\\_arrays.h](#).

#### 12.8.2.3 count

`int UI16_ARRAY::count`

Number of valid elements.

Definition at line 22 of file [int\\_arrays.h](#).

#### 12.8.2.4 ordered

`int UI16_ARRAY::ordered`

Flag indicating if the array is sorted.

Definition at line 24 of file [int\\_arrays.h](#).

#### 12.8.2.5 sha256

`unsigned char UI16_ARRAY::sha256[SHA256_DIGEST_LENGTH]`

SHA-256 over used payload.

Definition at line 25 of file [int\\_arrays.h](#).

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

- [include/int\\_arrays.h](#)

## 12.9 UI32\_ARRAY Struct Reference

Dynamic array for `uint32_t` values.

`#include <int_arrays.h>`

### Data Fields

- `int` [capacity](#)
- `int` [count](#)
- `uint32_t *` [array](#)
- `int` [ordered](#)
- `unsigned char` [sha256](#) [SHA256\_DIGEST\_LENGTH]

## 12.9.1 Detailed Description

Dynamic array for uint32\_t values.

Definition at line 29 of file [int\\_arrays.h](#).

## 12.9.2 Field Documentation

### 12.9.2.1 array

```
uint32_t* UI32_ARRAY::array
```

Contiguous element storage.

Definition at line 33 of file [int\\_arrays.h](#).

### 12.9.2.2 capacity

```
int UI32_ARRAY::capacity
```

Allocated element capacity.

Definition at line 31 of file [int\\_arrays.h](#).

### 12.9.2.3 count

```
int UI32_ARRAY::count
```

Number of valid elements.

Definition at line 32 of file [int\\_arrays.h](#).

### 12.9.2.4 ordered

```
int UI32_ARRAY::ordered
```

Flag indicating if the array is sorted.

Definition at line 34 of file [int\\_arrays.h](#).

### 12.9.2.5 sha256

```
unsigned char UI32_ARRAY::sha256[SHA256_DIGEST_LENGTH]
```

SHA-256 over used payload.

Definition at line 35 of file [int\\_arrays.h](#).

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

- [include/int\\_arrays.h](#)

## 12.10 UI64\_ARRAY Struct Reference

Dynamic array for uint64\_t values.

```
#include <int_arrays.h>
```

Data Fields

- int [capacity](#)
- int [count](#)
- uint64\_t \* [array](#)
- int [ordered](#)
- unsigned char [sha256](#) [SHA256\_DIGEST\_LENGTH]

### 12.10.1 Detailed Description

Dynamic array for uint64\_t values.

Definition at line 39 of file [int\\_arrays.h](#).

### 12.10.2 Field Documentation

#### 12.10.2.1 array

```
uint64_t* UI64_ARRAY::array
```

Contiguous element storage.

Definition at line 43 of file [int\\_arrays.h](#).

Referenced by [compute\\_max\\_vx\(\)](#), [iZm\\_clone\(\)](#), [SiZ\(\)](#), [SiZ\\_count\(\)](#), [SiZ\\_stream\(\)](#), [SiZm\(\)](#), [SiZm\\_vy\(\)](#), [SoEu\(\)](#), [SSoE\(\)](#), [vx\\_det\\_sieve\(\)](#), and [vx\\_search\\_prime\(\)](#).

#### 12.10.2.2 capacity

```
int UI64_ARRAY::capacity
```

Allocated element capacity.

Definition at line 41 of file [int\\_arrays.h](#).

Referenced by [iZm\\_clone\(\)](#).

#### 12.10.2.3 count

```
int UI64_ARRAY::count
```

Number of valid elements.

Definition at line 42 of file [int\\_arrays.h](#).

Referenced by [iZm\\_clone\(\)](#), [SiZ\(\)](#), [SiZ\\_count\(\)](#), [SiZ\\_stream\(\)](#), [SiZm\(\)](#), [SiZm\\_vy\(\)](#), [SoEu\(\)](#), [SSoE\(\)](#), [vx\\_det\\_sieve\(\)](#), and [vx\\_search\\_prime\(\)](#).

## 12.10.2.4 ordered

int UI64\_ARRAY::ordered

Flag indicating if the array is sorted.

Definition at line 44 of file [int\\_arrays.h](#).

Referenced by [SiZm\\_vy\(\)](#).

## 12.10.2.5 sha256

unsigned char UI64\_ARRAY::sha256[SHA256\_DIGEST\_LENGTH]

SHA-256 over used payload.

Definition at line 45 of file [int\\_arrays.h](#).

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

- include/[int\\_arrays.h](#)

## 12.11 VX\_SEG Struct Reference

Runtime state for one VX segment at a specific y.

#include <iZ\_toolkit.h>

## Data Fields

- int [vx](#)
- mpz\_t [y](#)
- mpz\_t [yvx](#)
- mpz\_t [root\\_limit](#)
- int [is\\_large\\_limit](#)
- int [mr\\_rounds](#)
- int [start\\_x](#)
- int [end\\_x](#)
- BITMAP \* [x5](#)
- BITMAP \* [x7](#)
- int [p\\_count](#)
- UI16\_ARRAY \* [p\\_gaps](#)
- int [bit\\_ops](#)
- int [p\\_test\\_ops](#)

## 12.11.1 Detailed Description

Runtime state for one VX segment at a specific y.

Definition at line 184 of file [iZ\\_toolkit.h](#).

## 12.11.2 Field Documentation

### 12.11.2.1 bit\_ops

int VX\_SEG::bit\_ops

Approximate deterministic mark operations.

Definition at line 198 of file [iZ\\_toolkit.h](#).

Referenced by [vx\\_det\\_sieve\(\)](#), and [vx\\_init\(\)](#).

### 12.11.2.2 end\_x

int VX\_SEG::end\_x

Inclusive end x for this segment.

Definition at line 193 of file [iZ\\_toolkit.h](#).

Referenced by [vx\\_det\\_sieve\(\)](#), [vx\\_init\(\)](#), and [vx\\_prob\\_sieve\(\)](#).

### 12.11.2.3 is\_large\_limit

int VX\_SEG::is\_large\_limit

Non-zero => requires probabilistic primality checks.

Definition at line 190 of file [iZ\\_toolkit.h](#).

Referenced by [vx\\_collect\\_p\\_gaps\(\)](#), [vx\\_det\\_sieve\(\)](#), [vx\\_full\\_sieve\(\)](#), [vx\\_prob\\_sieve\(\)](#), [vx\\_set\\_base\\_values\(\)](#), and [vx\\_stream\(\)](#).

### 12.11.2.4 mr\_rounds

int VX\_SEG::mr\_rounds

Miller-Rabin rounds when probabilistic checks are used.

Definition at line 191 of file [iZ\\_toolkit.h](#).

Referenced by [vx\\_init\(\)](#), [vx\\_prob\\_sieve\(\)](#), and [vx\\_stream\(\)](#).

### 12.11.2.5 p\_count

int VX\_SEG::p\_count

Count of primes found in this segment.

Definition at line 196 of file [iZ\\_toolkit.h](#).

Referenced by [SiZ\\_count\(\)](#), [SiZ\\_stream\(\)](#), [vx\\_collect\\_p\\_gaps\(\)](#), [vx\\_det\\_sieve\(\)](#), [vx\\_init\(\)](#), [vx\\_prob\\_sieve\(\)](#), and [vx\\_stream\(\)](#).

## 12.11.2.6 p\_gaps

[UI16\\_ARRAY\\*](#) VX\_SEG::p\_gaps

Optional prime-gap encoding for streamed output.

Definition at line 197 of file [iZ\\_toolkit.h](#).

Referenced by [vx\\_collect\\_p\\_gaps\(\)](#), and [vx\\_init\(\)](#).

## 12.11.2.7 p\_test\_ops

int VX\_SEG::p\_test\_ops

Probabilistic primality tests executed.

Definition at line 199 of file [iZ\\_toolkit.h](#).

Referenced by [vx\\_init\(\)](#), [vx\\_prob\\_sieve\(\)](#), and [vx\\_stream\(\)](#).

## 12.11.2.8 root\_limit

mpz\_t VX\_SEG::root\_limit

$\sqrt{iZ(yvx + vx, +1)}$ .

Definition at line 189 of file [iZ\\_toolkit.h](#).

Referenced by [vx\\_det\\_sieve\(\)](#), and [vx\\_set\\_base\\_values\(\)](#).

## 12.11.2.9 start\_x

int VX\_SEG::start\_x

Inclusive start x for this segment.

Definition at line 192 of file [iZ\\_toolkit.h](#).

Referenced by [vx\\_collect\\_p\\_gaps\(\)](#), [vx\\_det\\_sieve\(\)](#), [vx\\_init\(\)](#), [vx\\_prob\\_sieve\(\)](#), and [vx\\_stream\(\)](#).

## 12.11.2.10 vx

int VX\_SEG::vx

Segment width.

Definition at line 186 of file [iZ\\_toolkit.h](#).

Referenced by [vx\\_det\\_sieve\(\)](#), [vx\\_init\(\)](#), and [vx\\_set\\_base\\_values\(\)](#).

12.11.2.11 `x5`

`BITMAP* VX_SEG::x5`

Candidate bitmap for  $6x-1$ .

Definition at line 194 of file `iZ_toolkit.h`.

Referenced by `vx_collect_p_gaps()`, `vx_det_sieve()`, `vx_init()`, `vx_prob_sieve()`, and `vx_stream()`.

12.11.2.12 `x7`

`BITMAP* VX_SEG::x7`

Candidate bitmap for  $6x+1$ .

Definition at line 195 of file `iZ_toolkit.h`.

Referenced by `vx_collect_p_gaps()`, `vx_det_sieve()`, `vx_init()`, `vx_prob_sieve()`, and `vx_stream()`.

12.11.2.13 `y`

`mpz_t VX_SEG::y`

Segment index  $y$ .

Definition at line 187 of file `iZ_toolkit.h`.

Referenced by `vx_collect_p_gaps()`, `vx_det_sieve()`, and `vx_set_base_values()`.

12.11.2.14 `yvx`

`mpz_t VX_SEG::yvx`

Cached product  $y*vx$ .

Definition at line 188 of file `iZ_toolkit.h`.

Referenced by `vx_prob_sieve()`, `vx_set_base_values()`, and `vx_stream()`.

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

- `include/iZ_toolkit.h`



# Chapter 13

## File Documentation

### 13.1 include/bitmap.h File Reference

Bitmap container and bit operations used by sieve implementations.

#include <utils.h>

#### Data Structures

- struct [BITMAP](#)  
Packed bit-array with optional SHA-256 checksum.

#### Functions

- int [TEST\\_BITMAP](#) (int verbose)  
Run bitmap module tests.

#### Lifecycle

- [BITMAP](#) \* [bitmap\\_init](#) (size\_t size, int set\_bits)  
Allocate a bitmap with size bits.
- void [bitmap\\_free](#) ([BITMAP](#) \*\*bitmap)  
Free a bitmap and set the caller pointer to NULL.
- [BITMAP](#) \* [bitmap\\_clone](#) ([BITMAP](#) \*src)  
Deep-copy a bitmap, including data and checksum.

#### Single-bit Operations

- int [bitmap\\_get\\_bit](#) ([BITMAP](#) \*bitmap, size\_t idx)  
Read bit value at idx.
- void [bitmap\\_set\\_bit](#) ([BITMAP](#) \*bitmap, size\_t idx)  
Set bit at idx to 1.
- void [bitmap\\_flip\\_bit](#) ([BITMAP](#) \*bitmap, size\_t idx)  
Toggle bit at idx.
- void [bitmap\\_clear\\_bit](#) ([BITMAP](#) \*bitmap, size\_t idx)  
Clear bit at idx (set to 0).

## Bulk Operations

- void `bitmap_set_all` (`BITMAP *bitmap`)  
Set all bits to 1.
- void `bitmap_clear_all` (`BITMAP *bitmap`)  
Clear all bits to 0.
- void `bitmap_clear_steps` (`BITMAP *bitmap`, `uint64_t step`, `uint64_t start_idx`, `uint64_t limit`)  
Clear every step-th bit from `start_idx` up to `limit`.
- void `bitmap_clear_steps_simd` (`BITMAP *bitmap`, `uint64_t step`, `uint64_t start_idx`, `uint64_t limit`)  
SIMD-accelerated variant of `bitmap_clear_steps()`.

## Integrity and I/O

- void `bitmap_compute_hash` (`BITMAP *bitmap`)  
Compute SHA-256 over bitmap data and store it in `BITMAP::sha256`.
- int `bitmap_validate_hash` (`BITMAP *bitmap`)  
Verify `BITMAP::sha256` against current bitmap data.
- int `bitmap_fwrite` (`BITMAP *bitmap`, `FILE *file`)  
Write bitmap payload and checksum to a binary stream.
- `BITMAP * bitmap_fread` (`FILE *file`)  
Read bitmap payload and checksum from a binary stream.

### 13.1.1 Detailed Description

Bitmap container and bit operations used by sieve implementations.

The bitmap API is intentionally minimal and fast: a packed bit-array plus helpers for per-bit operations, stepped clearing (for sieving), hashing, and binary serialization.

Definition in file [bitmap.h](#).

## 13.2 bitmap.h

[Go to the documentation of this file.](#)

```
00001
00009
00010 #ifndef BITMAP_H
00011 #define BITMAP_H
00012
00013 #include <utils.h>
00014
00018
00026 typedef struct
00027 {
00028     size_t size;
00029     size_t byte_size;
00030     unsigned char *data;
00031     unsigned char sha256[SHA256_DIGEST_LENGTH];
00032 } BITMAP;
00033
00042 BITMAP *bitmap_init(size_t size, int set_bits);
00043
00048 void bitmap_free(BITMAP **bitmap);
00049
00055 BITMAP *bitmap_clone(BITMAP *src);
00057
00066 int bitmap_get_bit(BITMAP *bitmap, size_t idx);
00067
00073 void bitmap_set_bit(BITMAP *bitmap, size_t idx);
00074
00080 void bitmap_flip_bit(BITMAP *bitmap, size_t idx);
00081
```

```

00087 void bitmap\_clear\_bit(BITMAP *bitmap, size_t idx);
00089
00096 void bitmap\_set\_all(BITMAP *bitmap);
00097
00102 void bitmap\_clear\_all(BITMAP *bitmap);
00103
00114 void bitmap\_clear\_steps(BITMAP *bitmap, uint64_t step, uint64_t start_idx, uint64_t limit);
00115
00123 void bitmap\_clear\_steps\_simd(BITMAP *bitmap, uint64_t step, uint64_t start_idx, uint64_t limit);
00125
00132 void bitmap\_compute\_hash(BITMAP *bitmap);
00133
00139 int bitmap\_validate\_hash(BITMAP *bitmap);
00140
00147 int bitmap\_fwrite(BITMAP *bitmap, FILE *file);
00148
00154 BITMAP *bitmap\_fread(FILE *file);
00156
00162 int TEST\_BITMAP(int verbose);
00163
00165
00166 #endif // BITMAP_H

```

## 13.3 include/int\_arrays.h File Reference

Dynamic arrays for uint16\_t, uint32\_t, and uint64\_t values.

```
#include <utils.h>
```

### Data Structures

- struct [UI16\\_ARRAY](#)  
Dynamic array for uint16\_t values.
- struct [UI32\\_ARRAY](#)  
Dynamic array for uint32\_t values.
- struct [UI64\\_ARRAY](#)  
Dynamic array for uint64\_t values.

### Functions

- int [TEST\\_GENERIC\\_INT\\_ARRAYS](#) (int verbose)  
Run generic dispatch tests for C11 \_Generic helper macros.

### UI16 API

- [UI16\\_ARRAY](#) \* [ui16\\_init](#) (int capacity)  
Allocate a UI16 array with an initial capacity.
- void [ui16\\_free](#) ([UI16\\_ARRAY](#) \*\*array)  
Free a UI16 array and null the caller pointer.
- void [ui16\\_resize\\_to](#) ([UI16\\_ARRAY](#) \*array, int new\_capacity)  
Resize UI16 storage to new\_capacity (must be >= count).
- void [ui16\\_resize\\_to\\_fit](#) ([UI16\\_ARRAY](#) \*array)  
Shrink UI16 storage so capacity equals count.
- void [ui16\\_push](#) ([UI16\\_ARRAY](#) \*array, uint16\_t element)  
Append a uint16 value, growing storage if needed.
- void [ui16\\_sort](#) ([UI16\\_ARRAY](#) \*array)  
Sort values in ascending order.
- void [ui16\\_pop](#) ([UI16\\_ARRAY](#) \*array)

- Remove the last element if the array is non-empty.
- `void ui16_compute_hash (UI16_ARRAY *array)`  
Compute SHA-256 checksum over active payload.
- `int ui16_verify_hash (UI16_ARRAY *array)`  
Verify the stored checksum against current payload.
- `int ui16_fwrite (UI16_ARRAY *array, FILE *file)`  
Serialize count, payload, and checksum to a binary stream.
- `UI16_ARRAY * ui16_fread (FILE *file)`  
Deserialize a UI16 array from a binary stream.
- `int TEST_UI16_ARRAY (int verbose)`  
Execute UI16 test suite.

## UI32 API

- `UI32_ARRAY * ui32_init (int capacity)`  
Allocate a UI32 array with an initial capacity.
- `void ui32_free (UI32_ARRAY **array)`  
Free a UI32 array and null the caller pointer.
- `void ui32_resize_to (UI32_ARRAY *array, int new_capacity)`  
Resize UI32 storage to new\_capacity (must be >= count).
- `void ui32_resize_to_fit (UI32_ARRAY *array)`  
Shrink UI32 storage so capacity equals count.
- `void ui32_push (UI32_ARRAY *array, uint32_t element)`  
Append a uint32 value, growing storage if needed.
- `void ui32_sort (UI32_ARRAY *array)`  
Sort values in ascending order.
- `void ui32_pop (UI32_ARRAY *array)`  
Remove the last element if the array is non-empty.
- `void ui32_compute_hash (UI32_ARRAY *array)`  
Compute SHA-256 checksum over active payload.
- `int ui32_verify_hash (UI32_ARRAY *array)`  
Verify the stored checksum against current payload.
- `int ui32_fwrite (UI32_ARRAY *array, FILE *file)`  
Serialize count, payload, and checksum to a binary stream.
- `UI32_ARRAY * ui32_fread (FILE *file)`  
Deserialize a UI32 array from a binary stream.
- `int TEST_UI32_ARRAY (int verbose)`  
Execute UI32 test suite.

## UI64 API

- `UI64_ARRAY * ui64_init (int capacity)`  
Allocate a UI64 array with an initial capacity.
- `void ui64_free (UI64_ARRAY **array)`  
Free a UI64 array and null the caller pointer.
- `void ui64_resize_to (UI64_ARRAY *array, int new_capacity)`  
Resize UI64 storage to new\_capacity (must be >= count).
- `void ui64_resize_to_fit (UI64_ARRAY *array)`  
Shrink UI64 storage so capacity equals count.
- `void ui64_push (UI64_ARRAY *array, uint64_t element)`  
Append a uint64 value, growing storage if needed.
- `void ui64_sort (UI64_ARRAY *array)`  
Sort values in ascending order.
- `void ui64_pop (UI64_ARRAY *array)`  
Remove the last element if the array is non-empty.
- `void ui64_compute_hash (UI64_ARRAY *array)`  
Compute SHA-256 checksum over active payload.
- `int ui64_verify_hash (UI64_ARRAY *array)`

- Verify the stored checksum against current payload.
- `int ui64_fwrite (UI64_ARRAY *array, FILE *file)`  
Serialize count, payload, and checksum to a binary stream.
- `UI64_ARRAY * ui64_fread (FILE *file)`  
Deserialize a UI64 array from a binary stream.
- `int TEST_UI64_ARRAY (int verbose)`  
Execute UI64 test suite.

### 13.3.1 Detailed Description

Dynamic arrays for `uint16_t`, `uint32_t`, and `uint64_t` values.

The module provides uniform typed arrays with append/pop operations, deterministic resizing, optional SHA-256 integrity checks, and binary I/O.

Definition in file [int\\_arrays.h](#).

## 13.4 int\_arrays.h

[Go to the documentation of this file.](#)

```

00001
00008
00009 #ifndef INT_ARRAYS_H
00010 #define INT_ARRAYS_H
00011
00012 #include <utils.h>
00013
00017
00019 typedef struct
00020 {
00021     int capacity;
00022     int count;
00023     uint16_t *array;
00024     int ordered;
00025     unsigned char sha256[SHA256_DIGEST_LENGTH];
00026 } UI16_ARRAY;
00027
00029 typedef struct
00030 {
00031     int capacity;
00032     int count;
00033     uint32_t *array;
00034     int ordered;
00035     unsigned char sha256[SHA256_DIGEST_LENGTH];
00036 } UI32_ARRAY;
00037
00039 typedef struct
00040 {
00041     int capacity;
00042     int count;
00043     uint64_t *array;
00044     int ordered;
00045     unsigned char sha256[SHA256_DIGEST_LENGTH];
00046 } UI64_ARRAY;
00047
00051 UI16_ARRAY *ui16_init(int capacity);
00053 void ui16_free(UI16_ARRAY **array);
00055 void ui16_resize_to(UI16_ARRAY *array, int new_capacity);
00057 void ui16_resize_to_fit(UI16_ARRAY *array);
00059 void ui16_push(UI16_ARRAY *array, uint16_t element);
00061 void ui16_sort(UI16_ARRAY *array);
00063 void ui16_pop(UI16_ARRAY *array);
00065 void ui16_compute_hash(UI16_ARRAY *array);
00067 int ui16_verify_hash(UI16_ARRAY *array);
00069 int ui16_fwrite(UI16_ARRAY *array, FILE *file);
00071 UI16_ARRAY *ui16_fread(FILE *file);
00073 int TEST_UI16_ARRAY(int verbose);
00075
00079 UI32_ARRAY *ui32_init(int capacity);
00081 void ui32_free(UI32_ARRAY **array);
00083 void ui32_resize_to(UI32_ARRAY *array, int new_capacity);

```

```

00085 void ui32_resize_to_fit(UI32_ARRAY *array);
00087 void ui32_push(UI32_ARRAY *array, uint32_t element);
00089 void ui32_sort(UI32_ARRAY *array);
00091 void ui32_pop(UI32_ARRAY *array);
00093 void ui32_compute_hash(UI32_ARRAY *array);
00095 int ui32_verify_hash(UI32_ARRAY *array);
00097 int ui32_fwrite(UI32_ARRAY *array, FILE *file);
00099 UI32_ARRAY *ui32_fread(FILE *file);
00101 int TEST_UI32_ARRAY(int verbose);
00103
00107 UI64_ARRAY *ui64_init(int capacity);
00109 void ui64_free(UI64_ARRAY **array);
00111 void ui64_resize_to(UI64_ARRAY *array, int new_capacity);
00113 void ui64_resize_to_fit(UI64_ARRAY *array);
00115 void ui64_push(UI64_ARRAY *array, uint64_t element);
00117 void ui64_sort(UI64_ARRAY *array);
00119 void ui64_pop(UI64_ARRAY *array);
00121 void ui64_compute_hash(UI64_ARRAY *array);
00123 int ui64_verify_hash(UI64_ARRAY *array);
00125 int ui64_fwrite(UI64_ARRAY *array, FILE *file);
00127 UI64_ARRAY *ui64_fread(FILE *file);
00129 int TEST_UI64_ARRAY(int verbose);
00131
00137 int TEST_GENERIC_INT_ARRAYS(int verbose);
00138
00145 #if __STDC_VERSION__ >= 201112L
00146
00148 #define int_array_free(arr) _Generic((arr), \
00149     UI16_ARRAY *: ui16_free, \
00150     UI32_ARRAY *: ui32_free, \
00151     UI64_ARRAY *: ui64_free)(arr)
00152
00154 #define int_array_resize_to(arr, cap) _Generic((arr), \
00155     UI16_ARRAY *: ui16_resize_to, \
00156     UI32_ARRAY *: ui32_resize_to, \
00157     UI64_ARRAY *: ui64_resize_to)(arr, cap)
00158
00160 #define int_array_resize_to_fit(arr) _Generic((arr), \
00161     UI16_ARRAY *: ui16_resize_to_fit, \
00162     UI32_ARRAY *: ui32_resize_to_fit, \
00163     UI64_ARRAY *: ui64_resize_to_fit)(arr)
00164
00166 #define int_array_push(arr, val) _Generic((arr), \
00167     UI16_ARRAY *: ui16_push, \
00168     UI32_ARRAY *: ui32_push, \
00169     UI64_ARRAY *: ui64_push)(arr, val)
00170
00172 #define int_array_sort(arr) _Generic((arr), \
00173     UI16_ARRAY *: ui16_sort, \
00174     UI32_ARRAY *: ui32_sort, \
00175     UI64_ARRAY *: ui64_sort)(arr)
00176
00178 #define int_array_pop(arr) _Generic((arr), \
00179     UI16_ARRAY *: ui16_pop, \
00180     UI32_ARRAY *: ui32_pop, \
00181     UI64_ARRAY *: ui64_pop)(arr)
00182
00184 #define int_array_compute_hash(arr) _Generic((arr), \
00185     UI16_ARRAY *: ui16_compute_hash, \
00186     UI32_ARRAY *: ui32_compute_hash, \
00187     UI64_ARRAY *: ui64_compute_hash)(arr)
00188
00190 #define int_array_verify_hash(arr) _Generic((arr), \
00191     UI16_ARRAY *: ui16_verify_hash, \
00192     UI32_ARRAY *: ui32_verify_hash, \
00193     UI64_ARRAY *: ui64_verify_hash)(arr)
00194
00196 #define int_array_fwrite(arr, file) _Generic((arr), \
00197     UI16_ARRAY *: ui16_fwrite, \
00198     UI32_ARRAY *: ui32_fwrite, \
00199     UI64_ARRAY *: ui64_fwrite)(arr, file)
00200
00201 #endif // __STDC_VERSION__ >= 201112L
00203
00205
00206 #endif // INT_ARRAYS_H

```

## 13.5 include/iZ\_api.h File Reference

Public API for prime sieving, range scans, and prime generation.

```
#include <utils.h>
#include <iZ_toolkit.h>
```

## Data Structures

- struct [INPUT\\_SIEVE\\_RANGE](#)  
Input parameters for range sieving/counting.

## Functions

### Classic Prime Sieve Algorithms

Baseline sieves up to a numeric limit.

All classic sieve functions take a limit  $n$  and return an ascending list of primes  $\leq n$ .

- [UI64\\_ARRAY](#) \* [SoE](#) (uint64\_t n)  
Optimized Sieve of Eratosthenes.
- [UI64\\_ARRAY](#) \* [SSoE](#) (uint64\_t n)  
Segmented Sieve of Eratosthenes.
- [UI64\\_ARRAY](#) \* [SoEu](#) (uint64\_t n)  
Euler (linear) sieve.
- [UI64\\_ARRAY](#) \* [SoS](#) (uint64\_t n)  
Sieve of Sundaram.
- [UI64\\_ARRAY](#) \* [SoA](#) (uint64\_t n)  
Sieve of Atkin.

### iZ-based Sieve Algorithms

Sieve family operating in the  $6x-1$  /  $6x+1$  index space.

- [UI64\\_ARRAY](#) \* [SiZ](#) (uint64\_t n)  
Solid Sieve-iZ (wheel 6, iZ index space).
- [UI64\\_ARRAY](#) \* [SiZm](#) (uint64\_t n)  
Segmented Sieve-iZm (VX segmented, horizontal processing).
- [UI64\\_ARRAY](#) \* [SiZm\\_vy](#) (uint64\_t n)  
Segmented Sieve-iZm (vertical processing; faster, unordered output).

### Prime Generators

Probabilistic prime searches using iZm/VX machinery.

- int [vy\\_random\\_prime](#) (mpz\_t p, int bit\_size, int cores\_num)  
Search for a random prime using vertical (vy) sieving.
- int [vx\\_random\\_prime](#) (mpz\_t p, int bit\_size, int cores\_num)  
Search for a random prime using horizontal (vx) sieving.
- int [iZ\\_next\\_prime](#) (mpz\_t p, mpz\_t base, int forward)  
Advance to the next (or previous) prime from a base value.

## SiZ Range Variants

Count/stream primes over a numeric interval.

- typedef struct [INPUT\\_SIEVE\\_RANGE](#) [INPUT\\_SIEVE\\_RANGE](#)  
Input parameters for range sieving/counting.
- uint64\_t [SiZ\\_stream](#) ([INPUT\\_SIEVE\\_RANGE](#) \*range)  
Stream primes in a range to filepath (and return the count).
- uint64\_t [SiZ\\_count](#) ([INPUT\\_SIEVE\\_RANGE](#) \*input\_range, int cores\_num)  
Count primes in a range using multiple worker processes.

### 13.5.1 Detailed Description

Public API for prime sieving, range scans, and prime generation.

This header provides the stable, high-level entry points for the iZ library. Most functions return a heap-allocated `UI64_ARRAY*` that must be released with `ui64_free()`.

Definition in file `iZ_api.h`.

## 13.6 iZ\_api.h

[Go to the documentation of this file.](#)

```

00001
00009
00010 #ifndef IZ_API_H
00011 #define IZ_API_H
00012
00013 #include <utils.h>
00014 #include <iZ_toolkit.h>
00015
00026
00034
00041 UI64_ARRAY *SoE(uint64_t n);
00042
00049 UI64_ARRAY *SSoE(uint64_t n);
00050
00057 UI64_ARRAY *SoEu(uint64_t n);
00058
00065 UI64_ARRAY *SoS(uint64_t n);
00066
00073 UI64_ARRAY *SoA(uint64_t n);
00074
00076
00081
00088 UI64_ARRAY *SiZ(uint64_t n);
00089
00096 UI64_ARRAY *SiZm(uint64_t n);
00097
00108 UI64_ARRAY *SiZm_vy(uint64_t n);
00109
00111
00116
00124 typedef struct INPUT_SIEVE_RANGE
00125 {
00126     char *start;
00127     uint64_t range;
00128     int mr_rounds;
00129     char *filepath;
00130 } INPUT_SIEVE_RANGE;
00131
00137 uint64_t SiZ_stream(INPUT_SIEVE_RANGE *range);
00138
00145 uint64_t SiZ_count(INPUT_SIEVE_RANGE *input_range, int cores_num);
00146
00148
00153
00161 int vy_random_prime(mpz_t p, int bit_size, int cores_num);
00162
00170 int vx_random_prime(mpz_t p, int bit_size, int cores_num);
00171
00179 int iZ_next_prime(mpz_t p, mpz_t base, int forward);
00180
00182
00184
00185 #endif // IZ_API_H

```

## 13.7 include/iZ\_toolkit.h File Reference

Core iZ/iZm primitives used by SiZ-family algorithms.

```

#include <utils.h>
#include <int_arrays.h>
#include <bitmap.h>

```



## Data Structures

- struct [IZM](#)  
Precomputed iZm assets for repeated VX-segment sieving.
- struct [VX\\_SEG](#)  
Runtime state for one VX segment at a specific y.
- struct [IZM\\_RANGE\\_INFO](#)  
Precomputed iZm coordinates for an inclusive numeric interval.

## Macros

- `#define` [MR\\_ROUNDS](#) 25

## Standard VX Sizes (primorial products excluding 2,3)

- `#define` [VX2](#) (5 \* 7ULL)
- `#define` [VX3](#) ([VX2](#) \* 11ULL)
- `#define` [VX4](#) ([VX3](#) \* 13ULL)
- `#define` [VX5](#) ([VX4](#) \* 17ULL)
- `#define` [VX6](#) ([VX5](#) \* 19ULL)
- `#define` [VX7](#) ([VX6](#) \* 23ULL)
- `#define` [VX8](#) ([VX7](#) \* 29ULL)

## Functions

- void [iZm\\_construct\\_vx\\_base](#) (uint64\_t vx, [BITMAP](#) \*base\_x5, [BITMAP](#) \*base\_x7)  
Build pre-sieved base bitmaps for a VX segment.
- [IZM\\_RANGE\\_INFO range\\_info\\_init](#) ([INPUT\\_SIEVE\\_RANGE](#) \*input\_range, int vx)  
Map a decimal range input into iZm coordinates and segment bounds.
- void [range\\_info\\_free](#) ([IZM\\_RANGE\\_INFO](#) \*info)  
Clear all GMP fields owned by info.

## iZ Mapping Helpers

- uint64\_t [iZ](#) (uint64\_t x, int i)  
Map iZ coordinates to an integer: 6\*x + i.
- void [iZ\\_mpz](#) (mpz\_t z, mpz\_t x, int i)  
GMP variant of [iZ\(\)](#).
- void [process\\_iZ\\_bitmaps](#) ([UI64\\_ARRAY](#) \*primes, [BITMAP](#) \*x5, [BITMAP](#) \*x7, uint64\_t x↔\_limit)  
Traverse iZ bitmaps, emit surviving primes, and mark composites.
- void [get\\_root\\_primes](#) ([UI64\\_ARRAY](#) \*primes, uint64\_t limit)  
Generate primes up to limit for deterministic sieving.
- int [check\\_primality](#) (mpz\_t n, int rounds)  
Check the primality of a number using GMP's probabilistic test.

## IZM Lifecycle

- [IZM](#) \* [iZm\\_init](#) (size\_t vx)  
Allocate and initialize an [IZM](#) object for a given VX.
- [IZM](#) \* [iZm\\_clone](#) ([IZM](#) \*src)  
Deep-copy an [IZM](#) object for per-worker ownership.
- void [iZm\\_free](#) ([IZM](#) \*\*iZm)  
Release an [IZM](#) object and set the caller pointer to NULL.

### VX Selection Helpers

- `uint64_t compute_vx_k` (int k)  
Calculate  $VX_{\{k\}}$ .
- `uint64_t compute_l2_vx` (uint64\_t n)  
Choose VX using an L2-cache-aware heuristic.
- `void compute_max_vx` (mpz\_t vx, int bit\_size)  
Compute largest VX below  $2^{\text{bit\_size}}$ .

### Modular Hit Solvers

- `uint64_t iZm_solve_for_x0` (int m\_id, uint64\_t p, uint64\_t vx, uint64\_t y)  
Solve first x-hit of prime p for line m\_id in segment y.
- `uint64_t iZm_solve_for_x0_mpz` (int m\_id, uint64\_t p, uint64\_t vx, mpz\_t y)  
GMP variant of `iZm_solve_for_x0()` for very large y.
- `int64_t iZm_solve_for_y0` (int m\_id, uint64\_t p, uint64\_t vx, uint64\_t x)  
Solve first y-hit for fixed x in vertical (vy) scanning.

### VX Segment Lifecycle and Execution

- `VX_SEG * vx_init` (IZM \*iZm, int start\_x, int end\_x, char \*y\_str, int mr\_rounds)  
Initialize and deterministically sieve one VX segment.
- `void vx_free` (VX\_SEG \*\*vx\_obj)  
Free a VX segment and all owned resources.
- `void vx_collect_p_gaps` (VX\_SEG \*vx\_obj)  
Extract prime-gap encoding from a fully sieved segment.
- `void vx_full_sieve` (VX\_SEG \*vx\_obj, int collect\_p\_gaps)  
Complete segment processing (probabilistic stage and optional gaps).
- `void vx_stream` (VX\_SEG \*vx\_obj, FILE \*output)  
Stream segment primes to an output stream.

### Random Prime Search Routines

- `int vx_search_prime` (mpz\_t p, int m\_id, int vx, int bit\_size)  
Horizontal iZm/VX random-prime search.
- `int vy_search_prime` (mpz\_t p, int m\_id, mpz\_t vx)  
Vertical iZm/VY random-prime search.

### Toolkit Tests

- `int TEST_IZM` (int verbose)  
Run IZM construction and solver tests.
- `int TEST_VX_SEG` (int verbose)  
Run VX segment tests.

## 13.7.1 Detailed Description

Core iZ/iZm primitives used by SiZ-family algorithms.

This layer exposes index-space mapping helpers, VX sizing utilities, pre-sieved iZm base construction, per-segment sieve objects, and internal prime-search routines used by high-level APIs in [iZ\\_api.h](#).

Definition in file [iZ\\_toolkit.h](#).

## 13.8 iZ\_toolkit.h

[Go to the documentation of this file.](#)

```

00001
00009
00010 #ifndef IZ_TOOLKIT_H
00011 #define IZ_TOOLKIT_H
00012
00013 #include <utils.h> // Common utilities, types, and dependencies.
00014 #include <int_arrays.h> // Integer array containers.
00015 #include <bitmap.h> // Packed bit-array utilities.
00016
00018 typedef struct INPUT_SIEVE_RANGE INPUT_SIEVE_RANGE;
00019
00023
00032 uint64_t iZ(uint64_t x, int i);
00033
00040 void iZ_mpz(mpz_t z, mpz_t x, int i);
00041
00049 void process_iZ_bitmaps(UI64_ARRAY *primes, BITMAP *x5, BITMAP *x7, uint64_t x_limit);
00050
00056 void get_root_primes(UI64_ARRAY *primes, uint64_t limit);
00057
00064 int check_primality(mpz_t n, int rounds);
00066
00069 #define VX2 (5 * 7ULL)
00070 #define VX3 (VX2 * 11ULL)
00071 #define VX4 (VX3 * 13ULL)
00072 #define VX5 (VX4 * 17ULL)
00073 #define VX6 (VX5 * 19ULL)
00074 #define VX7 (VX6 * 23ULL)
00075 #define VX8 (VX7 * 29ULL)
00077
00079 #define MR_ROUNDS 25
00080
00084 typedef struct
00085 {
00086     int vx;
00087     int k_vx;
00088     BITMAP *base_x5;
00089     BITMAP *base_x7;
00090     UI64_ARRAY *root_primes;
00091 } IZM;
00092
00100 IZM *iZm_init(size_t vx);
00101
00107 IZM *iZm_clone(IZM *src);
00108
00113 void iZm_free(IZM **iZm);
00115
00123 uint64_t compute_vx_k(int k);
00124
00130 uint64_t compute_l2_vx(uint64_t n);
00131
00137 void compute_max_vx(mpz_t vx, int bit_size);
00139
00146 void iZm_construct_vx_base(uint64_t vx, BITMAP *base_x5, BITMAP *base_x7);
00147
00158 uint64_t iZm_solve_for_x0(int m_id, uint64_t p, uint64_t vx, uint64_t y);
00159
00168 uint64_t iZm_solve_for_x0_mpz(int m_id, uint64_t p, uint64_t vx, mpz_t y);
00169
00178 uint64_t iZm_solve_for_y0(int m_id, uint64_t p, uint64_t vx, uint64_t x);
00180
00184 typedef struct
00185 {
00186     int vx;
00187     mpz_t y;
00188     mpz_t yvx;
00189     mpz_t root_limit;
00190     int is_large_limit;
00191     int mr_rounds;
00192     int start_x;
00193     int end_x;
00194     BITMAP *x5;
00195     BITMAP *x7;
00196     int p_count;
00197     UI64_ARRAY *p_gaps;
00198     int bit_ops;
00199     int p_test_ops;
00200 } VX_SEG;
00201
00213 VX_SEG *vx_init(IZM *iZm, int start_x, int end_x, char *y_str, int mr_rounds);
00214
00219 void vx_free(VX_SEG **vx_obj);

```

```

00220
00225 void vx_collect_p_gaps(VX_SEG *vx_obj);
00226
00232 void vx_full_sieve(VX_SEG *vx_obj, int collect_p_gaps);
00233
00239 void vx_stream(VX_SEG *vx_obj, FILE *output);
00241
00254 typedef struct
00255 {
00256     int vx;
00257     mpz_t Zs;
00258     mpz_t Ze;
00259     mpz_t Xs;
00260     mpz_t Xe;
00261     mpz_t Ys;
00262     mpz_t Ye;
00263     int y_range;
00264 } IZM_RANGE_INFO;
00265
00273 IZM_RANGE_INFO range_info_init(INPUT_SIEVE_RANGE *input_range, int vx);
00275 void range_info_free(IZM_RANGE_INFO *info);
00276
00287 int vx_search_prime(mpz_t p, int m_id, int vx, int bit_size);
00288
00296 int vy_search_prime(mpz_t p, int m_id, mpz_t vx);
00298
00302 int TEST_IzM(int verbose);
00304 int TEST_VX_SEG(int verbose);
00306
00308
00309 #endif // IZ_TOOLKIT_H

```

## 13.9 include/logger.h File Reference

Header file for the logging system.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

### Macros

- #define [LOGGER\\_FORMAT\\_PRINTF](#)(fmt\_idx, arg\_idx)  
Compiler attribute wrapper for printf-style format checking.
- #define [LOG\\_DIR](#) "logs/"  
Directory where logs are stored.
- #define [LOG\\_FILE](#) [LOG\\_DIR](#) "log.txt"  
Default log file.
- #define [LOG\\_MAX\\_SIZE](#) 1024 \* 1024 \* 5  
Maximum log file size (5 MB).

### Enumerations

- enum [LogLevel](#) {  
[LOG\\_DEBUG](#) , [LOG\\_INFO](#) , [LOG\\_WARNING](#) , [LOG\\_ERROR](#) ,  
[LOG\\_FATAL](#) }  
Enumeration of log levels.

## Functions

- `const char * log_level_to_string (LogLevel level)`  
Returns a string representation of the log level.
- `void log_init (const char *log_file)`  
Initializes the logging system.
- `void log_shutdown (void)`  
Shuts down the logging system and cleans up resources.
- `void log_set_log_level (LogLevel level)`  
Sets the current log level. Messages below this level will not be logged.
- `void log_message (LogLevel level, const char *format,...) LOGGER\_FORMAT\_PRINTF(2)`  
Logs a formatted message at the given log level.
- `void log_message_extended (LogLevel level, const char *file_name, int line_number, const char *format,...) LOGGER\_FORMAT\_PRINTF(4)`  
Logs a formatted message with extended information (file name, line number).
- `void log_console (const char *format,...) LOGGER\_FORMAT\_PRINTF(1)`  
Logs a timestamped message to the console only, without requiring a log level.
- `void log_debug (const char *format,...) LOGGER\_FORMAT\_PRINTF(1)`  
Logs a debug message.
- `void log_info (const char *format,...) LOGGER\_FORMAT\_PRINTF(1)`  
Logs an info message.
- `void log_warn (const char *format,...) LOGGER\_FORMAT\_PRINTF(1)`  
Logs a warning message.
- `void log_error (const char *format,...) LOGGER\_FORMAT\_PRINTF(1)`  
Logs an error message.
- `void log_fatal (const char *format,...) LOGGER\_FORMAT\_PRINTF(1)`  
Logs a fatal message.

### 13.9.1 Detailed Description

Header file for the logging system.

This file contains function declarations and macros for a logging system that provides various logging levels (DEBUG, INFO, WARNING, ERROR, FATAL). The logging system supports thread-safe logging, log rotation, and console output. It also includes convenience functions for logging messages with extended information (file name, line number).

#### Note

This module is part of a larger project that includes various components requiring logging functionality. It is designed to be flexible and easy to use, allowing developers to log messages at different levels and with different formats.

#### API:

- `log_init`: Initializes the logging system.
- `log_shutdown`: Shuts down the logging system.
- `log_set_log_level`: Sets the current log level.
- `log_message`: Logs a formatted message at the given log level.

- `log_message_extended`: Logs a formatted message with extended information.
- `log_console`: Logs a formatted message to the console.
- `log_debug`: Logs a debug message.
- `log_info`: Logs an info message.
- `log_warn`: Logs a warning message.
- `log_error`: Logs an error message.
- `log_fatal`: Logs a fatal message.

Definition in file [logger.h](#).

## 13.10 logger.h

[Go to the documentation of this file.](#)

```

00001
00031
00032 #ifndef LOGGER_H
00033 #define LOGGER_H
00034
00035 #include <stdio.h>
00036 #include <stdlib.h>
00037 #include <string.h>
00038
00042
00047 #if defined(__GNUC__) || defined(__clang__)
00048 #define LOGGER_FORMAT_PRINTF(fmt_idx, arg_idx) __attribute__((format(printf, fmt_idx, arg_idx)))
00049 #else
00050 #define LOGGER_FORMAT_PRINTF(fmt_idx, arg_idx)
00051 #endif
00052
00053 // Log directory and file configuration
00054 #define LOG_DIR "logs/"
00055 #define LOG_FILE LOG_DIR "log.txt"
00056 #define LOG_MAX_SIZE 1024 * 1024 * 5
00057
00065 typedef enum
00066 {
00067     LOG_DEBUG,
00068     LOG_INFO,
00069     LOG_WARNING,
00070     LOG_ERROR,
00071     LOG_FATAL
00072 } LogLevel;
00073
00074 // Function declarations
00075
00082 const char *log_level_to_string(LogLevel level);
00083
00091 void log_init(const char *log_file);
00092
00096 void log_shutdown(void);
00097
00103 void log_set_log_level(LogLevel level);
00104
00113 void log_message(LogLevel level, const char *format, ...) LOGGER_FORMAT_PRINTF(2, 3);
00114
00125 void log_message_extended(LogLevel level, const char *file_name, int line_number, const char *format, ...)
    LOGGER_FORMAT_PRINTF(4, 5);
00126
00134 void log_console(const char *format, ...) LOGGER_FORMAT_PRINTF(1, 2);
00135
00136 // Convenience logging functions for different log levels
00142 void log_debug(const char *format, ...) LOGGER_FORMAT_PRINTF(1, 2);
00143
00149 void log_info(const char *format, ...) LOGGER_FORMAT_PRINTF(1, 2);
00150
00156 void log_warn(const char *format, ...) LOGGER_FORMAT_PRINTF(1, 2);
00157
00163 void log_error(const char *format, ...) LOGGER_FORMAT_PRINTF(1, 2);
00164
00170 void log_fatal(const char *format, ...) LOGGER_FORMAT_PRINTF(1, 2);
00171
00173
00174 #endif // LOGGER_H

```

## 13.11 include/test\_api.h File Reference

Public test/benchmark entry points for the iZ library.

```
#include <iZ_api.h>
```

### Data Structures

- struct [SIEVE\\_LIMIT](#)
- struct [SIEVE\\_MODEL](#)

### Typedefs

- typedef [UI64\\_ARRAY](#) [\\*\(\\* SIEVE\\_FN\)](#) (uint64\_t)

### Functions

#### Sieve model tests/benchmarks

- int [TEST\\_UTILS](#) (int verbose)  
Validate utility helpers (numeric/range parsers and shared utilities).
- int [TEST\\_SIEVE\\_MODELS\\_INTEGRITY](#) (int verbose)  
Run correctness checks across all registered sieve models.
- void [BENCHMARK\\_SIEVE\\_MODELS](#) (int save\_results)  
Benchmark registered sieve models and optionally persist results.

#### Range sieve tests/benchmarks

- int [TEST\\_SiZ\\_stream](#) (int verbose)  
Validate [SiZ\\_stream](#) behavior and output formatting.
- int [TEST\\_SiZ\\_count](#) (int verbose)  
Validate [SiZ\\_count](#) correctness across worker counts.
- void [BENCHMARK\\_SiZ\\_count](#) (int save\_results)  
Benchmark [SiZ\\_count](#) over increasing ranges starting from  $10^4$ ,  $10^6$ , ...,  $10^{100}$  using max cores.

#### Prime generation tests/benchmarks

- int [TEST\\_iZ\\_next\\_prime](#) (int verbose)  
Validate [iZ\\_next\\_prime](#) for forward/backward traversal cases.
- int [TEST\\_vy\\_random\\_prime](#) (int verbose)  
Validate [vy\\_random\\_prime](#) for primality and bit-length constraints.
- int [TEST\\_vx\\_random\\_prime](#) (int verbose)  
Validate [vx\\_random\\_prime](#) for primality and bit-length constraints.
- int [BENCHMARK\\_P\\_GEN\\_ALGORITHMS](#) (int bit\_size, int test\_rounds, int save\_results)  
Benchmark random-prime generation routines over repeated trials.

### 13.11.1 Detailed Description

Public test/benchmark entry points for the iZ library.

Doxygen can include this header when documenting the project to expose the test harness API and benchmark entry points.

Definition in file [test\\_api.h](#).

## 13.12 test\_api.h

[Go to the documentation of this file.](#)

```

00001
00008
00009 #ifndef TESTS_H
00010 #define TESTS_H
00011
00016
00017 #include <iZ_api.h>
00018
00020 typedef UI64_ARRAY *(*SIEVE_FN)(uint64_t);
00021
00023 typedef struct
00024 {
00025     int base;
00026     int exp;
00027 } SIEVE_LIMIT;
00028
00030 typedef struct
00031 {
00032     SIEVE_FN function;
00033     const char name[32];
00034 } SIEVE_MODEL;
00035
00038
00039 int TEST_UTILS(int verbose);
00041 int TEST_SIEVE_MODELS_INTEGRITY(int verbose);
00043 void BENCHMARK_SIEVE_MODELS(int save_results);
00045
00048
00049 int TEST_SiZ_stream(int verbose);
00051 int TEST_SiZ_count(int verbose);
00053 void BENCHMARK_SiZ_count(int save_results);
00055
00058
00059 int TEST_iZ_next_prime(int verbose);
00061 int TEST_vy_random_prime(int verbose);
00063 int TEST_vx_random_prime(int verbose);
00064
00066 int BENCHMARK_P_GEN_ALGORITHMS(int bit_size, int test_rounds, int save_results);
00068
00070
00071 #endif // TESTS_H

```

## 13.13 include/Utils.h File Reference

Shared utilities and common includes for the iZprime library.

```

#include <stdlib.h>
#include <stdint.h>
#include <stddef.h>
#include <string.h>
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <stdarg.h>
#include <time.h>
#include <logger.h>
#include <gmp.h>
#include <openssl/sha.h>

```



## Data Structures

- struct [STOPWATCH](#)  
Stopwatch state for elapsed wall-clock measurements.

## Macros

- `#define` [MIN](#)(a, b)  
Return the smaller of two expressions.
- `#define` [MAX](#)(a, b)  
Return the larger of two expressions.
- `#define` [DIR\\_output](#) `"/output"`
- `#define` [MAX\\_CORES](#) `get_cpu_cores_count()`

## Functions

- int [create\\_dir](#) (const char \*dir)  
Create a directory if it does not exist.
- int [is\\_numeric\\_str](#) (const char \*str)  
Return non-zero if str contains only ASCII digits.
- int [parse\\_numeric\\_expr\\_mpz](#) (mpz\_t out, const char \*expr)  
Parse an integer expression into an mpz value.
- int [parse\\_numeric\\_expr\\_u64](#) (const char \*expr, uint64\_t \*out)  
Parse an integer expression into uint64\_t.
- int [parse\\_inclusive\\_range\\_mpz](#) (const char \*range\_expr, mpz\_t lower, mpz\_t upper)  
Parse an inclusive range expression into lower/upper bounds.
- uint64\_t [gcd](#) (uint64\_t a, uint64\_t b)  
Compute the greatest common divisor of a and b.
- uint64\_t [modular\\_inverse](#) (uint64\_t a, uint64\_t m)  
Compute the modular inverse of a modulo m.
- void [gmp\\_seed\\_randstate](#) (gmp\_randstate\_t state)  
Seed a GMP random state.
- int [get\\_cpu\\_cores\\_count](#) (void)  
Get the number of online CPU cores ( $\geq 1$ ).
- int [get\\_cpu\\_L2\\_cache\\_size\\_bits](#) (void)  
Return the CPU L2 cache size in bits (best effort).
- void [print\\_sha256\\_hash](#) (const unsigned char \*hash)  
Print a SHA-256 digest as hex to stdout.
- void [print\\_line](#) (int length, char fill\_char)  
Print a repeated-character horizontal line.
- void [print\\_centered\\_text](#) (const char \*text, int line\_length, char fill\_char)  
Print text centered inside a padded line.
- void [print\\_test\\_table\\_header](#) (void)  
Print the generic test runner header.
- void [print\\_test\\_module\\_header](#) (char \*module\_name)  
Print a test-suite header banner for a module.
- void [print\\_test\\_fn\\_header](#) (char \*fn\_name)  
Print a formatted header for a test function.
- void [print\\_test\\_module\\_result](#) (int result, int test\_id, const char \*unit\_name, const char \*format,...)

- Print a single test-row result.
- void `print_test_summary` (char \*module\_name, int passed, int failed, int verbose)
  - Print module-level test summary.
- void `sw_start` (STOPWATCH \*sw)
  - Start or restart a stopwatch.
- void `sw_stop` (STOPWATCH \*sw)
  - Stop a running stopwatch, capturing the elapsed time in seconds.
- double `sw_elapsed_seconds` (const STOPWATCH \*sw)
  - Return elapsed seconds for the stopwatch.
- double `sw_elapsed_now_seconds` (void)
  - Capture the current monotonic time in seconds.

### 13.13.1 Detailed Description

Shared utilities and common includes for the iZprime library.

This header centralizes cross-cutting helpers (string checks, arithmetic, formatting, GMP helpers, and system queries).

Definition in file [utils.h](#).

## 13.14 utils.h

[Go to the documentation of this file.](#)

```

00001
00008
00009 #ifndef IZ_UTILS_H
00010 #define IZ_UTILS_H
00011
00012 // Standard library includes
00013 #include <stdlib.h> // For malloc, free, etc.
00014 #include <stdint.h> // For fixed-width integer types like uint64_t
00015 #include <stddef.h> // For size_t
00016 #include <string.h> // For string manipulation functions like snprintf
00017 #include <assert.h> // For assertions
00018 #include <math.h> // For math functions like sqrt
00019 #include <stdio.h> // For printf, FILE, fopen, fwrite, fread, etc.
00020 #include <sys/stat.h> // For creating directories (mkdir)
00021 #include <fcntl.h> // For open
00022 #include <unistd.h> // For close
00023 #include <signal.h> // For kill
00024 #include <sys/wait.h> // For waitpid
00025 #include <stdarg.h> // For variadic functions (va_list, va_start, va_end)
00026 #include <time.h> // For struct timespec and time-based helpers
00027
00028 // Platform-specific includes for sysctl (macOS/BSD)
00029 #if defined(__APPLE__) || defined(__FreeBSD__) || defined(__OpenBSD__) || defined(__NetBSD__)
00030 #include <sys/types.h>
00031 #include <sys/sysctl.h>
00032 #endif
00033
00034 // Internal includes
00035 #include <logger.h> // Logger module for logging timestamped messages.
00036
00037 // Third-party libraries
00038 #include <gmp.h> // GMP library for arbitrary precision arithmetic
00039 #include <openssl/sha.h> // For SHA-256 hashing
00040
00041
00042 #ifndef MIN
00043 #define MIN(a, b) (((a) < (b)) ? (a) : (b))
00044 #endif
00045
00046 #ifndef MAX
00047 #define MAX(a, b) (((a) > (b)) ? (a) : (b))
00048 #endif
00049
00050 #define DIR_output ".output"
```

```

00058 #define MAX_CORES get_cpu_cores_count()
00059
00060 // file utilities
00066 int create_dir(const char *dir);
00067
00068 // string utilities
00074 int is_numeric_str(const char *str);
00075
00089 int parse_numeric_expr_mpz(mpz_t out, const char *expr);
00090
00097 int parse_numeric_expr_u64(const char *expr, uint64_t *out);
00098
00116 int parse_inclusive_range_mpz(const char *range_expr, mpz_t lower, mpz_t upper);
00117
00118 // math utilities
00120 uint64_t gcd(uint64_t a, uint64_t b);
00125 uint64_t modular_inverse(uint64_t a, uint64_t m);
00126
00127 // gmp utilities
00133 void gmp_seed_randstate(gmp_randstate_t state);
00134
00135 // system utilities
00137 int get_cpu_cores_count(void);
00142 int get_cpu_L2_cache_size_bits(void);
00143
00145
00150
00155 void print_sha256_hash(const unsigned char *hash);
00156
00162 void print_line(int length, char fill_char);
00163
00170 void print_centered_text(const char *text, int line_length, char fill_char);
00171
00173 void print_test_table_header(void);
00174
00176 void print_test_module_header(char *module_name);
00177
00179 void print_test_fn_header(char *fn_name);
00180
00188 void print_test_module_result(int result, int test_id, const char *unit_name, const char *format, ...);
00189
00197 void print_test_summary(char *module_name, int passed, int failed, int verbose);
00198
00200
00205
00209 typedef struct
00210 {
00211     struct timespec start_time;
00212     struct timespec end_time;
00213     int running;
00214     double elapsed_sec;
00215 } STOPWATCH;
00216
00221 void sw_start(STOPWATCH *sw);
00222
00227 void sw_stop(STOPWATCH *sw);
00228
00238 double sw_elapsed_seconds(const STOPWATCH *sw);
00239
00244 double sw_elapsed_now_seconds(void);
00245
00247
00248 #endif // IZ_UTILS_H

```

## 13.15 src/iZ\_apps.c File Reference

High-level application routines built on top of the iZ\_toolkit.

#include <iZ\_api.h>

### Functions

SiZ Range Variants

Count/stream primes over a numeric interval.

- `uint64_t SiZ_stream (INPUT_SIEVE_RANGE *range)`  
Stream primes in a range to filepath (and return the count).
- `uint64_t SiZ_count (INPUT_SIEVE_RANGE *input_range, int cores_num)`  
Count primes in a range using multiple worker processes.

## Prime Generators

Probabilistic prime searches using iZm/VX machinery.

- `int vy_random_prime (mpz_t p, int bit_size, int cores_num)`  
Search for a random prime using vertical (vy) sieving.
- `int vx_random_prime (mpz_t p, int bit_size, int cores_num)`  
Search for a random prime using horizontal (vx) sieving.
- `int iZ_next_prime (mpz_t p, mpz_t base, int forward)`  
Advance to the next (or previous) prime from a base value.

## 13.15.1 Detailed Description

High-level application routines built on top of the iZ\_toolkit.

This module contains range-oriented entry points (stream/count) and probabilistic prime generation wrappers.

Definition in file `iZ_apps.c`.

## 13.16 iZ\_apps.c

[Go to the documentation of this file.](#)

```

00001
00009
00010 #include <iZ_api.h>
00011
00012 // =====
00013 // * SiZ Range Variants
00014 // =====
00015
00035 uint64_t SiZ_stream(INPUT_SIEVE_RANGE *input_range)
00036 {
00037     assert(input_range && input_range->start && "Invalid INPUT_SIEVE_RANGE passed to SiZ_stream.");
00038
00039     int has_output_file = (input_range->filepath && input_range->filepath[0] != '\0');
00040     FILE *output = stdout; // default to stdout if no valid filepath is provided
00041
00042     if (has_output_file)
00043     {
00044         output = fopen(input_range->filepath, "w");
00045         if (output == NULL)
00046         {
00047             log_error("Failed to open output file: %s", input_range->filepath);
00048             return 0;
00049         }
00050     }
00051
00052     uint64_t total = 0; // output: total prime count
00053
00054     int vx = VX6; // Use VX6 segment size (1,616,615 bits) for optimal results
00055     // Miller-Rabin rounds, bounded [5, 50]
00056     int mr_rounds = MIN(MAX(input_range->mr_rounds, 5), 50);
00057
00058     IZM_RANGE_INFO info = range_info_init(input_range, vx);
00059     if (info.y_range < 0)
00060     {
00061         if (has_output_file)
00062             fclose(output);
00063         range_info_free(&info);
00064         return 0;
00065     }

```

```

00066
00067     IZM *iZm = NULL;
00068     mpz_t current_y;
00069     mpz_init(current_y);
00070     mpz_set(current_y, info.Ys);
00071     int start_x = mpz_fdiv_ui(info.Xs, info.vx);
00072     int end_x = mpz_fdiv_ui(info.Xe, info.vx);
00073
00074     // if Ys = 0, use SiZm for the first segment
00075     if (mpz_cmp_ui(current_y, 0) == 0)
00076     {
00077         uint64_t limit = mpz_cmp_ui(info.Ye, 0) > 0 ? info.vx : end_x;
00078         UI64_ARRAY *primes = SiZm(limit * 6 + 1);
00079         if (!primes)
00080         {
00081             total = 0;
00082             goto stream_cleanup;
00083         }
00084
00085         uint64_t s = mpz_get_ui(info.Zs);
00086         uint64_t e = mpz_get_ui(info.Ze);
00087
00088         for (int i = 0; i < primes->count; i++)
00089         {
00090             // only primes in [Zs, Ze]
00091             if (primes->array[i] > s && primes->array[i] <= e)
00092             {
00093                 total++;
00094                 fprintf(output, "%llu ", primes->array[i]);
00095             }
00096         }
00097
00098         start_x = 1; // next segment starts at x=1
00099         mpz_add_ui(current_y, current_y, 1); // increment Ys for the next segment
00100
00101         ui64_free(&primes);
00102     }
00103
00104     // No remaining vx segments after the initial one.
00105     if (mpz_cmp(current_y, info.Ye) > 0)
00106     {
00107         goto stream_cleanup;
00108     }
00109
00110     // Initialize iZm structure for vx segments
00111     iZm = iZm_init(vx);
00112     if (!iZm)
00113     {
00114         total = 0;
00115         goto stream_cleanup;
00116     }
00117
00118     // Process remaining segments for y in [current_y:Ye]
00119     int first_segment = 1;
00120     while (mpz_cmp(current_y, info.Ye) <= 0)
00121     {
00122         int seg_start_x = first_segment ? start_x : 1;
00123         int seg_end_x = (mpz_cmp(current_y, info.Ye) == 0) ? end_x : vx;
00124         char *y_str = mpz_get_str(NULL, 10, current_y);
00125         if (!y_str)
00126         {
00127             total = 0;
00128             goto stream_cleanup;
00129         }
00130
00131         VX_SEG *vx_obj = vx_init(iZm, seg_start_x, seg_end_x, y_str, mr_rounds);
00132         free(y_str);
00133         if (!vx_obj)
00134         {
00135             // check logs for errors
00136             total = 0;
00137             goto stream_cleanup;
00138         }
00139
00140         vx_stream(vx_obj, output);
00141         total += vx_obj->p_count; // accumulate prime count
00142
00143         vx_free(&vx_obj);
00144         first_segment = 0;
00145         mpz_add_ui(current_y, current_y, 1); // increment Ys for the next segment
00146     }
00147
00148 stream_cleanup:
00149     iZm_free(&iZm);
00150     range_info_free(&info);
00151     mpz_clear(current_y);
00152     if (has_output_file)

```

```

00153     fclose(output);
00154     else
00155         fflush(stdout);
00156
00157     return total;
00158 }
00159
00184 uint64_t SiZ_count(INPUT_SIEVE_RANGE *input_range, int cores_num)
00185 {
00186     assert(input_range && input_range->start && input_range->range > 100 &&
00187            "Invalid INPUT_SIEVE_RANGE passed to SiZ_count.");
00188
00189     uint64_t total = 0;
00190     int vx = compute_l2_vx(pow(10, 9)); // Use a segment width that balances workload and overhead for 10^9 range
00191     cores_num = MIN(cores_num, get_cpu_cores_count());
00192     IZM *iZm = NULL;
00193     int (*pipe_fds)[2] = NULL;
00194     pid_t *pids = NULL;
00195
00196     IZM_RANGE_INFO info = range_info_init(input_range, vx);
00197     if (info.y_range < 0)
00198     {
00199         range_info_free(&info);
00200         return 0;
00201     }
00202
00203     mpz_t current_y;
00204     mpz_init(current_y);
00205     mpz_set(current_y, info.Ys);
00206     int start_x = mpz_fdiv_ui(info.Xs, vx);
00207     int end_x = mpz_fdiv_ui(info.Xe, vx);
00208
00209     // if current_y = 0, use sieve_iZm for the first segment
00210     if (mpz_cmp_ui(current_y, 0) == 0)
00211     {
00212         uint64_t limit = mpz_cmp_ui(info.Ye, 0) > 0 ? vx : end_x;
00213         UI64_ARRAY *primes = SiZm(limit * 6 + 1);
00214         if (!primes)
00215         {
00216             total = 0;
00217             goto count_cleanup;
00218         }
00219
00220         total += primes->count;
00221         uint64_t s = mpz_get_ui(info.Zs);
00222         uint64_t e = mpz_get_ui(info.Ze);
00223
00224         for (int i = 0; i < primes->count; i++)
00225         {
00226             // Exclude values outside [Zs, Ze] from the first solid segment.
00227             if (primes->array[i] < s || primes->array[i] > e)
00228             {
00229                 total--;
00230             }
00231         }
00232
00233         ui64_free(&primes);
00234         start_x = 1;
00235         mpz_add_ui(current_y, current_y, 1); // increment Ys for the next segment
00236     }
00237
00238     if (mpz_cmp(current_y, info.Ye) > 0)
00239     {
00240         goto count_cleanup;
00241     }
00242
00243     iZm = iZm_init(vx);
00244     if (!iZm)
00245     {
00246         total = 0;
00247         goto count_cleanup;
00248     }
00249
00250     // Handle edge cases:
00251     mpz_t prime_z;
00252     mpz_init(prime_z);
00253     // if Ys > 0 and Zs % 6 <= 1
00254     if (mpz_cmp_ui(current_y, 0) > 0 && mpz_fdiv_ui(info.Zs, 6) <= 1)
00255     {
00256         // if iZ(Xs, -1) < Zs and prime, decrement total
00257         iZ_mpz(prime_z, info.Xs, -1);
00258         if (mpz_cmp(prime_z, info.Zs) < 0)
00259         {
00260             if (mpz_probab_prime_p(prime_z, 25))
00261             {
00262                 total--;
00263             }
00264         }
00265     }

```

```

00264     }
00265 }
00266 // if Ye > 0 and Ze % 6 <= 1
00267 if (mpz_cmp_ui(info.Ye, 0) > 0 && mpz_fdiv_ui(info.Ze, 6) <= 1)
00268 {
00269     // if iZ(Xe, 1) > Ze and prime, decrement total
00270     iZ_mpz(prime_z, info.Xe, 1);
00271     if (mpz_cmp(prime_z, info.Ze) > 0)
00272     {
00273         if (mpz_probab_prime_p(prime_z, 25))
00274         {
00275             total--;
00276         }
00277     }
00278 }
00279 mpz_clear(prime_z);
00280
00281 int total_segments = (int)(mpz_get_ui(info.Ye) - mpz_get_ui(current_y) + 1);
00282 if (total_segments <= 0)
00283 {
00284     goto count_cleanup;
00285 }
00286
00287 // Single-process processing of all segments
00288 if (cores_num == 1)
00289 {
00290     int first_segment = 1;
00291     for (int i = 0; i < total_segments; i++)
00292     {
00293         int seg_start_x = first_segment ? start_x : 1;
00294         int seg_end_x = (i == total_segments - 1) ? end_x : vx;
00295         char *y_str = mpz_get_str(NULL, 10, current_y);
00296         if (!y_str)
00297         {
00298             total = 0;
00299             goto count_cleanup;
00300         }
00301
00302         VX_SEG *vx_obj = vx_init(iZm, seg_start_x, seg_end_x, y_str, input_range->mr_rounds);
00303         free(y_str);
00304         if (!vx_obj)
00305         {
00306             total = 0;
00307             goto count_cleanup;
00308         }
00309
00310         vx_full_sieve(vx_obj, 0);
00311         total += vx_obj->p_count;
00312
00313         vx_free(&vx_obj);
00314         first_segment = 0;
00315         mpz_add_ui(current_y, current_y, 1); // increment Ys for the next segment
00316     }
00317     goto count_cleanup;
00318 }
00319
00320 // Multi-process processing of remaining segments
00321 // if total_segments < cores_num, adjust cores_num
00322 if (total_segments < cores_num)
00323 {
00324     cores_num = total_segments;
00325 }
00326
00327 int segments_per_core = total_segments / cores_num;
00328 int remainder_segments = total_segments % cores_num;
00329
00330 pipe_fds = malloc((size_t)cores_num * sizeof(*pipe_fds));
00331 pids = malloc((size_t)cores_num * sizeof(*pids));
00332 if (!pipe_fds || !pids)
00333 {
00334     log_error("SiZ_count: Failed to allocate process bookkeeping arrays.");
00335     total = 0;
00336     goto count_cleanup;
00337 }
00338
00339 for (int i = 0; i < cores_num; i++)
00340 {
00341     pids[i] = -1;
00342     pipe_fds[i][0] = -1;
00343     pipe_fds[i][1] = -1;
00344 }
00345
00346 for (int core = 0; core < cores_num; core++)
00347 {
00348     if (pipe(pipe_fds[core]) == -1)
00349     {
00350         log_error("SiZ_count: Failed to create pipe for core %d. Aborting.", core);

```

```

00351 // cleanup and abort on any pipe error
00352 for (int j = 0; j <= core; j++)
00353 {
00354     if (pipe_fds[j][0] != -1)
00355         close(pipe_fds[j][0]);
00356     if (pipe_fds[j][1] != -1)
00357         close(pipe_fds[j][1]);
00358 }
00359 total = 0;
00360 goto count_cleanup;
00361 }
00362
00363 pids[core] = fork();
00364 if (pids[core] < 0)
00365 {
00366     log_error("SiZ_count: Failed to fork process for core %d. Aborting.", core);
00367     // close this core's pipe and any previous ones, then exit
00368     for (int j = 0; j <= core; j++)
00369     {
00370         if (pipe_fds[j][0] != -1)
00371             close(pipe_fds[j][0]);
00372         if (pipe_fds[j][1] != -1)
00373             close(pipe_fds[j][1]);
00374     }
00375     total = 0;
00376     goto count_cleanup;
00377 }
00378
00379 if (pids[core] == 0)
00380 {
00381     // Child process
00382     // Close unrelated pipes
00383     for (int j = 0; j < cores_num; j++)
00384     {
00385         if (j != core && pipe_fds[j][0] != -1)
00386             close(pipe_fds[j][0]);
00387         if (j != core && pipe_fds[j][1] != -1)
00388             close(pipe_fds[j][1]);
00389     }
00390     close(pipe_fds[core][0]); // close read end
00391
00392     // Compute this child's starting Y and number of segments
00393     int offset = core * segments_per_core + (core < remainder_segments ? core : remainder_segments);
00394     int local_segments = segments_per_core + (core < remainder_segments ? 1 : 0);
00395
00396     uint64_t child_total = 0;
00397
00398     if (local_segments > 0)
00399     {
00400         mpz_t local_Ys;
00401         mpz_init(local_Ys);
00402         mpz_set(local_Ys, current_y);
00403         mpz_add_ui(local_Ys, local_Ys, offset);
00404
00405         // Each child has its own IZM to avoid data races
00406         iZm *iZm_local = iZm_clone(iZm);
00407         if (!iZm_local)
00408         {
00409             mpz_clear(local_Ys);
00410             close(pipe_fds[core][1]);
00411             exit(1);
00412         }
00413
00414         for (int i = 0; i < local_segments; i++)
00415         {
00416             int global_segment = offset + i;
00417             int seg_start_x = (global_segment == 0) ? start_x : 1;
00418             int seg_end_x = (global_segment == total_segments - 1) ? end_x : vx;
00419             char *y_str = mpz_get_str(NULL, 10, local_Ys);
00420             if (!y_str)
00421             {
00422                 iZm_free(&iZm_local);
00423                 mpz_clear(local_Ys);
00424                 close(pipe_fds[core][1]);
00425                 exit(1);
00426             }
00427
00428             VX_SEG *vx_obj = vx_init(iZm_local, seg_start_x, seg_end_x, y_str, input_range->mr_rounds);
00429             free(y_str);
00430             if (!vx_obj)
00431             {
00432                 iZm_free(&iZm_local);
00433                 mpz_clear(local_Ys);
00434                 close(pipe_fds[core][1]);
00435                 exit(1);
00436             }
00437

```



```

00438         vx_full_sieve(vx_obj, 0);
00439         child_total += vx_obj->p_count;
00440
00441         vx_free(&vx_obj);
00442         mpz_add_ui(local_Ys, local_Ys, 1);
00443     }
00444
00445     iZm_free(&iZm_local);
00446     mpz_clear(local_Ys);
00447 }
00448
00449 // Send result to parent
00450 ssize_t written = write(pipe_fds[core][1], &child_total, sizeof(child_total));
00451 if (written != sizeof(child_total))
00452 {
00453     log_error("SiZ_count: Child %d failed to write result.", core);
00454 }
00455
00456 close(pipe_fds[core][1]);
00457 exit(0);
00458 }
00459 else
00460 {
00461     // Parent process
00462     close(pipe_fds[core][1]); // close write end
00463     pipe_fds[core][1] = -1;
00464 }
00465 }
00466
00467 // Collect results
00468 for (int core = 0; core < cores_num; core++)
00469 {
00470     if (pids[core] > 0 && pipe_fds[core][0] != -1)
00471     {
00472         uint64_t child_total = 0;
00473         ssize_t bytes_read = read(pipe_fds[core][0], &child_total, sizeof(child_total));
00474         if (bytes_read == sizeof(child_total))
00475         {
00476             total += child_total;
00477         }
00478         else if (bytes_read == 0)
00479         {
00480             log_error("SiZ_count: Child %d closed pipe without sending result.", core);
00481         }
00482         else if (bytes_read < 0)
00483         {
00484             log_error("SiZ_count: Failed to read result from child %d.", core);
00485         }
00486         else
00487         {
00488             log_error("SiZ_count: Partial read from child %d (got %zd bytes).", core, bytes_read);
00489         }
00490
00491         close(pipe_fds[core][0]);
00492         pipe_fds[core][0] = -1;
00493
00494         int status;
00495         waitpid(pids[core], &status, 0);
00496         if (WIFEXITED(status) && WEXITSTATUS(status) != 0)
00497         {
00498             log_error("SiZ_count: Child %d exited with status %d.", core, WEXITSTATUS(status));
00499             // propagate child failure as an overall error
00500             total = 0;
00501             goto count_cleanup;
00502         }
00503         else if (WIFSIGNALED(status))
00504         {
00505             log_error("SiZ_count: Child %d terminated by signal %d.", core, WTERMSIG(status));
00506             total = 0;
00507             goto count_cleanup;
00508         }
00509     }
00510     else if (pipe_fds[core][0] != -1)
00511     {
00512         close(pipe_fds[core][0]);
00513         pipe_fds[core][0] = -1;
00514     }
00515 }
00516
00517 count_cleanup:
00518     free(pids);
00519     free(pipe_fds);
00520     range_info_free(&info);
00521     iZm_free(&iZm);
00522     mpz_clear(current_y);
00523
00524     return total;

```

```

00525 }
00526
00527 // =====
00528 // * Random Prime Generation
00529 // =====
00530
00545 int vy_random_prime(mpz_t p, int bit_size, int cores_num)
00546 {
00547     int found = 0; // flag to indicate if a prime was found
00548     bit_size = MAX(bit_size, 10); // Set minimum bit size
00549
00550     // 1. Compute max vx for the given bit-size
00551     mpz_t vx;
00552     mpz_init(vx);
00553     compute_max_vx(vx, bit_size);
00554
00555     // 2. If < 2 cores, run the search in-process
00556     if (cores_num < 2)
00557     {
00558         found = vy_search_prime(p, 0, vx);
00559         mpz_clear(vx);
00560         return found;
00561     }
00562
00563     // 3. Else, fork multiple processes to search for a prime
00564     // Create a pipe for inter-process communication.
00565     int fd[2];
00566     if (pipe(fd) == -1)
00567     {
00568         log_error("Failed to create pipe in vy_random_prime. Falling back to in-process search.");
00569         found = vy_search_prime(p, 0, vx);
00570         mpz_clear(vx);
00571         return found;
00572     }
00573
00574     pid_t pids[cores_num];
00575     int forks_created = 0;
00576
00577     // Fork child processes.
00578     for (int i = 0; i < cores_num; i++)
00579     {
00580         pid_t pid = fork();
00581         if (pid < 0)
00582         {
00583             log_error("Failed to fork process %d in vy_random_prime", i);
00584             continue; // Skip this fork failure and continue creating other forks
00585         }
00586         else if (pid == 0)
00587         {
00588             // Child process: close the read-end.
00589             close(fd[0]);
00590
00591             mpz_t local_p; // local prime candidate
00592             mpz_init(local_p);
00593
00594             // Search for a candidate prime
00595             found = vy_search_prime(local_p, 0, vx);
00596
00597             // If a candidate is found, send it via the pipe.
00598             if (found)
00599             {
00600                 char *p_str = mpz_get_str(NULL, 10, local_p);
00601                 if (p_str)
00602                 {
00603                     write(fd[1], p_str, strlen(p_str) + 1);
00604                     free(p_str);
00605                 }
00606             }
00607             mpz_clear(local_p);
00608
00609             close(fd[1]); // Close the write-end.
00610             exit(0);
00611         }
00612         else
00613         {
00614             // Parent process saves child's PID.
00615             pids[i] = pid;
00616             forks_created++;
00617         }
00618     }
00619
00620     // 4. Parent reads first result from the pipe.
00621     close(fd[1]); // Close the write-end.
00622
00623     if (forks_created == 0)
00624     {
00625         log_error("No child processes were created in vy_random_prime, falling back to in-process search");

```

```

00626     close(fd[0]);
00627     found = vy_search_prime(p, 0, vx);
00628     mpz_clear(vx);
00629     return found;
00630 }
00631
00632 // Allocate a sufficiently large buffer to hold the prime string + null terminator
00633 // bit_size * 0.302 is approx digits, add extra padding for safety
00634 size_t buf_size = (size_t)(bit_size * 0.4);
00635 char *buf = malloc(buf_size);
00636 if (!buf)
00637 {
00638     log_error("Failed to allocate buffer in vy_random_prime");
00639     found = 0;
00640 }
00641 else
00642 {
00643     ssize_t n = read(fd[0], buf, buf_size - 1);
00644     if (n == -1)
00645     {
00646         log_error("Failed to read from pipe in vx_random_prime");
00647         found = 0;
00648     }
00649     else if (n > 0)
00650     {
00651         buf[n] = '\0'; // Ensure null-termination
00652         if (mpz_set_str(p, buf, 10) == 0)
00653         {
00654             found = 1;
00655         }
00656         else
00657         {
00658             log_error("Failed to set prime from buffer in vx_random_prime");
00659             found = 0;
00660         }
00661     }
00662     free(buf);
00663 }
00664 close(fd[0]); // Close the read-end of the pipe.
00665
00666 // 5. Terminate all child processes
00667 for (int i = 0; i < forks_created; i++)
00668 {
00669     kill(pids[i], SIGTERM); // Terminate child process
00670     waitpid(pids[i], NULL, 0); // Wait for child process to terminate
00671 }
00672
00673 mpz_clear(vx);
00674
00675 return found;
00676 }
00677
00692 int vx_random_prime(mpz_t p, int bit_size, int cores_num)
00693 {
00694     int found = 0;
00695     bit_size = MAX(bit_size, 10);
00696     int vx = bit_size <= 2048 ? VX5 : VX6;
00697
00698     // 2. If < 2 cores, run the search in-process
00699     if (cores_num < 2)
00700     {
00701         found = vx_search_prime(p, 0, vx, bit_size);
00702         return found;
00703     }
00704
00705     // 3. Multi-core: fork processes
00706     int fd[2];
00707     if (pipe(fd) == -1)
00708     {
00709         log_error("Failed to create pipe in vx_random_prime. Falling back to in-process search.");
00710         found = vx_search_prime(p, 0, vx, bit_size);
00711         return found;
00712     }
00713
00714     pid_t pids[cores_num];
00715     int forks_created = 0;
00716
00717     for (int i = 0; i < cores_num; i++)
00718     {
00719         pid_t pid = fork();
00720         if (pid < 0)
00721         {
00722             log_error("Failed to fork process %d in vx_random_prime", i);
00723             continue;
00724         }
00725         else if (pid == 0)
00726         {

```

```

00727 // CHILD PROCESS: Initialize everything fresh
00728 close(fd[0]);
00729
00730 // Search for prime with child's own objects
00731 mpz_t local_p;
00732 mpz_init(local_p);
00733 int child_found = vx_search_prime(local_p, 0, vx, bit_size);
00734
00735 if (child_found)
00736 {
00737     char *p_str = mpz_get_str(NULL, 10, local_p);
00738     if (p_str != NULL)
00739     {
00740         write(fd[1], p_str, strlen(p_str) + 1);
00741         free(p_str);
00742     }
00743 }
00744
00745 // Cleanup child resources
00746 mpz_clear(local_p);
00747 close(fd[1]);
00748 exit(0);
00749 }
00750 else
00751 {
00752     pids[i] = pid;
00753     forks_created++;
00754 }
00755 }
00756
00757 // 4. Parent process reads result
00758 close(fd[1]);
00759
00760 if (forks_created == 0)
00761 {
00762     log_error("No child processes were created in vx_random_prime, falling back to in-process search");
00763     close(fd[0]);
00764     // fallback to in-process search
00765     found = vx_search_prime(p, 0, vx, bit_size);
00766     return found;
00767 }
00768
00769 // Allocate a sufficiently large buffer to hold the prime string + null terminator
00770 size_t buf_size = (size_t)(bit_size * 0.4);
00771 char *buf = malloc(buf_size);
00772 if (!buf)
00773 {
00774     log_error("Failed to allocate buffer in vx_random_prime");
00775     found = 0;
00776 }
00777 else
00778 {
00779     ssize_t n = read(fd[0], buf, buf_size - 1);
00780     if (n > 0)
00781     {
00782         buf[n] = '\0'; // Ensure null-termination
00783         if (mpz_set_str(p, buf, 10) == 0)
00784         {
00785             found = 1;
00786         }
00787         else
00788         {
00789             log_error("Failed to set prime from buffer in vx_random_prime");
00790             found = 0;
00791         }
00792     }
00793     free(buf);
00794 }
00795
00796 close(fd[0]); // Close the read-end of the pipe.
00797
00798 // Terminate children
00799 for (int i = 0; i < forks_created; i++)
00800 {
00801     kill(pids[i], SIGTERM);
00802     waitpid(pids[i], NULL, 0);
00803 }
00804
00805 return found;
00806 }
00807
00819 int iz_next_prime(mpz_t p, mpz_t base, int forward)
00820 {
00821     // 1. Initialization
00822     int found = 0; // flag to indicate if a prime was found
00823
00824     // tmp variable to hold the value until a prime is found

```

```

00825     mpz_t z;
00826     mpz_init_set(z, base); // set tmp = base
00827
00828     // a. Edge cases:
00829     // if forward and base is iZ-, check next iZ+
00830     if (mpz_fdiv_ui(z, 6) == 5 && forward)
00831     {
00832         mpz_add_ui(z, z, 2); // increment tmp by 2
00833         if (mpz_probab_prime_p(z, MR_ROUNDS))
00834         {
00835             mpz_set(p, z); // set p = tmp + 2
00836             mpz_clear(z);
00837             return 1;
00838         }
00839     }
00840     // if backward and base is iZ+, check previous iZ-
00841     else if (mpz_fdiv_ui(z, 6) == 1 && !forward)
00842     {
00843         mpz_sub_ui(z, z, 2); // decrement tmp by 2
00844         if (mpz_probab_prime_p(z, MR_ROUNDS))
00845         {
00846             mpz_set(p, z); // set p = tmp - 2
00847             mpz_clear(z);
00848             return 1;
00849         }
00850     }
00851
00852     int vx = (mpz_sizeinbase(base, 2) > 2048) ? VX6 : VX5;
00853     iZm *iZm = iZm_init(vx);
00854     if (!iZm)
00855     {
00856         log_error("iZm initialization failed in iZ_next_prime.");
00857         mpz_clear(z);
00858         return 0;
00859     }
00860
00861     // c. Initialize and set y and yvx
00862     mpz_t y, yvx, x_p;
00863     mpz_init(y);
00864     mpz_init(yvx);
00865     mpz_init(x_p);
00866
00867     mpz_div_ui(y, base, 6 * vx); // compute y = base / 6 * vx
00868     mpz_mul_ui(yvx, y, vx); // compute yvx = y * vx
00869     mpz_div_ui(x_p, z, 6); // compute x_p = tmp / 6
00870
00871     // 2. Iterate over the x5 and x7 bitmaps to find a prime
00872     // set start_x = x_p % vx +/- 1
00873     int step = forward ? 1 : -1;
00874     int start_x = mpz_fdiv_ui(x_p, vx) + step;
00875     int end_x = forward ? vx : 1;
00876
00877     int i = 0; // segment counter
00878
00879     while (!found)
00880     {
00881         if (forward)
00882         {
00883             if (i > 0)
00884                 start_x = 1; // start from the beginning of the bitmap
00885
00886             for (int x = start_x; x <= end_x; x++)
00887             {
00888                 // check if x5[x] is set
00889                 if (bitmap_get_bit(iZm->base_x5, x))
00890                 {
00891                     mpz_add_ui(x_p, yvx, x); // set x_p = yvx + x
00892                     iZ_mpz(z, x_p, 1); // compute p = iZ(x_p, 1)
00893                     // check if tmp is prime
00894                     found = mpz_probab_prime_p(z, MR_ROUNDS);
00895
00896                     if (found)
00897                         break;
00898                 }
00899
00900                 // check if x7[x] is set
00901                 if (bitmap_get_bit(iZm->base_x7, x))
00902                 {
00903                     mpz_add_ui(x_p, yvx, x); // set x_p = yvx + x
00904                     iZ_mpz(z, x_p, 1); // compute tmp = iZ(x_p, 1)
00905                     // check if tmp is prime
00906                     found = mpz_probab_prime_p(z, MR_ROUNDS);
00907
00908                     if (found)
00909                         break;
00910                 }
00911             }

```

```

00912
00913     mpz_add_ui(yvx, yvx, vx); // increment yvx by vx for next segment
00914 }
00915 else // backward search
00916 {
00917     if (i > 0)
00918         start_x = vx; // start from the end of the bitmaps
00919
00920     // check iZ+ first if backward
00921     for (int x = start_x; x >= end_x; x--)
00922     {
00923         // check if x7[x] is set
00924         if (bitmap_get_bit(iZm->base_x7, x))
00925         {
00926             mpz_add_ui(x_p, yvx, x); // set x_p = yvx + x
00927             iZ_mpz(z, x_p, 1); // compute tmp = iZ(x_p, 1)
00928             // check if tmp is prime
00929             found = mpz_probab_prime_p(z, MR_ROUNDS);
00930
00931             if (found)
00932                 break;
00933         }
00934
00935         // check iZ-
00936         if (bitmap_get_bit(iZm->base_x5, x))
00937         {
00938             mpz_add_ui(x_p, yvx, x); // set x_p = yvx + x
00939             iZ_mpz(z, x_p, -1); // compute p = iZ(x_p, -1)
00940             // check if tmp is prime
00941             found = mpz_probab_prime_p(z, MR_ROUNDS);
00942
00943             if (found)
00944                 break;
00945         }
00946     }
00947
00948     mpz_sub_ui(yvx, yvx, vx); // decrement yvx by vx for next segment
00949 }
00950
00951 i++; // increment segment counter
00952 }
00953
00954 // 3. Set the found prime
00955 if (found)
00956     mpz_set(p, z); // set p = tmp
00957 else
00958     log_debug("No prime found :/");
00959
00960 // cleanup
00961 iZm_free(&iZm);
00962 mpz_clears(y, yvx, x_p, z, NULL);
00963
00964 return found;
00965 }

```

## 13.17 src/playground.c File Reference

Scratch space for testing new ideas.

```
#include <iZ_api.h>
```

### 13.17.1 Detailed Description

Scratch space for testing new ideas.

Definition in file [playground.c](#).

## 13.18 playground.c

[Go to the documentation of this file.](#)

```

00001
00005
00006 #include <iZ_api.h>

```

## 13.19 src/prime\_sieve.c File Reference

Implementations of classical and SiZ-family sieve algorithms.

```
#include <iZ_api.h>
```

### Macros

- `#define N_LIMIT` (1000000000000ULL)
- `#define ASSERT_LIMIT(n)`  
Assert that input `n` is within the valid range for sieve functions.
- `#define Pi(n)`  
`Pi(n)` is an approximation of the number of primes up to `n`, used for initial array sizing.

### Functions

- static void `process_N_bitmap` (`UI64_ARRAY` \*primes, `BITMAP` \*sieve\_bitmap, `uint64_t` n)  
A helper function to process the bitmap for the Sieve of Eratosthenes and collect primes.

#### Classic Prime Sieve Algorithms

Baseline sieves up to a numeric limit.

All classic sieve functions take a limit `n` and return an ascending list of primes  $\leq n$ .

- `UI64_ARRAY` \* `SoE` (`uint64_t` n)  
Optimized Sieve of Eratosthenes.
- `UI64_ARRAY` \* `SSoE` (`uint64_t` n)  
Segmented Sieve of Eratosthenes.
- `UI64_ARRAY` \* `SoEu` (`uint64_t` n)  
Euler (linear) sieve.
- `UI64_ARRAY` \* `SoS` (`uint64_t` n)  
Sieve of Sundaram.
- `UI64_ARRAY` \* `SoA` (`uint64_t` n)  
Sieve of Atkin.

#### iZ-based Sieve Algorithms

Sieve family operating in the  $6x-1$  /  $6x+1$  index space.

- `UI64_ARRAY` \* `SiZ` (`uint64_t` n)  
Solid Sieve-iZ (wheel 6, iZ index space).
- `UI64_ARRAY` \* `SiZm` (`uint64_t` n)  
Segmented Sieve-iZm (VX segmented, horizontal processing).
- `UI64_ARRAY` \* `SiZm_vy` (`uint64_t` n)  
Segmented Sieve-iZm (vertical processing; faster, unordered output).

### 13.19.1 Detailed Description

Implementations of classical and SiZ-family sieve algorithms.

This file contains the implementations of various prime sieving algorithms, including classical algorithms (SoE/SSoE/SoEu/SoS/SoA) as well as SiZ-family algorithms (SiZ/SiZm/SiZm\_vy). All functions are single-threaded, take an upper limit `n` and return a pointer to a `UI64_ARRAY` containing the prime numbers up to `n`.

Definition in file `prime_sieve.c`.

## 13.19.2 Macro Definition Documentation

### 13.19.2.1 ASSERT\_LIMIT

```
#define ASSERT_LIMIT(  
    n)
```

Value:

```
assert((n) > 10 && (n) <= N_LIMIT && "Input must be in the range (10, 10^12).")
```

Assert that input n is within the valid range for sieve functions.

Definition at line 23 of file [prime\\_sieve.c](#).

Referenced by [SiZ\(\)](#), [SiZm\(\)](#), [SiZm\\_vy\(\)](#), [SoA\(\)](#), [SoE\(\)](#), [SoEu\(\)](#), [SoS\(\)](#), and [SSoE\(\)](#).

### 13.19.2.2 N\_LIMIT

```
#define N_LIMIT (1000000000000ULL)
```

Maximum supported sieve limit for standard entry points ( $10^{12}$ ).

Definition at line 20 of file [prime\\_sieve.c](#).

### 13.19.2.3 Pi

```
#define Pi(  
    n)
```

Value:

```
((n / log(n)))
```

[Pi\(n\)](#) is an approximation of the number of primes up to n, used for initial array sizing.

Definition at line 26 of file [prime\\_sieve.c](#).

Referenced by [SiZ\(\)](#), [SiZm\(\)](#), [SiZm\\_vy\(\)](#), [SoA\(\)](#), [SoE\(\)](#), [SoEu\(\)](#), [SoS\(\)](#), and [SSoE\(\)](#).

## 13.19.3 Function Documentation

### 13.19.3.1 process\_N\_bitmap()

```
void process_N_bitmap (  
    UI64_ARRAY * primes,  
    BITMAP * sieve_bitmap,  
    uint64_t n) [static]
```

A helper function to process the bitmap for the Sieve of Eratosthenes and collect primes.

Parameters

primes	Output prime array.
--------	---------------------



sieve_bitmap	Candidate bitmap.
n	Inclusive numeric upper bound represented in sieve_bitmap.

Definition at line 38 of file [prime\\_sieve.c](#).

References [bitmap\\_clear\\_steps\\_simd\(\)](#), [bitmap\\_get\\_bit\(\)](#), and [ui64\\_push\(\)](#).

Referenced by [SoE\(\)](#), and [SSoE\(\)](#).

## 13.20 prime\_sieve.c

[Go to the documentation of this file.](#)

```

00001
00012
00013 #include <iZ_api.h>
00014
00015 // =====
00016 // * Internal helper macros:
00017 // =====
00018
00020 #define N_LIMIT (1000000000000ULL)
00021
00023 #define ASSERT_LIMIT(n) assert((n) > 10 && (n) <= N_LIMIT && "Input must be in the range (10, 10^12).")
00024
00026 #define Pi(n) ((n / log(n)))
00027
00028 // =====
00029 // * Classic Sieve Algorithms
00030 // =====
00031
00038 static void process_N_bitmap(UI64_ARRAY *primes, BITMAP *sieve_bitmap, uint64_t n)
00039 {
00040     ui64_push(primes, 2); // Add 2 to primes to focus on odd candidates
00041     uint64_t n_sqrt = sqrt(n);
00042
00043     // Sieve logic:
00044     // Iterate through odd numbers i starting from 3,
00045     // check if i is marked as prime in the bitmap,
00046     // if so, collect i as a prime and mark its odd multiples if i <= sqrt(n).
00047     for (uint64_t i = 3; i <= n; i += 2)
00048     {
00049         if (bitmap_get_bit(sieve_bitmap, i))
00050         {
00051             ui64_push(primes, i);
00052             if (i <= n_sqrt)
00053                 bitmap_clear_steps_simd(sieve_bitmap, 2 * i, i * i, n + 1);
00054         }
00055     }
00056 }
00057
00072 UI64_ARRAY *SoE(uint64_t n)
00073 {
00074     ASSERT_LIMIT(n); // Validate input limit
00075
00076     // Initialize the primes object with an estimated capacity
00077     UI64_ARRAY *primes = ui64_init(Pi(n) * 1.4); // 40% over-estimation to avoid reallocs
00078     assert(primes && "Memory allocation failed for primes array.");
00079
00080     // Create a bitmap to mark prime numbers
00081     BITMAP *sieve = bitmap_init(n + 1, 1);
00082     if (!sieve)
00083     {
00084         ui64_free(&primes);
00085         return NULL;
00086     }
00087
00088     // Sieve logic
00089     process_N_bitmap(primes, sieve, n);
00090
00091     // cleanup and finalize
00092     bitmap_free(&sieve); // free bitmap
00093     ui64_resize_to_fit(primes); // Trim excess memory in primes array
00094
00095     return primes;

```

```

00096 }
00097
00114 UI64_ARRAY *SSoE(uint64_t n)
00115 {
00116     ASSERT_LIMIT(n); // Validate input limit
00117
00118     // Initialize UI64_ARRAY with an estimated capacity
00119     UI64_ARRAY *primes = ui64_init(Pi(n) * 1.4); // 40% over-estimation to avoid reallocs
00120     assert(primes && "Memory allocation failed for primes array.");
00121
00122     // Define the segment size; can be tuned based on memory constraints
00123     uint64_t segment_size = (uint64_t)sqrt(n);
00124
00125     // * Step 1: Sieve small primes up to sqrt(n) using the traditional sieve
00126     BITMAP *sieve = bitmap_init(segment_size + 8, 1);
00127     if (!sieve)
00128     {
00129         ui64_free(&primes);
00130         return NULL;
00131     }
00132
00133     // process first segment to collect root primes
00134     process_N_bitmap(primes, sieve, segment_size);
00135
00136     // * Step 2: Segmented sieve
00137     uint64_t low = segment_size + 1;
00138     uint64_t high = low + segment_size - 1;
00139
00140     // Iterate over segments
00141     while (low <= n)
00142     {
00143         bitmap_set_all(sieve); // Reset segment bitmap
00144         uint64_t root_limit = sqrt(high);
00145
00146         // Sieve the current segment using primes <= sqrt(high)
00147         for (int i = 1; i < primes->count; i++) // skip 2
00148         {
00149             uint64_t p = primes->array[i];
00150             if (p > root_limit)
00151                 break;
00152
00153             // Find the minimum number in [low, high] that is a multiple of p
00154             uint64_t start = (low / p) * p;
00155             start += (start < low) ? p : 0; // ensure start >= low
00156             start += (start % 2 == 0) ? p : 0; // ensure start is odd
00157             start = MAX(p * p, start); // start from p^2 or higher
00158
00159             // Mark multiples of p within the segment
00160             bitmap_clear_steps_simd(sieve, 2 * p, start - low, high - low + 1);
00161         }
00162
00163         // Collect primes from the current segment
00164         uint64_t i = (low % 2 == 0) ? low + 1 : low;
00165         for (; i <= high; i += 2) // skip even numbers
00166         {
00167             if (bitmap_get_bit(sieve, i - low))
00168                 ui64_push(primes, i);
00169         }
00170
00171         // Move to the next segment
00172         low += segment_size;
00173         high += segment_size;
00174         if (high > n)
00175             high = n;
00176     }
00177
00178     // * Step 3: Cleanup and finalize
00179     bitmap_free(&sieve); // free bitmap
00180     ui64_resize_to_fit(primes); // Trim excess memory in primes array
00181
00182     return primes;
00183 }
00184
00200 UI64_ARRAY *SoEu(uint64_t n)
00201 {
00202     ASSERT_LIMIT(n); // Validate input limit
00203
00204     // initialization
00205     UI64_ARRAY *primes = ui64_init(Pi(n) * 1.4); // 40% over-estimation to avoid reallocs
00206     assert(primes && "Memory allocation failed for primes array.");
00207
00208     BITMAP *sieve = bitmap_init(n + 1, 1);
00209     if (sieve == NULL)
00210     {
00211         ui64_free(&primes);
00212         return NULL;
00213     }

```

```

00214
00215 // starting the prime list with 2 to skip reading even numbers
00216 ui64_push(primes, 2);
00217
00218 // sieve logic: iterate through odd numbers, marking composites by clearing multiples of primes
00219 for (uint64_t i = 3; i <= n; i += 2)
00220 {
00221     if (bitmap_get_bit(sieve, i))
00222         ui64_push(primes, i);
00223
00224     // Mark multiples of the current prime
00225     for (int j = 1; j < primes->count; ++j)
00226     {
00227         uint64_t p = primes->array[j];
00228
00229         if (p * i > n)
00230             break;
00231
00232         bitmap_clear_bit(sieve, p * i);
00233
00234         if (i % p == 0)
00235             break;
00236     }
00237 }
00238
00239 // cleanup
00240 bitmap_free(&sieve);
00241
00242 // Resize primes array to fit the exact number of primes found
00243 ui64_resize_to_fit(primes);
00244
00245 return primes;
00246 }
00247
00264 UI64_ARRAY *SoS(uint64_t n)
00265 {
00266     ASSERT_LIMIT(n); // Validate input limit
00267
00268     // Calculate k as the odd numbers up to n.
00269     uint64_t k = (n - 1) / 2 + 1;
00270
00271     // Initialize the primes object with an estimated capacity
00272     UI64_ARRAY *primes = ui64_init(Pi(n) * 1.4); // 40% over-estimation to avoid reallocs
00273     assert(primes && "Memory allocation failed for primes array.");
00274
00275     ui64_push(primes, 2);
00276
00277     // Create a bitmap with size k+1.
00278     BITMAP *sieve = bitmap_init(k + 8, 1);
00279     if (sieve == NULL)
00280     {
00281         ui64_free(&primes);
00282         return NULL;
00283     }
00284
00285     uint64_t n_sqrt = sqrt(n) + 1;
00286
00287     // iterate through odd numbers
00288     for (uint64_t i = 1; i < k; ++i)
00289     {
00290         if (bitmap_get_bit(sieve, i))
00291         {
00292             uint64_t p = 2 * i + 1;
00293             ui64_push(primes, p);
00294             if (p < n_sqrt)
00295             {
00296                 // First composite mark xp in the bitmap is given by:
00297                 //  $x_p = 2 * i^2 + 2 * i = p * i + i$ , corresponding to  $p^2$  in the odd set
00298                 uint64_t xp = p * i + i;
00299                 // Mark composites of p additively in the bitmap using step size p
00300                 bitmap_clear_steps_simd(sieve, p, xp, k);
00301             }
00302         }
00303     }
00304
00305     // Cleanup
00306     bitmap_free(&sieve);
00307
00308     // Resize primes array to fit the exact number of primes found
00309     ui64_resize_to_fit(primes);
00310
00311     return primes;
00312 }
00313
00328 UI64_ARRAY *SoA(uint64_t n)
00329 {
00330     ASSERT_LIMIT(n); // Validate input limit

```

```

00331
00332 UI64_ARRAY *primes = ui64_init(Pi(n) * 1.4); // 40% over-estimation to avoid reallocs
00333 assert(primes && "Memory allocation failed for primes array.");
00334
00335 // Create a bitmap to mark potential primes
00336 BITMAP *sieve = bitmap_init(n + 1, 0);
00337 if (sieve == NULL)
00338 {
00339     ui64_free(&primes);
00340     return NULL;
00341 }
00342
00343 // Add 2 and 3 to primes
00344 ui64_push(primes, 2);
00345 ui64_push(primes, 3);
00346
00347 // 1. Mark potential primes in the bitmap using the Atkin conditions
00348 // Condition 1: for all  $4x^2 + y^2 \leq n$ 
00349 for (uint64_t x = 1; 4 * x * x < n; x++)
00350 {
00351     uint64_t a = 4 * x * x;
00352     for (uint64_t y = 1; a + y * y <= n; ++y)
00353     {
00354         uint64_t b = a + y * y;
00355         // if  $4x^2 + y^2 \equiv 1 \text{ or } 5 \pmod{12}$ , flip the bit
00356         if (b % 12 == 1 || b % 12 == 5)
00357             bitmap_flip_bit(sieve, b);
00358     }
00359 }
00360
00361 // Condition 2: for all  $3x^2 + y^2 \leq n$ 
00362 for (uint64_t x = 1; 3 * x * x < n; x++)
00363 {
00364     uint64_t a = 3 * x * x;
00365     for (uint64_t y = 1; a + y * y <= n; ++y)
00366     {
00367         uint64_t b = a + y * y;
00368         // if  $3x^2 + y^2 \equiv 7 \pmod{12}$ , flip the bit
00369         if (b % 12 == 7)
00370             bitmap_flip_bit(sieve, b);
00371     }
00372 }
00373
00374 // Condition 3: for all  $3x^2 - y^2 \leq n$  and  $x > y$ 
00375 for (uint64_t x = 1; 2 * x * x < n; x++) // Approximation for loop bound
00376 {
00377     uint64_t a = 3 * x * x;
00378     for (uint64_t y = x - 1; y > 0; y--)
00379     {
00380         uint64_t b = a - y * y;
00381         if (b > n)
00382             break; // b increases as y decreases
00383
00384         // if  $3x^2 - y^2 \equiv 11 \pmod{12}$ , flip the bit
00385         if (b % 12 == 11)
00386             bitmap_flip_bit(sieve, b);
00387     }
00388 }
00389
00390 // 2. Remove remaining composites by sieving out multiples of squares of root primes
00391 uint64_t n_sqrt = sqrt(n);
00392 for (uint64_t p = 5; p <= n_sqrt; p += 2)
00393 {
00394     if (bitmap_get_bit(sieve, p))
00395     {
00396         // Mark odd multiples of  $p^2$  as non-prime
00397         // starting at  $p^2$  with step size  $2p^2$ 
00398         bitmap_clear_steps_simd(sieve, 2 * p * p, p * p, n + 1);
00399     }
00400 }
00401
00402 // 3. Collect primes from the bitmap
00403 for (uint64_t p = 5; p <= n; p += 2)
00404 {
00405     if (bitmap_get_bit(sieve, p))
00406         ui64_push(primes, p);
00407 }
00408
00409 // cleanup
00410 bitmap_free(&sieve);
00411
00412 // Resize primes array to fit the exact number of primes found
00413 ui64_resize_to_fit(primes);
00414
00415 return primes;
00416 }
00417

```

```

00418 // =====
00419 // * Sieve-iZ Algorithms
00420 // =====
00421
00444 UI64_ARRAY *SiZ(uint64_t n)
00445 {
00446     ASSERT_LIMIT(n); // Validate input limit
00447
00448     // Initialize primes object with enough initial estimation
00449     UI64_ARRAY *primes = ui64_init(Pi(n) * 1.4); // 40% over-estimation to avoid reallocs
00450     assert(primes && "Memory allocation failed for primes array.");
00451
00452     // Add 2, 3 to primes
00453     ui64_push(primes, 2);
00454     ui64_push(primes, 3);
00455
00456     // Calculate x_n, max x value in iZ space for given n
00457     uint64_t x_n = n / 6 + 1;
00458
00459     // Create bitmap X-Arrays x5, x7, each of size x_n + 1 bits
00460     BITMAP *x5 = bitmap_init(x_n + 1, 1);
00461     BITMAP *x7 = bitmap_init(x_n + 1, 1);
00462
00463     // Memory allocation failed, check logs
00464     if (!x5 || !x7)
00465     {
00466         ui64_free(&primes);
00467         return NULL;
00468     }
00469
00470     // Sieve logic
00471     process_iZ_bitmaps(primes, x5, x7, x_n);
00472
00473     // Cleanup: free memory of x5, x7
00474     bitmap_free(&x5);
00475     bitmap_free(&x7);
00476
00477     // Handle edge case: if last prime > n, remove it
00478     if (primes->array[primes->count - 1] > n)
00479         ui64_pop(primes);
00480
00481     // Trim unused memory in primes object
00482     ui64_resize_to_fit(primes);
00483
00484     return primes;
00485 }
00486
00516 UI64_ARRAY *SiZm(uint64_t n)
00517 {
00518     ASSERT_LIMIT(n); // Validate input limit
00519
00520     // if n < 10000, return SiZ(n), doesn't worth segmenting
00521     if (n < 10000)
00522         return SiZ(n);
00523
00524     // * 1. Initialization:
00525     // Initialize primes array with enough capacity to avoid reallocs
00526     UI64_ARRAY *primes = ui64_init(Pi(n) * 1.4); // 40% over-estimation to avoid reallocs
00527     assert(primes && "Memory allocation failed for primes array in SiZm.");
00528
00529     // Compute vx wheel size that fits in L2 cache
00530     int vx = compute_l2_vx(n);
00531
00532     // Initialize and construct base bitmaps for iZm/vx
00533     BITMAP *base_x5 = bitmap_init(vx + 8, 1);
00534     BITMAP *base_x7 = bitmap_init(vx + 8, 1);
00535     if (!base_x5 || !base_x7)
00536     {
00537         ui64_free(&primes);
00538         return NULL;
00539     }
00540     iZm_construct_vx_base(vx, base_x5, base_x7);
00541
00542     // Add the pre-sieved k primes to primes array
00543     const uint64_t base_primes[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
00544     int k = 0;
00545     while ((6 * vx) % base_primes[k] == 0)
00546         ui64_push(primes, base_primes[k++]);
00547
00548     // * 2. Process first segment (y = 0) to collect root primes:
00549     uint64_t x_n = n / 6 + 1; // max x value up to n
00550     // uint64_t root_limit = sqrt(6 * x_limit) + 1;
00551     // Initialize active sieve bitmaps from base
00552     BITMAP *x5 = bitmap_clone(base_x5);
00553     BITMAP *x7 = bitmap_clone(base_x7);
00554     if (!x5 || !x7)
00555     {

```

```

00556     ui64_free(&primes);
00557     bitmap_free(&base_x5);
00558     bitmap_free(&base_x7);
00559     return NULL;
00560 }
00561 process_iZ_bitmaps(primes, x5, x7, vx + 1);
00562
00563 // * 3. Process remaining segments (y >= 1) to collect primes:
00564 int y_limit = x_n / vx; // number of full segments to process
00565 uint64_t yvx = vx; // current base value (y * vx)
00566 for (int y = 1; y <= y_limit; y++)
00567 {
00568     // * a. Reset active bitmaps to base state
00569     memcpy(x5->data, base_x5->data, x5->byte_size);
00570     memcpy(x7->data, base_x7->data, x7->byte_size);
00571
00572     int x_limit = (y < y_limit) ? vx : x_n % vx; // local x limit adjusted for last segment
00573     uint64_t root_limit = sqrt(6 * (yvx + x_limit)) + 1; // local root limit for current segment
00574
00575     // * b. Mark composites of root primes in current segment
00576     for (int i = k; i < primes->count; i++)
00577     {
00578         uint64_t p = primes->array[i];
00579         if (p > root_limit)
00580             break;
00581
00582         // mark composites of p in current segment
00583         bitmap_clear_steps_simd(x5, p, iZm_solve_for_x0(-1, p, vx, y), x_limit);
00584         bitmap_clear_steps_simd(x7, p, iZm_solve_for_x0(1, p, vx, y), x_limit);
00585     }
00586
00587     // * c. Collect unmarked indices as primes in current segment
00588     for (int x = 2; x <= x_limit; x++)
00589     {
00590         if (bitmap_get_bit(x5, x)) // i.e. iZ- prime
00591             ui64_push(primes, iZ(yvx + x, -1));
00592
00593         if (bitmap_get_bit(x7, x)) // i.e. iZ+ prime
00594             ui64_push(primes, iZ(yvx + x, 1));
00595     }
00596
00597     yvx += vx; // advance yvx for next segment
00598 }
00599
00600 // * 4. Clean up and finalize
00601 bitmap_free(&x5);
00602 bitmap_free(&x7);
00603 bitmap_free(&base_x5);
00604 bitmap_free(&base_x7);
00605
00606 // Handle edge case: if last prime > n, remove it
00607 if (primes->array[primes->count - 1] > n)
00608     ui64_pop(primes);
00609
00610 ui64_resize_to_fit(primes); // Trim excess memory in primes array
00611 return primes;
00612 }
00613
00623 UI64_ARRAY *SiZm_vy(uint64_t n)
00624 {
00625     ASSERT_LIMIT(n); // Validate input limit
00626
00627     // if n is less than 10000, return SiZ(n)
00628     if (n < 10000)
00629         return SiZ(n);
00630
00631     // * 1. Initialization:
00632     // Initialize primes array with enough capacity to avoid reallocs
00633     UI64_ARRAY *primes = ui64_init(Pi(n) * 1.4);
00634     assert(primes && "Memory allocation failed for primes array in SiZm.");
00635
00636     uint64_t x_n = n / 6 + 1; // iZ limit for n
00637     uint64_t root_limit = sqrt(n) + 1;
00638
00639     get_root_primes(primes, root_limit);
00640     int root_count = primes->count;
00641
00642     int k = 4; // pointing at 11 in root_primes
00643     int vx = 35;
00644     if (n >= pow(10, 9))
00645     {
00646         vx *= 11;
00647         k++;
00648     }
00649     if (n >= pow(10, 11))
00650     {
00651         vx *= 13;

```

```

00652     k++;
00653 }
00654
00655 int vy = x_n / vx;
00656
00657 BITMAP *sieve = bitmap_init(vy + 8, 1);
00658
00659 // * 2. Sieve logic: Process iZm's columns as segments
00660 for (int x = 2; x <= vx; x++)
00661 {
00662     // Handle iZ- case:
00663     // crucial, ensure iZ(x, -1) is coprime to vx before sieving
00664     if (gcd(iZ(x, -1), vx) == 1)
00665     {
00666         // * a. reset sieve bitmap for new segment
00667         bitmap_set_all(sieve); // set all bits
00668
00669         // * b. mark composites of root primes in sieve
00670         for (int i = k; i < root_count; i++)
00671         {
00672             uint64_t p = primes->array[i];
00673             int64_t y_0 = iZm_solve_for_y0(-1, p, vx, x);
00674             bitmap_clear_steps_simd(sieve, p, y_0, vy);
00675         }
00676
00677         // * c. collect primes from sieve up to y = vy-1
00678         for (int y = 0; y < vy; y++)
00679         {
00680             if (bitmap_get_bit(sieve, y))
00681                 ui64_push(primes, iZ(y * vx + x, -1));
00682         }
00683         // handle partial last row where y = vy (check if p < n before pushing)
00684         if (bitmap_get_bit(sieve, vy))
00685         {
00686             uint64_t p = iZ(vy * vx + x, -1);
00687             if (p < n)
00688                 ui64_push(primes, p);
00689         }
00690     }
00691
00692     // Handle iZ+ case similarly:
00693     if (gcd(iZ(x, 1), vx) == 1)
00694     {
00695         bitmap_set_all(sieve); // reset sieve bitmap for new segment
00696         for (int i = k; i < root_count; i++)
00697         {
00698             uint64_t p = primes->array[i];
00699             int64_t y_0 = iZm_solve_for_y0(1, p, vx, x);
00700             bitmap_clear_steps_simd(sieve, p, y_0, vy);
00701         }
00702
00703         for (int y = 0; y < vy; y++)
00704         {
00705             if (bitmap_get_bit(sieve, y))
00706                 ui64_push(primes, iZ(y * vx + x, 1));
00707         }
00708         if (bitmap_get_bit(sieve, vy))
00709         {
00710             uint64_t p = iZ(vy * vx + x, 1);
00711             if (p < n)
00712                 ui64_push(primes, p);
00713         }
00714     }
00715 }
00716
00717 // * 3. Clean up and finalize
00718 bitmap_free(&sieve);
00719 ui64_resize_to_fit(primes); // Trim excess memory in primes array
00720
00721 primes->ordered = 0; // Mark the array as unordered
00722 return primes;
00723 }

```

## 13.21 src/toolkit/bitmap.c File Reference

Implementation of the bitmap module for efficient bit array operations.

#include <bitmap.h>

## Functions

- **BITMAP \* bitmap\_init** (size\_t size, int set\_bits)  
Creates and initializes a new bitmap with the specified number of bits.
- void **bitmap\_free** (BITMAP \*\*bitmap)  
Frees all memory associated with a bitmap and nullifies the pointer.
- void **bitmap\_set\_bit** (BITMAP \*bitmap, size\_t idx)  
Sets a specific bit to 1.
- int **bitmap\_get\_bit** (BITMAP \*bitmap, size\_t idx)  
Gets the value of a specific bit.
- void **bitmap\_flip\_bit** (BITMAP \*bitmap, size\_t idx)  
Flips the value of a specific bit.
- void **bitmap\_clear\_bit** (BITMAP \*bitmap, size\_t idx)  
Clears (sets to 0) a specific bit.
- void **bitmap\_set\_all** (BITMAP \*bitmap)  
Sets all bits in the array to 1.
- void **bitmap\_clear\_all** (BITMAP \*bitmap)  
Clears all bits in the bitmap (sets them to 0).
- void **bitmap\_clear\_steps** (BITMAP \*bitmap, uint64\_t step, uint64\_t start\_idx, uint64\_t limit)  
Clear every step-th bit from start\_idx up to limit.
- void **bitmap\_clear\_steps\_simd** (BITMAP \*bitmap, uint64\_t step, uint64\_t start\_idx, uint64\_t limit)  
SIMD-optimized version of bitmap\_clear\_steps.
- **BITMAP \* bitmap\_clone** (BITMAP \*src)  
Creates a deep copy (clone) of an existing bitmap.
- void **bitmap\_compute\_hash** (BITMAP \*bitmap)  
Generates a SHA-256 hash for the bitmap->data.
- int **bitmap\_validate\_hash** (BITMAP \*bitmap)  
Validates the SHA-256 hash stored in bitmap->sha256.
- int **bitmap\_fwrite** (BITMAP \*bitmap, FILE \*file)  
Writes the bitmap to a binary file.
- **BITMAP \* bitmap\_fread** (FILE \*file)  
Reads a bitmap from a binary file.

### 13.21.1 Detailed Description

Implementation of the bitmap module for efficient bit array operations.

This file provides the complete implementation of bitmap functions including memory management, bit manipulation, data integrity verification via SHA-256 hashing, and binary file I/O with automatic hash validation.

#### 13.21.1.1 Implementation Notes

- All functions include NULL pointer and bounds checking
- File I/O operations verify all read/write operations succeed
- Overflow protection is implemented in loop-based operations
- Memory is managed with proper cleanup on errors
- SHA-256 hashing uses OpenSSL library



## 13.21.1.2 Performance Characteristics

- [bitmap\\_init\(\)](#):  $O(n/8)$  where  $n$  is number of bits
- Individual bit ops:  $O(1)$
- Bulk operations:  $O(n/8)$
- Hash computation:  $O(n/8)$
- File I/O:  $O(n/8)$

Author

iZprime.com

Date

October 2025

Version

1.0

See also

[bitmap.h](#) for API documentation  
[test/test\\_bitmap.c](#) for test suite

Definition in file [bitmap.c](#).

## 13.22 bitmap.c

[Go to the documentation of this file.](#)

```

00001
00030
00031 #include <bitmap.h>
00032
00033 #ifdef __aarch64__
00034 #include <arm_neon.h>
00035 #elif defined(__x86_64__) || defined(_M_X64)
00036 #include <immintrin.h>
00037 #endif
00038
00057 BITMAP *bitmap_init(size_t size, int set_bits)
00058 {
00059     assert(size > 0 && "Bitmap size must be positive");
00060
00061     BITMAP *bitmap = (BITMAP *)malloc(sizeof(BITMAP));
00062     if (bitmap == NULL)
00063     {
00064         log_error("Memory allocation failed for BITMAP struct");
00065         return NULL;
00066     }
00067
00068     bitmap->size = size;
00069     bitmap->byte_size = (size + 7) / 8;
00070     bitmap->data = (unsigned char *)malloc(bitmap->byte_size);
00071     if (bitmap->data == NULL)
00072     {
00073         free(bitmap);
00074         log_error("bitmap_create: Memory allocation failed for BITMAP");

```

```

00075     return NULL;
00076 }
00077
00078 if (set_bits)
00079     memset(bitmap->data, 0xFF, bitmap->byte_size);
00080 else
00081     memset(bitmap->data, 0x00, bitmap->byte_size);
00082
00083 memset(bitmap->sha256, 0, SHA256_DIGEST_LENGTH); // Initialize SHA-256 to zero
00084
00085 return bitmap;
00086 }
00087
00103 void bitmap_free(BITMAP **bitmap)
00104 {
00105     // Safe guard against double free and NULL pointer
00106     if (bitmap && *bitmap)
00107     {
00108         if ((*bitmap)->data)
00109         {
00110             free((*bitmap)->data);
00111             (*bitmap)->data = NULL;
00112         }
00113
00114         free(*bitmap);
00115         *bitmap = NULL;
00116     }
00117 }
00118
00125 void bitmap_set_bit(BITMAP *bitmap, size_t idx)
00126 {
00127     bitmap->data[idx / 8] |= (1 « (idx % 8));
00128 }
00129
00137 int bitmap_get_bit(BITMAP *bitmap, size_t idx)
00138 {
00139     return (bitmap->data[idx / 8] & (1 « (idx % 8))) != 0;
00140 }
00141
00148 void bitmap_flip_bit(BITMAP *bitmap, size_t idx)
00149 {
00150     bitmap->data[idx / 8] ^= (1 « (idx % 8));
00151 }
00152
00159 void bitmap_clear_bit(BITMAP *bitmap, size_t idx)
00160 {
00161     // Clear the bit by ANDing with the negation of the bit mask
00162     bitmap->data[idx / 8] &= ~(1 « (idx % 8));
00163 }
00164
00170 void bitmap_set_all(BITMAP *bitmap)
00171 {
00172     memset(bitmap->data, 0xFF, (bitmap->size + 7) / 8);
00173 }
00174
00180 void bitmap_clear_all(BITMAP *bitmap)
00181 {
00182     memset(bitmap->data, 0x00, (bitmap->size + 7) / 8);
00183 }
00184
00210 void bitmap_clear_steps(BITMAP *bitmap, uint64_t step, uint64_t start_idx, uint64_t limit)
00211 {
00212     assert(step > 0 && "Step must be positive in bitmap_clear_steps.");
00213     limit = MIN(limit, bitmap->size - 1);
00214
00215     for (uint64_t idx = start_idx; idx <= limit; idx += step)
00216     {
00217         bitmap->data[idx / 8] &= ~(1 « (idx % 8));
00218     }
00219 }
00220
00233 void bitmap_clear_steps_simd(BITMAP *bitmap, uint64_t step, uint64_t start_idx, uint64_t limit)
00234 {
00235     assert(step > 0 && "Step must be positive in bitmap_clear_steps_simd.");
00236     limit = MIN(limit, bitmap->size - 1);
00237
00238 #ifdef __aarch64__
00239     // NEON implementation for ARM64
00240     uint64_t idx = start_idx;
00241
00242     // Process 4 steps at a time
00243     // Condition: idx + 3*step <= limit
00244
00245     if (limit >= 3 * step && idx <= limit - 3 * step)
00246     {
00247         uint64x2_t v_step2 = vdupq_n_u64(2 * step);
00248         uint64x2_t v_step4 = vdupq_n_u64(4 * step);

```

```

00249
00250 // Initialize indices: [idx, idx+step] and [idx+2step, idx+3step]
00251 uint64x2_t v_idx_01 = vdupq_n_u64(idx);
00252 v_idx_01 = vsetq_lane_u64(idx + step, v_idx_01, 1);
00253
00254 uint64x2_t v_idx_23 = vaddq_u64(v_idx_01, v_step2);
00255
00256 while (idx <= limit - 3 * step)
00257 {
00258     // Extract indices and perform operations
00259     // Note: NEON doesn't have efficient scatter for bytes, so we extract and do scalar stores
00260
00261     uint64_t i0 = vgetq_lane_u64(v_idx_01, 0);
00262     uint64_t i1 = vgetq_lane_u64(v_idx_01, 1);
00263     uint64_t i2 = vgetq_lane_u64(v_idx_23, 0);
00264     uint64_t i3 = vgetq_lane_u64(v_idx_23, 1);
00265
00266     bitmap->data[i0 / 8] &= ~(1 « (i0 % 8));
00267     bitmap->data[i1 / 8] &= ~(1 « (i1 % 8));
00268     bitmap->data[i2 / 8] &= ~(1 « (i2 % 8));
00269     bitmap->data[i3 / 8] &= ~(1 « (i3 % 8));
00270
00271     // Advance indices
00272     v_idx_01 = vaddq_u64(v_idx_01, v_step4);
00273     v_idx_23 = vaddq_u64(v_idx_23, v_step4);
00274
00275     idx += 4 * step;
00276 }
00277 }
00278
00279 // Handle remaining steps
00280 for (; idx <= limit; idx += step)
00281 {
00282     bitmap->data[idx / 8] &= ~(1 « (idx % 8));
00283 }
00284
00285 #elif defined(__AVX2__)
00286 // AVX2 implementation for x86_64
00287 uint64_t idx = start_idx;
00288
00289 // Process 4 steps at a time
00290 if (limit >= 3 * step && idx <= limit - 3 * step)
00291 {
00292     __m256i v_step4 = _mm256_set1_epi64x(4 * step);
00293
00294     // Initialize indices: [idx, idx+step, idx+2step, idx+3step]
00295     // Note: __mm256_set_epi64x takes arguments in reverse order (e3, e2, e1, e0)
00296     __m256i v_idx = _mm256_set_epi64x(idx + 3 * step, idx + 2 * step, idx + step, idx);
00297
00298     while (idx <= limit - 3 * step)
00299     {
00300         // Extract indices
00301         uint64_t i0 = _mm256_extract_epi64(v_idx, 0);
00302         uint64_t i1 = _mm256_extract_epi64(v_idx, 1);
00303         uint64_t i2 = _mm256_extract_epi64(v_idx, 2);
00304         uint64_t i3 = _mm256_extract_epi64(v_idx, 3);
00305
00306         bitmap->data[i0 / 8] &= ~(1 « (i0 % 8));
00307         bitmap->data[i1 / 8] &= ~(1 « (i1 % 8));
00308         bitmap->data[i2 / 8] &= ~(1 « (i2 % 8));
00309         bitmap->data[i3 / 8] &= ~(1 « (i3 % 8));
00310
00311         // Advance indices
00312         v_idx = _mm256_add_epi64(v_idx, v_step4);
00313         idx += 4 * step;
00314     }
00315 }
00316
00317 // Handle remaining steps
00318 for (; idx <= limit; idx += step)
00319 {
00320     bitmap->data[idx / 8] &= ~(1 « (idx % 8));
00321 }
00322
00323 #elif defined(__SSE2__)
00324 // SSE2 implementation for x86_64
00325 uint64_t idx = start_idx;
00326
00327 // Process 2 steps at a time
00328 if (limit >= step && idx <= limit - step)
00329 {
00330     __m128i v_step2 = _mm_set1_epi64x(2 * step);
00331     // Note: _mm_set_epi64x takes arguments in reverse order (e1, e0)
00332     __m128i v_idx = _mm_set_epi64x(idx + step, idx);
00333
00334     while (idx <= limit - step)
00335     {

```

```

00336         // Extract indices
00337         // _mm_cvtsi128_si64 extracts the lower 64 bits (element 0)
00338         uint64_t i0 = _mm_cvtsi128_si64(v_idx);
00339         // Unpack high to low to extract element 1
00340         uint64_t i1 = _mm_cvtsi128_si64(_mm_unpackhi_epi64(v_idx, v_idx));
00341
00342         bitmap->data[i0 / 8] &= ~(1 « (i0 % 8));
00343         bitmap->data[i1 / 8] &= ~(1 « (i1 % 8));
00344
00345         // Advance indices
00346         v_idx = _mm_add_epi64(v_idx, v_step2);
00347         idx += 2 * step;
00348     }
00349 }
00350
00351 // Handle remaining steps
00352 for (; idx <= limit; idx += step)
00353 {
00354     bitmap->data[idx / 8] &= ~(1 « (idx % 8));
00355 }
00356
00357 #else
00358 // Fallback to scalar implementation
00359 for (uint64_t idx = start_idx; idx <= limit; idx += step)
00360 {
00361     bitmap->data[idx / 8] &= ~(1 « (idx % 8));
00362 }
00363 #endif
00364 }
00365
00382 BITMAP *bitmap_clone(BITMAP *src)
00383 {
00384     assert(src && src->data && "Invalid source bitmap passed to bitmap_clone.");
00385
00386     BITMAP *dest = bitmap_init(src->size, 0);
00387     memcpy(dest->data, src->data, src->byte_size);
00388     memcpy(dest->sha256, src->sha256, SHA256_DIGEST_LENGTH);
00389
00390     return dest;
00391 }
00392
00398 void bitmap_compute_hash(BITMAP *bitmap)
00399 {
00400     assert(bitmap && bitmap->data && "Invalid bitmap passed to bitmap_compute_hash.");
00401
00402     // Generate SHA-256 hash and store it in the struct
00403     SHA256((unsigned char *)bitmap->data, bitmap->byte_size, bitmap->sha256);
00404 }
00405
00412 int bitmap_validate_hash(BITMAP *bitmap)
00413 {
00414     assert(bitmap && bitmap->data && "Invalid bitmap");
00415
00416     unsigned char correct_hash[SHA256_DIGEST_LENGTH]; // Buffer to hold the computed hash
00417
00418     // Generate SHA-256 hash and store it in correct_hash
00419     SHA256((unsigned char *)bitmap->data, bitmap->byte_size, correct_hash);
00420
00421     // Compare actual_hash with the stored hash in bitmap->sha256
00422     if (memcmp(correct_hash, bitmap->sha256, SHA256_DIGEST_LENGTH) == 0)
00423     {
00424         return 1; // SHA-256 match
00425     }
00426     else
00427     {
00428         log_error("SHA-256 checksum validation failed.");
00429         return 0; // SHA-256 mismatch
00430     }
00431 }
00432
00440 int bitmap_fwrite(BITMAP *bitmap, FILE *file)
00441 {
00442     assert(bitmap && bitmap->data && "Invalid bitmap passed to bitmap_fwrite.");
00443     assert(file && "File pointer is NULL in bitmap_fwrite.");
00444
00445     // Write size
00446     if (fwrite(&bitmap->size, sizeof(size_t), 1, file) != 1)
00447     {
00448         log_error("Failed to write bitmap size to file.");
00449         return 0;
00450     }
00451
00452     // Write data
00453     if (fwrite(bitmap->data, sizeof(unsigned char), bitmap->byte_size, file) != bitmap->byte_size)
00454     {
00455         log_error("Failed to write bitmap data to file.");
00456         return 0;

```

```

00457     }
00458
00459     // Compute SHA-256 hash if not already computed
00460     // Check if hash is all zeros by examining all bytes
00461     int hash_is_zero = 1;
00462     for (size_t i = 0; i < SHA256_DIGEST_LENGTH; i++)
00463     {
00464         if (bitmap->sha256[i] != 0)
00465         {
00466             hash_is_zero = 0;
00467             break;
00468         }
00469     }
00470
00471     if (hash_is_zero)
00472     {
00473         bitmap_compute_hash(bitmap);
00474     }
00475
00476     // Write SHA-256 hash
00477     if (fwrite(bitmap->sha256, 1, SHA256_DIGEST_LENGTH, file) != SHA256_DIGEST_LENGTH)
00478     {
00479         log_error("Failed to write SHA-256 hash to file.");
00480         return 0;
00481     }
00482
00483     return 1; // Success
00484 }
00485
00492 BITMAP *bitmap_fread(FILE *file)
00493 {
00494     assert(file && "File pointer is NULL in bitmap_fread.");
00495
00496     // Read size
00497     size_t size;
00498     if (fread(&size, sizeof(size_t), 1, file) != 1)
00499     {
00500         log_error("Failed to read bitmap size from file.");
00501         return NULL;
00502     }
00503
00504     BITMAP *bitmap = bitmap_init(size, 0);
00505     if (bitmap == NULL)
00506     {
00507         log_error("Failed to initialize bitmap.");
00508         return NULL;
00509     }
00510
00511     // Read data directly into the already-allocated buffer
00512     if (fread(bitmap->data, sizeof(unsigned char), bitmap->byte_size, file) != bitmap->byte_size)
00513     {
00514         log_error("Failed to read complete bitmap data from file.");
00515         bitmap_free(&bitmap);
00516         return NULL;
00517     }
00518
00519     // Read SHA-256 hash
00520     if (fread(bitmap->sha256, 1, SHA256_DIGEST_LENGTH, file) != SHA256_DIGEST_LENGTH)
00521     {
00522         log_error("Failed to read SHA-256 hash from file.");
00523         bitmap_free(&bitmap);
00524         return NULL;
00525     }
00526
00527     if (!bitmap_validate_hash(bitmap))
00528     {
00529         log_error("SHA-256 hash validation failed.");
00530         bitmap_free(&bitmap);
00531         return NULL;
00532     }
00533
00534     return bitmap;
00535 }

```

## 13.23 src/toolkit/int\_arrays.c File Reference

Implementation of dynamic integer array module for 16-bit, 32-bit, and 64-bit unsigned integers.

```

#include <int_arrays.h>
#include "templates/int_array_impl.inc"

```

### 13.23.1 Detailed Description

Implementation of dynamic integer array module for 16-bit, 32-bit, and 64-bit unsigned integers.

This file provides the complete implementation of [UI16\\_ARRAY](#), [UI32\\_ARRAY](#), and [UI64\\_ARRAY](#) functions including memory management, automatic capacity growth, data integrity verification via SHA-256 hashing, and binary file I/O with automatic hash validation.

#### 13.23.1.1 Implementation Notes

- All functions include NULL pointer checking
- Automatic capacity doubling (2x growth) ensures amortized O(1) append operations
- File I/O operations automatically compute and validate SHA-256 hashes
- Memory is managed with proper cleanup on errors
- SHA-256 hashing uses OpenSSL library

#### 13.23.1.2 Performance Characteristics

- `init()`: O(1)
- `push()` (amortized): O(1)
- `push()` (worst case): O(n) during resize
- Hash operations: O(n)
- File I/O: O(n)

Author

iZprime.com

Date

October 2025

Version

1.0

See also

[int\\_arrays.h](#) for API documentation

Definition in file [int\\_arrays.c](#).

## 13.24 int\_arrays.c

[Go to the documentation of this file.](#)

```
00001
00029
00030 #include <int_arrays.h>
00031
00032 //
=====
00033 // UI16_ARRAY IMPLEMENTATION
00034 //
=====
00036 #define TEMPLATE_TYPE uint16_t
00037 #define TEMPLATE_STRUCT UI16_ARRAY
00038 #define TEMPLATE_FUNC(name) ui16_##name
00039 #define TEMPLATE_NAME_STR "UI16_ARRAY"
00041 #include "templates/int_array_impl.inc"
00043 #undef TEMPLATE_TYPE
00044 #undef TEMPLATE_STRUCT
00045 #undef TEMPLATE_FUNC
00046 #undef TEMPLATE_NAME_STR
00048
00049 //
=====
00050 // UI32_ARRAY IMPLEMENTATION
00051 //
=====
00053 #define TEMPLATE_TYPE uint32_t
00054 #define TEMPLATE_STRUCT UI32_ARRAY
00055 #define TEMPLATE_FUNC(name) ui32_##name
00056 #define TEMPLATE_NAME_STR "UI32_ARRAY"
00058 #include "templates/int_array_impl.inc"
00060 #undef TEMPLATE_TYPE
00061 #undef TEMPLATE_STRUCT
00062 #undef TEMPLATE_FUNC
00063 #undef TEMPLATE_NAME_STR
00065
00066 //
=====
00067 // UI64_ARRAY IMPLEMENTATION
00068 //
=====
00070 #define TEMPLATE_TYPE uint64_t
00071 #define TEMPLATE_STRUCT UI64_ARRAY
00072 #define TEMPLATE_FUNC(name) ui64_##name
00073 #define TEMPLATE_NAME_STR "UI64_ARRAY"
00075 #include "templates/int_array_impl.inc"
00077 #undef TEMPLATE_TYPE
00078 #undef TEMPLATE_STRUCT
00079 #undef TEMPLATE_FUNC
00080 #undef TEMPLATE_NAME_STR
```

## 13.25 src/toolkit/iZ\_toolkit.c File Reference

Implementation of iZ index space helpers and iZm/VX segment machinery.

```
#include <iZ_api.h>
```

### Functions

- `uint64_t iZ (uint64_t x, int i)`  
Computes  $6x + i$  for a given  $x$  and  $i$ .
- `void iZ_mpz (mpz_t z, mpz_t x, int i)`  
Computes  $6x + i$  for arbitrary precision values using GMP.
- `static int compute_k_vx (IZM *iZm)`  
Count how many small primes ( $>3$ ) divide  $vx$ .
- `void iZm_construct_vx_base (uint64_t vx, BITMAP *base_x5, BITMAP *base_x7)`

- Build pre-sieved base bitmaps for a VX segment.
- static int `vx_set_base_values` (`VX_SEG` \*vx\_obj, char \*y\_str)
  - Initialize mpz-dependent base fields for a VX segment object.
- static void `vx_det_sieve` (`IZM` \*iZm, `VX_SEG` \*vx\_obj)
  - Deterministic phase: mark composites using root primes.
- static void `vx_prob_sieve` (`VX_SEG` \*vx\_obj)
  - Perform probabilistic sieve cleanup for large numeric ranges.
- `IZM_RANGE_INFO` `range_info_init` (`INPUT_SIEVE_RANGE` \*input\_range, int vx)
  - Map a decimal range input into iZm coordinates and segment bounds.
- void `range_info_free` (`IZM_RANGE_INFO` \*info)
  - Clear all GMP fields owned by info.

### iZ Mapping Helpers

- void `process_iZ_bitmaps` (`UI64_ARRAY` \*primes, `BITMAP` \*x5, `BITMAP` \*x7, uint64\_t x↔\_limit)
  - Traverse iZ bitmaps, emit surviving primes, and mark composites.
- void `get_root_primes` (`UI64_ARRAY` \*primes, uint64\_t limit)
  - Generate primes up to limit for deterministic sieving.
- int `check_primality` (mpz\_t n, int rounds)
  - Check the primality of a number using GMP's probabilistic test.

### IZM Lifecycle

- `IZM` \* `iZm_init` (size\_t vx)
  - Allocate and initialize an `IZM` object for a given VX.
- `IZM` \* `iZm_clone` (`IZM` \*src)
  - Deep-copy an `IZM` object for per-worker ownership.
- void `iZm_free` (`IZM` \*\*iZm)
  - Release an `IZM` object and set the caller pointer to NULL.

### VX Selection Helpers

- uint64\_t `compute_vx_k` (int k)
  - Calculate `VX_{ k }`.
- uint64\_t `compute_l2_vx` (uint64\_t n)
  - Choose VX using an L2-cache-aware heuristic.
- void `compute_max_vx` (mpz\_t vx, int bit\_size)
  - Compute largest VX below  $2^{\text{bit\_size}}$ .

### Modular Hit Solvers

- uint64\_t `iZm_solve_for_x0` (int m\_id, uint64\_t p, uint64\_t vx, uint64\_t y)
  - Solve first x-hit of prime p for line m\_id in segment y.
- uint64\_t `iZm_solve_for_x0_mpz` (int m\_id, uint64\_t p, uint64\_t vx, mpz\_t y)
  - GMP variant of `iZm_solve_for_x0()` for very large y.
- int64\_t `iZm_solve_for_y0` (int m\_id, uint64\_t p, uint64\_t vx, uint64\_t x)
  - Solve first y-hit for fixed x in vertical (vy) scanning.

### VX Segment Lifecycle and Execution

- `VX_SEG` \* `vx_init` (`IZM` \*iZm, int start\_x, int end\_x, char \*y\_str, int mr\_rounds)
  - Initialize and deterministically sieve one VX segment.
- void `vx_free` (`VX_SEG` \*\*vx\_obj)



- Free a VX segment and all owned resources.
- void [vx\\_collect\\_p\\_gaps](#) ([VX\\_SEG](#) \*vx\_obj)  
Extract prime-gap encoding from a fully sieved segment.
- void [vx\\_full\\_sieve](#) ([VX\\_SEG](#) \*vx\_obj, int collect\_p\_gaps)  
Complete segment processing (probabilistic stage and optional gaps).
- void [vx\\_stream](#) ([VX\\_SEG](#) \*vx\_obj, FILE \*output)  
Stream segment primes to an output stream.

#### Random Prime Search Routines

- int [vx\\_search\\_prime](#) (mpz\_t p, int m\_id, int vx, int bit\_size)  
Horizontal iZm/VX random-prime search.
- int [vy\\_search\\_prime](#) (mpz\_t p, int m\_id, mpz\_t vx)  
Vertical iZm/VY random-prime search.

#### Variables

- static const int [s\\_primes](#) []
- static int [s\\_primes\\_count](#) = sizeof([s\\_primes](#)) / sizeof(int)

### 13.25.1 Detailed Description

Implementation of iZ index space helpers and iZm/VX segment machinery.

Definition in file [iZ\\_toolkit.c](#).

### 13.25.2 Function Documentation

#### 13.25.2.1 compute\_k\_vx()

```
int compute_k_vx (
    IZM * iZm) [static]
```

Count how many small primes (>3) divide vx.

#### Parameters

iZm	Initialized toolkit context.
-----	------------------------------

#### Returns

Number of pre-sieved small primes encoded in vx.

Definition at line 52 of file [iZ\\_toolkit.c](#).

References [s\\_primes](#), and [IZM::vx](#).

Referenced by [iZm\\_init\(\)](#).

#### 13.25.2.2 vx\_det\_sieve()

```
void vx_det_sieve (
    IZM * iZm,
    VX\_SEG * vx_obj) [static]
```

Deterministic phase: mark composites using root primes.

#### Parameters

iZm	Toolkit context containing root-prime table and base bitmaps.
-----	---

<code>vx_obj</code>	Segment object to update.
---------------------	---------------------------

Definition at line 566 of file [iZ\\_toolkit.c](#).

References [UI64\\_ARRAY::array](#), [VX\\_SEG::bit\\_ops](#), [bitmap\\_clear\\_steps\\_simd\(\)](#), [bitmap\\_get\\_bit\(\)](#), [UI64\\_ARRAY::count](#), [VX\\_SEG::end\\_x](#), [VX\\_SEG::is\\_large\\_limit](#), [iZm\\_solve\\_for\\_x0\(\)](#), [iZm\\_solve\\_for\\_x0\\_mpz\(\)](#), [IZM::k\\_vx](#), [VX\\_SEG::p\\_count](#), [VX\\_SEG::root\\_limit](#), [IZM::root\\_primes](#), [VX\\_SEG::start\\_x](#), [VX\\_SEG::vx](#), [VX\\_SEG::x5](#), [VX\\_SEG::x7](#), and [VX\\_SEG::y](#).

Referenced by [vx\\_init\(\)](#).

### 13.25.2.3 vx\_prob\_sieve()

```
void vx_prob_sieve (
    VX\_SEG * vx_obj) [static]
```

Perform probabilistic sieve cleanup for large numeric ranges.

Parameters

<code>vx_obj</code>	Segment object containing deterministic survivors.
---------------------	--

Definition at line 633 of file [iZ\\_toolkit.c](#).

References [bitmap\\_clear\\_bit\(\)](#), [bitmap\\_get\\_bit\(\)](#), [check\\_primalty\(\)](#), [VX\\_SEG::end\\_x](#), [VX\\_SEG::is\\_large\\_limit](#), [iZ\\_mpz\(\)](#), [log\\_info\(\)](#), [VX\\_SEG::mr\\_rounds](#), [VX\\_SEG::p\\_count](#), [VX\\_SEG::p\\_test\\_ops](#), [VX\\_SEG::start\\_x](#), [VX\\_SEG::x5](#), [VX\\_SEG::x7](#), and [VX\\_SEG::yvx](#).

Referenced by [vx\\_full\\_sieve\(\)](#).

### 13.25.2.4 vx\_set\_base\_values()

```
int vx_set_base_values (
    VX\_SEG * vx_obj,
    char * y_str) [static]
```

Initialize mpz-dependent base fields for a VX segment object.

Parameters

<code>vx_obj</code>	Segment object to populate.
<code>y_str</code>	Decimal y-coordinate for the segment.

Returns

1 on success, 0 on parse/initialization failure.

Definition at line 529 of file [iZ\\_toolkit.c](#).

References [VX\\_SEG::is\\_large\\_limit](#), [iZ\\_mpz\(\)](#), [log\\_error\(\)](#), [VX\\_SEG::root\\_limit](#), [VX\\_SEG::vx](#), [VX\\_SEG::y](#), and [VX\\_SEG::yvx](#).

Referenced by [vx\\_init\(\)](#).

### 13.25.3 Variable Documentation

#### 13.25.3.1 s\_primes

```
const int s_primes[] [static]
```

Initial value:

```
= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,  
   43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}
```

Small primes used to compute and construct wheel structures.

Definition at line 10 of file [iZ\\_toolkit.c](#).

Referenced by [compute\\_k\\_vx\(\)](#), [compute\\_l2\\_vx\(\)](#), [compute\\_vx\\_k\(\)](#), and [iZm\\_construct\\_vx\\_base\(\)](#).

#### 13.25.3.2 s\_primes\_count

```
int s_primes_count = sizeof(s_primes) / sizeof(int) [static]
```

Number of entries available in [s\\_primes](#).

Definition at line 13 of file [iZ\\_toolkit.c](#).

Referenced by [iZm\\_construct\\_vx\\_base\(\)](#).

## 13.26 iZ\_toolkit.c

[Go to the documentation of this file.](#)

```
00001
00006
00007 #include <iZ_api.h>
00008
00010 static const int s_primes[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
00011                               43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97};
00013 static int s_primes_count = sizeof(s_primes) / sizeof(int);
00014
00024 inline uint64_t iZ(uint64_t x, int i)
00025 {
00026     return 6 * x + i;
00027 }
00028
00037 void iZ_mpz(mpz_t z, mpz_t x, int i)
00038 {
00039     mpz_mul_ui(z, x, 6); // z = 6 * x
00040
00041     if (i > 0)
00042         mpz_add_ui(z, z, i); // z = z + i
00043     else
00044         mpz_sub_ui(z, z, -i); // z = z - i
00045 }
00046
00052 static int compute_k_vx(IZM *iZm)
00053 {
00054     int k = 0;
00055     while (iZm->vx % s_primes[k + 2] == 0)
00056     {
00057         k++;
00058     }
00059     return k;
00060 }
00061
00070 void process_iZ_bitmaps(UI64_ARRAY *primes, BITMAP *x5, BITMAP *x7, uint64_t x_limit)
00071 {
00072     uint64_t root_limit = sqrt(6 * x_limit) + 1;
00073 }
```

```

00074 // Iterate through x values in range 0 < x < x_n
00075 for (uint64_t x = 1; x < x_limit; x++)
00076 {
00077     // if x5[x], implying it's iZ- prime
00078     if (bitmap_get_bit(x5, x))
00079     {
00080         uint64_t p = iZ(x, -1); // compute p = iZ(x, -1)
00081         ui64_push(primes, p); // add p to primes
00082
00083         // if p is root prime, mark its multiples in x5, x7
00084         if (p < root_limit)
00085         {
00086             bitmap_clear_steps_simd(x5, p, p * x + x, x_limit);
00087             bitmap_clear_steps_simd(x7, p, p * x - x, x_limit);
00088         }
00089     }
00090
00091     // Do the same if x7[x], inverting the xp signs
00092     if (bitmap_get_bit(x7, x))
00093     {
00094         uint64_t p = iZ(x, 1);
00095         ui64_push(primes, p);
00096
00097         if (p < root_limit)
00098         {
00099             bitmap_clear_steps_simd(x5, p, p * x - x, x_limit);
00100             bitmap_clear_steps_simd(x7, p, p * x + x, x_limit);
00101         }
00102     }
00103 }
00104 }
00105
00112 void get_root_primes(UI64_ARRAY *primes, uint64_t limit)
00113 {
00114     // Add 2, 3 to primes
00115     ui64_push(primes, 2);
00116     ui64_push(primes, 3);
00117
00118     // Calculate x_n, max x value in iZ space for given n
00119     uint64_t x_n = limit / 6 + 1;
00120
00121     // Create bitmap X-Arrays x5, x7, each of size x_n + 1 bits
00122     BITMAP *x5 = bitmap_init(x_n + 1, 1);
00123     BITMAP *x7 = bitmap_init(x_n + 1, 1);
00124
00125     // Memory allocation failed, check logs
00126     if (!x5 || !x7)
00127     {
00128         ui64_free(&primes);
00129         return;
00130     }
00131
00132     // Sieve logic: mark composites and collect primes up to limit
00133     process_iZ_bitmaps(primes, x5, x7, x_n);
00134
00135     // Cleanup: free memory of x5, x7
00136     bitmap_free(&x5);
00137     bitmap_free(&x7);
00138
00139     return;
00140 }
00141
00153 int check_primality(mpz_t n, int rounds)
00154 {
00155     // GMP's mpz_probab_prime_p returns:
00156     // 0 if n is composite,
00157     // 1 if n is probably prime (without being certain),
00158     // 2 if n is definitely prime (only for small n).
00159     int is_prime = mpz_probab_prime_p(n, rounds);
00160     // ToDo: implement Baillie-PSW primality test for a deterministic alternative to Miller-Rabin, which would return a
00161     // boolean result instead of a probability.
00162     return is_prime;
00163 }
00164
00165 // =====
00166 // * iZm structure:
00167 // =====
00168
00184 IZM *iZm_init(size_t vx)
00185 {
00186     // validate vx
00187     assert(vx >= 35 && "vx must be at least 35.");
00188
00189     IZM *iZm = malloc(sizeof(IZM));
00190     if (!iZm)
00191     {

```

```

00192     log_error("Memory allocation failed for iZm.");
00193     return NULL;
00194 }
00195
00196 iZm->vx = vx;
00197 // get root primes for deterministic sieving
00198 iZm->root_primes = SiZ(vx);
00199 // iZm->root_primes = root_limit < pow(10, 7) ? SiZ(root_limit) : SiZm(root_limit);
00200 if (!iZm->root_primes)
00201 {
00202     log_error("Root primes generation failed for iZm.");
00203     free(iZm);
00204     return NULL;
00205 }
00206
00207 iZm->k_vx = compute_k_vx(iZm);
00208
00209 // initialize base bitmaps
00210 iZm->base_x5 = bitmap_init(vx + 10, 1);
00211 iZm->base_x7 = bitmap_init(vx + 10, 1);
00212 if (!iZm->base_x5 || !iZm->base_x7)
00213 {
00214     log_error("Bitmap initialization failed for iZm base segment.");
00215     iZm_free(&iZm);
00216     ui64_free(&iZm->root_primes);
00217     return NULL;
00218 }
00219
00220 // construct pre-sieved base_x5, base_x7 bitmaps
00221 iZm_construct_vx_base(vx, iZm->base_x5, iZm->base_x7);
00222
00223 return iZm;
00224 }
00225
00232 IZM *iZm_clone(IZM *src)
00233 {
00234     assert(src && "Invalid source IZM structure for cloning.");
00235
00236     IZM *clone = malloc(sizeof(IZM));
00237     assert(clone && "Memory allocation failed for iZm clone.");
00238
00239     clone->vx = src->vx;
00240     clone->k_vx = src->k_vx;
00241     clone->root_primes = ui64_init(src->root_primes->capacity);
00242     if (clone->root_primes == NULL)
00243     {
00244         log_error("Memory allocation failed for root primes in iZm clone.");
00245         iZm_free(&clone);
00246         return NULL;
00247     }
00248     // copy root primes
00249     memcpy(clone->root_primes->array, src->root_primes->array, src->root_primes->count * sizeof(uint64_t));
00250     clone->root_primes->count = src->root_primes->count;
00251
00252     // clone base bitmaps
00253     clone->base_x5 = bitmap_clone(src->base_x5);
00254     clone->base_x7 = bitmap_clone(src->base_x7);
00255
00256     return clone;
00257 }
00258
00266 void iZm_free(IZM **iZm)
00267 {
00268     if (iZm == NULL || *iZm == NULL)
00269         return;
00270
00271     // Free root primes object allocated in iZm_init
00272     ui64_free(&(*iZm)->root_primes);
00273     bitmap_free(&(*iZm)->base_x5);
00274     bitmap_free(&(*iZm)->base_x7);
00275     free(*iZm);
00276     *iZm = NULL;
00277 }
00278
00286 uint64_t compute_vx_k(int k)
00287 {
00288     uint64_t vx = 1;
00289     k += 2; // adjust k to account for skipping 2 and 3
00290     int i = 2; // start from prime 5
00291
00292     while (i < k && s_primes[i] * vx < UINT64_MAX)
00293     {
00294         vx *= s_primes[i++];
00295     }
00296
00297     return vx;
00298 }

```

```

00299
00306 uint64_t compute_l2_vx(uint64_t n)
00307 {
00308     uint64_t l2 = get_cpu_L2_cache_size_bits();
00309     uint64_t x_n = n / 6;
00310     uint64_t vx = 35; // minimum useful vx size
00311     int k = 4;        // pointing to prime 11 in s_primes
00312
00313     // while vx * k-th prime doesn't exceed x_n and l2 cache size
00314     while (vx * s_primes[k] < MIN(l2, x_n))
00315     {
00316         vx *= s_primes[k];
00317         k++;
00318     }
00319
00320     return vx;
00321 }
00322
00331 void compute_max_vx(mpz_t vx, int bit_size)
00332 {
00333     // get some primes to compute the primorial vx
00334     UI64_ARRAY *primes = SIZ(10000);
00335     assert(primes && "Failed to generate primes");
00336
00337     int i = 2; // to skip 2, 3
00338     mpz_set_ui(vx, primes->array[i]); // set vx = 5
00339
00340     // while vx * primes->array[i] < 2^bit_size
00341     while (mpz_sizeinbase(vx, 2) < (size_t)bit_size)
00342     {
00343         i++;
00344         mpz_mul_ui(vx, vx, primes->array[i]);
00345     }
00346
00347     mpz_div_ui(vx, vx, primes->array[i]); // divide by the last prime
00348 }
00349
00363 void iZm_construct_vx_base(uint64_t vx, BITMAP *base_x5, BITMAP *base_x7)
00364 {
00365     assert(base_x5 && base_x7 && "Invalid bitmaps passed to iZm_construct_vx_base.");
00366
00367     bitmap_set_all(base_x5);
00368     bitmap_set_all(base_x7);
00369     bitmap_clear_bit(base_x5, 0); // clear the 0th bit
00370     bitmap_clear_bit(base_x7, 0); // clear the 0th bit
00371
00372     // iterate through small primes that divide vx
00373     for (int i = 2; i < s_primes_count; i++)
00374     {
00375         int p = s_primes[i];
00376         if (vx % p != 0)
00377             continue;
00378
00379         int ip = (p % 6 == 1) ? 1 : -1;
00380         int xp = (p + 1) / 6;
00381         // clear composites in iZ- and iZ+
00382         if (ip == -1)
00383         {
00384             bitmap_clear_bit(base_x5, xp); // clear the prime itself
00385             bitmap_clear_steps_simd(base_x5, p, p * xp + xp, vx); // mark composites in iZ-
00386             bitmap_clear_steps_simd(base_x7, p, p * xp - xp, vx); // mark composites in iZ+
00387         }
00388         else
00389         {
00390             bitmap_clear_bit(base_x7, xp); // clear the prime itself
00391             bitmap_clear_steps_simd(base_x5, p, p * xp - xp, vx); // mark composites in iZ-
00392             bitmap_clear_steps_simd(base_x7, p, p * xp + xp, vx); // mark composites in iZ+
00393         }
00394     }
00395 }
00396
00415 uint64_t iZm_solve_for_x0(int m_id, uint64_t p, uint64_t vx, uint64_t y)
00416 {
00417     uint64_t xp = (p + 1) / 6;
00418     int ip = (p % 6 == 1) ? 1 : -1;
00419
00420     // Immediate solution check for y = 0
00421     if (y == 0)
00422     {
00423         // compute p * xp + m_id * ip * xp
00424         return p * xp + m_id * ip * xp;
00425     }
00426
00427     // Normalize x_p to x_p if p_id = m_id, else to p - x_p
00428     xp = (m_id == ip) ? xp : p - xp;
00429     uint64_t yvx = vx * y;
00430

```

```

00431 // Compute the first composite hit of p in the yth vx segment in iZm index space
00432 uint64_t x = p < vx ? p - (yvx - xp) % p : (yvx - xp) % p;
00433 return x;
00434 }
00435
00449 uint64_t iZm_solve_for_x0_mpz(int m_id, uint64_t p, uint64_t vx, mpz_t y)
00450 {
00451     mpz_t tmp;
00452     mpz_init(tmp);
00453
00454     // 1. Normalize x_p to x_p if p_id = m_id, else to p - x_p
00455     uint64_t xp = (p + 1) / 6;
00456     int ip = (p % 6 == 1) ? 1 : -1;
00457     xp = (m_id == ip) ? xp : p - xp;
00458
00459     mpz_mul_ui(tmp, y, vx);
00460     mpz_sub_ui(tmp, tmp, xp);
00461     mpz_mod_ui(tmp, tmp, p);
00462
00463     uint64_t x = p < vx ? p - mpz_get_ui(tmp) : mpz_get_ui(tmp);
00464
00465     mpz_clear(tmp);
00466     return x;
00467 }
00468
00490 int64_t iZm_solve_for_y0(int m_id, uint64_t p, uint64_t vx, uint64_t x)
00491 {
00492     // No solution check, if vx and p are not coprime
00493     if (gcd(vx, p) != 1)
00494     {
00495         printf("There's no solution for the given parameters\n");
00496         return -1; // No solution can be found
00497     }
00498
00499     uint64_t xp = (p + 1) / 6;
00500     int ip = (p % 6 == 1) ? 1 : -1;
00501     xp = (m_id == ip) ? xp : p - xp;
00502
00503     // Immediate solution check
00504     if (x % p == xp)
00505         return 0;
00506
00507     // Compute delta
00508     uint64_t x_mod_p = x % p;
00509     uint64_t delta = (xp + p - x_mod_p) % p;
00510
00511     // Find modular inverse
00512     uint64_t vx_inv = modular_inverse(vx % p, p);
00513
00514     // Compute and return y
00515     uint64_t y = (delta * vx_inv) % p;
00516     return y;
00517 }
00518
00519 // =====
00520 // * VX_SEG structure:
00521 // =====
00522
00529 static int vx_set_base_values(VX_SEG *vx_obj, char *y_str)
00530 {
00531
00532     // assert y_str is numeric
00533     mpz_t y_tmp;
00534     mpz_init(y_tmp);
00535     if (mpz_set_str(y_tmp, y_str, 10) != 0)
00536     {
00537         log_error("Invalid numeric string for y in vx_set_base_values\n");
00538         mpz_clear(y_tmp);
00539         return 0;
00540     }
00541
00542     mpz_init_set(vx_obj->y, y_tmp);
00543     mpz_clear(y_tmp);
00544
00545     // Compute yvx = y * vx
00546     mpz_init(vx_obj->yvx);
00547     mpz_mul_ui(vx_obj->yvx, vx_obj->y, vx_obj->vx);
00548
00549     // Compute root_limit = sqrt(iZ(vx * (y+1), 1))
00550     mpz_init(vx_obj->root_limit);
00551     mpz_add_ui(vx_obj->root_limit, vx_obj->yvx, vx_obj->vx);
00552     iZ_mpz(vx_obj->root_limit, vx_obj->root_limit, 1);
00553     mpz_sqrt(vx_obj->root_limit, vx_obj->root_limit);
00554
00555     // Set is_large_limit to determine if probabilistic primality test is needed
00556     // if root_limit > vx
00557     vx_obj->is_large_limit = mpz_cmp_ui(vx_obj->root_limit, vx_obj->vx) > 0;

```

```

00558     return 1;
00559 }
00560
00566 static void vx_det_sieve(IZM *iZm, VX_SEG *vx_obj)
00567 {
00568     assert(iZm && "iZm is NULL in vx_det_sieve");
00569     assert(vx_obj && "vx_obj is NULL in vx_det_sieve");
00570
00571     int vx = vx_obj->vx; // segment size
00572
00573     // * Deterministic Sieve: Mark composites of primes < vx in x5, x7
00574     int start_x = vx_obj->start_x;
00575     int end_x = vx_obj->end_x;
00576     int k = 2 + iZm->k_vx; // skip 2, 3 and pre-sieved k_vx primes
00577     UI64_ARRAY *root_primes = iZm->root_primes;
00578
00579     // if y < 2^64, use iZm_solve_for_x0 version for efficiency
00580     if (mpz_sizeinbase(vx_obj->y, 2) <= 64)
00581     {
00582         uint64_t y = mpz_get_ui(vx_obj->y);
00583         uint64_t root_limit = mpz_get_ui(vx_obj->root_limit);
00584
00585         // Iterate through root primes, skipping the first k pre-sieved primes
00586         for (int i = k; i < root_primes->count; i++)
00587         {
00588             uint64_t p = root_primes->array[i];
00589
00590             if (p > root_limit)
00591                 break;
00592
00593             // Mark composites of p in x5 and x7
00594             bitmap_clear_steps_simd(vx_obj->x5, p, iZm_solve_for_x0(-1, p, vx, y), end_x);
00595             bitmap_clear_steps_simd(vx_obj->x7, p, iZm_solve_for_x0(1, p, vx, y), end_x);
00596
00597             vx_obj->bit_ops += (2 * end_x) / p; // approximate number of bit operations
00598         }
00599     }
00600     else
00601     {
00602         // the same as above but using iZm_solve_for_x0_mpz version
00603         for (int i = k; i < root_primes->count; i++)
00604         {
00605             int p = root_primes->array[i];
00606
00607             bitmap_clear_steps_simd(vx_obj->x5, p, iZm_solve_for_x0_mpz(-1, p, vx, vx_obj->y), end_x);
00608             bitmap_clear_steps_simd(vx_obj->x7, p, iZm_solve_for_x0_mpz(1, p, vx, vx_obj->y), end_x);
00609
00610             vx_obj->bit_ops += (2 * end_x) / p;
00611         }
00612     }
00613
00614     // if no further probabilistic testing is needed,
00615     // count unmarked bits in x5 and x7 as p_count
00616     if (!vx_obj->is_large_limit)
00617     {
00618         vx_obj->p_count = 0;
00619         for (int x = start_x; x <= end_x; x++)
00620         {
00621             if (bitmap_get_bit(vx_obj->x5, x))
00622                 vx_obj->p_count++;
00623             if (bitmap_get_bit(vx_obj->x7, x))
00624                 vx_obj->p_count++;
00625         }
00626     }
00627 }
00628
00633 static void vx_prob_sieve(VX_SEG *vx_obj)
00634 {
00635     // If is_large_limit is false, no need for probabilistic testing
00636     if (!vx_obj->is_large_limit)
00637     {
00638         log_info("vx_obj->is_large_limit is false, no need for probabilistic testing");
00639         return;
00640     }
00641
00642     // Initialize GMP reusable variables p, x_p
00643     mpz_t p, x_p;
00644     mpz_init(p);
00645     mpz_init(x_p);
00646
00647     int r = vx_obj->mr_rounds;
00648     int s = vx_obj->start_x <= 1 ? 1 : vx_obj->start_x;
00649
00650     // Iterate through x values in the range start_x <= x <= end_x
00651     for (int x = s; x <= vx_obj->end_x; x++)
00652     {
00653         // Check if iZ(vx * y + x, -1) is prime, if not, clear x in x5

```



```

00654     if (bitmap_get_bit(vx_obj->x5, x))
00655     {
00656         // Compute x_p = yvx + x
00657         mpz_add_ui(x_p, vx_obj->yvx, x);
00658         iZ_mpz(p, x_p, -1); // Compute p = iZ(x_p, -1)
00659         int is_prime = check_primalty(p, r);
00660         vx_obj->p_test_ops++;
00661
00662         // if is_prime, increment count, else clear composite from x5
00663         if (is_prime)
00664         {
00665             vx_obj->p_count++;
00666         }
00667         else
00668         {
00669             bitmap_clear_bit(vx_obj->x5, x); // Clear composite from x5
00670         }
00671     }
00672
00673     // same for the 6x+1 candidate
00674     if (bitmap_get_bit(vx_obj->x7, x))
00675     {
00676         mpz_add_ui(x_p, vx_obj->yvx, x);
00677         iZ_mpz(p, x_p, 1); // Compute p = iZ(x_p, 1)
00678         int is_prime = check_primalty(p, r);
00679         vx_obj->p_test_ops++;
00680
00681         if (is_prime)
00682         {
00683             vx_obj->p_count++;
00684         }
00685         else
00686         {
00687             bitmap_clear_bit(vx_obj->x7, x); // Clear composite from x7
00688         }
00689     }
00690 }
00691
00692 vx_obj->is_large_limit = 0; // all composites cleared
00693
00694 // Cleanup
00695 mpz_clears(p, x_p, NULL);
00696 }
00697
00713 VX_SEG *vx_init(IZM *iZm, int start_x, int end_x, char *y_str, int mr_rounds)
00714 {
00715     // assert vx > 5 and not a multiple of 2 or 3
00716     assert(iZm && "iZm is NULL in vx_init");
00717     assert(y_str && "y_str is NULL in vx_init");
00718
00719     VX_SEG *vx_obj = malloc(sizeof(VX_SEG));
00720     if (vx_obj == NULL)
00721     {
00722         log_error("Memory allocation failed in vx_init\n");
00723         return NULL;
00724     }
00725
00726     // Initialize struct members
00727     vx_obj->vx = iZm->vx;
00728
00729     // Set base values
00730     if (!vx_set_base_values(vx_obj, y_str))
00731     {
00732         // check logs
00733         free(vx_obj);
00734         return NULL;
00735     }
00736
00737     vx_obj->mr_rounds = (mr_rounds == 0) ? MR_ROUNDS : mr_rounds; // default 25 rounds
00738
00739     vx_obj->start_x = MAX(start_x, 1);
00740     vx_obj->end_x = MIN(end_x, vx_obj->vx);
00741     vx_obj->x5 = bitmap_clone(iZm->base_x5);
00742     vx_obj->x7 = bitmap_clone(iZm->base_x7);
00743     vx_obj->p_count = 0;
00744     vx_obj->p_gaps = NULL;
00745     vx_obj->bit_ops = 0;
00746     vx_obj->p_test_ops = 0;
00747
00748     // perform deterministic sieving to prepare for probabilistic sieving or streaming
00749     vx_det_sieve(iZm, vx_obj);
00750
00751     return vx_obj;
00752 }
00753
00759 void vx_free(VX_SEG **vx_obj)
00760 {

```

```

00761     if (vx_obj == NULL || *vx_obj == NULL)
00762         return;
00763
00764     // clear bitmaps
00765     bitmap_free(&(*vx_obj)->x5);
00766     bitmap_free(&(*vx_obj)->x7);
00767
00768     // clear p_gaps array
00769     ui16_free(&(*vx_obj)->p_gaps);
00770
00771     // clear mpz_t variables
00772     mpz_clears((*vx_obj)->y, (*vx_obj)->yvx, (*vx_obj)->root_limit, NULL);
00773
00774     free(*vx_obj);
00775     *vx_obj = NULL;
00776 }
00777
00787 void vx_collect_p_gaps(VX_SEG *vx_obj)
00788 {
00789     assert(vx_obj && vx_obj->p_count > 0 && "Invalid vx_obj in vx_collect_p_gaps");
00790     assert(mpz_cmp_ui(vx_obj->y, 0) > 0 && "First segment requires special handling in vx_collect_p_gaps");
00791     if (vx_obj->is_large_limit)
00792     {
00793         vx_full_sieve(vx_obj, 0);
00794     }
00795
00796     vx_obj->p_gaps = ui16_init(vx_obj->p_count + 2);
00797     assert(vx_obj->p_gaps && "Memory allocation failed for vx_obj->p_gaps in vx_collect_p_gaps");
00798
00799     // Initialize gap counter
00800     int gap = 0;
00801
00802     // Iterate through x values in the range start_x <= x <= end_x
00803     for (int x = vx_obj->start_x; x <= vx_obj->end_x; x++)
00804     {
00805         // Increment gap by 4 since: iZ(x, -1) - iZ(x-1, 1) = 4
00806         gap += 4;
00807
00808         // Check if iZ(vx * y + x, -1) is prime
00809         if (bitmap_get_bit(vx_obj->x5, x))
00810         {
00811             // Append gap to p_gaps
00812             ui16_push(vx_obj->p_gaps, gap);
00813             gap = 0; // Reset gap
00814         }
00815
00816         // Increment gap by 2
00817         gap += 2;
00818
00819         // Check if iZ(vx * y + x, 1) is prime
00820         if (bitmap_get_bit(vx_obj->x7, x))
00821         {
00822             // Append gap to p_gaps
00823             ui16_push(vx_obj->p_gaps, gap);
00824             gap = 0; // Reset gap
00825         }
00826     }
00827
00828     // append final gap for backward calculations
00829     ui16_push(vx_obj->p_gaps, gap);
00830 }
00831
00847 void vx_full_sieve(VX_SEG *vx_obj, int collect_p_gaps)
00848 {
00849     assert(vx_obj && "vx_obj is NULL in vx_full_sieve");
00850
00851     // If is_large_limit is true, perform probabilistic primality tests
00852     if (vx_obj->is_large_limit)
00853         vx_prob_sieve(vx_obj);
00854
00855     // If collect_p_gaps is true, populate the p_gaps array
00856     if (collect_p_gaps)
00857         vx_collect_p_gaps(vx_obj);
00858 }
00859
00866 void vx_stream(VX_SEG *vx_obj, FILE *output)
00867 {
00868     assert(vx_obj && "vx_obj is NULL in vx_stream");
00869     assert(output && "output stream is NULL in vx_stream");
00870
00871     // Initialize GMP reusable variables p, x_p
00872     mpz_t p, x_p;
00873     mpz_init(p);
00874     mpz_init(x_p);
00875
00876     int r = vx_obj->mr_rounds;
00877

```

```

00878 // Iterate through x values in the range start_x <= x <= end_x
00879 for (int x = vx_obj->start_x; x <= vx_obj->end_x; x++)
00880 {
00881     // Check if iZ(vx * y + x, -1) is prime, if not, clear x in x5
00882     if (bitmap_get_bit(vx_obj->x5, x))
00883     {
00884         // Compute x_p = yvx + x
00885         mpz_add_ui(x_p, vx_obj->yvx, x);
00886         iZ_mpz(p, x_p, -1); // Compute p = iZ(x_p, -1)
00887         int is_prime = 1;
00888
00889         if (vx_obj->is_large_limit)
00890         {
00891             vx_obj->p_test_ops++;
00892             is_prime = check_primal(p, r);
00893         }
00894
00895         if (is_prime)
00896         {
00897             if (vx_obj->is_large_limit)
00898             {
00899                 vx_obj->p_count++; // otherwise already counted in det_sieve
00900             }
00901             gmp_printf(output, "%Zd ", p);
00902         }
00903         else
00904         {
00905             bitmap_clear_bit(vx_obj->x5, x); // Clear composite from x5
00906         }
00907     }
00908
00909     // test iZ(vx * y + x, 1) for primality
00910     if (bitmap_get_bit(vx_obj->x7, x))
00911     {
00912         mpz_add_ui(x_p, vx_obj->yvx, x);
00913         iZ_mpz(p, x_p, 1); // Compute p = iZ(x_p, 1)
00914         int is_prime = 1;
00915
00916         if (vx_obj->is_large_limit)
00917         {
00918             vx_obj->p_test_ops++;
00919             is_prime = check_primal(p, r);
00920         }
00921
00922         if (is_prime)
00923         {
00924             if (vx_obj->is_large_limit)
00925             {
00926                 vx_obj->p_count++;
00927             }
00928             gmp_printf(output, "%Zd ", p);
00929         }
00930         else
00931         {
00932             bitmap_clear_bit(vx_obj->x7, x); // Clear composite from x7
00933         }
00934     }
00935 }
00936
00937 mpz_clears(p, x_p, NULL);
00938 }
00939
00940 // =====
00941 // * IZM_RANGE_INFO structure:
00942 // =====
00943
00944 IZM_RANGE_INFO range_info_init(INPUT_SIEVE_RANGE *input_range, int vx)
00945 {
00946     IZM_RANGE_INFO info = {0};
00947     info.vx = vx;
00948     info.y_range = -1;
00949
00950     mpz_inits(&info.Zs, &info.Ze, &info.Xs, &info.Xe, &info.Ys, &info.Ye, NULL);
00951
00952     if (!input_range || !input_range->start || vx < 35 || mpz_set_str(&info.Zs, input_range->start, 10) != 0)
00953     {
00954         log_error("range_info_init: invalid input.");
00955         return info;
00956     }
00957
00958     if (input_range->range == 0)
00959     {
00960         mpz_set(&info.Ze, &info.Zs);
00961     }
00962     else
00963     {
00964         // inclusive upper bound for range size semantics: [start, start + range - 1]

```

```

00965     mpz_add_ui(info.Ze, info.Zs, input_range->range - 1);
00966 }
00967
00968 mpz_fdiv_q_ui(info.Xs, info.Zs, 6);
00969 mpz_fdiv_q_ui(info.Xe, info.Ze, 6);
00970 mpz_fdiv_q_ui(info.Ys, info.Xs, vx);
00971 mpz_fdiv_q_ui(info.Ye, info.Xe, vx);
00972
00973 mpz_t y_delta;
00974 mpz_init(y_delta);
00975 mpz_sub(y_delta, info.Ye, info.Ys);
00976
00977 if (mpz_sgn(y_delta) < 0 || !mpz_fits_sint_p(y_delta))
00978 {
00979     log_error("range_info_init: computed y-range is out of supported bounds.");
00980     mpz_clear(y_delta);
00981     return info;
00982 }
00983
00984 info.y_range = (int)mpz_get_si(y_delta);
00985 mpz_clear(y_delta);
00986
00987 return info;
00988 }
00989 void range_info_free(IZM_RANGE_INFO *info)
00990 {
00991     if (info == NULL)
00992         return;
00993
00994     mpz_clears(info->Zs, info->Ze, info->Xs, info->Xe, info->Ys, info->Ye, NULL);
00995 }
00996
00997 // =====
00998 // * iZm Search primes routines:
00999 // =====
01000
01017 int vx_search_prime(mpz_t p, int m_id, int vx, int bit_size)
01018 {
01019     // assert p is not NULL
01020     assert(p && "p cannot be NULL in vx_search_prime.");
01021
01022     bit_size = MAX(bit_size, 10);
01023
01024     // set m_id randomly if not provided
01025     if (m_id != -1 && m_id != 1)
01026         m_id = (rand() % 2) ? 1 : -1;
01027
01028     gmp_randstate_t state;
01029     gmp_randinit_default(state);
01030     gmp_seed_randstate(state);
01031
01032     UI64_ARRAY *root_primes = SiZm(vx);
01033     if (!root_primes)
01034     {
01035         log_error("Failed to initialize root primes in vx_search_prime");
01036         gmp_randclear(state);
01037         return 0;
01038     }
01039
01040     mpz_t z, x_z, y, yvx;
01041     mpz_init(z);
01042     mpz_init(x_z);
01043     mpz_init(y);
01044     mpz_init(yvx);
01045
01046     mpz_urandomb(y, state, bit_size);
01047     mpz_fdiv_q_ui(y, y, 6 * vx);
01048     mpz_mul_ui(yvx, y, vx); // compute yvx = vx * y
01049
01050     int found = 0;
01051     while (!found)
01052     {
01053         BITMAP *bitmap = bitmap_init(vx + 10, 1);
01054
01055         // sieve the bitmap with root primes skipping 2 and 3
01056         for (int i = 2; i < root_primes->count; i++)
01057         {
01058             uint64_t q = root_primes->array[i];
01059             // mark composites of q in the bitmap
01060             bitmap_clear_steps(bitmap, q, iZm_solve_for_x0_mpz(m_id, q, vx, y), vx);
01061         }
01062
01063         int random_x = rand() % (vx / 2); // random x < vx/2
01064         for (int x = random_x; x < vx; x++)
01065         {
01066             // check if z is prime candidate in bitmap
01067             if (bitmap_get_bit(bitmap, x))

```

```

01068     {
01069         // compute x_z = vx * y + x
01070         mpz_add_ui(x_z, yvx, x);
01071         iZ_mpz(z, x_z, m_id);
01072         // check if z is prime
01073         found = check_primalty(z, MR_ROUNDS);
01074
01075         // if z is prime, set p = z
01076         if (found)
01077         {
01078             mpz_set(p, z);
01079             break;
01080         }
01081     }
01082 }
01083
01084 // cleanup
01085 bitmap_free(&bitmap);
01086
01087 // increment y by 1 for next vx segment
01088 mpz_add_ui(y, y, 1);
01089 mpz_add_ui(yvx, yvx, vx);
01090 }
01091
01092 ui64_free(&root_primes);
01093 mpz_clears(z, x_z, y, yvx, NULL);
01094 gmp_randclear(state);
01095
01096 return found;
01097 }
01098
01114 int vy_search_prime(mpz_t p, int m_id, mpz_t vx)
01115 {
01116     assert(p && vx && "Input parameters cannot be NULL");
01117
01118     int found = 0; // flag to indicate if a prime was found
01119
01120     // set m_id randomly if not provided
01121     if (m_id != -1 && m_id != 1)
01122         m_id = (rand() % 2) ? 1 : -1;
01123
01124     gmp_randstate_t state;
01125     gmp_randinit_default(state);
01126     gmp_seed_randstate(state); // seed random state
01127
01128     mpz_t z, g;
01129     mpz_init(z);
01130     mpz_init(g);
01131
01132     // set random x value in the range of vx
01133     mpz_urandomm(z, state, vx);
01134     // compute z = 6 * z + i
01135     iZ_mpz(z, z, m_id);
01136
01137     // search for x value such that gcd(iZ(x, m_id), z) = 1
01138     for (;;)
01139     {
01140         // increment z by 6 to advance x by 1
01141         mpz_add_ui(z, z, 6);
01142
01143         // Compute g = gcd(vx, z)
01144         mpz_gcd(g, vx, z);
01145
01146         // break if g = 1,
01147         // implying current x value can yield primes of the form iZ(x + vx * y, p_id)
01148         if (mpz_cmp_ui(g, 1) == 0)
01149             break;
01150     }
01151
01152     // set g = 6 * vx to use as a step
01153     mpz_mul_ui(g, vx, 6);
01154     // add z by rand * g
01155     int rand_steps = rand() % 100;
01156     mpz_addmul_ui(z, g, rand_steps);
01157
01158     while (!found)
01159     {
01160         // increment z by 6 * vx to advance y by 1
01161         mpz_add(z, z, g);
01162
01163         // check if z is prime
01164         found = check_primalty(z, MR_ROUNDS);
01165
01166         // if z is prime, set p = z
01167         if (found)
01168             mpz_set(p, z);
01169     }

```

```

01170
01171 // cleanup
01172 gmp_randclear(state);
01173 mpz_clears(z, g, NULL);
01174
01175 return found;
01176 }

```

## 13.27 src/toolkit/logger.c File Reference

Logging module.

### 13.27.1 Detailed Description

Logging module.

This module provides functions for logging messages at different levels.

Definition in file [logger.c](#).

## 13.28 logger.c

[Go to the documentation of this file.](#)

```

00001
00008 #ifdef ENABLE_LOGGING
00009
00010 #include "logger.h"
00011 #include <stdio.h>
00012 #include <time.h>
00013 #include <stdarg.h>
00014 #include <string.h>
00015 #include <sys/stat.h>
00016 #include <pthread.h>
00017 #include <stdlib.h>
00018 #include <errno.h>
00019
00020 // Initialize the mutex for thread-safe logging
00021 static pthread_mutex_t log_mutex = PTHREAD_MUTEX_INITIALIZER;
00022 static LogLevel current_log_level = LOG_DEBUG; // Default log level
00023
00030 const char *log_level_to_string(LogLevel level)
00031 {
00032     switch (level)
00033     {
00034     case LOG_DEBUG:
00035         return "DEBUG";
00036     case LOG_INFO:
00037         return "INFO";
00038     case LOG_WARNING:
00039         return "WARNING";
00040     case LOG_ERROR:
00041         return "ERROR";
00042     case LOG_FATAL:
00043         return "FATAL";
00044     default:
00045         return "UNKNOWN";
00046     }
00047 }
00048
00055 void get_current_timestamp(char *buffer, size_t size)
00056 {
00057     time_t now = time(NULL);
00058     struct tm *t = localtime(&now);
00059     strftime(buffer, size, "%Y-%m-%d %H:%M:%S", t);
00060 }
00061
00072 void log_rotate(const char *log_file, size_t max_size)
00073 {
00074     struct stat st;

```

```

00075     if (stat(log_file, &st) == 0 && st.st_size >= (off_t)max_size) // Cast max_size to off_t to match types
00076     {
00077         char old_log_file[256];
00078         for (int i = 5; i > 0; --i)
00079         {
00080             snprintf(old_log_file, sizeof(old_log_file), "%s.%d", log_file, i);
00081             if (i == 5)
00082             {
00083                 remove(old_log_file); // Remove the oldest log file
00084             }
00085             else
00086             {
00087                 char prev_log_file[256];
00088                 snprintf(prev_log_file, sizeof(prev_log_file), "%s.%d", log_file, i + 1);
00089                 rename(old_log_file, prev_log_file);
00090             }
00091         }
00092
00093         char new_log_file[256];
00094         snprintf(new_log_file, sizeof(new_log_file), "%s.1", log_file);
00095         rename(log_file, new_log_file);
00096     }
00097 }
00098
00106 void log_init(const char *log_file)
00107 {
00108     struct stat st = {0};
00109     if (stat(LOG_DIR, &st) == -1)
00110     {
00111         mkdir(LOG_DIR, 0700); // Create log directory if it doesn't exist
00112     }
00113
00114     log_rotate(log_file, LOG_MAX_SIZE); // Rotate logs if needed
00115 }
00116
00120 void log_shutdown(void)
00121 {
00122     pthread_mutex_destroy(&log_mutex);
00123 }
00124
00130 void log_set_log_level(LogLevel level)
00131 {
00132     current_log_level = level;
00133 }
00134
00145 static void log_message_va(LogLevel level, const char *format, va_list args)
00146 {
00147     if (level < current_log_level)
00148         return; // Don't log messages below the current log level
00149
00150     char message[1024];
00151     // Format string comes from a variadic function parameter - suppress warning
00152     #pragma clang diagnostic push
00153     #pragma clang diagnostic ignored "-Wformat-nonliteral"
00154     vsnprintf(message, sizeof(message), format, args);
00155     #pragma clang diagnostic pop
00156
00157     char timestamp[20];
00158     get_current_timestamp(timestamp, sizeof(timestamp));
00159
00160     pthread_mutex_lock(&log_mutex);
00161
00162     FILE *file = fopen(LOG_FILE, "a");
00163     if (!file)
00164     {
00165         perror("Failed to open log file");
00166         pthread_mutex_unlock(&log_mutex); // Ensure mutex is unlocked
00167         return;
00168     }
00169
00170     fprintf(file, "[%s] [%s] %s\n", timestamp, log_level_to_string(level), message);
00171     fclose(file);
00172
00173     pthread_mutex_unlock(&log_mutex);
00174 }
00175
00185 void log_message(LogLevel level, const char *format, ...)
00186 {
00187     va_list args;
00188     va_start(args, format);
00189     log_message_va(level, format, args);
00190     va_end(args);
00191 }
00192
00201 void log_message_extended(LogLevel level, const char *file_name, int line_number, const char *format, ...)
00202 {
00203     if (level < current_log_level)

```

```

00204     return; // Don't log messages below the current log level
00205
00206     va_list args;
00207     va_start(args, format);
00208     char message[1024];
00209     // Format string comes from a variadic function parameter - suppress warning
00210     #pragma clang diagnostic push
00211     #pragma clang diagnostic ignored "-Wformat-nonliteral"
00212     vsnprintf(message, sizeof(message), format, args);
00213     #pragma clang diagnostic pop
00214     va_end(args);
00215
00216     char timestamp[20];
00217     get_current_timestamp(timestamp, sizeof(timestamp));
00218
00219     pthread_mutex_lock(&log_mutex);
00220
00221     FILE *file = fopen(LOG_FILE, "a");
00222     if (!file)
00223     {
00224         perror("Failed to open log file");
00225         pthread_mutex_unlock(&log_mutex);
00226         return;
00227     }
00228
00229     fprintf(file, "[%s] [%s] %s (File: %s, Line: %d)\n",
00230             timestamp, log_level_to_string(level), message, file_name, line_number);
00231     fclose(file);
00232
00233     pthread_mutex_unlock(&log_mutex);
00234 }
00235
00243 void log_console(const char *format, ...)
00244 {
00245     va_list args;
00246     va_start(args, format);
00247     char message[1024];
00248     // Format string comes from a variadic function parameter - suppress warning
00249     #pragma clang diagnostic push
00250     #pragma clang diagnostic ignored "-Wformat-nonliteral"
00251     vsnprintf(message, sizeof(message), format, args);
00252     #pragma clang diagnostic pop
00253     va_end(args);
00254
00255     char timestamp[20];
00256     get_current_timestamp(timestamp, sizeof(timestamp));
00257
00258     // Print the message with a timestamp to the console
00259     printf("[%s] %s\n", timestamp, message);
00260 }
00261
00267 void log_debug(const char *format, ...)
00268 {
00269     va_list args;
00270     va_start(args, format);
00271     log_message_va(LOG_DEBUG, format, args);
00272     va_end(args);
00273 }
00274
00280 void log_info(const char *format, ...)
00281 {
00282     va_list args;
00283     va_start(args, format);
00284     log_message_va(LOG_INFO, format, args);
00285     va_end(args);
00286 }
00287
00293 void log_warn(const char *format, ...)
00294 {
00295     va_list args;
00296     va_start(args, format);
00297     log_message_va(LOG_WARNING, format, args);
00298     va_end(args);
00299 }
00300
00306 void log_error(const char *format, ...)
00307 {
00308     va_list args;
00309     va_start(args, format);
00310     log_message_va(LOG_ERROR, format, args);
00311     va_end(args);
00312 }
00313
00319 void log_fatal(const char *format, ...)
00320 {
00321     va_list args;
00322     va_start(args, format);

```



```
00323     log_message_va(LOG_FATAL, format, args);
00324     va_end(args);
00325 }
00326
00327 #endif // ENABLE_LOGGING
```

## 13.29 src/toolkit/print\_utils.c File Reference

Print formatting helpers for tests, benchmarks, and CLI output.

```
#include <utils.h>
```

### Functions

- void [print\\_line](#) (int length, char fill\_char)  
Print a repeated-character horizontal line.
- void [print\\_centered\\_text](#) (const char \*text, int line\_length, char fill\_char)  
Print text centered inside a padded line.
- void [print\\_sha256\\_hash](#) (const unsigned char \*hash)  
Print a SHA-256 digest as hex to stdout.
- void [print\\_test\\_module\\_header](#) (char \*module\_name)  
Print a test-suite header banner for a module.
- void [print\\_test\\_table\\_header](#) (void)  
Print the generic test runner header.
- void [print\\_test\\_module\\_result](#) (int result, int test\_id, const char \*unit\_name, const char \*format,...)  
Print a single test-row result.
- void [print\\_test\\_summary](#) (char \*module\_name, int passed, int failed, int verbose)  
Print module-level test summary.
- void [print\\_test\\_fn\\_header](#) (char \*fn\_name)  
Print a formatted header for a test function.

### 13.29.1 Detailed Description

Print formatting helpers for tests, benchmarks, and CLI output.

Definition in file [print\\_utils.c](#).

## 13.30 print\_utils.c

[Go to the documentation of this file.](#)

```

00001
00006
00007 #include <utils.h>
00008
00014 void print_line(int length, char fill_char)
00015 {
00016     if (fill_char == '\0')
00017         fill_char = '-';
00018
00019     for (int i = 0; i < length; ++i)
00020         printf("%c", fill_char);
00021     printf("\n");
00022 }
00023
00030 void print_centered_text(const char *text, int line_length, char fill_char)
00031 {
00032     if (text == NULL)
00033         text = "";
00034
00035     int text_length = (int)strlen(text);
00036     if (text_length >= line_length)
00037     {
00038         printf("%s\n", text);
00039         return;
00040     }
00041
00042     int left_padding = (line_length - text_length) / 2;
00043     int right_padding = line_length - text_length - left_padding;
00044
00045     for (int i = 0; i < left_padding; ++i)
00046         printf("%c", fill_char);
00047     printf("%s", text);
00048     for (int i = 0; i < right_padding; ++i)
00049         printf("%c", fill_char);
00050     printf("\n");
00051 }
00052
00058 void print_sha256_hash(const unsigned char *hash)
00059 {
00060     for (int i = 0; i < SHA256_DIGEST_LENGTH; i++)
00061         printf("%02x", hash[i]);
00062
00063     printf("\n");
00064 }
00065
00070 void print_test_module_header(char *module_name)
00071 {
00072     print_line(60, '*');
00073     printf("** %s MODULE TEST SUITE\n", module_name);
00074     print_line(60, '*');
00075 }
00076
00080 void print_test_table_header(void)
00081 {
00082     print_line(92, '-');
00083     printf("[%s] %-30s %s %-66s\n", "ID", "Unit Name", "Result", "Details");
00084     print_line(92, '-');
00085 }
00086
00096 void print_test_module_result(int result, int test_id, const char *unit_name, const char *format, ...)
00097 {
00098     va_list args;
00099     va_start(args, format);
00100
00101     char message[1024];
00102     #pragma clang diagnostic push
00103     #pragma clang diagnostic ignored "-Wformat-nonliteral"
00104     vsnprintf(message, sizeof(message), format, args);
00105     #pragma clang diagnostic pop
00106     va_end(args);
00107
00108     if (result)
00109     {
00110         printf("[%02d] %-30s [PASS] %s\n", test_id, unit_name, message);
00111     }
00112     else
00113     {
00114         printf("[%02d] %-30s [FAIL] %s\n", test_id, unit_name, message);
00115     }
00116 }
00117
00126 void print_test_summary(char *module_name, int passed, int failed, int verbose)

```

```

00127 {
00128     (void)verbose;
00129
00130     print_line(60, '*');
00131     printf("Results Summary for %s\n", module_name);
00132     print_line(60, '-');
00133     printf("%-32s: %d\n", "Total Tests", passed + failed);
00134     printf("%-32s: %d\n", "Passed", passed);
00135     printf("%-32s: %d\n", "Failed", failed);
00136     print_line(60, '-');
00137     if (failed == 0)
00138     {
00139         printf("[SUCCESS] ALL %s TESTS PASSED!\n", module_name);
00140     }
00141     else
00142     {
00143         printf("[FAILURE] SOME %s TESTS FAILED :\\n", module_name);
00144     }
00145     print_line(60, '*');
00146 }
00147
00148 void print_test_fn_header(char *fn_name)
00149 {
00150     char header[64];
00151     snprintf(header, sizeof(header), " Testing %s ", fn_name);
00152
00153     print_line(60, '*');
00154     print_centered_text(header, 60, '=');
00155     print_line(60, '*');
00156     printf("\n");
00157 }

```

## 13.31 src/toolkit/stopwatch.c File Reference

Stopwatch helpers based on monotonic wall-clock time.

#include <utils.h>

### Functions

- void [sw\\_start](#) (STOPWATCH \*sw)  
Start or restart a stopwatch.
- void [sw\\_stop](#) (STOPWATCH \*sw)  
Stop a running stopwatch, capturing the elapsed time in seconds.
- double [sw\\_elapsed\\_seconds](#) (const STOPWATCH \*sw)  
Return elapsed seconds for the stopwatch.
- double [sw\\_elapsed\\_now\\_seconds](#) (void)  
Capture the current monotonic time in seconds.

### 13.31.1 Detailed Description

Stopwatch helpers based on monotonic wall-clock time.

Definition in file [stopwatch.c](#).

## 13.32 stopwatch.c

[Go to the documentation of this file.](#)

```

00001
00006
00007 #include <utils.h>
00008
00010 static struct timespec sw_now_timespec(void)
00011 {
00012     struct timespec ts;
00013     clock_gettime(CLOCK_MONOTONIC, &ts);
00014     return ts;
00015 }
00016
00017 static double sw_elapsed_between(struct timespec start, struct timespec end)
00018 {
00019     time_t sec = end.tv_sec - start.tv_sec;
00020     long nsec = end.tv_nsec - start.tv_nsec;
00021     if (nsec < 0)
00022     {
00023         sec -= 1;
00024         nsec += 1000000000L;
00025     }
00026     return (double)sec + (double)nsec / 1000000000.0;
00027 }
00029
00030 void sw_start(STOPWATCH *sw)
00031 {
00032     assert(sw && "Invalid stopwatch passed to sw_start.");
00033     sw->start_time = sw_now_timespec();
00034     sw->end_time = sw->start_time;
00035     sw->running = 1;
00036 }
00037
00038 void sw_stop(STOPWATCH *sw)
00039 {
00040     assert(sw && "Invalid stopwatch passed to sw_stop.");
00041     if (!sw->running)
00042         return;
00043     sw->end_time = sw_now_timespec();
00044     sw->running = 0;
00045     sw->elapsed_sec = sw_elapsed_between(sw->start_time, sw->end_time);
00046 }
00047
00048 double sw_elapsed_seconds(const STOPWATCH *sw)
00049 {
00050     assert(sw && "Invalid stopwatch passed to sw_elapsed_seconds.");
00051     if (sw->running)
00052     {
00053         struct timespec now = sw_now_timespec();
00054         return sw_elapsed_between(sw->start_time, now);
00055     }
00056     return sw->elapsed_sec;
00057 }
00058
00059 double sw_elapsed_now_seconds(void)
00060 {
00061     struct timespec now = sw_now_timespec();
00062     return (double)now.tv_sec + (double)now.tv_nsec / 1000000000.0;
00063 }

```

## 13.33 src/toolkit/utils.c File Reference

Implementations for the shared utility layer.

```

#include <utils.h>
#include <ctype.h>

```

### Functions

- int [is\\_numeric\\_str](#) (const char \*str)

- Check if a string is numeric.
- int [parse\\_numeric\\_expr\\_mpz](#) (mpz\_t out, const char \*expr)  
Parse an integer expression into an mpz value.
- int [parse\\_numeric\\_expr\\_u64](#) (const char \*expr, uint64\_t \*out)  
Parse an integer expression into uint64\_t.
- int [parse\\_inclusive\\_range\\_mpz](#) (const char \*range\_expr, mpz\_t lower, mpz\_t upper)  
Parse an inclusive range expression into lower/upper bounds.
- int [create\\_dir](#) (const char \*dir)  
Create a directory if it does not exist.
- uint64\_t [gcd](#) (uint64\_t a, uint64\_t b)  
Compute the greatest common divisor (GCD) of two integers.
- uint64\_t [modular\\_inverse](#) (uint64\_t a, uint64\_t m)  
Compute the modular inverse of a modulo m.
- void [gmp\\_seed\\_randstate](#) (gmp\_randstate\_t state)  
Seed the GMP random state.
- int [get\\_cpu\\_cores\\_count](#) (void)  
Compute the number of CPU cores available.
- int [get\\_cpu\\_L2\\_cache\\_size\\_bits](#) (void)  
Get the CPU L2 cache size in bits.

### 13.33.1 Detailed Description

Implementations for the shared utility layer.

Definition in file [utils.c](#).

## 13.34 utils.c

[Go to the documentation of this file.](#)

```

00001
00006
00007 #include <utils.h>
00008 #include <ctype.h>
00009
00011 static const char *skip_spaces(const char *s)
00012 {
00013     while (*s && isspace((unsigned char)*s))
00014         s++;
00015     return s;
00016 }
00017
00018 static char *dup_trimmed(const char *src)
00019 {
00020     if (src == NULL)
00021         return NULL;
00022
00023     const char *start = skip_spaces(src);
00024     const char *end = start + strlen(start);
00025     while (end > start && isspace((unsigned char)end[-1]))
00026         --end;
00027
00028     size_t len = (size_t)(end - start);
00029     char *out = malloc(len + 1);
00030     if (!out)
00031         return NULL;
00032     memcpy(out, start, len);
00033     out[len] = '\0';
00034     return out;
00035 }
00036
00037 static char *normalize_decimal_token(const char *token)
00038 {

```

```

00039 char *trimmed = dup_trimmed(token);
00040 if (trimmed == NULL)
00041     return NULL;
00042
00043 const char *s = trimmed;
00044 if (*s == '+')
00045     ++s;
00046
00047 if (*s == '\0')
00048 {
00049     free(trimmed);
00050     return NULL;
00051 }
00052
00053 size_t cap = strlen(s) + 1;
00054 char *normalized = malloc(cap);
00055 if (!normalized)
00056 {
00057     free(trimmed);
00058     return NULL;
00059 }
00060
00061 size_t n = 0;
00062 int has_comma = strchr(s, ',') != NULL;
00063
00064 if (!has_comma)
00065 {
00066     for (const char *p = s; *p; ++p)
00067     {
00068         if (*p == ' ')
00069             continue;
00070         if (!isdigit((unsigned char)*p))
00071         {
00072             free(normalized);
00073             free(trimmed);
00074             return NULL;
00075         }
00076         normalized[n++] = *p;
00077     }
00078 }
00079 else
00080 {
00081     const char *seg = s;
00082     int group_idx = 0;
00083     while (1)
00084     {
00085         const char *comma = strchr(seg, ',');
00086         size_t seg_len = comma ? (size_t)(comma - seg) : strlen(seg);
00087         if (seg_len == 0)
00088         {
00089             free(normalized);
00090             free(trimmed);
00091             return NULL;
00092         }
00093
00094         if (group_idx == 0)
00095         {
00096             if (seg_len < 1 || seg_len > 3)
00097             {
00098                 free(normalized);
00099                 free(trimmed);
00100                 return NULL;
00101             }
00102         }
00103         else if (seg_len != 3)
00104         {
00105             free(normalized);
00106             free(trimmed);
00107             return NULL;
00108         }
00109
00110         for (size_t i = 0; i < seg_len; ++i)
00111         {
00112             if (!isdigit((unsigned char)seg[i]))
00113             {
00114                 free(normalized);
00115                 free(trimmed);
00116                 return NULL;
00117             }
00118             normalized[n++] = seg[i];
00119         }
00120
00121         group_idx++;
00122         if (comma == NULL)
00123             break;
00124         seg = comma + 1;
00125     }

```

```

00126     }
00127
00128     normalized[n] = '\0';
00129     free(trimmed);
00130     return normalized;
00131 }
00132
00133 static int parse_integer_token_mpz(mpz_t out, const char *token)
00134 {
00135     char *normalized = normalize_decimal_token(token);
00136     if (normalized == NULL)
00137         return 0;
00138
00139     int ok = (mpz_set_str(out, normalized, 10) == 0);
00140     free(normalized);
00141     return ok;
00142 }
00143
00144 static int parse_exponent_u64(const char *token, unsigned long *exp_out)
00145 {
00146     mpz_t exp_mpz;
00147     mpz_init(exp_mpz);
00148     int ok = parse_integer_token_mpz(exp_mpz, token);
00149     if (!ok || !mpz_fits_ulong_p(exp_mpz))
00150     {
00151         mpz_clear(exp_mpz);
00152         return 0;
00153     }
00154     *exp_out = mpz_get_ui(exp_mpz);
00155     mpz_clear(exp_mpz);
00156     return 1;
00157 }
00158
00159 static int parse_numeric_term_mpz(mpz_t out, const char *term)
00160 {
00161     char *trimmed = dup_trimmed(term);
00162     if (trimmed == NULL || trimmed[0] == '\0')
00163     {
00164         free(trimmed);
00165         return 0;
00166     }
00167
00168     char *pow_op = strchr(trimmed, '^');
00169     char *sci_op = strchr(trimmed, 'e');
00170     if (!sci_op)
00171         sci_op = strchr(trimmed, 'E');
00172
00173     if (pow_op && sci_op)
00174     {
00175         free(trimmed);
00176         return 0;
00177     }
00178
00179     if (pow_op != NULL)
00180     {
00181         if (strchr(pow_op + 1, '^') != NULL || strchr(pow_op + 1, 'e') != NULL || strchr(pow_op + 1, 'E') != NULL)
00182         {
00183             free(trimmed);
00184             return 0;
00185         }
00186
00187         *pow_op = '\0';
00188         const char *base_str = trimmed;
00189         const char *exp_str = pow_op + 1;
00190
00191         mpz_t base;
00192         mpz_init(base);
00193         unsigned long exp = 0;
00194         int ok = parse_integer_token_mpz(base, base_str) && parse_exponent_u64(exp_str, &exp);
00195         if (ok)
00196             mpz_pow_ui(out, base, exp);
00197
00198         mpz_clear(base);
00199         free(trimmed);
00200         return ok;
00201     }
00202
00203     if (sci_op != NULL)
00204     {
00205         if (strchr(sci_op + 1, 'e') != NULL || strchr(sci_op + 1, 'E') != NULL || strchr(sci_op + 1, '^') != NULL)
00206         {
00207             free(trimmed);
00208             return 0;
00209         }
00210
00211         *sci_op = '\0';
00212         const char *base_str = trimmed;

```

```

00213     const char *exp_str = sci_op + 1;
00214
00215     mpz_t base, pow10;
00216     mpz_inits(base, pow10, NULL);
00217     unsigned long exp = 0;
00218     int ok = parse_integer_token_mpz(base, base_str) && parse_exponent_u64(exp_str, &exp);
00219     if (ok)
00220     {
00221         mpz_ui_pow_ui(pow10, 10, exp);
00222         mpz_mul(out, base, pow10);
00223     }
00224
00225     mpz_clears(base, pow10, NULL);
00226     free(trimmed);
00227     return ok;
00228 }
00229
00230 int ok = parse_integer_token_mpz(out, trimmed);
00231 free(trimmed);
00232 return ok;
00233 }
00234
00235 static int parse_range_parts(const char *left_expr, const char *right_expr, mpz_t lower, mpz_t upper)
00236 {
00237     mpz_t left, right;
00238     mpz_inits(left, right, NULL);
00239
00240     int ok = parse_numeric_expr_mpz(left, left_expr) && parse_numeric_expr_mpz(right, right_expr) &&
    mpz_cmp(right, left) >= 0;
00241     if (ok)
00242     {
00243         mpz_set(lower, left);
00244         mpz_set(upper, right);
00245     }
00246
00247     mpz_clears(left, right, NULL);
00248     return ok;
00249 }
00250
00251 int is_numeric_str(const char *str)
00252 {
00253     if (str == NULL || *str == '\\0')
00254         return 0;
00255
00256     while (*str)
00257     {
00258         if (*str < '0' || *str > '9')
00259             return 0;
00260         ++str;
00261     }
00262     return 1;
00263 }
00264
00265 int parse_numeric_expr_mpz(mpz_t out, const char *expr)
00266 {
00267     if (expr == NULL)
00268         return 0;
00269
00270     size_t expr_len = strlen(expr);
00271     char *copy = malloc(expr_len + 1);
00272     if (copy == NULL)
00273         return 0;
00274     memcpy(copy, expr, expr_len + 1);
00275
00276     mpz_set_ui(out, 0);
00277     int has_terms = 0;
00278
00279     char *cursor = copy;
00280     while (cursor != NULL)
00281     {
00282         char *plus = strchr(cursor, '+');
00283         if (plus != NULL)
00284             *plus = '\\0';
00285
00286         mpz_t term_value;
00287         mpz_init(term_value);
00288         int ok = parse_numeric_term_mpz(term_value, cursor);
00289         if (!ok)
00290         {
00291             mpz_clear(term_value);
00292             free(copy);
00293             return 0;
00294         }
00295
00296         mpz_add(out, out, term_value);
00297         mpz_clear(term_value);
00298         has_terms = 1;
00299     }
00300
00301     return has_terms;
00302 }

```



```

00306
00307     cursor = (plus != NULL) ? (plus + 1) : NULL;
00308 }
00309
00310 free(copy);
00311 return has_terms;
00312 }
00313
00314 int parse_numeric_expr_u64(const char *expr, uint64_t *out)
00315 {
00316     if (out == NULL)
00317         return 0;
00318
00319     mpz_t value;
00320     mpz_init(value);
00321     if (!parse_numeric_expr_mpz(value, expr) || mpz_sgn(value) < 0 || mpz_sizeinbase(value, 2) > 64)
00322     {
00323         mpz_clear(value);
00324         return 0;
00325     }
00326
00327     uint64_t parsed = 0;
00328     size_t words = 0;
00329     mpz_export(&parsed, &words, 1, sizeof(parsed), 0, 0, value);
00330     *out = (words == 0) ? 0 : parsed;
00331     mpz_clear(value);
00332     return 1;
00333 }
00334
00335 int parse_inclusive_range_mpz(const char *range_expr, mpz_t lower, mpz_t upper)
00336 {
00337     if (range_expr == NULL)
00338         return 0;
00339
00340     char *range = dup_trimmed(range_expr);
00341     if (range == NULL || range[0] == '\\0')
00342     {
00343         free(range);
00344         return 0;
00345     }
00346
00347     size_t len = strlen(range);
00348     if (len > 7 && strncmp(range, "range[", 6) == 0 && range[len - 1] == ']')
00349     {
00350         memmove(range, range + 6, len - 7);
00351         range[len - 7] = '\\0';
00352     }
00353
00354     len = strlen(range);
00355     if (len > 2 && range[0] == '[' && range[len - 1] == ']')
00356     {
00357         memmove(range, range + 1, len - 2);
00358         range[len - 2] = '\\0';
00359     }
00360
00361     char *sep = strstr(range, "..");
00362     if (sep != NULL)
00363     {
00364         *sep = '\\0';
00365         int ok = parse_range_parts(range, sep + 2, lower, upper);
00366         free(range);
00367         return ok;
00368     }
00369
00370     sep = strchr(range, ',');
00371     if (sep != NULL)
00372     {
00373         *sep = '\\0';
00374         int ok = parse_range_parts(range, sep + 1, lower, upper);
00375         free(range);
00376         return ok;
00377     }
00378
00379     for (char *comma = range; (comma = strchr(comma, ',')) != NULL; ++comma)
00380     {
00381         char saved = *comma;
00382         *comma = '\\0';
00383         int ok = parse_range_parts(range, comma + 1, lower, upper);
00384         *comma = saved;
00385         if (ok)
00386         {
00387             free(range);
00388             return 1;
00389         }
00390     }
00391
00392     free(range);

```

```

00393     return 0;
00394 }
00395
00402 int create_dir(const char *dir)
00403 {
00404     struct stat st = {0};
00405     if (stat(dir, &st) == -1)
00406     {
00407         if (mkdir(dir, 0700) != 0)
00408         {
00409             // log error if mkdir fails
00410             log_error("mkdir failed to create %s :\\", dir);
00411             return -1;
00412         }
00413     }
00414
00415     return 0; // Directory already exists or was successfully created
00416 }
00417
00425 uint64_t gcd(uint64_t a, uint64_t b)
00426 {
00427     while (b != 0)
00428     {
00429         uint64_t temp = b;
00430         b = a % b;
00431         a = temp;
00432     }
00433     return a;
00434 }
00435
00453 uint64_t modular_inverse(uint64_t a, uint64_t m)
00454 {
00455     if (m == 1)
00456         return 0;
00457
00458     uint64_t m0 = m;
00459     int64_t x0 = 0, x1 = 1;
00460
00461     while (a > 1)
00462     {
00463         uint64_t q = a / m;
00464         uint64_t t = m;
00465
00466         m = a % m;
00467         a = t;
00468         t = x0;
00469
00470         x0 = x1 - (int64_t)q * x0;
00471         x1 = t;
00472     }
00473
00474     if (x1 < 0)
00475         x1 += (int64_t)m0;
00476     return (uint64_t)x1;
00477 }
00478
00486 void gmp_seed_randstate(gmp_randstate_t state)
00487 {
00488     unsigned long seed;
00489     int random_fd = open("/dev/urandom", O_RDONLY); // Open /dev/urandom for reading
00490     if (random_fd == -1)
00491     {
00492         // Fallback if /dev/urandom cannot be opened
00493         seed = (unsigned long)time(NULL);
00494     }
00495     else
00496     {
00497         read(random_fd, &seed, sizeof(seed)); // Read random bytes from /dev/urandom
00498         close(random_fd);
00499     }
00500
00501     gmp_randseed_ui(state, seed); // Seed the state with the random value
00502 }
00503
00509 int get_cpu_cores_count(void)
00510 {
00511     int cores = sysconf(_SC_NPROCESSORS_ONLN);
00512     if (cores < 1)
00513         cores = 1; // Fallback to 1 if unable to determine
00514     return cores;
00515 }
00516
00531 int get_cpu_L2_cache_size_bits(void)
00532 {
00533
00534     #ifdef __linux__
00535         // Linux: Try reading from sysfs

```

```

00536 FILE *fp = fopen("/sys/devices/system/cpu/cpu0/cache/index2/size", "r");
00537 if (fp != NULL)
00538 {
00539     char buffer[32];
00540     if (fgets(buffer, sizeof(buffer), fp) != NULL)
00541     {
00542         // Parse size with K suffix (e.g., "256K")
00543         int size_kb = 0;
00544         if (sscanf(buffer, "%dK", &size_kb) == 1)
00545         {
00546             fclose(fp);
00547             return size_kb * 1024 * 8; // Convert KB to bits
00548         }
00549     }
00550     fclose(fp);
00551 }
00552 #endif
00553
00554 #if defined(__APPLE__) || defined(__FreeBSD__) || defined(__OpenBSD__) || defined(__NetBSD__)
00555 // macOS and BSD: Use sysctl
00556 size_t size_bytes = 0;
00557 size_t len = sizeof(size_bytes);
00558
00559 // Try hw.l2cachesize (macOS/some BSDs)
00560 if (sysctlbyname("hw.l2cachesize", &size_bytes, &len, NULL, 0) == 0)
00561 {
00562     if (size_bytes > 0)
00563     {
00564         return (int)(size_bytes * 8); // Convert bytes to bits
00565     }
00566 }
00567
00568 // Fallback: Try machdep.cpu.cache.L2_cache_size (older macOS)
00569 if (sysctlbyname("machdep.cpu.cache.L2_cache_size", &size_bytes, &len, NULL, 0) == 0)
00570 {
00571     if (size_bytes > 0)
00572     {
00573         return (int)(size_bytes * 8); // Convert bytes to bits
00574     }
00575 }
00576 #endif
00577
00578 // Conservative fallback: 256 KB is common for L2 cache per core
00579 return 256 * 1024 * 8; // Convert 256 KB to bits
00580 }

```



# Chapter 14

## Examples

### 14.1 `/Users/hazemmounir/Workspace/iZproject/iZprime/src/toolkit/↵ bitmap.c`

Clears bits at regular intervals, typically used in sieve algorithms.

Clears bits at regular intervals, typically used in sieve algorithms. Starting from `start_idx`, this function clears every `step`-th bit up to the specified limit (inclusive). This is commonly used in prime sieve algorithms where you need to mark all multiples of a number as composite.

The function includes:

- Automatic capping of limit to bitmap size
- Integer overflow protection in the loop
- Early termination to prevent infinite loops

#### Parameters

bitmap	Pointer to the bitmap to modify
step	Interval between bits to clear (must be > 0, typically a prime)
start_idx	Starting bit index (inclusive, typically first multiple)
limit	Upper bound for clearing (inclusive, auto-capped to size-1)

#### Note

If `limit > bitmap->size`, it is automatically reduced to `bitmap->size - 1`

No operation if `bitmap` is `NULL`, `step` is 0, or `start_idx > limit`

Includes overflow prevention: stops if next index would overflow

// For Sieve of Eratosthenes: mark multiples of 3 starting from 9 bitmap\_clear\_steps(bitmap, 3, 9, 1000); // Clears 9, 12, 15, 18, ...

```
#include <bitmap.h>

#ifdef __aarch64__
#include <arm_neon.h>
#elif defined(__x86_64__) || defined(_M_X64)
#include <immintrin.h>
#endif

BITMAP *bitmap_init(size_t size, int set_bits)
{
    assert(size > 0 && "Bitmap size must be positive");

    BITMAP *bitmap = (BITMAP *)malloc(sizeof(BITMAP));
    if (bitmap == NULL)
    {
        log_error("Memory allocation failed for BITMAP struct");
        return NULL;
    }

    bitmap->size = size;
    bitmap->byte_size = (size + 7) / 8;
    bitmap->data = (unsigned char *)malloc(bitmap->byte_size);
    if (bitmap->data == NULL)
    {
        free(bitmap);
        log_error("bitmap_create: Memory allocation failed for BITMAP");
        return NULL;
    }

    if (set_bits)
        memset(bitmap->data, 0xFF, bitmap->byte_size);
    else
        memset(bitmap->data, 0x00, bitmap->byte_size);

    memset(bitmap->sha256, 0, SHA256_DIGEST_LENGTH); // Initialize SHA-256 to zero

    return bitmap;
}

void bitmap_free(BITMAP **bitmap)
{
    // Safe guard against double free and NULL pointer
    if (bitmap && *bitmap)
    {
        if ((*bitmap)->data)
        {
            free((*bitmap)->data);
            (*bitmap)->data = NULL;
        }

        free(*bitmap);
        *bitmap = NULL;
    }
}

void bitmap_set_bit(BITMAP *bitmap, size_t idx)
{
    bitmap->data[idx / 8] |= (1 « (idx % 8));
}

int bitmap_get_bit(BITMAP *bitmap, size_t idx)
{
    return (bitmap->data[idx / 8] & (1 « (idx % 8))) != 0;
}

void bitmap_flip_bit(BITMAP *bitmap, size_t idx)
{
    bitmap->data[idx / 8] ^= (1 « (idx % 8));
}

void bitmap_clear_bit(BITMAP *bitmap, size_t idx)
{
    // Clear the bit by ANDing with the negation of the bit mask
    bitmap->data[idx / 8] &= ~(1 « (idx % 8));
}

void bitmap_set_all(BITMAP *bitmap)
{
    memset(bitmap->data, 0xFF, (bitmap->size + 7) / 8);
}

void bitmap_clear_all(BITMAP *bitmap)
```

```

{
    memset(bitmap->data, 0x00, (bitmap->size + 7) / 8);
}

void bitmap_clear_steps(BITMAP *bitmap, uint64_t step, uint64_t start_idx, uint64_t limit)
{
    assert(step > 0 && "Step must be positive in bitmap_clear_steps.");
    limit = MIN(limit, bitmap->size - 1);

    for (uint64_t idx = start_idx; idx <= limit; idx += step)
    {
        bitmap->data[idx / 8] &= ~(1 « (idx % 8));
    }
}

void bitmap_clear_steps_simd(BITMAP *bitmap, uint64_t step, uint64_t start_idx, uint64_t limit)
{
    assert(step > 0 && "Step must be positive in bitmap_clear_steps_simd.");
    limit = MIN(limit, bitmap->size - 1);

#ifdef __aarch64__
    // NEON implementation for ARM64
    uint64_t idx = start_idx;

    // Process 4 steps at a time
    // Condition: idx + 3*step <= limit

    if (limit >= 3 * step && idx <= limit - 3 * step)
    {
        uint64x2_t v_step2 = vdupq_n_u64(2 * step);
        uint64x2_t v_step4 = vdupq_n_u64(4 * step);

        // Initialize indices: [idx, idx+step] and [idx+2step, idx+3step]
        uint64x2_t v_idx_01 = vdupq_n_u64(idx);
        v_idx_01 = vsetq_lane_u64(idx + step, v_idx_01, 1);

        uint64x2_t v_idx_23 = vaddq_u64(v_idx_01, v_step2);

        while (idx <= limit - 3 * step)
        {
            // Extract indices and perform operations
            // Note: NEON doesn't have efficient scatter for bytes, so we extract and do scalar stores

            uint64_t i0 = vgetq_lane_u64(v_idx_01, 0);
            uint64_t i1 = vgetq_lane_u64(v_idx_01, 1);
            uint64_t i2 = vgetq_lane_u64(v_idx_23, 0);
            uint64_t i3 = vgetq_lane_u64(v_idx_23, 1);

            bitmap->data[i0 / 8] &= ~(1 « (i0 % 8));
            bitmap->data[i1 / 8] &= ~(1 « (i1 % 8));
            bitmap->data[i2 / 8] &= ~(1 « (i2 % 8));
            bitmap->data[i3 / 8] &= ~(1 « (i3 % 8));

            // Advance indices
            v_idx_01 = vaddq_u64(v_idx_01, v_step4);
            v_idx_23 = vaddq_u64(v_idx_23, v_step4);

            idx += 4 * step;
        }
    }

    // Handle remaining steps
    for (; idx <= limit; idx += step)
    {
        bitmap->data[idx / 8] &= ~(1 « (idx % 8));
    }
}

#elif defined(__AVX2__)
    // AVX2 implementation for x86_64
    uint64_t idx = start_idx;

    // Process 4 steps at a time
    if (limit >= 3 * step && idx <= limit - 3 * step)
    {
        __m256i v_step4 = _mm256_set1_epi64x(4 * step);

        // Initialize indices: [idx, idx+step, idx+2step, idx+3step]
        // Note: _mm256_set_epi64x takes arguments in reverse order (e3, e2, e1, e0)
        __m256i v_idx = _mm256_set_epi64x(idx + 3 * step, idx + 2 * step, idx + step, idx);

        while (idx <= limit - 3 * step)
        {
            // Extract indices
            uint64_t i0 = _mm256_extract_epi64(v_idx, 0);
            uint64_t i1 = _mm256_extract_epi64(v_idx, 1);
            uint64_t i2 = _mm256_extract_epi64(v_idx, 2);

```

```

uint64_t i3 = _mm256_extract_epi64(v_idx, 3);

bitmap->data[i0 / 8] &= ~(1 « (i0 % 8));
bitmap->data[i1 / 8] &= ~(1 « (i1 % 8));
bitmap->data[i2 / 8] &= ~(1 « (i2 % 8));
bitmap->data[i3 / 8] &= ~(1 « (i3 % 8));

// Advance indices
v_idx = _mm256_add_epi64(v_idx, v_step4);
idx += 4 * step;
}
}

// Handle remaining steps
for (; idx <= limit; idx += step)
{
    bitmap->data[idx / 8] &= ~(1 « (idx % 8));
}

#elif defined(__SSE2__)
// SSE2 implementation for x86_64
uint64_t idx = start_idx;

// Process 2 steps at a time
if (limit >= step && idx <= limit - step)
{
    _mm128i v_step2 = _mm_set1_epi64x(2 * step);
    // Note: _mm_set_epi64x takes arguments in reverse order (e1, e0)
    _mm128i v_idx = _mm_set_epi64x(idx + step, idx);

    while (idx <= limit - step)
    {
        // Extract indices
        // _mm_cvtsi128_si64 extracts the lower 64 bits (element 0)
        uint64_t i0 = _mm_cvtsi128_si64(v_idx);
        // Unpack high to low to extract element 1
        uint64_t i1 = _mm_cvtsi128_si64(_mm_unpackhi_epi64(v_idx, v_idx));

        bitmap->data[i0 / 8] &= ~(1 « (i0 % 8));
        bitmap->data[i1 / 8] &= ~(1 « (i1 % 8));

        // Advance indices
        v_idx = _mm_add_epi64(v_idx, v_step2);
        idx += 2 * step;
    }
}

// Handle remaining steps
for (; idx <= limit; idx += step)
{
    bitmap->data[idx / 8] &= ~(1 « (idx % 8));
}

#else
// Fallback to scalar implementation
for (uint64_t idx = start_idx; idx <= limit; idx += step)
{
    bitmap->data[idx / 8] &= ~(1 « (idx % 8));
}
#endif
}

BITMAP *bitmap_clone(BITMAP *src)
{
    assert(src && src->data && "Invalid source bitmap passed to bitmap_clone.");

    BITMAP *dest = bitmap_init(src->size, 0);
    memcpy(dest->data, src->data, src->byte_size);
    memcpy(dest->sha256, src->sha256, SHA256_DIGEST_LENGTH);

    return dest;
}

void bitmap_compute_hash(BITMAP *bitmap)
{
    assert(bitmap && bitmap->data && "Invalid bitmap passed to bitmap_compute_hash.");

    // Generate SHA-256 hash and store it in the struct
    SHA256((unsigned char *)bitmap->data, bitmap->byte_size, bitmap->sha256);
}

int bitmap_validate_hash(BITMAP *bitmap)
{

```



```

assert(bitmap && bitmap->data && "Invalid bitmap");

unsigned char correct_hash[SHA256_DIGEST_LENGTH]; // Buffer to hold the computed hash

// Generate SHA-256 hash and store it in correct_hash
SHA256((unsigned char *)bitmap->data, bitmap->byte_size, correct_hash);

// Compare actual_hash with the stored hash in bitmap->sha256
if (memcmp(correct_hash, bitmap->sha256, SHA256_DIGEST_LENGTH) == 0)
{
    return 1; // SHA-256 match
}
else
{
    log_error("SHA-256 checksum validation failed.");
    return 0; // SHA-256 mismatch
}
}

int bitmap_fwrite(BITMAP *bitmap, FILE *file)
{
    assert(bitmap && bitmap->data && "Invalid bitmap passed to bitmap_fwrite.");
    assert(file && "File pointer is NULL in bitmap_fwrite.");

    // Write size
    if (fwrite(&bitmap->size, sizeof(size_t), 1, file) != 1)
    {
        log_error("Failed to write bitmap size to file.");
        return 0;
    }

    // Write data
    if (fwrite(bitmap->data, sizeof(unsigned char), bitmap->byte_size, file) != bitmap->byte_size)
    {
        log_error("Failed to write bitmap data to file.");
        return 0;
    }

    // Compute SHA-256 hash if not already computed
    // Check if hash is all zeros by examining all bytes
    int hash_is_zero = 1;
    for (size_t i = 0; i < SHA256_DIGEST_LENGTH; i++)
    {
        if (bitmap->sha256[i] != 0)
        {
            hash_is_zero = 0;
            break;
        }
    }

    if (hash_is_zero)
    {
        bitmap_compute_hash(bitmap);
    }

    // Write SHA-256 hash
    if (fwrite(bitmap->sha256, 1, SHA256_DIGEST_LENGTH, file) != SHA256_DIGEST_LENGTH)
    {
        log_error("Failed to write SHA-256 hash to file.");
        return 0;
    }

    return 1; // Success
}

BITMAP *bitmap_fread(FILE *file)
{
    assert(file && "File pointer is NULL in bitmap_fread.");

    // Read size
    size_t size;
    if (fread(&size, sizeof(size_t), 1, file) != 1)
    {
        log_error("Failed to read bitmap size from file.");
        return NULL;
    }

    BITMAP *bitmap = bitmap_init(size, 0);
    if (bitmap == NULL)
    {
        log_error("Failed to initialize bitmap.");
        return NULL;
    }

    // Read data directly into the already-allocated buffer
    if (fread(bitmap->data, sizeof(unsigned char), bitmap->byte_size, file) != bitmap->byte_size)
    {

```

```
    log_error("Failed to read complete bitmap data from file.");
    bitmap_free(&bitmap);
    return NULL;
}

// Read SHA-256 hash
if (fread(bitmap->sha256, 1, SHA256_DIGEST_LENGTH, file) != SHA256_DIGEST_LENGTH)
{
    log_error("Failed to read SHA-256 hash from file.");
    bitmap_free(&bitmap);
    return NULL;
}

if (!bitmap_validate_hash(bitmap))
{
    log_error("SHA-256 hash validation failed.");
    bitmap_free(&bitmap);
    return NULL;
}

return bitmap;
}
```

# Index

- array
  - [UI16\\_ARRAY](#), [109](#)
  - [UI32\\_ARRAY](#), [111](#)
  - [UI64\\_ARRAY](#), [112](#)
- [ASSERT\\_LIMIT](#)
  - [prime\\_sieve.c](#), [148](#)
- base
  - [SIEVE\\_LIMIT](#), [106](#)
- [base\\_x5](#)
  - [IZM](#), [103](#)
- [base\\_x7](#)
  - [IZM](#), [103](#)
- [Benchmarks](#), [3](#)
- [bit\\_ops](#)
  - [VX\\_SEG](#), [114](#)
- [BITMAP](#), [99](#)
  - [byte\\_size](#), [99](#)
  - [data](#), [99](#)
  - [sha256](#), [100](#)
  - [size](#), [100](#)
- [Bitmap Module](#), [27](#)
  - [bitmap\\_clear\\_all](#), [29](#)
  - [bitmap\\_clear\\_bit](#), [29](#)
  - [bitmap\\_clear\\_steps](#), [29](#)
  - [bitmap\\_clear\\_steps\\_simd](#), [30](#)
  - [bitmap\\_clone](#), [31](#)
  - [bitmap\\_compute\\_hash](#), [32](#)
  - [bitmap\\_flip\\_bit](#), [32](#)
  - [bitmap\\_fread](#), [33](#)
  - [bitmap\\_free](#), [33](#)
  - [bitmap\\_fwrite](#), [34](#)
  - [bitmap\\_get\\_bit](#), [35](#)
  - [bitmap\\_init](#), [36](#)
  - [bitmap\\_set\\_all](#), [37](#)
  - [bitmap\\_set\\_bit](#), [37](#)
  - [bitmap\\_validate\\_hash](#), [38](#)
  - [TEST\\_BITMAP](#), [38](#)
- [bitmap\\_clear\\_all](#)
  - [Bitmap Module](#), [29](#)
- [bitmap\\_clear\\_bit](#)
  - [Bitmap Module](#), [29](#)
- [bitmap\\_clear\\_steps](#)
  - [Bitmap Module](#), [29](#)
- [bitmap\\_clear\\_steps\\_simd](#)
  - [Bitmap Module](#), [30](#)
- [bitmap\\_clone](#)
  - [Bitmap Module](#), [31](#)
- [bitmap\\_compute\\_hash](#)
  - [Bitmap Module](#), [32](#)
- [bitmap\\_flip\\_bit](#)
  - [Bitmap Module](#), [32](#)
- [bitmap\\_fread](#)
  - [Bitmap Module](#), [33](#)
- [bitmap\\_free](#)
  - [Bitmap Module](#), [33](#)
- [bitmap\\_fwrite](#)
  - [Bitmap Module](#), [34](#)
- [bitmap\\_get\\_bit](#)
  - [Bitmap Module](#), [35](#)
- [bitmap\\_init](#)
  - [Bitmap Module](#), [36](#)
- [bitmap\\_set\\_all](#)
  - [Bitmap Module](#), [37](#)
- [bitmap\\_set\\_bit](#)
  - [Bitmap Module](#), [37](#)
- [bitmap\\_validate\\_hash](#)
  - [Bitmap Module](#), [38](#)
- [byte\\_size](#)
  - [BITMAP](#), [99](#)
- [capacity](#)
  - [UI16\\_ARRAY](#), [109](#)
  - [UI32\\_ARRAY](#), [111](#)
  - [UI64\\_ARRAY](#), [112](#)
- [check\\_primality](#)
  - [Toolkit \(iZ/iZm\)](#), [59](#)
- [compute\\_k\\_vx](#)
  - [iZ\\_toolkit.c](#), [165](#)
- [compute\\_l2\\_vx](#)
  - [Toolkit \(iZ/iZm\)](#), [60](#)
- [compute\\_max\\_vx](#)
  - [Toolkit \(iZ/iZm\)](#), [60](#)
- [compute\\_vx\\_k](#)
  - [Toolkit \(iZ/iZm\)](#), [61](#)
- [Contributing to iZprime](#), [9](#)
- [count](#)
  - [UI16\\_ARRAY](#), [110](#)
  - [UI32\\_ARRAY](#), [111](#)
  - [UI64\\_ARRAY](#), [112](#)
- [create\\_dir](#)
  - [Utilities](#), [86](#)
- [data](#)
  - [BITMAP](#), [99](#)
- [DIR\\_output](#)
  - [Utilities](#), [85](#)
- [elapsed\\_sec](#)
  - [STOPWATCH](#), [108](#)

- end\_time
  - STOPWATCH, 108
- end\_x
  - VX\_SEG, 114
- exp
  - SIEVE\_LIMIT, 106
- filepath
  - INPUT\_SIEVE\_RANGE, 101
- function
  - SIEVE\_MODEL, 107
- gcd
  - Utilities, 86
- get\_cpu\_cores\_count
  - Utilities, 86
- get\_cpu\_L2\_cache\_size\_bits
  - Utilities, 87
- get\_root\_primes
  - Toolkit (iZ/iZm), 61
- gmp\_seed\_randstate
  - Utilities, 87
- include/bitmap.h, 117, 118
- include/int\_arrays.h, 119, 121
- include/iZ\_api.h, 122, 124
- include/iZ\_toolkit.h, 124, 127
- include/logger.h, 128, 130
- include/test\_api.h, 131, 132
- include/utls.h, 132, 134
- INPUT\_SIEVE\_RANGE, 101
  - filepath, 101
  - iZ Public API, 43
  - mr\_rounds, 101
  - range, 101
  - start, 102
- Integer Arrays, 39
  - TEST\_GENERIC\_INT\_ARRAYS, 41
- is\_large\_limit
  - VX\_SEG, 114
- is\_numeric\_str
  - Utilities, 88
- iZ
  - Toolkit (iZ/iZm), 62
- iZ Public API, 42
  - INPUT\_SIEVE\_RANGE, 43
  - iZ\_next\_prime, 44
  - SiZ, 44
  - SiZ\_count, 45
  - SiZ\_stream, 47
  - SiZm, 48
  - SiZm\_vy, 49
  - SoA, 49
  - SoE, 50
  - SoEu, 51
  - SoS, 52
  - SSoE, 53
  - vx\_random\_prime, 53
  - vy\_random\_prime, 54
- iZ\_mpz
  - Toolkit (iZ/iZm), 62
- iZ\_next\_prime
  - iZ Public API, 44
- iZ\_toolkit.c
  - compute\_k\_vx, 165
  - s\_primes, 167
  - s\_primes\_count, 167
  - vx\_det\_sieve, 165
  - vx\_prob\_sieve, 166
  - vx\_set\_base\_values, 166
- IZM, 102
  - base\_x5, 103
  - base\_x7, 103
  - k\_vx, 103
  - root\_primes, 103
  - vx, 103
- iZm\_clone
  - Toolkit (iZ/iZm), 63
- iZm\_construct\_vx\_base
  - Toolkit (iZ/iZm), 63
- iZm\_free
  - Toolkit (iZ/iZm), 64
- iZm\_init
  - Toolkit (iZ/iZm), 64
- IZM\_RANGE\_INFO, 104
  - vx, 104
  - Xe, 104
  - Xs, 105
  - y\_range, 105
  - Ye, 105
  - Ys, 105
  - Ze, 105
  - Zs, 106
- iZm\_solve\_for\_x0
  - Toolkit (iZ/iZm), 65
- iZm\_solve\_for\_x0\_mpz
  - Toolkit (iZ/iZm), 66
- iZm\_solve\_for\_y0
  - Toolkit (iZ/iZm), 68
- izprime CLI, 7
- izprime User Manual, 1
- k\_vx
  - IZM, 103
- log\_console
  - Logging, 78
- LOG\_DEBUG
  - Logging, 77
- log\_debug
  - Logging, 78
- LOG\_DIR
  - Logging, 77
- LOG\_ERROR
  - Logging, 78
- log\_error
  - Logging, 78
- LOG\_FATAL

- Logging, [78](#)
- log\_fatal
  - Logging, [79](#)
- LOG\_FILE
  - Logging, [77](#)
- LOG\_INFO
  - Logging, [78](#)
- log\_info
  - Logging, [79](#)
- log\_init
  - Logging, [79](#)
- log\_level\_to\_string
  - Logging, [80](#)
- LOG\_MAX\_SIZE
  - Logging, [77](#)
- log\_message
  - Logging, [80](#)
- log\_message\_extended
  - Logging, [80](#)
- log\_set\_log\_level
  - Logging, [82](#)
- log\_warn
  - Logging, [82](#)
- LOG\_WARNING
  - Logging, [78](#)
- LOGGER\_FORMAT\_PRINTF
  - Logging, [77](#)
- Logging, [76](#)
  - log\_console, [78](#)
  - LOG\_DEBUG, [77](#)
  - log\_debug, [78](#)
  - LOG\_DIR, [77](#)
  - LOG\_ERROR, [78](#)
  - log\_error, [78](#)
  - LOG\_FATAL, [78](#)
  - log\_fatal, [79](#)
  - LOG\_FILE, [77](#)
  - LOG\_INFO, [78](#)
  - log\_info, [79](#)
  - log\_init, [79](#)
  - log\_level\_to\_string, [80](#)
  - LOG\_MAX\_SIZE, [77](#)
  - log\_message, [80](#)
  - log\_message\_extended, [80](#)
  - log\_set\_log\_level, [82](#)
  - log\_warn, [82](#)
  - LOG\_WARNING, [78](#)
  - LOGGER\_FORMAT\_PRINTF, [77](#)
  - LogLevel, [77](#)
- LogLevel
  - Logging, [77](#)
- Makefile documentation (iZprime), [13](#)
- MAX
  - Utilities, [85](#)
- MAX\_CORES
  - Utilities, [85](#)
- MIN
  - Utilities, [85](#)
- modular\_inverse
  - Utilities, [88](#)
- MR\_ROUNDS
  - Toolkit (iZ/iZm), [58](#)
- mr\_rounds
  - INPUT\_SIEVE\_RANGE, [101](#)
  - VX\_SEG, [114](#)
- N\_LIMIT
  - prime\_sieve.c, [148](#)
- name
  - SIEVE\_MODEL, [107](#)
- ordered
  - UI16\_ARRAY, [110](#)
  - UI32\_ARRAY, [111](#)
  - UI64\_ARRAY, [112](#)
- p\_count
  - VX\_SEG, [114](#)
- p\_gaps
  - VX\_SEG, [114](#)
- p\_test\_ops
  - VX\_SEG, [115](#)
- parse\_inclusive\_range\_mpz
  - Utilities, [89](#)
- parse\_numeric\_expr\_mpz
  - Utilities, [90](#)
- parse\_numeric\_expr\_u64
  - Utilities, [90](#)
- Pi
  - prime\_sieve.c, [148](#)
- prime\_sieve.c
  - ASSERT\_LIMIT, [148](#)
  - N\_LIMIT, [148](#)
  - Pi, [148](#)
  - process\_N\_bitmap, [148](#)
- print\_centered\_text
  - Printer, implemented in toolkit/print\_utils.c, [92](#)
- print\_line
  - Printer, implemented in toolkit/print\_utils.c, [93](#)
- print\_sha256\_hash
  - Printer, implemented in toolkit/print\_utils.c, [93](#)
- print\_test\_fn\_header
  - Printer, implemented in toolkit/print\_utils.c, [93](#)
- print\_test\_module\_header
  - Printer, implemented in toolkit/print\_utils.c, [93](#)
- print\_test\_module\_result
  - Printer, implemented in toolkit/print\_utils.c, [94](#)
- print\_test\_summary
  - Printer, implemented in toolkit/print\_utils.c, [94](#)
- print\_test\_table\_header

- Printer, implemented in toolkit/print\_utils.c, 96
- Printer, implemented in toolkit/print\_utils.c, 91
  - print\_centered\_text, 92
  - print\_line, 93
  - print\_sha256\_hash, 93
  - print\_test\_fn\_header, 93
  - print\_test\_module\_header, 93
  - print\_test\_module\_result, 94
  - print\_test\_summary, 94
  - print\_test\_table\_header, 96
- process\_iZ\_bitmaps
  - Toolkit (iZ/iZm), 69
- process\_N\_bitmap
  - prime\_sieve.c, 148
- range
  - INPUT\_SIEVE\_RANGE, 101
- range\_info\_free
  - Toolkit (iZ/iZm), 69
- range\_info\_init
  - Toolkit (iZ/iZm), 69
- Releasing iZprime, 17
- root\_limit
  - VX\_SEG, 115
- root\_primes
  - IZM, 103
- running
  - STOPWATCH, 108
- s\_primes
  - iZ\_toolkit.c, 167
- s\_primes\_count
  - iZ\_toolkit.c, 167
- sha256
  - BITMAP, 100
  - UI16\_ARRAY, 110
  - UI32\_ARRAY, 111
  - UI64\_ARRAY, 113
- SIEVE\_FN
  - Tests and Benchmarks, 83
- SIEVE\_LIMIT, 106
  - base, 106
  - exp, 106
- SIEVE\_MODEL, 107
  - function, 107
  - name, 107
- SiZ
  - iZ Public API, 44
- SiZ\_count
  - iZ Public API, 45
- SiZ\_stream
  - iZ Public API, 47
- size
  - BITMAP, 100
- SiZm
  - iZ Public API, 48
- SiZm\_vy
  - iZ Public API, 49
- SoA
  - iZ Public API, 49
- SoE
  - iZ Public API, 50
- SoEu
  - iZ Public API, 51
- SoS
  - iZ Public API, 52
- src/iZ\_apps.c, 135, 136
- src/playground.c, 146
- src/prime\_sieve.c, 147, 149
- src/toolkit/bitmap.c, 155, 157
- src/toolkit/int\_arrays.c, 161, 163
- src/toolkit/iZ\_toolkit.c, 163, 167
- src/toolkit/logger.c, 178
- src/toolkit/print\_utils.c, 181, 182
- src/toolkit/stopwatch.c, 183, 184
- src/toolkit/utils.c, 184, 185
- SSoE
  - iZ Public API, 53
- start
  - INPUT\_SIEVE\_RANGE, 102
- start\_time
  - STOPWATCH, 108
- start\_x
  - VX\_SEG, 115
- STOPWATCH, 108
  - elapsed\_sec, 108
  - end\_time, 108
  - running, 108
  - start\_time, 108
- Stopwatch, 96
  - sw\_elapsed\_now\_seconds, 97
  - sw\_elapsed\_seconds, 97
  - sw\_start, 97
  - sw\_stop, 97
- sw\_elapsed\_now\_seconds
  - Stopwatch, 97
- sw\_elapsed\_seconds
  - Stopwatch, 97
- sw\_start
  - Stopwatch, 97
- sw\_stop
  - Stopwatch, 97
- TEST\_BITMAP
  - Bitmap Module, 38
- TEST\_GENERIC\_INT\_ARRAYS
  - Integer Arrays, 41
- Tests, 19
- Tests and Benchmarks, 82
  - SIEVE\_FN, 83
- Toolkit (iZ/iZm), 55
  - check\_primalty, 59
  - compute\_l2\_vx, 60
  - compute\_max\_vx, 60
  - compute\_vx\_k, 61
  - get\_root\_primes, 61
  - iZ, 62

- iZ\_mpz, 62
- iZm\_clone, 63
- iZm\_construct\_vx\_base, 63
- iZm\_free, 64
- iZm\_init, 64
- iZm\_solve\_for\_x0, 65
- iZm\_solve\_for\_x0\_mpz, 66
- iZm\_solve\_for\_y0, 68
- MR\_ROUNDS, 58
- process\_iZ\_bitmaps, 69
- range\_info\_free, 69
- range\_info\_init, 69
- VX2, 58
- VX3, 58
- VX4, 58
- VX5, 58
- VX6, 58
- VX7, 59
- VX8, 59
- vx\_collect\_p\_gaps, 70
- vx\_free, 70
- vx\_full\_sieve, 71
- vx\_init, 71
- vx\_search\_prime, 72
- vx\_stream, 73
- vy\_search\_prime, 75
- UI16\_ARRAY, 109
  - array, 109
  - capacity, 109
  - count, 110
  - ordered, 110
  - sha256, 110
- UI32\_ARRAY, 110
  - array, 111
  - capacity, 111
  - count, 111
  - ordered, 111
  - sha256, 111
- UI64\_ARRAY, 112
  - array, 112
  - capacity, 112
  - count, 112
  - ordered, 112
  - sha256, 113
- Utilities, 84
  - create\_dir, 86
  - DIR\_output, 85
  - gcd, 86
  - get\_cpu\_cores\_count, 86
  - get\_cpu\_L2\_cache\_size\_bits, 87
  - gmp\_seed\_randstate, 87
  - is\_numeric\_str, 88
  - MAX, 85
  - MAX\_CORES, 85
  - MIN, 85
  - modular\_inverse, 88
  - parse\_inclusive\_range\_mpz, 89
  - parse\_numeric\_expr\_mpz, 90
  - parse\_numeric\_expr\_u64, 90
- vx
  - IZM, 103
  - IZM\_RANGE\_INFO, 104
  - VX\_SEG, 115
- VX2
  - Toolkit (iZ/iZm), 58
- VX3
  - Toolkit (iZ/iZm), 58
- VX4
  - Toolkit (iZ/iZm), 58
- VX5
  - Toolkit (iZ/iZm), 58
- VX6
  - Toolkit (iZ/iZm), 58
- VX7
  - Toolkit (iZ/iZm), 59
- VX8
  - Toolkit (iZ/iZm), 59
- vx\_collect\_p\_gaps
  - Toolkit (iZ/iZm), 70
- vx\_det\_sieve
  - iZ\_toolkit.c, 165
- vx\_free
  - Toolkit (iZ/iZm), 70
- vx\_full\_sieve
  - Toolkit (iZ/iZm), 71
- vx\_init
  - Toolkit (iZ/iZm), 71
- vx\_prob\_sieve
  - iZ\_toolkit.c, 166
- vx\_random\_prime
  - iZ Public API, 53
- vx\_search\_prime
  - Toolkit (iZ/iZm), 72
- VX\_SEG, 113
  - bit\_ops, 114
  - end\_x, 114
  - is\_large\_limit, 114
  - mr\_rounds, 114
  - p\_count, 114
  - p\_gaps, 114
  - p\_test\_ops, 115
  - root\_limit, 115
  - start\_x, 115
  - vx, 115
  - x5, 115
  - x7, 116
  - y, 116
  - yvx, 116
- vx\_set\_base\_values
  - iZ\_toolkit.c, 166
- vx\_stream
  - Toolkit (iZ/iZm), 73
- vy\_random\_prime
  - iZ Public API, 54
- vy\_search\_prime
  - Toolkit (iZ/iZm), 75

x5  
    VX\_SEG, [115](#)  
x7  
    VX\_SEG, [116](#)  
Xe  
    IZM\_RANGE\_INFO, [104](#)  
Xs  
    IZM\_RANGE\_INFO, [105](#)  
  
y  
    VX\_SEG, [116](#)  
y\_range  
    IZM\_RANGE\_INFO, [105](#)  
Ye  
    IZM\_RANGE\_INFO, [105](#)  
Ys  
    IZM\_RANGE\_INFO, [105](#)  
yvx  
    VX\_SEG, [116](#)  
  
Ze  
    IZM\_RANGE\_INFO, [105](#)  
Zs  
    IZM\_RANGE\_INFO, [106](#)