libzahl version 1.1

Short contents

Contents

Chapter 1

What is libzahl?

In this chapter, it is discussed what libzahl is, why it is called libzahl, why it exists, why you should use it, what makes it different, and what is its limitations.

Contents

1.1 The name and the what

In mathematics, the set of all integers is represented by a bold uppercase ‘Z’ (**Z**), or sometimes double-stroked (blackboard bold) ($\mathbb{Z}$). This symbol is derived from the german word for integers: ‘Zahlen’ [ˈtsaːlən], whose singular is ‘Zahl’ [tsaːl]. libzahl [lˌɪbˈtsaːl] is a C library capable of representing very large integers, limited by the memory address space and available memory. Whilst this is almost none of the elements in **Z**, it is substantially more than available using the intrinsic integer types in C. libzahl of course also implements functions for performing arithmetic operations over integers represented using libzahl. Libraries such as libzahl are called bigint libraries, big integer libraries, multiple precision integer libraries, arbitrary precision integer libraries,[1] or bignum libraries, or any of the previous with ‘number’ substituted for ‘integer’. Some libraries that refer to themselves as bignum libraries or any of using the word ‘number’ support other number types than integers. libzahl only supports integers.

[1] ‘Multiple precision integer’ and ‘arbitrary precision integer’ are misnomers, precision is only relevant for floating-point numbers.

1.2 Why does it exist?

libzahl's main competitors are GNU MP (gmp),[2] LibTomMath (ltm), TomsFastMath (tfm) and Hebimath. All of these have problems:

- GNU MP is extremely bloated, can only be complied with GCC, and requires that you use glibc unless another C standard library was used when GNU MP was compiled. Additionally, whilst its performance is generally good, it can still be improved. Furthermore, GNU MP cannot be used for robust applications.
- LibTomMath is very slow, in fact performance is not its priority, rather its simplicity is the priority. Despite this, it is not really that simple.
- TomsFastMath is slow, complicated, and is not a true big integer library and is specifically targeted at cryptography.

libzahl is developed under the suckless.org umbrella. As such, it attempts to follow the suckless philosophy.[3] libzahl is simple, very fast, simple to use, and can be used in robust applications. Currently however, it does not support multithreading, but it has better support for multiprocessing and distributed computing than its competitors.

Lesser "competitors" (less known) to libzahl include Hebimath and bsdnt.

- Hebimath is far from stable, some fundamental functions are not implemented and some functions are broken. The author of libzahl thinks Hebimath is promising, but that it could be better designed. Like libzahl, Hebimath aims to follow the suckless philosophy.
- bsdnt has not been thoroughly investigated, but it does not look promising.

[2]GNU Multiple Precision Arithmetic Library
[3]http://suckless.org/philosophy

1.3 How is it different?

All big number libraries have in common that both input and output integers are parameters for the functions. There are however two variants of this: input parameters followed by output parameters, and output parameters followed by input parameters. The former variant is the conventional for C functions. The latter is more in style with primitive operations, pseudo-code, mathematics, and how it would look if the output was return. In libzahl, the latter convention is used. That is, we write

```
zadd(sum, augend, addend);
```

rather than

```
zadd(augend, addend, sum);
```

This can be compared to

$$sum \leftarrow augend + addend$$

versus

$$augend + addend \rightarrow sum.$$

libzahl, GNU MP, and Hebimath use the output-first convention.[4] LibTomMath and TomsFastMath use the input-first convention.[5]

Unlike other bignum libraries, errors in libzahl are caught using `setjmp`. This ensure that it can be used in robust applications, catching errors does not become a mess, and it minimises the overhead of catching errors. Errors are only checked when they can occur, not also after each function return.

Additionally, libzahl tries to keep the functions' names simple and natural rather than technical or mathematical. The names resemble those of the standard integer operators. For example, the left-shift, right-shift and truncation bit-operations in libzahl are called `zlsh`, `zrsh` and `ztrunc`, respectively. In GNU MP, they are called `mpz_mul_2exp`, `mpz_tdiv_q_2exp` and `mpz_tdiv_r_2exp`. The need of complicated names are diminished by resisting to implement all possible variants of each operations. Variants of a function simply append a short description of the difference in plain text. For example, a variant of `zadd` that makes the assumption that both operands are non-negative (or if not so, calculates the sum of their absolute values) is called `zadd_unsigned`. If libzahl would have had floored and

[4] GNU MP-style.
[5] BSD MP-style.

ceiled variants of `zdiv` (truncated division), they would have been called `zdiv_floor` and `zdiv_ceiling`. `zdiv` and `zmod` (modulus) are variants of `zdivmod` that throw away one of the outputs. These names can be compared to GNU MP's variants of truncated division: `mpz_tdiv_q`, `mpz_tdiv_r` and `mpz_tdiv_qr`.

1.4 Limitations

libzahl is not recommended for cryptographic applications, it is not mature enough, and its author does not have the necessary expertise. And in particular, it does not implement constant time operations, and it does not clear pooled memory. Using libzahl in cryptographic application is insecure; your application may become susceptible attacks such as timing attacks, power-monitoring attacks, electromagnetic attacks, acoustic cryptanalysis, and data remanence attacks. libzahl is known to be susceptible to timing attacks (due to lack of constant time operations) and data remanence attacks (due to pooling memory for reuse without clearing the content of the memory allocations.) Additionally, libzahl is not thread-safe.

libzahl is also only designed for POSIX systems. It will probably run just fine on any modern system. But it makes some assumption that POSIX stipulates or are unpractical not to implement for machines that should support POSIX (or even support modern software):

- Bytes are octets.
- There is an integer type that is 64-bits wide. (The compiler needs to support it, but it is not strictly necessary for it to be an CPU-intrinsic, but that would be favourable for performance.)
- Two's complement is used. (The compiler needs to support it, but it is not strictly necessary for it to be an CPU-intrinsic, but that would be favourable for performance.)

Because of the prevalence of theses properties in contemporary machines, and the utilisation of these properties in software, especially software for POSIX and popular platforms with similar properties, any new general-purpose machine most have these properties lest, it but useless with today's software. Therefore, libzahl can make the assumption that the machine has these properties. If the machine does not have these properties, the compiler must compensate for these machines deficiencies, making it generally slower.

These limitations may be removed later. And there is some code that does not make these assumptions but acknowledge that it may be a case. On the other hand, these limitations could be fixed, and agnostic code could be rewritten to assume that these restrictions are met.

Chapter 2

libzahl's design

In this chapter, the design of libzahl is discussed.

Contents

2.1 Memory pool

Allocating memory dynamically is an expensive operation. To improve performance, libzahl never deallocates memory before the library is uninitialised, instead it pools memory, that is no longer needed, for reuse.

Because of the memory pooling, this is a pattern to the allocation sizes. In an allocation, a power of two elements, plus a few elements that are discussed in Section 2.3 [Integer structure], page 10, are allocated. That is, the number multiplied by the size of an element. Powers of two (growth factor 2) is not the most memory efficient way to do this, but it is the simplest and performance efficient. This power of two (sans the few extra elements) is used to calculate — getting the index of the only set bit — the index of the bucket in which the allocation is stored when pooled. The buckets are dynamic arrays with the growth factor 1.5. The growth factor 1.5 is often used for dynamic arrays, it is a good compromise between memory usage and performance.

libzahl also avoids allocating memory by having a set of temporary variables predefined.

2.2 Error handling

In C, it is traditional to return a sentiel value in case an error has occurred, and set the value of a global variable to describe the error that has occurred. The programmer can choose whether to check for errors, ignore errors where it does not matter, or simple ignore errors altogether and let the program eventually crash. This is a simple technique that gives the programmer a better understanding of what can happen. A great advantage C has over most programming languages.

Another technique is to use long jumps on error. This technique is not too common, but is has one significant advantage. Error-checks need only be preformed where the error can first be detected. There is no need to check the return value at every function return. This leads to cleaner code, if there are many functions that can raise exceptional conditions, and greater performance under some conditions. This is why this technique is sometimes used in high-performance libraries. libzahl uses this technique.

Rather than writing

```
if (zadd(a, b, c))
    goto out;
```

or a bit cleaner, if there are a lot of calls,

```
#define TRY(...) do if (__VA_ARGS__) goto out; while (0)
/* ... */
TRY(zadd(a, b, c));
```

we write

```
jmp_buf env;
if (setjmp(env))
    goto out;
zsetup(env);
/* ... */
zadd(a, b, c);
```

You only need to call `setjmp` and `zsetup` once, but can update the return point by calling them once more.

If you don't need to check for errors, you can disable error detection at compile-time. By defining the `ZAHL_UNSAFE` C preprocessor definition when compiling libzahl, and when compiling your software that uses libzahl.

2.3 Integer structure

The data type used to represent a big integer with libzahl is `z_t`,[1] defined as

```
typedef struct zahl z_t[1];
```

where `struct zahl` is defined as

```
struct zahl {
    int sign;            /* not short for 'signum' */
    size_t used;
    size_t alloced;      /* short for 'allocated' */
    zahl_char_t *chars;  /* short for 'characters' */
};
```

where `zahl_char_t` is defined as

```
typedef uint64_t zahl_char_t;
```

As a user, try not to think about anything else than

```
typedef /* ignore what is here */ z_t[1];
```

details can change in future versions of libzahl.

`z_t` is defined as a single-element array. This is often called a reference, or a call-by-reference. There are some flexibility issues with this, why `struct zahl` has beed added, but for most uses with big integers, it makes things simpler. Particularly, you need not work prepend `&` to variable when making function calls, but the existence of `struct zahl` allows you do so if you so choose.

The `.sign` member, is either -1, 0, or 1, when the integer is negative, zero, or positive, respectively. Whenever `.sign` is 0, the value of `.used` and `.chars` are undefined.

`.used` holds to the number of elements used in `.chars`, and `.alloced` holds the allocation side of `.chars` measured in elements minus a few extra elements that are always added to the allocation. `.chars` is a little-endian array of 64-bit digits, these 64-bit digits are called 'characters' in libzahl. `.chars` holds the absolute value of the represented value.

Unless `.sign` is 0, `.chars` always contains four extra elements, refered to as fluff. These are merely allocated so functions can assume that they can always manipulate groups of four characters, and need not care about cases where the number of characters is not a multiple of four. There are of course a few cases when the precise number of characters is important.

[1] This name actually violates the naming convention; it should be `Z`, or `Zahl` to avoid single-letter names. But this violation is common-place.

2.4 Parameters

The general order of parameters in libzahl functions are: output integers, input integers, input data, output data, parametric values. For example, in addition, the out parameter is the first parameter. But for marshalling and unmarshalling the buffer is last. For random number generation the order is: output, device, distribution, distribution parameters. Whilst the distribution parameters are big integers, they are not considered input integers. The order of the input parameters are that of the order you would write them using mathematical notation, this also holds true if you include the output parameter (as long as there is exactly one output), for example

$$a \leftarrow b^c \mod d$$

is written

```
zmodpow(a, b, c, d);
```

or

```
zmodpowu(a, b, c, d);
```

Like any self respecting bignum library, libzahl supports using the same big integer reference as for output as input, as long as all the output parameters are mutually unique. For example

```
a += b;
```

or

```
a = a + b;
```

is written, using libzahl, as

```
zadd(a, a, b);
```

For commutative functions, like `zadd`, the implementation is optimised to assume that this order is more likely to be used than the alternative. That is, we should, for example, write

```
zadd(a, a, b);
```

rather than

```
zadd(a, b, a);
```

This assumption is not made for non-commutative functions.

When writting your own functions, be aware, input parameters are generally not declared `const` in libzahl. Currently, some functions actually make modifications (that do not affect the value) to input parameters.

Chapter 3

Get started

In this chapter, you will learn the basics of libzahl. You should read the sections in order.

Contents

3.1 Initialisation

Before using libzahl, it must be initialised. When initialising, you must select a location whither libzahl long jumps on error.

```
#include <zahl.h>

int
main(void)
{
    jmp_buf jmpenv;
    if (setjmp(jmpenv))
        return 1; /* Exit on error */
    zsetup(jmpenv);
    /* ... */
    return 0;
}
```

`zsetup` also initialises temporary variables used by libzahl's functions, and constants used by libzahl's functions. Furthermore, it initialises the memory pool and a stack which libzahl uses to keep track of temporary allocations that need to be pooled for use if a function fails.

It is recommended to also uninitialise libzahl when you are done using it, for example before the program exits.

```
int
main(void)
{
    jmp_buf jmpenv;
    if (setjmp(jmpenv))
        return 1; /* Exit on error */
    zsetup(jmpenv);
    /* ... */
    zunsetup();
    return 0;
}
```

`zunsetup` frees all memory that has been reclaimed to the memory pool, and all memory allocated by `zsetup`. Note that this does not free integers that are still in use. It is possible to simply call `zunsetup` directly followed by `zsetup` to free all pooled memory.

3.2 Exceptional conditions

Exceptional conditions, casually called 'errors', are treated in libzahl using long jumps.

```
int
main(int argc, char *argv[])
{
    jmp_buf jmpenv;
    if (setjmp(jmpenv))
        return 1; /* Exit on error */
    zsetup(jmpenv);
    return 0;
}
```

Just exiting on error is not a particularly good idea. Instead, you may want to print an error message. This is done with `zperror`.

```
if (setjmp(jmpenv)) {
    zperror(*argv);
    return 1;
}
```

`zperror` works just like `perror`. It outputs an error description to standard error. A line break is printed at the end of the message. If the argument passed to `zperror` is neither `NULL` nor an empty string, it is printed in front of the description, with a colon and a space separating the passed string and the description. For example, `zperror("my-app")` may output

```
my-app: Cannot allocate memory
```

libzahl also provides `zerror`. Calling this function will provide you with an error code and a textual description.

```
if (setjmp(jmpenv)) {
    const char *description;
    zerror(&description);
    fprintf(stderr, "%s: %s\n", *argv, description);
    return 1;
}
```

This code behaves like the example above that calls `zperror`. If you are interested in the error code, you instead look at the return value.

```
if (setjmp(jmpenv)) {
    enum zerror e = zerror(NULL);
    switch (e) {
    case ZERROR_ERRNO_SET:
        perror("");
        return 1;
    case ZERROR_0_POW_0:
        fprintf(stderr, "Indeterminate form: 0^0\n");
        return 1;
    case ZERROR_0_DIV_0:
        fprintf(stderr, "Indeterminate form: 0/0\n");
        return 1;
    case ZERROR_DIV_0:
        fprintf(stderr, "Do not divide by zero, dummy\n");
        return 1;
    case ZERROR_NEGATIVE:
        fprintf(stderr, "Undefined (negative input)\n");
        return 1;
    case ZERROR_INVALID_RADIX:
        fprintf(stderr, "Radix must be at least 2\n");
        return 1;
    default:
        zperror("");
        return 1;
    }
}
```

To change the point whither libzahl's functions jump, call `setjmp` and `zsetup` again.

```
jmp_buf jmpenv;
if (setjmp(jmpenv)) {
    /* ... */
}
zsetup(jmpenv);
/* ... */
if (setjmp(jmpenv)) {
    /* ... */
}
zsetup(jmpenv);
```

3.3 Create an integer

To do any real work with libzahl, we need integers. The data type for a big integer in libzahl is `z_t` (see Section 2.3 [Integer structure], page 10). Before a `z_t` can be assigned a value, it must be initialised.

```
z_t a;
/* ... */
zsetup(jmpenv);
zinit(a);
/* ... */
zunsetup();
```

`zinit(a)` is actually a less cumbersome and optimised alternative to calling `memset(a, 0, sizeof(z_t))`. It sets the values of two members: `.alloced` and `.chars`, to 0 and `NULL`. This is necessary, otherwise the memory allocated could be fooled to deallocate a false pointer, causing the program to abort.

Once the reference has been initialised, you may assign it a value. The simplest way to do this is by calling

```
void zseti(z_t a, int64_t value);
```

For example `zseti(a, 1)`, assignes the value 1 to the `z_t a`.

When you are done using a big integer reference, you should call `zfree` to let libzahl know that it should pool the allocation of the `.chars` member.

```
z_t a;
zinit(a);
/* ... */
zfree(a); /* before zunsetup */
```

Instead of calling `zfree(a)`, it is possible — but strongly discouraged — to call `free(a->chars)`. Note however, by doing so, the allocation is not pooled for reuse.

If you plan to reuse the variable later, you need to reinitialise it by calling `zinit` again.

Alternatives to `zseti` include (see Section 4.1 [Assignment], page 20):

```
void zsetu(z_t a, uint64_t value);
void zsets(z_t a, const char *value);
void zset(z_t a, z_t value); /* copy value into a */
```

Chapter 4

Miscellaneous

In this chapter, we will learn some miscellaneous functions. It might seem counterintuitive to start with miscellanea, but it is probably a good idea to read this before arithmetics and more advanced topics. You may read Section 4.4 [Marshalling], page 26 later. Before reading this chapter you should have read Chapter 3 [Get started], page 13.

Contents

4.1 Assignment

To be able to do anything useful, we must assign values to integers. There are three functions for this: `zseti`, `zsetu`, and `zsets`. The last letter in the names of these function describe the data type of the input, 'i', 'u', and 's' stand for 'integer', 'unsigned integer', and 'string', respectively. These resemble the rules for the format strings in the family of `printf`-functions. 'Integer' of course refer to 'signed integer'; for integer types in C, part from `char`, the keyword `signed` is implicit.

Consider `zseti`,

```
z_t two;
zinit(two);
zseti(two, 2);
```

assignes `two` the value 2. The data type of the second parameter of `zseti` is `int64_t`. It will accept any integer value in the range $[-2^{63},\ 2^{63}-1] = [-9223372036854775808,\ 9223372036854775807]$, independently of the machine.[1] If this range so not wide enough, it may be possible to use `zsetu`. Its second parameter of the type `uint64_t`, and thus its range is $[0,\ 2^{64}-1] = [0,\ 18446744073709551615]$. If a need negative value is desired, `zsetu` can be combined with `zneg` (see Section 5.6 [Sign manipulation], page 38).

For enormous constants or textual input, `zsets` can be used. `zsets` will accept any numerical value encoded in decimal ASCII, that only contain digits, *not* decimal points, whitespace, apostrophes, et cetera. However, an optional plus sign or, for negative numbers, an ASCII minus sign may be used as the very first character. Note that a proper UCS minus sign is not supported.

Using what we have learned so far, and `zstr` which we will learn about in Section 4.2 [String output], page 23, we can construct a simple program that calculates the sum of a set of numbers.

```
#include <stdio.h>
#include <stdlib.h>
#include <zahl.h>

int
main(int argc, char *argv[]) {
    z_t sum, temp;
    jmp_buf failenv;
    char *sbuf, *argv0 = *argv;
```

[1] `int64_t` is defined to be a signed 64-bit integer using two's complement representation.

```
        if (setjmp(failenv)) {
            zperror(argv0);
            return 1;
        }
        zsetup(failenv);
        zinit(sum);
        zinit(term);
        zsetu(sum, 0);
        for (argv++; *argv; argv++) {
            zsets(term, *argv);
            zadd(sum, sum, term);
        }
        printf("%s\n", (sbuf = zstr(sum, NULL, 0)));
        free(sbuf);
        zfree(sum);
        zfree(term);
        zunsetup();
        return 0;
}
```

Another form of assignment available in libzahl is copy-assignment. This done using `zset`. As easily observable, `zset` is named like `zseti`, `zsetu`, and `zsetu`, but without the input-type suffix. The lack of a input-type suffix means that the input type is `z_t`. `zset` copies value of second parameter into the reference in the first. For example, if `v`, of the type `z_t`, has value 10, then `a` will too after the instruction

```
zset(a, v);
```

`zset` does not necessarily make an exact copy of the input. If, in the example above, the `a->alloced` is greater than or equal to `v->used`, `a->alloced` and `a->chars` are preserved, of course, the content of `a->chars` is overridden. If however, `a->alloced` is less then `v->used`, `a->alloced` is assigned a minimal value at least as great as `v->used` that is a power of 2, and `a->chars` is updated accordingly as described in Section 2.3 [Integer structure], page 10. This of course does not apply if `v` has the value 0; in such cases `a->sign` is simply set to 0.

`zset`, `zseti`, `zsetu`, and `zsets` require that the output-parameter has been initialised with `zinit` or an equally acceptable method as described in Section 3.3 [Create an integer], page 17.

`zset` is often unnecessary, of course there are cases where it is needed. In some case `zswap` is enough, and advantageous. `zswap` is defined as

```
static inline void
zswap(z_t a, z_t b)
{
    z_t t;
    *t = *a;
    *a = *b;
    *b = *t;
}
```

however its implementation is optimised to be around three times as fast. It just swaps the members of the parameters, and thereby the values. There is no rewriting of `.chars` involved; thus it runs in constant time. It also does not require that any argument has been initialised. After the call, `a` will be initialised if and only if `b` was initialised, and vice versa.

4.2 String output

Few useful things can be done without creating textual output of calculations. To convert a z_t to ASCII string in decimal, we use the function zstr, declared as

```
char *zstr(z_t a, char *buf, size_t n);
```

zstr will store the string it creates into buf and return buf. However, if buf is NULL, a new memory segment is allocated and returned. n should be at least the length of the resulting string sans NUL termiantion, but not larger than the allocation size of buf minus 1 byte for NUL termiantion. If buf is NULL, n may be 0. However if buf is not NULL, it is unsafe to let n be 0, unless buf has been allocated by zstr for a value of a at least as larger as the value of a in the new call to zstr. Combining non-NULL buf with 0 n is unsafe because zstr will use a very fast formula for calculating a value that is at least as large as the resulting output length, rather than the exact length.

The length of the string output by zstr can be predicted by zstr_length, decleared as

```
size_t zstr_length(z_t a, unsigned long long int radix);
```

It will calculated the length of a represented in radix radix, sans NUL termination. If radix is 10, the length for a decimal representation is calculated.

Sometimes it is possible to never allocate a buf for zstr. For example, in an implementation of factor, you can reuse the string of the value to factorise, since all of its factors are guaranteed to be no longer than the factored value.

```
void
factor(char *value)
{
    size_t n = strlen(value);
    z_t product, factor;
    zsets(product, value);
    printf("%s:", value);
    while (next_factor(product, factor))
        printf(" %s", zstr(factor, value, n));
    printf("\n");
}
```

Other times it is possible to allocate just once, for example of creating a sorted output. In such cases, the allocation can be done almost transparently.

```
void
output_presorted_decending(z_t *list, size_t n)
{
    char *buf = NULL;
    while (n--)
        printf("%s\n", (buf = zstr(*list++, buf, 0)));
}
```

Note, this example assumes that all values are non-negative.

4.3 Comparison

libzahl defines four functions for comparing integers: `zcmp`, `zcmpi`, `zcmpu`, and `zcmpmag`. These follow the same naming convention as `zset`, `zseti`, and `zsetu`, as described in Section 4.1 [Assignment], page 20. `zcmpmag` compares the absolute value, the magnitude, rather than the proper value. These functions are declared as

```
int zcmp(z_t a, z_t b);
int zcmpi(z_t a, int64_t b);
int zcmpu(z_t a, uint64_t b);
int zcmpmag(z_t a, z_t b);
```

They behave similar to `memcmp` and `strcmp`.[2] The return value is defined

$$\operatorname{sgn}(a-b)=\begin{cases}-1 & \text{if } a<b\\ 0 & \text{if } a=b\\ +1 & \text{if } a>b\end{cases}$$

for `zcmp`, `zcmpi`, and `zcmpu`. The return for `zcmpmag` value is defined

$$\operatorname{sgn}(|a|-|b|)=\begin{cases}-1 & \text{if } |a|<|b|\\ 0 & \text{if } |a|=|b|\\ +1 & \text{if } |a|>|b|\end{cases}$$

It is discouraged, stylistically, to compare against -1 and $+1$, rather, you should always compare against 0. Think of it as returning $a-b$, or $|a|-|b|$ in the case of `zcmpmag`.

[2] And `wmemcmp` and `wcscmp` if you are into that mess.

4.4 Marshalling

libzahl is designed to provide efficient communication for multi-processes applications, including running on multiple nodes on a cluster computer. However, these facilities require that it is known that all processes run the same version of libzahl, and run on compatible microarchitectures, that is, the processors must have endianness, and the intrinsic integer types in C must have the same widths on all processors. When this is not the case, string conversion can be used (see Section 4.1 [Assignment], page 20 and Section 4.2 [String output], page 23), but when it is the case `zsave` and `zload` can be used. `zsave` and `zload` are declared as

```
size_t zsave(z_t a, char *buf);
size_t zload(z_t a, const char *buf);
```

`zsave` stores a version- and microarchitecture-dependent binary representation of `a` in `buf`, and returns the number of bytes written to `buf`. If `buf` is `NULL`, the numbers bytes that would have be written is returned. `zload` unmarshals an integers from `buf`, created with `zsave`, into `a`, and returns the number of read bytes. `zload` returns the value returned by `zsave`.

Chapter 5

Arithmetic

In this chapter, we will learn how to perform basic arithmetic with libzahl: addition, subtraction, multiplication, division, modulus, exponentiation, and sign manipulation. Section 5.4 [Division], page 32 is of special importance.

Contents

5.1 Addition

To calculate the sum of two terms, we perform addition using `zadd`.

$r \leftarrow a + b$

is written as

```
zadd(r, a, b);
```

libzahl also provides `zadd_unsigned` which has slightly lower overhead. The calculates the sum of the absolute values of two integers.

$r \leftarrow |a| + |b|$

is written as

```
zadd_unsigned(r, a, b);
```

`zadd_unsigned` has lower overhead than `zadd` because it does not need to inspect or change the sign of the input, the low-level function that performs the addition inherently calculates the sum of the absolute values of the input.

In libzahl, addition is implemented using a technique called ripple-carry. It is derived from that observation that

$f : \mathbf{Z}_n, \mathbf{Z}_n \rightarrow \mathbf{Z}_n$
$f : a, b \mapsto a + b + 1$

only wraps at most once, that is, the carry cannot exceed 1. CPU:s provide an instruction specifically for performing addition with ripple-carry over multiple words, adds twos numbers plus the carry from the last addition. libzahl uses assembly to implement this efficiently. If, however, an assembly implementation is not available for the on which machine it is running, libzahl implements ripple-carry less efficiently using compiler extensions that check for overflow. In the event that neither an assembly implementation is available nor the compiler is known to support this extension, it is implemented using inefficient pure C code. This last resort manually predicts whether an addition will overflow; this could be made more efficient, by never using the highest bit in each character, except to detect overflow. This optimisation is however not implemented because it is not deemed important enough and would be detrimental to libzahl's simplicity.

`zadd` and `zadd_unsigned` support in-place operation:

```
zadd(a, a, b);
zadd(b, a, b);           /* should be avoided */
zadd_unsigned(a, a, b);
zadd_unsigned(b, a, b);  /* should be avoided */
```

Use this whenever possible, it will improve your performance, as it will involve less CPU instructions for each character addition and it may be possible to eliminate some character additions.

5.2 Subtraction

TODO

5.3 Multiplication

TODO

5.4 Division

To calculate the quotient or modulus of two integers, use either of

```
void zdiv(z_t quotient, z_t dividend, z_t divisor);
void zmod(z_t remainder, z_t dividend, z_t divisor);
void zdivmod(z_t quotient, z_t remainder,
             z_t dividend, z_t divisor);
```

These function *do not* allow `NULL` for the output parameters: `quotient` and `remainder`. The quotient and remainder are calculated simultaneously and indivisibly, hence `zdivmod` is provided to calculated both; if you are only interested in the quotient or only interested in the remainder, use `zdiv` or `zmod`, respectively.

These functions calculate a truncated quotient. That is, the result is rounded towards zero. This means for example that if the quotient is in $(-1,\ 1)$, `quotient` gets 0. That is, this would not be the case for one of the sides of zero. For example, if the quotient would have been floored, negative quotients would have been rounded away from zero. libzahl only provides truncated division.

The remainder is defined such that $n = qd + r$ after calling `zdivmod(q, r, n, d)`. There is no difference in the remainer between `zdivmod` and `zmod`. The sign of `d` has no affect on `r`, `r` will always, unless it is zero, have the same sign as `n`.

There are of course other ways to define integer division (that is, $\mathbf{Z}$ being the codomain) than as truncated division. For example integer divison in Python is floored — yes, you did just read 'integer divison in Python is floored,' and you are correct, that is not the case in for example C. Users that want another definition for division than truncated division are required to implement that themselves. We will however lend you a hand.

```
#define isneg(x) (zsignum(x) < 0)
static z_t one;
__attribute__((constructor)) static
void init(void) { zinit(one), zseti(one, 1); }

static int
cmpmag_2a_b(z_t a, z_b b)
{
    int r;
    zadd(a, a, a), r = zcmpmag(a, b), zrsh(a, a, 1);
    return r;
}
```

```
void /* All arguments must be unique. */
divmod_floor(z_t q, z_t r, z_t n, z_t d)
{
    zdivmod(q, r, n, d);
    if (!zzero(r) && isneg(n) != isneg(d))
        zsub(q, q, one), zadd(r, r, d);
}

void /* All arguments must be unique. */
divmod_ceiling(z_t q, z_t r, z_t n, z_t d)
{
    zdivmod(q, r, n, d);
    if (!zzero(r) && isneg(n) == isneg(d))
        zadd(q, q, one), zsub(r, r, d);
}

/* This is how we normally round numbers. */
void /* All arguments must be unique. */
divmod_half_from_zero(z_t q, z_t r, z_t n, z_t d)
{
    zdivmod(q, r, n, d);
    if (!zzero(r) && cmpmag_2a_b(r, d) >= 0) {
        if (isneg(n) == isneg(d))
            zadd(q, q, one), zsub(r, r, d);
        else
            zsub(q, q, one), zadd(r, r, d);
    }
}
```

Now to the weird ones that will more often than not award you a face-slap.

```
void /* All arguments must be unique. */
divmod_half_to_zero(z_t q, z_t r, z_t n, z_t d)
{
    zdivmod(q, r, n, d);
    if (!zzero(r) && cmpmag_2a_b(r, d) > 0) {
        if (isneg(n) == isneg(d))
            zadd(q, q, one), zsub(r, r, d);
        else
            zsub(q, q, one), zadd(r, r, d);
    }
}
```

```
void /* All arguments must be unique. */
divmod_half_up(z_t q, z_t r, z_t n, z_t d)
{
    int cmp;
    zdivmod(q, r, n, d);
    if (!zzero(r) && (cmp = cmpmag_2a_b(r, d)) >= 0) {
        if (isneg(n) == isneg(d))
            zadd(q, q, one), zsub(r, r, d);
        else if (cmp)
            zsub(q, q, one), zadd(r, r, d);
    }
}

void /* All arguments must be unique. */
divmod_half_down(z_t q, z_t r, z_t n, z_t d)
{
    int cmp;
    zdivmod(q, r, n, d);
    if (!zzero(r) && (cmp = cmpmag_2a_b(r, d)) >= 0) {
        if (isneg(n) != isneg(d))
            zsub(q, q, one), zadd(r, r, d);
        else if (cmp)
            zadd(q, q, one), zsub(r, r, d);
    }
}

void /* All arguments must be unique. */
divmod_half_to_even(z_t q, z_t r, z_t n, z_t d)
{
    int cmp;
    zdivmod(q, r, n, d);
    if (!zzero(r) && (cmp = cmpmag_2a_b(r, d)) >= 0) {
        if (cmp || zodd(q)) {
            if (isneg(n) != isneg(d))
                zsub(q, q, one), zadd(r, r, d);
            else
                zadd(q, q, one), zsub(r, r, d);
        }
    }
}
```

```
void /* All arguments must be unique. */
divmod_half_to_odd(z_t q, z_t r, z_t n, z_t d)
{
    int cmp;
    zdivmod(q, r, n, d);
    if (!zzero(r) && (cmp = cmpmag_2a_b(r, d)) >= 0) {
        if (cmp || zeven(q)) {
            if (isneg(n) != isneg(d))
                zsub(q, q, one), zadd(r, r, d);
            else
                zadd(q, q, one), zsub(r, r, d);
        }
    }
}
```

Currently, libzahl uses an almost trivial division algorithm. It operates on positive numbers. It begins by left-shifting the divisor as much as possible with letting it exceed the dividend. Then, it subtracts the shifted divisor from the dividend and add 1, left-shifted as much as the divisor, to the quotient. The quotient begins at 0. It then right-shifts the shifted divisor as little as possible until it no longer exceeds the diminished dividend and marks the shift in the quotient. This process is repeated until the unshifted divisor is greater than the diminished dividend. The final diminished dividend is the remainder.

5.5 Exponentiation

Exponentiation refers to raising a number to a power. libzahl provides two functions for regular exponentiation, and two functions for modular exponentiation. libzahl also provides a function for raising a number to the second power, see Section 5.3 [Multiplication], page 31 for more details on this. The functions for regular exponentiation are

```
void zpow(z_t power, z_t base, z_t exponent);
void zpowu(z_t, z_t, unsigned long long int);
```

They are identical, except `zpowu` expects an intrinsic type as the exponent. Both functions calculate

$$power \leftarrow base^{exponent}$$

The functions for modular exponentiation are

```
void zmodpow(z_t, z_t, z_t, z_t modulator);
void zmodpowu(z_t, z_t, unsigned long long int, z_t);
```

They are identical, except `zmodpowu` expects and intrinsic type as the exponent. Both functions calculate

$$power \leftarrow base^{exponent} \mod modulator$$

The sign of `modulator` does not affect the result, `power` will be negative if and only if `base` is negative and `exponent` is odd, that is, under the same circumstances as for `zpow` and `zpowu`.

These four functions are implemented using exponentiation by squaring. `zmodpow` and `zmodpowu` are optimised, they modulate results for multiplication and squaring at every multiplication and squaring, rather than modulating the result at the end. Exponentiation by modulation is a very simple algorithm which can be expressed as a simple formula

$$a^b = \prod_{k \in \mathbf{Z}_+ \,:\, \left\lfloor \frac{b}{2^k} \bmod 2 \right\rfloor = 1} a^{2^k}$$

This is a natural extension to the observations[1]

$$\forall b \in \mathbf{Z}_+ \exists B \subset \mathbf{Z}_+ : b = \sum_{i \in B} 2^i \quad \text{and} \quad a^{\sum x} = \prod a^x.$$

[1] The first of course being that any non-negative number can be expressed with the binary positional system. The latter should be fairly self-explanatory.

The algorithm can be expressed in psuedocode as

$r, f \leftarrow 1, a$
while $b \neq 0$ **do**
  $r \leftarrow r \cdot f$ **unless** $2|b$
  $f \leftarrow f^2$ $\{f \leftarrow f \cdot f\}$
  $b \leftarrow \lfloor b/2 \rfloor$
end while
return r

Modular exponentiation ($a^b \mod m$) by squaring can be expressed as

$r, f \leftarrow 1, a$
while $b \neq 0$ **do**
  $r \leftarrow r \cdot f \mod m$ **unless** $2|b$
  $f \leftarrow f^2 \mod m$
  $b \leftarrow \lfloor b/2 \rfloor$
end while
return r

`zmodpow` does *not* calculate the modular inverse if the exponent is negative, rather, you should expect the result to be 1 and 0 depending of whether the base is 1 or not 1.

5.6 Sign manipulation

libzahl provides two functions for manipulating the sign of integers:

```
void zabs(z_t r, z_t a);
void zneg(z_t r, z_t a);
```

`zabs` stores the absolute value of `a` in `r`, that is, it creates a copy of `a` to `r`, unless `a` and `r` are the same reference, and then removes its sign; if the value is negative, it becomes positive.

$$r \leftarrow |a| = \begin{cases} -a & \text{if } a \le 0 \\ +a & \text{if } a \ge 0 \end{cases}$$

`zneg` stores the negated of `a` in `r`, that is, it creates a copy of `a` to `r`, unless `a` and `r` are the same reference, and then flips sign; if the value is negative, it becomes positive, if the value is positive, it becomes negative.

$$r \leftarrow -a$$

Note that there is no function for

$$r \leftarrow -|a| = \begin{cases} a & \text{if } a \le 0 \\ -a & \text{if } a \ge 0 \end{cases}$$

calling `zabs` followed by `zneg` should be sufficient for most users:

```
#define my_negabs(r, a)  (zabs(r, a), zneg(r, r))
```

Chapter 6

Bit operations

libzahl provides a number of functions that operate on bits. These can sometimes be used instead of arithmetic functions for increased performance. You should read the sections in order.

Contents

6.1 Boundary

To retrieve the index of the lowest set bit, use

```
size_t zlsb(z_t a);
```

It will return a zero-based index, that is, if the least significant bit is indeed set, it will return 0.

If a is a power of 2, it will return the power of which 2 is raised, effectively calculating the binary logarithm of a. Note, this is only if a is a power of two. More generally, it returns the number of trailing binary zeroes, if equivalently the number of times a can evenly be divided by 2. However, in the special case where $a = 0$, SIZE_MAX is returned.

A similar function is

```
size_t zbit(z_t a);
```

It returns the minimal number of bits require to represent an integer. That is, $\lceil \log_2 a \rceil - 1$, or equivalently, the number of times a can be divided by 2 before it gets the value 0. However, in the special case where $a = 0$, 1 is returned. 0 is never returned. If you want the value 0 to be returned if $a = 0$, write

```
zzero(a) ? 0 : zbits(a)
```

The definition "it returns the minimal number of bits required to represent an integer," holds true if $a = 0$, the other divisions do not hold true if $a = 0$.

6.2 Shift

There are two functions for shifting bits in integers:

```
void zlsh(z_t r, z_t a, size_t b);
void zrsh(z_t r, z_t a, size_t b);
```

`zlsh` performs a left-shift, and `zrsh` performs a right-shift. That is, `zlsh` adds `b` trailing binary zeroes, and `zrsh` removes the lowest `b` binary digits. So if

$a = 10000101_2$ then
$r = 1000010100_2$ after calling `zlsh(r, a, 2)`, and
$r = 100001_2$ after calling `zrsh(r, a, 2)`.

`zlsh(r, a, b)` is equivalent to $r \leftarrow a \cdot 2^b$, and `zrsh(r, a, b)` is equivalent to $r \leftarrow a \div 2^b$, with truncated division, `zlsh` and `zrsh` are significantly faster than `zpowu` and should be used whenever possible. `zpowu` does not check if it is possible for it to use `zlsh` instead, even if it would, `zlsh` and `zrsh` would still be preferable in most cases because it removes the need for `zmul` and `zdiv`, respectively.

`zlsh` and `zrsh` are implemented in two steps: (1) shift whole characters, that is, groups of aligned 64 bits, and (2) shift on a bit-level between characters.

If you are implementing a calculator, you may want to create a wrapper for `zpow` that uses `zlsh` whenever possible. One such wrapper could be

```
void
pow(z_t r, z_t a, z_t b)
{
    size_t s1, s2;
    if ((s1 = zlsb(a)) + 1 == zbits(a) &&
                  zbits(b) <= 8 * sizeof(SIZE_MAX)) {
        s2 = zzero(b) ? 0 : b->chars[0];
        if (s1 <= SIZE_MAX / s2) {
            zsetu(r, 1);
            zlsh(r, r, s1 * s2);
            return;
        }
    }
    zpow(r, a, b);
}
```

6.3 Truncation

In Section 6.2 [Shift], page 41 we have seen how bit-shift operations can be used to multiply or divide by a power of two. There is also a bit-truncation operation: `ztrunc`, which is used to keep only the lowest bits, or equivalently, calculate the remainder of a division by a power of two.

```
void ztrunc(z_t r, z_t a, size_t b);
```

is consistent with `zmod`; like `zlsh` and `zrsh`, `a`'s sign is preserved into `r` assuming the result is non-zero.

`ztrunc(r, a, b)` stores only the lowest `b` bits in `a` into `r`, or equivalently, calculates $r \leftarrow a \mod 2^b$. For example, if

$a = 100011000_2$ then

$r = 1000_2$ after calling `ztrunc(r, a, 4)`.

6.4 Split

In Section 6.2 [Shift], page 41 and Section 6.3 [Truncation], page 42 we have seen how bit operations can be used to calculate division by a power of two and modulus a power of two efficiently using bit-shift and bit-truncation operations. libzahl also has a bit-split operation that can be used to efficiently calculate both division and modulus a power of two efficiently in the same operation, or equivalently, storing low bits in one integer and high bits in another integer. This function is

```
void zsplit(z_t high, z_t low, z_t a, size_t b);
```

Unlike `zdivmod`, it is not more efficient than calling `zrsh` and `ztrunc`, but it is more convenient. `zsplit` requires that `high` and `low` are from each other distinct references.

Calling `zsplit(high, low, a, b)` is equivalent to

```
ztrunc(low, a, delim);
zrsh(high, a, delim);
```

assuming `a` and `low` are not the same reference (reverse the order of the functions if they are the same reference.)

`zsplit` copies the lowest `b` bits of `a` to `low`, and the rest of the bits to `high`, with the lowest `b` removesd. For example, if $a = 1010101111_2$, then $high = 101010_2$ and $low = 1111_2$ after calling `zsplit(high, low, a, 4)`.

`zsplit` is especially useful in divide-and-conquer algorithms.

6.5 Bit manipulation

The function

```
void zbset(z_t r, z_t a, size_t bit, int mode);
```

is used to manipulate single bits in `a`. It will copy `a` into `r` and then, in `r`, either set, clear, or flip, the bit with the index `bit` — the least significant bit has the index 0. The action depend on the value of `mode`:

- $mode > 0$ (+1): set
- $mode = 0$ (0): clear
- $mode < 0$ (−1): flip

6.6 Bit test

libzahl provides a function for testing whether a bit in a big integer is set:

```
int zbtest(z_t a, size_t bit);
```

it will return 1 if the bit with the index `bit` is set in `a`, counting from the least significant bit, starting at zero. 0 is returned otherwise. The sign of `a` is ignored.

We can think of this like so: consider

$$|a| = \sum_{i=0}^{\infty} k_i 2^i, \ \ k_i \in \{0, 1\},$$

`zbtest(a, b)` returns k_b. Equivalently, we can think that `zbtest(a, b)` return whether $b \in B$ where B is defined by

$$|a| = \sum_{b \in B} 2^b, \ \ B \subset \mathbf{Z}_+,$$

or as right-shifting a by b bits and returning whether the least significant bit is set.

`zbtest` always returns 1 or 0, but for good code quality, you should avoid testing against 1, rather you should test whether the value is a truth-value or a falsehood-value. However, there is nothing wrong with depending on the value being restricted to being either 1 or 0 if you want to sum up returned values or otherwise use them in new values.

6.7 Connectives

libzahl implements the four basic logical connectives: and, or, exclusive or, and not. The functions for these are named `zand`, `zor`, `zxor`, and `znot`, respectively.

The connectives apply to each bit in the integers, as well as the sign. The sign is treated as a bit that is set if the integer is negative, and as cleared otherwise. For example (integers are in binary):

```
zand(r, a, b)               zor(r, a, b)
a = +1010  (input)          a = +1010  (input)
b = -1100  (input)          b = -1100  (input)
r = +1000  (output)         r = -1110  (output)

zxor(r, a, b)               znot(r, a)
a = +1010  (input)          a = +1010  (input)
b = -1100  (input)          r = -0101  (output)
r = -0110  (output)
```

Remember, in libzahl, integers are represented with sign and magnitude, not two's complement, even when using these connectives. Therefore, more work than just changing the name of the called function may be required when moving between big integer libraries. Consequently, `znot` does not flip bits that are higher than the highest set bit, which means that `znot` is nilpotent rather than self dual.

Below is a list of the value of `a` when `znot(a, a)` is called repeatedly.

```
10101010
-1010101
  101010
  -10101
    1010
    -101
      10
      -1
       0
       0
       0
```

Chapter 7

Number theory

In this chapter, you will learn about the number theoretic functions in libzahl.

Contents

7.1 Odd or even

There are four functions available for testing the oddness and evenness of an integer:

```
int zodd(z_t a);
int zeven(z_t a);
int zodd_nonzero(z_t a);
int zeven_nonzero(z_t a);
```

`zodd` returns 1 if `a` contains an odd value, or 0 if `a` contains an even number. Conversely, `zeven` returns 1 if `a` contains an even value, or 0 if `a` contains an odd number. `zodd_nonzero` and `zeven_nonzero` behave exactly like `zodd` and `zeven`, respectively, but assumes that `a` contains a non-zero value, if not undefined behaviour is invoked, possibly in the form of a segmentation fault; they are thus sligtly faster than `zodd` and `zeven`.

It is discouraged to test the returned value against 1, we should always test against 0, treating all non-zero value as equivalent to 1. For clarity, we use also avoid testing that the returned value is zero, for example, rather than `!zeven(a)` we write `zodd(a)`.

7.2 Signum

There are two functions available for testing the sign of an integer, one of the can be used to retrieve the sign:

```
int zsignum(z_t a);
int zzero(z_t a);
```

`zsignum` returns -1 if $a < 0$, 0 if $a = 0$, and $+1$ if $a > 0$, that is,

$$\text{sgn}\ a = \begin{cases} -1 & \text{if } a < 0 \\ 0 & \text{if } a = 0 \\ +1 & \text{if } a > 0 \end{cases}$$

It is discouraged to compare the returned value against -1 and $+1$; always compare against 0, for example:

```
if (zsignum(a) >  0)  "positive";
if (zsignum(a) >= 0)  "non-negative";
if (zsignum(a) == 0)  "zero";
if (!zsignum(a))      "zero";
if (zsignum(a) <= 0)  "non-positive";
if (zsignum(a) <  0)  "negative";
if (zsignum(a))       "non-zero";
```

However, when we are doing arithmetic with the signum, we may relay on the result never being any other value than -1, 0, and $+0$. For example:

```
zset(sgn, zsignum(a));
zadd(b, sgn);
```

`zzero` returns 0 if $a = 0$ or 1 if $a \neq 0$. Like with `zsignum`, avoid testing the returned value against 1, rather test that the returned value is not 0. When however we are doing arithmetic with the result, we may relay on the result never being any other value than 0 or 1.

7.3 Greatest common divisor

There is no single agreed upon definition for the greatest common divisor of two integer, that cover non-positive integers. In libzahl we define it as

$$\gcd(a,b) = \begin{cases} -k & \text{if } a < 0, b < 0 \\ b & \text{if } a = 0 \\ a & \text{if } b = 0 \\ k & \text{otherwise} \end{cases},$$

where k is the largest integer that divides both $|a|$ and $|b|$. This definion ensures

$$\frac{a}{\gcd(a,b)} \begin{cases} > 0 & \text{if } a < 0, b < 0 \\ < 0 & \text{if } a < 0, b > 0 \\ = 1 & \text{if } b = 0, a \neq 0 \\ = 0 & \text{if } a = 0, b \neq 0 \\ \in \mathbf{N} & \text{otherwise if } a \neq 0, b \neq 0 \end{cases},$$

and analogously for $\frac{b}{\gcd(a,b)}$. Note however, the convension $\gcd(0,0) = 0$ is adhered. Therefore, before dividing with $\gcd(a,b)$ you may want to check whether $\gcd(a,b) = 0$. $\gcd(a,b)$ is calculated with `zgcd(a, b)`.

`zgcd` calculates the greatest common divisor using the Binary GCD algorithm.

return $a + b$ **if** $ab = 0$
return $-\gcd(|a|,|b|)$ **if** $a < 0$ **and** $b < 0$
$s \leftarrow \max s : 2^s|a,b$
$u, v \leftarrow |a| \div 2^s, |b| \div 2^s$
while $u \neq v$ **do**
  $v \leftrightarrow u$ **if** $v < u$
  $v \leftarrow v - u$
  $v \leftarrow v \div 2^x$, where $x = \max x : 2^x|v$
end while
return $u \cdot 2^s$

$\max x : 2^x|z$ is returned by `zlsb(z)` (see Section 6.1 [Boundary], page 40).

7.4 Primality test

The primality of an integer can be tested with

```
enum zprimality zptest(z_t w, z_t a, int t);
```

`zptest` uses Miller–Rabin primality test, with `t` runs of its witness loop, to determine whether `a` is prime. `zptest` returns either

- `PRIME` = 2: `a` is prime. This is only returned for known prime numbers: 2 and 3.
- `PROBABLY_PRIME` = 1: `a` is probably a prime. The certainty will be $1 - 4^{-t}$.
- `NONPRIME` = 0: `a` is either composite, non-positive, or 1. It is certain that `a` is not prime.

If and only if `NONPRIME` is returned, a value will be assigned to `w` — unless `w` is `NULL`. This will be the witness of `a`'s completeness. If $a \le 2$, it is not really composite, and the value of `a` is copied into `w`.

$\gcd(w, a)$ can be used to extract a factor of a. This factor is however not necessarily, and unlikely so, prime, but can be composite, or even 1. In the latter case this becomes utterly useless. Therefore using this method for prime factorisation is a bad idea.

Below is pseudocode for the Miller–Rabin primality test with witness return.

return NONPRIME ($w \leftarrow a$) **if** $a \le 1$
return PRIME **if** $a \le 3$
return NONPRIME ($w \leftarrow 2$) **if** $2|a$
$r \leftarrow \max r : 2^r|(a-1)$
$d \leftarrow (a-1) \div 2^r$
repeat t **times**
  $k \xleftarrow{\$} \mathbf{Z}_{a-2} \setminus \mathbf{Z}_2$ {Uniformly random assignment.}
  $x \leftarrow k^d \mod a$
  continue if $x = 1$ **or** $x = a - 1$
  repeat r **times or until** $x = 1$ **or** $x = a - 1$
    $x \leftarrow x^2 \mod a$
  end repeat
  if $x = 1$ **return** NONPRIME ($w \leftarrow k$)
end repeat
return PROBABLY PRIME

$\max x : 2^x|z$ is returned by `zlsb(z)` (see Section 6.1 [Boundary], page 40).

Chapter 8

Random numbers

TODO

Contents

8.1 Generation

TODO

8.2 Devices

TODO

8.3 Distributions

TODO

Chapter 9

Not implemented

In this chapter we maintain a list of features we have chosen not to implement, but would fit into libzahl, had we not have our priorities straight. Functions listed herein will only be implemented if it is shown that it would be overwhelmingly advantageous. For each feature, a sample implementation or a mathematical expression on which you can base your implementation is included. The sample implementations create temporary integer references to simplify the examples. You should try to use dedicated variables; in case of recursion, a robust program should store temporary variables on a stack, so they can be cleaned up if something happens.

Research problems, like prime factorisation and discrete logarithms, do not fit in the scope of bignum libraries and therefore do not fit into libzahl, and will not be included in this chapter. Operators and functions that grow so ridiculously fast that a tiny lookup table constructed to cover all practical input will also not be included in this chapter, nor in libzahl.

Contents

9.1 Extended greatest common divisor

```
void
extgcd(z_t bézout_coeff_1, z_t bézout_coeff_2, z_t gcd
       z_t quotient_1, z_t quotient_2, z_t a, z_t b)
{
#define old_r gcd
#define old_s bézout_coeff_1
#define old_t bézout_coeff_2
#define s quotient_2
#define t quotient_1
    z_t r, q, qs, qt;
    int odd = 0;
    zinit(r), zinit(q), zinit(qs), zinit(qt);
    zset(r, b), zset(old_r, a);
    zseti(s, 0), zseti(old_s, 1);
    zseti(t, 1), zseti(old_t, 0);
    while (!zzero(r)) {
        odd ^= 1;
        zdivmod(q, old_r, old_r, r), zswap(old_r, r);
        zmul(qs, q, s), zsub(old_s, old_s, qs);
        zmul(qt, q, t), zsub(old_t, old_t, qt);
        zswap(old_s, s), zswap(old_t, t);
    }
    odd ? abs(s, s) : abs(t, t);
    zfree(r), zfree(q), zfree(qs), zfree(qt);
}
```

Perhaps you are asking yourself "wait a minute, doesn't the extended Euclidean algorithm only have three outputs if you include the greatest common divisor, what is this shenanigans?" No[1], it has five outputs, most implementations just ignore two of them. If this confuses you, or you want to know more about this, I refer you to Wikipeida.

[1]Well, technically yes, but it calculates two values for free in the same ways as division calculates the remainder for free.

9.2 Least common multiple

$\mathrm{lcm}(a,b) = \dfrac{|a \cdot b|}{\gcd(a,b)}$

$\mathrm{lcm}(a,b)$ is undefined when a or b is zero, because division by zero is undefined. Note however that $\gcd(a,b)$ is only zero when both a and b is zero.

9.3 Modular multiplicative inverse

```
int
modinv(z_t inv, z_t a, z_t m)
{
    z_t x, _1, _2, _3, gcd, mabs, apos;
    int invertible, aneg = zsignum(a) < 0;
    zinit(x), zinit(_1), zinit(_2), zinit(_3), zinit(gcd);
    *mabs = *m;
    zabs(mabs, mabs);
    if (aneg) {
        zinit(apos);
        zset(apos, a);
        if (zcmpmag(apos, mabs))
            zmod(apos, apos, mabs);
        zadd(apos, apos, mabs);
    }
    extgcd(inv, _1, _2, _3, gcd, apos, mabs);
    if ((invertible = !zcmpi(gcd, 1))) {
        if (zsignum(inv) < 0)
            (zsignum(m) < 0 ? zsub : zadd)(x, x, m);
        zswap(x, inv);
    }
    if (aneg)
        zfree(apos);
    zfree(x), zfree(_1), zfree(_2), zfree(_3), zfree(gcd);
    return invertible;
}
```

9.4 Random prime number generation

TODO

9.5 Symbols

9.5.1 Legendre symbol

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod p, \ \left(\frac{a}{p}\right) \in \{-1,\ 0,\ 1\},\ p \in \mathbf{P},\ p > 2$$

That is, unless $a^{\frac{p-1}{2}} \mod p \le 1$, $a^{\frac{p-1}{2}} \mod p = p-1$, so $\left(\frac{a}{p}\right) = -1$.

It should be noted that $\left(\frac{a}{p}\right) = \left(\frac{a \text{ Mod } p}{p}\right)$, so a compressed lookup table can be used for small p.

9.5.2 Jacobi symbol

$$\left(\frac{a}{n}\right) = \prod_k \left(\frac{a}{p_k}\right)^{n_k}, \text{ where } n = \prod_k p_k^{n_k} > 0, \text{ and } p_k \in \mathbf{P}.$$

Like the Legendre symbol, the Jacobi symbol is n-periodic over a. If n, is prime, the Jacobi symbol is the Legendre symbol, but the Jacobi symbol is defined for all odd numbers n, where the Legendre symbol is defined only for odd primes n.

Use the following algorithm to calculate the Jacobi symbol:

$a \leftarrow a \mod n$
$r \leftarrow \text{lsb } a$
$a \leftarrow a \cdot 2^{-r}$
$r \leftarrow \begin{cases} 1 & \text{if } n \equiv 1,7 \pmod 8 \text{ or } r \equiv 0 \pmod 2 \\ -1 & \text{otherwise} \end{cases}$
if $a = 1$ **then**
  return r
else if $\gcd(a,n) \ne 1$ **then**
  return 0
end if
$(a,n) = (n,a)$
start over

9.5.3 Kronecker symbol

The Kronecker symbol $\left(\frac{a}{n}\right)$ is a generalisation of the Jacobi symbol, where n can be any integer. For positive odd n, the Kronecker symbol is equal to the Jacobi symbol. For even n, the Kronecker symbol is $2n$-periodic over a, the Kronecker symbol is zero for all (a,n) with both a and n are even.

$$\left(\frac{a}{2^k \cdot n}\right) = \left(\frac{a}{n}\right) \cdot \left(\frac{a}{2}\right)^k, \text{ where } \left(\frac{a}{2}\right) = \begin{cases} 1 & \text{if } a \equiv 1, 7 \pmod 8 \\ -1 & \text{if } a \equiv 3, 5 \pmod 8 \\ 0 & \text{otherwise} \end{cases}$$

$$\left(\frac{-a}{n}\right) = \left(\frac{a}{n}\right) \cdot \left(\frac{a}{-1}\right), \text{ where } \left(\frac{a}{-1}\right) = \begin{cases} 1 & \text{if } a \geq 0 \\ -1 & \text{if } a < 0 \end{cases}$$

However, for $n = 0$, the symbol is defined as

$$\left(\frac{a}{0}\right) = \begin{cases} 1 & \text{if } a = \pm 1 \\ 0 & \text{otherwise.} \end{cases}$$

9.5.4 Power residue symbol

TODO

9.5.5 Pochhammer k-symbol

$$(x)_{n,k} = \prod_{i=1}^{n} (x + (i-1)k)$$

9.6 Logarithm

TODO

9.7 Roots

TODO

9.8 Modular roots

TODO

9.9 Combinatorial

9.9.1 Factorial

$$n! = \begin{cases} \prod\limits_{i=1}^{n} i & \text{if } n \ge 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

This can be implemented much more efficiently than using the naïve method, and is a very important function for many combinatorial applications, therefore it may be implemented in the future if the demand is high enough.

An efficient, yet not optimal, implementation of factorials that about halves the number of required multiplications compared to the naïve method can be derived from the observation

$$n! = n!!\ \lfloor n/2 \rfloor!\ 2^{\lfloor n/2 \rfloor},\ n \text{ odd}.$$

The resulting algorithm can be expressed as

```
void
fact(z_t r, uint64_t n)
{
    z_t p, f, two;
    uint64_t *ns, s = 1, i = 1;
    zinit(p), zinit(f), zinit(two);
    zseti(r, 1), zseti(p, 1), zseti(f, n), zseti(two, 2);
    ns = alloca(zbits(f) * sizeof(*ns));
    while (n > 1) {
        if (n & 1) {
            ns[i++] = n;
            s += n >>= 1;
        } else {
            zmul(r, r, (zsetu(f, n), f));
            n -= 1;
        }
    }
    for (zseti(f, 1); i-- > 0; zmul(r, r, p);)
        for (n = ns[i]; zcmpu(f, n); zadd(f, f, two))
            zmul(p, p, f);
    zlsh(r, r, s);
    zfree(two), zfree(f), zfree(p);
}
```

9.9.2 Subfactorial

$$!n = \begin{cases} n(!(n-1)) + (-1)^n & \text{if } n > 0 \\ 1 & \text{if } n = 0 \\ \text{undefined} & \text{otherwise} \end{cases} = n! \sum_{i=0}^{n} \frac{(-1)^i}{i!}$$

9.9.3 Alternating factorial

$$\operatorname{af}(n) = \sum_{i=1}^{n} (-1)^{n-i} i!$$

9.9.4 Multifactorial

$$n!^{(k)} = \begin{cases} 1 & \text{if } n = 0 \\ n & \text{if } 0 < n \le k \\ n((n-k)!^{(k)}) & \text{if } n > k \\ \text{undefined} & \text{otherwise} \end{cases}$$

9.9.5 Quadruple factorial

$(4n-2)!^{(4)}$

9.9.6 Superfactorial

$\operatorname{sf}(n) = \prod_{k=1}^{n} k^{1+n-k}$, undefined for $n < 0$.

9.9.7 Hyperfactorial

$H(n) = \prod_{k=1}^{n} k^k$, undefined for $n < 0$.

9.9.8 Raising factorial

$x^{(n)} = \dfrac{(x+n-1)!}{(x-1)!}$, undefined for $n < 0$.

9.9.9 Falling factorial

$(x)_n = \dfrac{x!}{(x-n)!}$, undefined for $n < 0$.

9.9.10 Primorial

$$n\# = \prod_{\{i \in \mathbf{P} \,:\, i \leq n\}} i$$

$$p_n\# = \prod_{i \in \mathbf{P}_{\pi(n)}} i$$

9.9.11 Gamma function

$\Gamma(n) = (n-1)!$, undefined for $n \leq 0$.

9.9.12 K-function

$$K(n) = \begin{cases} \prod\limits_{i=1}^{n-1} i^i & \text{if } n \geq 0 \\ 1 & \text{if } n = -1 \\ 0 & \text{otherwise (result is truncated)} \end{cases}$$

9.9.13 Binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{1}{(n-k)!} \prod_{i=k+1}^{n} i = \frac{1}{k!} \prod_{i=n-k+1}^{n} i$$

9.9.14 Catalan number

$$C_n = \binom{2n}{n} \Big/ (n+1)$$

9.9.15 Fuss–Catalan number

$$A_m(p,r) = \frac{r}{mp+r} \binom{mp+r}{m}$$

9.10 Fibonacci numbers

Fibonacci numbers can be computed efficiently using the following algorithm:

```
static void
fib_ll(z_t f, z_t g, z_t n)
{
    z_t a, k;
    int odd;
    if (zcmpi(n, 1) <= 0) {
        zseti(f, !zzero(n));
        zseti(f, zzero(n));
        return;
    }
    zinit(a), zinit(k);
    zrsh(k, n, 1);
    if (zodd(n)) {
        odd = zodd(k);
        fib_ll(a, g, k);
        zadd(f, a, a);
        zadd(k, f, g);
        zsub(f, f, g);
        zmul(f, f, k);
        zseti(k, odd ? -2 : +2);
        zadd(f, f, k);
        zadd(g, g, g);
        zadd(g, g, a);
        zmul(g, g, a);
    } else {
        fib_ll(g, a, k);
        zadd(f, a, a);
        zadd(f, f, g);
        zmul(f, f, g);
        zsqr(a, a);
        zsqr(g, g);
        zadd(g, a);
    }
    zfree(k), zfree(a);
}
```

```
void
fib(z_t f, z_t n)
{
    z_t tmp, k;
    zinit(tmp), zinit(k);
    zset(k, n);
    fib_ll(f, tmp, k);
    zfree(k), zfree(tmp);
}
```

This algorithm is based on the rules

$$F_{2k+1} = 4F_k^2 - F_{k-1}^2 + 2(-1)^k = (2F_k + F_{k-1})(2F_k - F_{k-1}) + 2(-1)^k$$

$$F_{2k} = F_k \cdot (F_k + 2F_{k-1})$$

$$F_{2k-1} = F_k^2 + F_{k-1}^2$$

Each call to `fib_ll` returns F_n and F_{n-1} for any input n. F_k is only correctly returned for $k \geq 0$. F_n and F_{n-1} is used for calculating F_{2n} or F_{2n+1}. The algorithm can be sped up with a larger lookup table than one covering just the base cases. Alternatively, a naïve calculation could be used for sufficiently small input.

9.11 Lucas numbers

Lucas [lykɑ] numbers can be calculated by utilising `fib_ll` from Section 9.10 [Fibonacci numbers], page 70:

```
void
lucas(z_t l, z_t n)
{
    z_t k;
    int odd;
    if (zcmp(n, 1) <= 0) {
        zset(l, 1 + zzero(n));
        return;
    }
    zinit(k);
    zrsh(k, n, 1);
    if (zeven(n)) {
        lucas(l, k);
        zsqr(l, l);
        zseti(k, zodd(k) ? +2 : -2);
        zadd(l, k);
    } else {
        odd = zodd(k);
        fib_ll(l, k, k);
        zadd(l, l, l);
        zadd(l, l, k);
        zmul(l, l, k);
        zseti(k, 5);
        zmul(l, l, k);
        zseti(k, odd ? +4 : -4);
        zadd(l, l, k);
    }
    zfree(k);
}
```

This algorithm is based on the rules

$$L_{2k} = L_k^2 - 2(-1)^k$$
$$L_{2k+1} = 5F_{k-1} \cdot (2F_k + F_{k-1}) - 4(-1)^k$$

Alternatively, the function can be implemented trivially using the rule

$$L_k = F_k + 2F_{k-1}$$

9.12 Bit operation

9.12.1 Bit scanning

Scanning for the next set or unset bit can be trivially implemented using `zbtest`. A more efficient, although not optimally efficient, implementation would be

```
size_t
bscan(z_t a, size_t whence, int direction, int value)
{
    size_t ret;
    z_t t;
    zinit(t);
    value ? zset(t, a) : znot(t, a);
    ret = direction < 0
        ? (ztrunc(t, t, whence + 1), zbits(t) - 1)
        : (zrsh(t, t, whence), zlsb(t) + whence);
    zfree(t);
    return ret;
}
```

9.12.2 Population count

The following function can be used to compute the population count, the number of set bits, in an integer, counting the sign bit:

```
size_t
popcount(z_t a)
{
    size_t i, ret = zsignum(a) < 0;
    for (i = 0; i < a->used; i++) {
        ret += __builtin_popcountll(a->chars[i]);
    }
    return ret;
}
```

It requires a compiler extension; if it's not available, there are other ways to computer the population count for a word: manually bit-by-bit, or with a fully unrolled

```
int s;
for (s = 1; s < 64; s <<= 1)
    w = (w >> s) + w;
```

9.12.3 Hamming distance

A simple way to compute the Hamming distance, the number of differing bits between two numbers is with the function

```
size_t
hammdist(z_t a, z_t b)
{
    size_t ret;
    z_t t;
    zinit(t);
    zxor(t, a, b);
    ret = popcount(t);
    zfree(t);
    return ret;
}
```

The performance of this function could be improved by comparing character by character manually using `zxor`.

9.13 Miscellaneous

9.13.1 Character retrieval

```
uint64_t
getu(z_t a)
{
    return zzero(a) ? 0 : a->chars[0];
}
```

9.13.2 Fit test

Some libraries have functions for testing whether a big integer is small enough to fit into an intrinsic type. Since libzahl does not provide conversion to intrinsic types this is irrelevant. But additionally, it can be implemented with a single one-line macro that does not have any side-effects.

```
#define fits_in(a, type)  (zbits(a) <= 8 * sizeof(type))
/* Just be sure the type is integral. */
```

9.13.3 Reference duplication

This could be useful for creating duplicates with modified sign, but only if neither `r` nor `a` will be modified whilst both are in use. Because it is unsafe, fairly simple to create an implementation with acceptable performance — `*r = *a`, — and probably seldom useful, this has not been implemented.

```
void
refdup(z_t r, z_t a)
{
    /* Almost fully optimised, but perfectly portable *r = *a; */
    r->sign    = a->sign;
    r->used    = a->used;
    r->alloced = a->alloced;
    r->chars   = a->chars;
}
```

9.13.4 Variadic initialisation

Most bignum libraries have variadic functions for initialisation and uninitialisation. This is not available in libzahl, because it is not useful enough and has performance overhead. And what's next, support `va_list`, variadic addition, variadic multiplication, power towers, set manipulation? Anyone can implement variadic wrapper for `zinit` and `zfree` if they really need it. But if you want to avoid the overhead, you can use something like this:

```
/* Call like this: MANY(zinit, (a), (b), (c)) */
#define MANY(f, ...)  (_MANY1(f, __VA_ARGS__,,,,,,,,,))

#define _MANY1(f, a, ...)  (void)f a, _MANY2(f, __VA_ARGS__)
#define _MANY2(f, a, ...)  (void)f a, _MANY3(f, __VA_ARGS__)
#define _MANY3(f, a, ...)  (void)f a, _MANY4(f, __VA_ARGS__)
#define _MANY4(f, a, ...)  (void)f a, _MANY5(f, __VA_ARGS__)
#define _MANY5(f, a, ...)  (void)f a, _MANY6(f, __VA_ARGS__)
#define _MANY6(f, a, ...)  (void)f a, _MANY7(f, __VA_ARGS__)
#define _MANY7(f, a, ...)  (void)f a, _MANY8(f, __VA_ARGS__)
#define _MANY8(f, a, ...)  (void)f a, _MANY9(f, __VA_ARGS__)
#define _MANY9(f, a, ...)  (void)f a
```

Chapter 10

Exercises

1. [▶*02*] **Saturated subtraction**

   Implement the function

   ```
   void monus(z_t r, z_t a, z_t b);
   ```

   which calculates $r = a \dot{-} b = \max\{0,\ a - b\}$.

2. [▶*10*] **Modular powers of 2**

   What is the advantage of using `zmodpow` over `zbset` or `zlsh` in combination with `zmod`?

3. [*M15*] **Convergence of the Lucas Number ratios**

   Find an approximation for $\lim\limits_{n\to\infty} \dfrac{L_{n+1}}{L_n}$, where L_n is the n^{th} Lucas Number (see Section 9.11 [Lucas numbers], page 72).

   $$L_n \stackrel{\text{def}}{=} \begin{cases} 2 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ L_{n-1} + L_{n+1} & \text{otherwise} \end{cases}$$

4. [*MP12*] **Factorisation of factorials**

   Implement the function

   ```
   void factor_fact(z_t n);
   ```

   which prints the prime factorisation of $n!$ (the n^{th} factorial). The function shall be efficient for all n where all primes $p \leq n$ can be found efficiently. You can assume that $n \geq 2$. You should not evaluate $n!$.

5. [*M20*] **Reverse factorisation of factorials**

   You should already have solved "Factorisation of factorials" before you solve this problem.

   Implement the function

```
void unfactor_fact(z_t x, z_t *P,
     unsigned long long int *K, size_t n);
```

which given the factorsation of $x!$ determines x. The factorisation of $x!$ is $\prod_{i=1}^{n} P_i^{K_i}$, where P_i is `P[i - 1]` and K_i is `K[i - 1]`.

6. [▶*MP17*] **Factorials inverted**

   Implement the function

   ```
   void unfact(z_t x, z_t n);
   ```

   which given a factorial number n, i.e. on the form $x! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot x$, calculates $x = n!^{-1}$. You can assume that n is a perfect factorial number and that $x \geq 1$. Extra credit if you can detect when the input, n, is not a factorial number. Such function would of course return an `int` 1 if the input is a factorial and 0 otherwise, or alternatively 0 on success and -1 with `errno` set to `EDOM` if the input is not a factorial.

7. [*05*] **Fast primality test**

   $(x + y)^p \equiv x^p + y^p \pmod p$ for all primes p and for a few composites p, which are know as pseudoprimes. Use this to implement a fast primality tester.

8. [*10*] **Fermat primality test**

   $a^{p-1} \equiv 1 \pmod p \ \forall \ 1 < a < p$ for all primes p and for a few composites p, which are know as pseudoprimes[1]. Use this to implement a heuristic primality tester. Try to mimic `zptest` as much as possible. GNU MP uses $a = 210$, but you don't have to. (a is called a base.)

9. [*11*] **Lucas–Lehmer primality test**

   The Lucas–Lehmer primality test can be used to determine whether a Mersenne numbers $M_n = 2^n - 1$ is a prime (a Mersenne prime). M_n is a prime if and only if $s_{n-1} \equiv 0 \pmod{M_n}$, where

   $$s_i = \begin{cases} 4 & \text{if } i = 0 \\ s_{i-1}^2 - 2 & \text{otherwise.} \end{cases}$$

   The Lucas–Lehmer primality test requires that $n \geq 3$, however $M_2 = 2^2 - 1 = 3$ is a prime. Implement a version of the Lucas–Lehmer primality test that takes n as the input. For some more fun, when you are done, you can implement a version that takes M_n as the input.

   For improved performance, instead of using `zmod`, you can use the recursive function $k \bmod (2^n - 1) = \left((k \bmod 2^n) + \lfloor k \div 2^n \rfloor\right) \bmod (2^n - 1)$,

[1] If p is composite but passes the test for all a, p is a Carmichael number.

where $k \mod 2^n$ is efficiently calculated using `zand(`k`, `$2^n - 1$`)`. (This optimisation is not part of the difficulty rating of this problem.)

10. [*20*] **Fast primality test with bounded perfection**

Implement a primality test that is both very fast and never returns `PROBABLY_PRIME` for input less than or equal to a preselected number.

11. [*30*] **Powers of the golden ratio**

Implement function that returns φ^n rounded to the nearest integer, where n is the input and φ is the golden ratio.

12. [►*05*] **zlshu and zrshu**

Why does libzahl have

```
void zlsh(z_t, z_t, size_t);
void zrsh(z_t, z_t, size_t);
```

rather than

```
void zlsh(z_t, z_t, z_t);
void zrsh(z_t, z_t, z_t);
void zlshu(z_t, z_t, size_t);
void zrshu(z_t, z_t, size_t);
```

13. [►*M15*] **Modular left-shift**

Implement a function that calculates $2^a \mod b$, using `zmod` and only cheap functions. You can assume $a \geq 0$, $b \geq 1$. You can also assume that all parameters are unique pointers.

14. [►*08*] **Modular left-shift, extended**

You should already have solved ‘‘Modular left-shift’’ before you solve this problem.

Extend the function you wrote in ‘‘Modular left-shift’’ to accept negative b and non-unique pointers.

15. [*13*] **The totient**

The totient of n is the number of integer a, $0 < a < n$ that are relatively prime to n. Implement Euler's totient function $\varphi(n)$ which calculates the totient of n. Its formula is

$$\varphi(n) = |n| \prod_{p \in \mathbf{P} : p | n} \left(1 - \frac{1}{p}\right).$$

Note that $\varphi(-n) = \varphi(n)$, $\varphi(0) = 0$, and $\varphi(1) = 1$.

16. [*M13*] **The totient from factorisation**

Implement the function

```
void totient_fact(z_t t, z_t *P,
                  unsigned long long int *K, size_t n);
```

which calculates the totient $t = \varphi(n)$, where $n = \prod_{i=1}^{n} P_i^{K_i} > 0$, and P_i = `P[i - 1]` $\in \mathbf{P}$, K_i = `K[i - 1]` ≥ 1. All values `P` are mutually unique. `P` and `K` make up the prime factorisation of n.

You can use the following rules:

$$\begin{array}{ll} \varphi(1) = 1 & \\ \varphi(p) = p - 1 & \text{if } p \in \mathbf{P} \\ \varphi(nm) = \varphi(n)\varphi(m) & \text{if } \gcd(n, m) = 1 \\ n^a\varphi(n) = \varphi(n^{a+1}) & \end{array}$$

17. [*HMP32*] **Modular tetration**

Implement the function

```
void modtetra(z_t r, z_t b, unsigned long n, z_t m);
```

which calculates $r = {}^{n}b \bmod m$, where ${}^{0}b = 1$, ${}^{1}b = b$, ${}^{2}b = b^b$, ${}^{3}b = b^{b^b}$, ${}^{4}b = b^{b^{b^b}}$, and so on. You can assume $b > 0$ and $m > 0$. You can also assume `r`, `b`, and `m` are mutually unique pointers.

18. [*13*] **Modular generalised power towers**

This problem requires a working solution for "Modular tetration".

Modify your solution for "Modular tetration" to evaluate any expression on the forms a^b, a^{b^c}, $a^{b^{c^d}}$, $\ldots \bmod m$.

Chapter 11

Solutions

1. **Saturated subtraction**

```
void monus(z_t r, z_t a, z_t b)
{
    zsub(r, a, b);
    if (zsignum(r) < 0)
        zsetu(r, 0);
}
```

2. **Modular powers of 2**

   `zbset` and `zbit` requires $\Theta(n)$ memory to calculate 2^n. `zmodpow` only requires $\mathcal{O}(\min\{n, \log m\})$ memory to calculate $2^n \bmod m$. $\Theta(n)$ memory complexity becomes problematic for very large n.

3. **Convergence of the Lucas Number ratios**

   It would be a mistake to use bignum, and bigint in particular, to solve this problem. Good old mathematics is a much better solution.

$$\lim_{n\to\infty} \frac{L_{n+1}}{L_n} = \lim_{n\to\infty} \frac{L_n}{L_{n-1}} = \lim_{n\to\infty} \frac{L_{n-1}}{L_{n-2}}$$

$$\frac{L_n}{L_{n-1}} = \frac{L_{n-1}}{L_{n-2}}$$

$$\frac{L_{n-1} + L_{n-2}}{L_{n-1}} = \frac{L_{n-1}}{L_{n-2}}$$

$$1 + \frac{L_{n-2}}{L_{n-1}} = \frac{L_{n-1}}{L_{n-2}}$$

$$1 + \varphi = \frac{1}{\varphi}$$

So the ratio tends toward the golden ratio.

4. **Factorisation of factorials**

Base your implementation on

$$n! = \prod_{p \,\in\, \mathbf{P}}^{n} p^{k_p}, \text{ where } k_p = \sum_{a=1}^{\lfloor \log_p n \rfloor} \lfloor np^{-a} \rfloor.$$

There is no need to calculate $\lfloor \log_p n \rfloor$, you will see when $p^a > n$.

5. **Reverse factorisation of factorials**

$x = \max_{p \,\in\, P} p \cdot f(p, k_p)$, where k_p is the power of p in the factorisation of $x!$. $f(p, k)$ is defined as:

$k' \leftarrow 0$
while $k > 0$ **do**
  $a \leftarrow 0$
  while $p^a \leq k$ **do**
    $k \leftarrow k - p^a$
    $a \leftarrow a + 1$
  end while
  $k' \leftarrow k' + p^{a-1}$
end while
return k'

6. **Factorials inverted**

Use `zlsb` to get the power of 2 in the prime factorisation of n, that is, the number of times n is divisible by 2. If we write n on the form $1 \cdot 2 \cdot 3 \cdot \ldots \cdot x$, every 2^{nd} factor is divisible by 2, every 4^{th} factor is divisible by 2^2, and so on. From calling `zlsb` we know how many times, n is divisible by 2, but know how many of the factors are divisible by 2, but this can be calculated with the following algorithm, where k is the number times n is divisible by 2:

$k' \leftarrow 0$
while $k > 0$ **do**
  $a \leftarrow 0$
  while $2^a \leq k$ **do**
    $k \leftarrow k - 2^a$
    $a \leftarrow a + 1$
  end while
  $k' \leftarrow k' + 2^{a-1}$
end while
return k'

Note that 2^a is efficiently calculated with, `zlsh`, but it is more efficient to use `zbset`.

Now that we know k', the number of factors ni $1 \cdot \ldots \cdot x$ that are divisible by 2, we have two choices for x: k' and $k'+1$. To check which, we calculate $(k'-1)!!$ (the semifactoral, i.e. $1 \cdot 3 \cdot 5 \cdot \ldots \cdot (k'-1)$) naïvely and shift the result k steps to the left. This gives us $k'!$. If $x! \neq k'!$, then $x = k' + 1$ unless n is not factorial number. Of course, if $x! = k'!$, then $x = k'$.

7. **Fast primality test**

If we select $x = y = 1$ we get $2^p \equiv 2 \pmod{p}$. This gives us

```
enum zprimality
ptest_fast(z_t p)
{
    z_t a;
    int c = zcmpu(p, 2);
    if (c <= 0)
        return c ? NONPRIME : PRIME;
    zinit(a);
    zsetu(a, 1);
    zlsh(a, a, p);
    zmod(a, a, p);
    c = zcmpu(a, 2);
    zfree(a);
    return c ? NONPRIME : PROBABLY_PRIME;
}
```

8. **Fermat primality test**

Below is a simple implementation. It can be improved by just testing against a fix base, such as $a = 210$, it $t = 0$. It could also do an exhaustive test (all a such that $1 < a < p$) if $t < 0$.

```
enum zprimality
ptest_fermat(z_t witness, z_t p, int t)
{
    enum zprimality rc = PROBABLY_PRIME;
    z_t a, p1, p3, temp;
    int c;

    if ((c = zcmpu(p, 2)) <= 0) {
        if (!c)
            return PRIME;
        if (witness && witness != p)
            zset(witness, p);
        return NONPRIME;
    }

    zinit(a), zinit(p1), zinit(p3), zinit(temp);
    zsetu(temp, 3), zsub(p3, p, temp);
    zsetu(temp, 1), zsub(p1, p, temp);

    zsetu(temp, 2);
    while (t--) {
        zrand(a, DEFAULT_RANDOM, UNIFORM, p3);
        zadd(a, a, temp);
        zmodpow(a, a, p1, p);
        if (zcmpu(a, 1)) {
            if (witness)
                zswap(witness, a);
            rc = NONPRIME;
            break;
        }
    }

    zfree(temp), zfree(p3), zfree(p1), zfree(a);
    return rc;
}
```

9. **Lucas–Lehmer primality test**

```
enum zprimality
ptest_llt(z_t n)
{
    z_t s, M;
    int c;
```

```
    size_t p;

    if ((c = zcmpu(n, 2)) <= 0)
        return c ? NONPRIME : PRIME;

    if (n->used > 1) {
        /* An optimised implementation would not need this */
        errno = ENOMEM;
        return (enum zprimality)(-1);
    }

    zinit(s), zinit(M), zinit(2);

    p = (size_t)(n->chars[0]);
    zsetu(s, 1), zsetu(M, 0);
    zbset(M, M, p, 1), zsub(M, M, s);
    zsetu(s, 4);
    zsetu(two, 2);

    p -= 2;
    while (p--) {
        zsqr(s, s);
        zsub(s, s, two);
        zmod(s, s, M);
    }
    c = zzero(s);

    zfree(two), zfree(M), zfree(s);
    return c ? PRIME : NONPRIME;
}
```

M_n is composite if n is composite, therefore, if you do not expect prime-only values on n, the performance can be improved by using some other primality test (or this same test if n is a Mersenne number) to first check that n is prime.

10. Fast primality test with bounded perfection

First we select a fast primality test. We can use $2^p \equiv 2 \ (\texttt{Mod}\ p) \ \forall\ p \in \mathbf{P}$, as describe in the solution for the problem *Fast primality test*.

Next, we use this to generate a large list of primes and pseudoprimes. Use a perfect primality test, such as a naïve test or the AKS primality test, to filter out all primes and retain only the pseudoprimes. This is not in runtime so it does not matter that this is slow, but to speed it up, we can use a probabilistic test such the Miller–Rabin primality test (`zptest`) before we use the perfect test.

Now that we have a quite large — but not humongous — list of pseudoprimes, we can incorporate it into our fast primality test. For any input that passes the test, and is less or equal to the largest pseudoprime we found, binary search our list of pseudoprime for the input.

For input, larger than our limit, that passes the test, we can run it through `zptest` to reduce the number of false positives.

As an alternative solution, instead of comparing against known pseudoprimes. Find a minimal set of primes that includes divisors for all known pseudoprimes, and do trail division with these primes when a number passes the test. No pseudoprime need to have more than one divisor included in the set, so this set cannot be larger than the set of pseudoprimes.

11. **Powers of the golden ratio**

This was an information gathering exercise. For $n \geq 2$, $L_n = [\varphi^n]$, where L_n is the n^{th} Lucas number.

$$L_n \stackrel{\text{def}}{=} \begin{cases} 2 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ L_{n-1} + L_{n+1} & \text{otherwise} \end{cases}$$

but for efficiency and briefness, we will use `lucas` from Section 9.11 [Lucas numbers], page 72.

```
void
golden_pow(z_t r, z_t n)
{
    if (zsignum(n) <= 0)
        zsetu(r, zcmpi(n, -1) >= 0);
    else if (!zcmpu(n, 1))
        zsetu(r, 2);
    else
        lucas(r, n);
}
```

12. **zlshu and zrshu**

You are in big trouble, memorywise, of you need to evaluate $2^{2^{64}}$.

13. **Modular left-shift**

```
void
modlsh(z_t r, z_t a, z_t b)
{
    z_t t, at;
    size_t s = zbits(b);

    zinit(t), zinit(at);
    zset(at, a);
    zsetu(r, 1);
    zsetu(t, s);

    while (zcmp(at, t) > 0) {
        zsub(at, at, t);
        zlsh(r, r, t);
        zmod(r, r, b);
        if (zzero(r))
            break;
    }
    if (!zzero(a) && !zzero(b)) {
        zlsh(r, r, a);
        zmod(r, r, b);
    }

    zfree(at), zfree(t);
}
```

It is worth noting that this function is not necessarily faster than `zmodpow`.

14. Modular left-shift, extended

The sign of `b` shall not effect the result. Use `zabs` to create a copy of `b` with its absolute value.

15. The totient

$$\varphi(n) = n \prod_{p \in \mathbf{P} : p \mid n} \left(1 - \frac{1}{p}\right) = n \prod_{p \in \mathbf{P} : p \mid n} \left(\frac{p-1}{p}\right)$$

So, if we set $a = n$ and $b = 1$, then we iterate of all integers p, $2 \le p \le n$. For which p that is prime, we set $a \leftarrow a \cdot (p-1)$ and $b \leftarrow b \cdot p$. After the iteration, $b|a$, and $\varphi(n) = \frac{a}{b}$. However, if $n < 0$, then, $\varphi(n) = \varphi|n|$.

16. The totient from factorisation

```
void
```

```
totient_fact(z_t t, z_t *P,
             unsigned long long *K, size_t n)
{
    z_t a, one;
    zinit(a), zinit(one);
    zseti(t, 1);
    zseti(one, 1);
    while (n--) {
        zpowu(a, P[n], K[n] - 1);
        zmul(t, t, a);
        zsub(a, P[n], one);
        zmul(t, t, a);
    }
    zfree(a), zfree(one);
}
```

17. Modular tetration

Let `totient` be Euler's totient function. It is described in the problem "The totient".

We need two help function: `tetra(r, b, n)` which calculated $r = {}^{n}b$, and `cmp_tetra(a, b, n)` which is compares a to ${}^{n}b$.

```
void
tetra(z_t r, z_t b, unsigned long n)
{
    zsetu(r, 1);
    while (n--)
        zpow(r, b, r);
}

int
cmp_tetra(z_t a, z_t b, unsigned long n)
{
    z_t B;
    int cmp;

    if (!n || !zcmpu(b, 1))
        return zcmpu(a, 1);
    if (n == 1)
        return zcmp(a, b);
    if (zcmp(a, b) >= 0)
        return +1;
```

```
    zinit(B);
    zsetu(B, 1);
    while (n) {
        zpow(B, b, B);
        if (zcmp(a, B) < 0) {
            zfree(B);
            return -1;
        }
    }
    cmp = zcmp(a, B);
    zfree(B);
    return cmp;

}
```

`tetra` can generate unmaintainably huge numbers. Will however only call `tetra` when this is not the case.

```
void
modtetra(z_t r, z_t b, unsigned long n, z_t m)
{
    z_t t, temp;

    if (n <= 1) {
        if (!n)
            zsetu(r, zcmpu(m, 1));
        else
            zmod(r, b, m);
        return;
    }

    zmod(r, b, m);
    if (zcmpu(r, 1) <= 0)
        return;

    zinit(t);
    zinit(temp);

    t = totient(m);
    zgcd(temp, b, m);

    if (!zcmpu(temp, 1)) {
        modtetra(temp, b, n - 1, t);
```

```
        zmodpow(r, r, temp, m);
    } else if (cmp_tetra(t, b, n - 1) > 0) {
        temp = tetra(b, n - 1);
        zpowmod(r, r, temp, m);
    } else {
        modtetra(temp, b, n - 1, t);
        zmodpow(temp, r, temp, m);
        zmodpow(r, r, t, m);
        zmodmul(r, r, temp, m);
    }

    zfree(temp);
    zfree(t);
}
```

18. **Modular generalised power towers**

Instead of the signature

```
void modtetra(z_t r, z_t b, unsigned long n, z_t m);
```

you want to use the signature

```
void modtower_(z_t r, z_t *a, size_t n, z_t m);
```

Instead of using, `b` (in `modtetra`), use `*a`. At every recursion, instead of passing on `a`, pass on `a + 1`.

The function `tetra` is modified into

```
void tower(z_t r, z_t *a, size_t n)
{
    zsetu(r, 1);
    while (n--);
        zpow(r, a[n], r);
}
```

`cmp_tetra` is changed analogously.

To avoid problems in the evaluation, define

```
void modtower(z_t r, z_t *a, size_t n, z_t m);
```

which cuts the power short at the first element of of `a` that equals 1. For example, if $a = \{2, 3, 4, 5, 1, 2, 3\}$, and $n = 7$ ($n = |a|$), then `modtower`, sets $n = 4$, and effectively $a = \{2, 3, 4, 5\}$. After this `modtower` calls `modtower_`.