2.10. math.ak

//// This module contains some basic Math utilities. Standard arithmetic

//// operations on integers are available through native operators:

////

//// Operator | Description

//// --- | :---

//// `+` | Arithmetic sum

//// `-` | Arithmetic difference

//// `/` | Whole division

//// `\*` | Arithmetic multiplication

//// `%` | Remainder by whole division

////

//// Here are a few examples:

////

//// ```aiken

//// 1 + 1 // 2

//// 10 - 2 // 8

//// 40 / 14 // 2

//// 3 \* 4 // 12

//// 10 % 3 // 1

use aiken/builtin

/// Calculate the absolute value of an integer.

///

/// ```aiken

/// math.abs(-42) == 42

/// math.abs(14) == 14

/// ```

pub fn abs(self: Int) -> Int {

if self < 0 {

0 - self

} else {

self

}

}

test abs\_1() {

abs(14) == 14

}

test abs\_2() {

abs(-42) == 42

}

/// Restrict the value of an integer between two min and max bounds

///

/// ```aiken

/// math.clamp(14, min: 0, max: 10) == 10

/// ```

pub fn clamp(self: Int, min: Int, max: Int) -> Int {

if self < min {

min

} else {

if self > max {

max

} else {

self

}

}

}

test clamp\_1() {

clamp(14, min: 0, max: 10) == 10

}

test clamp\_2() {

clamp(7, min: 0, max: 10) == 7

}

test clamp\_3() {

clamp(7, min: 10, max: 100) == 10

}

/// Return the maximum of two integers.

pub fn max(a: Int, b: Int) -> Int {

if a > b {

a

} else {

b

}

}

test max\_1() {

max(0, 0) == 0

}

test max\_2() {

max(14, 42) == 42

}

test max\_3() {

max(42, 14) == 42

}

/// Return the minimum of two integers.

pub fn min(a: Int, b: Int) -> Int {

if a > b {

b

} else {

a

}

}

test min\_1() {

min(0, 0) == 0

}

test min\_2() {

min(14, 42) == 14

}

test min\_3() {

min(42, 14) == 14

}

/// Calculates a number to the power of `e` using the exponentiation by

/// squaring method.

///

/// ```aiken

/// math.pow(3, 5) == 243

/// math.pow(7, 2) == 49

/// math.pow(3, -4) == 0

/// math.pow(0, 0) == 1

/// math.pow(513, 3) == 135005697

/// ```

pub fn pow(self: Int, e: Int) -> Int {

if e < 0 {

0

} else if e == 0 {

1

} else if e % 2 == 0 {

pow(self \* self, e / 2)

} else {

self \* pow(self \* self, ( e - 1 ) / 2)

}

}

test pow\_3\_5() {

pow(3, 5) == 243

}

test pow\_7\_2() {

pow(7, 2) == 49

}

test pow\_3\_\_4() {

// negative powers round to zero

pow(3, -4) == 0

}

test pow\_0\_0() {

// sorry math

pow(0, 0) == 1

}

test pow\_513\_3() {

pow(513, 3) == 135005697

}

test pow\_2\_4() {

pow(2, 4) == 16

}

test pow\_2\_42() {

pow(2, 42) == 4398046511104

}

/// Calculates the power of 2 for a given exponent `e`. Much cheaper than

/// using `pow(2, \_)` for small exponents (0 < e < 256).

///

/// ```aiken

/// math.pow2(-2) == 0

/// math.pow2(0) == 1

/// math.pow2(1) == 2

/// math.pow2(4) == 16

/// math.pow2(42) == 4398046511104

/// ```

pub fn pow2(e: Int) -> Int {

// do\_pow2(e, 1)

if e < 8 {

if e < 0 {

0

} else {

builtin.index\_bytearray(#[1, 2, 4, 8, 16, 32, 64, 128], e)

}

} else if e < 32 {

256 \* pow2(e - 8)

} else {

4294967296 \* pow2(e - 32)

}

}

test pow2\_neg() {

pow2(-2) == 0

}

test pow2\_0() {

pow2(0) == 1

}

test pow2\_1() {

pow2(1) == 2

}

test pow2\_4() {

pow2(4) == 16

}

test pow2\_42() {

pow2(42) == 4398046511104

}

test pow2\_256() {

pow2(256) == 115792089237316195423570985008687907853269984665640564039457584007913129639936

}

/// The logarithm in base `b` of an element using integer divisions.

///

/// ```aiken

/// math.log(10, base: 2) == 3

/// math.log(42, base: 2) == 5

/// math.log(42, base: 3) == 3

/// math.log(5, base: 0) == 0

/// math.log(4, base: 4) == 1

/// math.log(4, base: 42) == 0

/// ```

pub fn log(self: Int, base: Int) -> Int {

if base <= 0 {

0

} else if self == base {

1

} else if self < base {

0

} else {

1 + log(self / base, base)

}

}

test log\_10\_2() {

log(10, base: 2) == 3

}

test log\_42\_2() {

log(42, base: 2) == 5

}

test log\_42\_3() {

log(42, base: 3) == 3

}

test log\_5\_0() {

log(5, base: 0) == 0

}

test log\_4\_4() {

log(4, base: 4) == 1

}

test log\_4\_43() {

log(4, base: 43) == 0

}

/// The greatest common divisor of two integers.

///

/// ```aiken

/// math.gcd(42, 14) == 14

/// math.gcd(14, 42) == 14

/// math.gcd(0, 0) == 0

/// ```

pub fn gcd(x: Int, y: Int) -> Int {

abs(do\_gcd(x, y))

}

fn do\_gcd(x: Int, y: Int) -> Int {

when y is {

0 -> x

\_ -> do\_gcd(y, x % y)

}

}

test gcd\_test1() {

gcd(10, 300) == 10

}

test gcd\_test2() {

gcd(-10, 300) == 10

}

test gcd\_test3() {

gcd(42, 14) == 14

}

/// Calculates the square root of an integer using the [Babylonian

/// method](https://en.wikipedia.org/wiki/Methods\_of\_computing\_square\_roots#Babylonian\_method). This returns either the exact result or the smallest integer

/// nearest to the square root.

///

/// Returns `None` for negative values.

///

/// ```aiken

/// math.sqrt(0) == Some(0)

/// math.sqrt(25) == Some(5)

/// math.sqrt(44203) == Some(210)

/// math.sqrt(-42) == None

/// ```

pub fn sqrt(self: Int) -> Option<Int> {

if self < 0 {

None

} else if self <= 1 {

Some(self)

} else {

Some(sqrt\_babylonian(self, self, ( self + 1 ) / 2))

}

}

// The basic idea is that if x is an overestimate to the square root of a

// non-negative real number S then S/x will be an underestimate, or vice versa,

// and so the average of these two numbers may reasonably be expected to provide a

// better approximation (though the formal proof of that assertion depends on the

// inequality of arithmetic and geometric means that shows this average is always

// an overestimate of the square root.

fn sqrt\_babylonian(self: Int, x: Int, y: Int) -> Int {

if y >= x {

x

} else {

sqrt\_babylonian(self, y, ( y + self / y ) / 2)

}

}

test sqrt1() {

sqrt(0) == Some(0)

}

test sqrt2() {

sqrt(1) == Some(1)

}

test sqrt3() {

sqrt(25) == Some(5)

}

test sqrt4() {

sqrt(44203) == Some(210)

}

test sqrt5() {

sqrt(975461057789971041) == Some(987654321)

}

test sqrt6() {

sqrt(-42) == None

}

/// Checks if an integer has a given integer square root x.

/// The check has constant time complexity (O(1)).

///

/// ```aiken

/// math.is\_sqrt(0, 0)

/// math.is\_sqrt(25, 5)

/// ! math.is\_sqrt(25, -5)

/// math.is\_sqrt(44203, 210)

/// ```

pub fn is\_sqrt(self: Int, x: Int) -> Bool {

x \* x <= self && ( x + 1 ) \* ( x + 1 ) > self

}

test is\_sqrt1() {

is\_sqrt(44203, 210)

}

test is\_sqrt2() {

is\_sqrt(975461057789971041, 987654321)

}