Table of Contents

BigInt Package Documentation

This package provides arbitrary-precision integer arithmetic through the BigInt type. BigInt allows you to work with integers of unlimited size, making it perfe ct for cryptographic operations, mathematical computations, and any scenario whe re standard integer types are insufficient.

Creating BigInt Values

There are several ways to create BigInt values:

```
1
2   test "creating bigint values" {
3
4     let big1 = 123456789012345678 90N
5     inspect(big1, content="123456789012345678 90")
6
7
8     let big2 = @bigint BigInt::from_int(42)
9     inspect(big2, content="42")
10
11
12    let big3 = @bigint BigInt::from_int64(9223372036854775807L)
13    inspect(big3, content="9223372036854775807")
14
15
16    let big4 = @bigint BigInt::from_string("12345678901234567890123456789 0
17    inspect(big4, content="12345678901234567890123456789 0")
18
19
20    let big5 = @bigint BigInt::from_hex("1a2b3c4d5e6f")
21    inspect(big5, content="28772997619311")
22  }
```

Basic Arithmetic Operations

BigInt supports all standard arithmetic operations:

```
1
2   test "arithmetic operations" {
3     let a = 12345678901234567890123456789N
4     let b = 98765432109876543210987654321N
5
6
7     let sum = a + b
8     inspect(sum, content="111111111011111111011111111100")
9
10
11    let diff = b - a
12    inspect(diff, content="864197532086419753208641975320")
13
14
15    let product = @bigint BigInt::from_int(123) * @bigint BigInt::from_int
16    inspect(product, content="56088")
17
18
19    let quotient = @bigint BigInt::from_int(1000) / @bigint BigInt::from_i
20    inspect(quotient, content="142")
21
22
23    let remainder = @bigint BigInt::from_int(1000) % @bigint BigInt::from_
24    inspect(remainder, content="6")
25
26
27    let neg = -a
28    inspect(neg, content="-12345678901234567890123456789")
29  }
```

Comparison Operations

Compare BigInt values with each other and with regular integers:

```
1
2   test "comparisons" {
3     let big = 12345N
4     let small = 123N
5
6
7     inspect(big > small, content="true")
8     inspect(big == small, content="false")
9     inspect(small < big, content="true")
10
11
12    inspect(big equal_int(12345), content="true")
13    inspect(big compare_int(12345), content="0")
14    inspect(big compare_int(1000), content="1")
15    inspect(small compare_int(200), content="-1")
16
17
18    let big64 = @bigint BigInt::from_int64(92233720368547758O7L)
19    inspect(big64 equal_int64(92233720368547758O7L), content="true")
20  }
```

Bitwise Operations

BigInt supports bitwise operations for bit manipulation:

```
1
2   test "bitwise operations" {
3     let a = 0b11110000N
4     let b = 0b10101010N
5
6
7     let and_result = a & b
8     inspect(and_result, content="160")
9
10
11    let or_result = a | b
12    inspect(or_result, content="250")
13
14
15    let xor_result = a ^ b
16    inspect(xor_result, content="90")
17
18
19    let big_num = 255N
20    inspect(big_num bit_length(), content="8")
21
22
23    let with_zeros = 1000N
24    let ctz = with_zeros ctz()
25    inspect(ctz >= 0, content="true")
26  }
```

Power and Modular Arithmetic

BigInt provides efficient power and modular exponentiation:

```
1
2   test "power operations" {
3
4     let base = 2N
5     let exponent = 10N
6     let power = base pow(exponent)
7     inspect(power, content="1024")
8
9
10    let base2 = 3N
11    let exp2 = 5N
12    let modulus = 7N
13    let mod_power = base2 pow(exp2, modulus~)
14    inspect(mod_power, content="5")
15
16
17    let large_base = 123N
18    let large_exp = 20N
19    let large_mod = 1000007N
20    let result = large_base pow(large_exp, modulus=large_mod)
21    inspect(result, content="378446")
22  }
```

String and Hexadecimal Conversion

Convert BigInt to and from various string representations:

```
1
2   test "string conversions" {
3     let big = 255N
4
5
6     let decimal = big to_string()
7     inspect(decimal, content="255")
8
9
10    let hex_lower = big to_hex()
11    inspect(hex_lower, content="FF")
12
13
14    let hex_upper = big to_hex(uppercase=true)
15    inspect(hex_upper, content="FF")
16
17
18    let from_hex = @bigint BigInt::from_hex("deadbeef")
19    inspect(from_hex, content="3735928559")
20
21
22    let original = 98765432109876543210N
23    let as_string = original to_string()
24    let parsed_back = @bigint BigInt::from_string(as_string)
25    inspect(original == parsed_back, content="true")
26  }
```

Byte Array Conversion

Convert BigInt to and from byte arrays:

```
1
2   test "byte conversions" {
3     let big = 0x123456789abcdefN
4
5
6     let bytes = big to_octets()
7     inspect(bytes length() > 0, content="true")
8
9
10    let from_bytes = @bigint BigInt::from_octets(bytes)
11    inspect(from_bytes == big, content="true")
12
13
14    let fixed_length = @bigint BigInt::from_int(255) to_octets(length=4)
15    inspect(fixed_length length(), content="4")
16
17
18
19
20
21
22
23  }
```

Type Conversions

Convert BigInt to standard integer types:

```
1
2   test "type conversions" {
3     let big = 12345N
4
5
6     let as_int = big to_int()
7     inspect(as_int, content="12345")
8
9
10    let as_int64 = big to_int64()
11    inspect(as_int64, content="12345")
12
13
14    let as_uint = big to_uint()
15    inspect(as_uint, content="12345")
16
17
18    let small = 255N
19    let as_int16 = small to_int16()
20    inspect(as_int16, content="255")
21    let as_uint16 = small to_uint16()
22    inspect(as_uint16, content="255")
23  }
```

JSON Serialization

BigInt values can be serialized to and from JSON:

```
1
2   test "json serialization" {
3     let big = 123456789012345678 90N
4
5
6     let json = big to_json()
7     inspect(json, content="String(\"123456789012345678 90\")")
8
9
10    let very_big = @bigint BigInt::from_string("123456789012345678 90123456
11    let big_json = very_big to_json()
12    inspect(big_json, content="String(\"123456789012345678 901234567890\")"
13  }
```

Utility Functions

Check properties of BigInt values:

```
1
2   test "utility functions" {
3     let zero = 0N
4     let positive = 42N
5     let negative = -42N
6
7
8     inspect(zero is_zero(), content="true")
9     inspect(positive is_zero(), content="false")
10
11
12    inspect(positive > zero, content="true")
13    inspect(negative < zero, content="true")
14    inspect(zero == zero, content="true")
15  }
```

Use Cases and Applications

BigInt is particularly useful for:

- **Cryptography**: RSA encryption, digital signatures, and key generation
- **Mathematical computations**: Factorial calculations, Fibonacci sequences, prime nu
mber testing
- **Financial calculations**: High-precision monetary computations
- **Scientific computing**: Large integer calculations in physics and chemistry
- **Data processing**: Handling large numeric IDs and checksums

Performance Considerations

- BigInt operations are slower than regular integer operations due to arbitrary pr
ecision
- Addition and subtraction are generally fast
- Multiplication and division become slower with larger numbers
- Modular exponentiation is optimized for cryptographic use cases
- String conversions can be expensive for very large numbers

Best Practices

- **Use regular integers when possible**: Only use BigInt when you need arbitrary prec
ision
- **Cache string representations**: If you need to display the same BigInt multiple ti
mes
- **Use modular arithmetic**: For cryptographic applications, always use modular expon
entiation
- **Be careful with conversions**: Converting very large BigInt to regular integers wi
ll truncate
- **Consider memory usage**: Very large BigInt values consume more memory