Table of Contents

1 Bench Package Documentation
1.1 Basic Benchmarking
1.2 Benchmark Collection
1.3 Benchmarking Different Algorithms
1.4 Data Structure Benchmarks
1.5 String Operations Benchmarking
1.6 Memory Usage Prevention
1.7 Iteration Count Control
1.8 Benchmarking Best Practices
1.8.1 1. Isolate What You’re Measuring
1.8.2 2. Warm Up Before Measuring
1.8.3 3. Use Meaningful Names
1.9 Performance Analysis
1.10 Integration with Testing
1.11 Common Benchmarking Patterns
1.12 Tips for Accurate Benchmarks

Bench Package Documentation

This package provides benchmarking utilities for measuring the performance of MoonBit code. It includes functions for timing code execution, collecting statistics, and generating performance reports.

Basic Benchmarking

Use the single_bench function to benchmark individual operations:

```

#skip("slow tests")
test "basic benchmarking" {
  fn simple_calc(n : Int) -> Int {
    n * 2 + 1
  }

  let summary = @bench.single_bench(name="simple_calc", fn() {
    ignore(simple_calc(5))
  })


  inspect(summary.to_json().stringify().length() > 0, content="true")
}
```

Benchmark Collection

Use the T type to collect multiple benchmarks:

```

#skip("slow tests")
test "benchmark collection" {
  let bencher = @bench.new()


  bencher.bench(name="array_creation", fn() {
    let arr = Array::new()
    for i in 0..<5 {
      arr.push(i)
    }
  })
  bencher.bench(name="array_iteration", fn() {
    let arr = [1, 2, 3, 4, 5]
    let mut sum = 0
    for x in arr {
      sum = sum + x
    }
  })


  let report = bencher.dump_summaries()
  inspect(report.length() > 0, content="true")
}
```

Benchmarking Different Algorithms

Compare the performance of different implementations:

```

#skip("slow tests")
test "algorithm comparison" {
  let bencher = @bench.new()


  bencher.bench(name="linear_search", fn() {
    let arr = [1, 2, 3, 4, 5]
    let target = 3
    let mut found = false
    for x in arr {
      if x == target {
        found = true
        break
      }
    }
    ignore(found)
  })


  bencher.bench(name="builtin_contains", fn() {
    let arr = [1, 2, 3, 4, 5]
    ignore(arr.contains(3))
  })
  let results = bencher.dump_summaries()
  inspect(results.length() > 10, content="true")
}
```

Data Structure Benchmarks

Benchmark different data structure operations:

```

#skip("slow tests")
test "data structure benchmarks" {
  let bencher = @bench.new()


  bencher.bench(name="array_append", fn() {
    let arr = Array::new()
    for i in 0..<5 {
      arr.push(i)
    }
  })


  bencher.bench(name="fixedarray_access", fn() {
    let arr = [0, 1, 2, 3, 4]
    let mut sum = 0
    for i in 0..<arr.length() {
      sum = sum + arr[i]
    }
    ignore(sum)
  })
  let report = bencher.dump_summaries()
  inspect(report.length() > 50, content="true")
}
```

String Operations Benchmarking

Measure string manipulation performance:

```

#skip("slow tests")
test "string benchmarks" {
  let bencher = @bench.new()


  bencher.bench(name="string_concat", fn() {
    let mut result = ""
    for i in 0..<5 {
      result = result + "x"
    }
  })


  bencher.bench(name="stringbuilder", fn() {
    let builder = StringBuilder::new()
    for i in 0..<5 {
      builder.write_string("x")
    }
    ignore(builder.to_string())
  })
  let results = bencher.dump_summaries()
  inspect(results.length() > 50, content="true")
}
```

Memory Usage Prevention

Use keep to prevent compiler optimizations from eliminating benchmarked code:

```

#skip("slow tests")
test "preventing optimization" {
  let bencher = @bench.new()
  bencher.bench(name="with_keep", fn() {
    let result = Array::makei(5, fn(i) { i * i })

    bencher.keep(result)
  })
  let report = bencher.dump_summaries()
  inspect(report.length() > 30, content="true")
}
```

Iteration Count Control

Control the number of benchmark iterations:

```

#skip("slow tests")
test "iteration control" {
  let bencher = @bench.new()


  bencher.bench(
    name="stable_benchmark",
    fn() {
      let arr = [1, 2, 3, 4, 5]
      let sum = arr.fold(init=0, fn(acc, x) { acc + x })
      ignore(sum)
    },
    count=20,
  )


  bencher.bench(
    name="quick_benchmark",
    fn() {
      let mut result = 0
      for i in 0..<10 {
        result = result + i
      }
      ignore(result)
    },
    count=2,
  )
  let results = bencher.dump_summaries()
  inspect(results.length() > 50, content="true")
}
```

Benchmarking Best Practices

1. Isolate What You’re Measuring

```

#skip("slow tests")
test "isolation example" {
  let bencher = @bench.new()


  let data = Array::makei(10, fn(i) { i })
  bencher.bench(name="array_sum", fn() {
    let mut sum = 0
    for x in data {
      sum = sum + x
    }
    bencher.keep(sum)
  })
  let results = bencher.dump_summaries()
  inspect(results.length() > 0, content="true")
}
```

2. Warm Up Before Measuring

```

#skip("slow tests")
test "warmup example" {
  let bencher = @bench.new()
  fn expensive_operation() -> Int {
    let mut result = 0
    for i in 0..<5 {
      result = result + i * i
    }
    result
  }


  for _ in 0..<5 {
    ignore(expensive_operation())
  }


  bencher.bench(name="warmed_up", fn() {
    let result = expensive_operation()
    bencher.keep(result)
  })
  let report = bencher.dump_summaries()
  inspect(report.length() > 30, content="true")
}
```

3. Use Meaningful Names

```

#skip("slow tests")
test "meaningful names" {
  let bencher = @bench.new()


  bencher.bench(name="array_insert_10_items", fn() {
    let arr = Array::new()
    for i in 0..<10 {
      arr.push(i * 2)
    }
    bencher.keep(arr)
  })
  bencher.bench(name="array_search_sorted_10", fn() {
    let arr = Array::makei(10, fn(i) { i })
    let result = arr.contains(5)
    bencher.keep(result)
  })
  let results = bencher.dump_summaries()
  inspect(results.length() > 50, content="true")
}
```

Performance Analysis

The benchmark results include statistical information:

- **Timing measurements**: Microsecond precision timing
- **Statistical analysis**: Median, percentiles, and outlier detection
- **Batch sizing**: Automatic adjustment for stable measurements
- **JSON output**: Machine-readable results for analysis

Integration with Testing

Benchmarks can be integrated into your testing workflow:

```

#skip("slow tests")
test "performance regression test" {
  let bencher = @bench.new()


  bencher.bench(name="critical_algorithm", fn() {
    let data = [5, 2, 8, 1, 9, 3, 7, 4, 6]
    let sorted = Array::new()
    for x in data {
      sorted.push(x)
    }
    sorted.sort()
    bencher.keep(sorted)
  })
  let results = bencher.dump_summaries()

  inspect(results.length() > 50, content="true")
}
```

Common Benchmarking Patterns

- **Before/After comparisons**: Benchmark code before and after optimizations
- **Algorithm comparison**: Compare different implementations of the same functionality
- **Scaling analysis**: Benchmark with different input sizes
- **Memory vs. speed tradeoffs**: Compare memory-efficient vs. speed-optimized approaches
- **Platform differences**: Compare performance across different targets (JS, WASM, native)

Tips for Accurate Benchmarks

- Run benchmarks multiple times and look for consistency
- Be aware of system load and other processes affecting timing
- Use appropriate iteration counts (more for stable results, fewer for quick feedback)
- Measure what matters to your use case
- Consider both average case and worst case performance
- Profile memory usage separately if memory performance is important

The bench package provides essential tools for performance analysis and optimization in MoonBit applications.