HOFs for the win!

Google's MapReduce

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution

static.googleusercontent.com/media/research.google.com/en/us/archive/mapreduce-osdi04.pdf

Write a **recursive** Racket function called `half-count` that takes a positive number *n* and returns the number of times *n* can be halved using integer division, before the result is 1.

```
> (half-count 1)
0
> (half-count 2)
1
> (half-count 4)
2
> (half-count 5)
2
> (half-count 8)
3
```

Firstname Lastname Th. 10 / 11

(Your response)

Write a **recursive** Racket function called `half-count` that takes a positive number *n* and returns the number of times *n* can be halved using integer division, before the result is 1.

```
> (half-count 1)
0
> (half-count 2)
1
> (half-count 4)
2
> (half-count 5)
2
> (half-count 8)
3
```

Firstname Lastname **Th. 10 / 11**

```
(define (half-count n)
  (if (= n 1)
      0
      (+ 1 (half-count (quotient n 2)))))
```

Three problems to consider

`remove`

Given a list L, and an element e, create a new list L' that contains all the elements of L *except* the first instance of e.

`uniq`

Given a list L, create a new list L' that contains only the unique elements of L.

`sublists`

Given a list L, generate a list L' of all sublists that can be made from the elements of L.

```
> (uniq '(c a l i f o r n i a))
'(c l f o r n i a)

> (uniq '(m u d d))
'(m u d)

> (uniq '(m i s s i p p i))
'(m s p i)
```

```
> (sublists '())
'(())

> (sublists '(1))
'((1) ())

> (sublists '(1 2))
'((1 2) (1) (2) ())
```

“Use it or lose it”

(a recursive search technique)

Review: recursion

a problem-solving strategy — fill in the **pieces**

Base case(s)

Recursive case(s)

1. **It:** a piece of the problem

2. **Smaller version:** What does the input look like without it?

3. **Lose-it solution:** How could we solve the smaller version?

4. How would we **combine** it and lose-it to solve the full problem?

```
> (sum '())
0

> (sum '(1 2 3))
6

> (sum '(7 7 7 7 7 7))
42
```

Review: recursion

a problem-solving strategy — fill in the **pieces**

Base case(s)

empty list → 0

Recursive case(s)

1. **It:** a piece of the problem

   the first element

2. **Smaller version:** What does the input look like without it?

   the rest of the list

3. **Lose-it solution:** How could we solve the smaller version?

   sum of the rest of the elements in the list

4. How would we **combine** *it* and *lose-it* to solve the full problem?

   add them

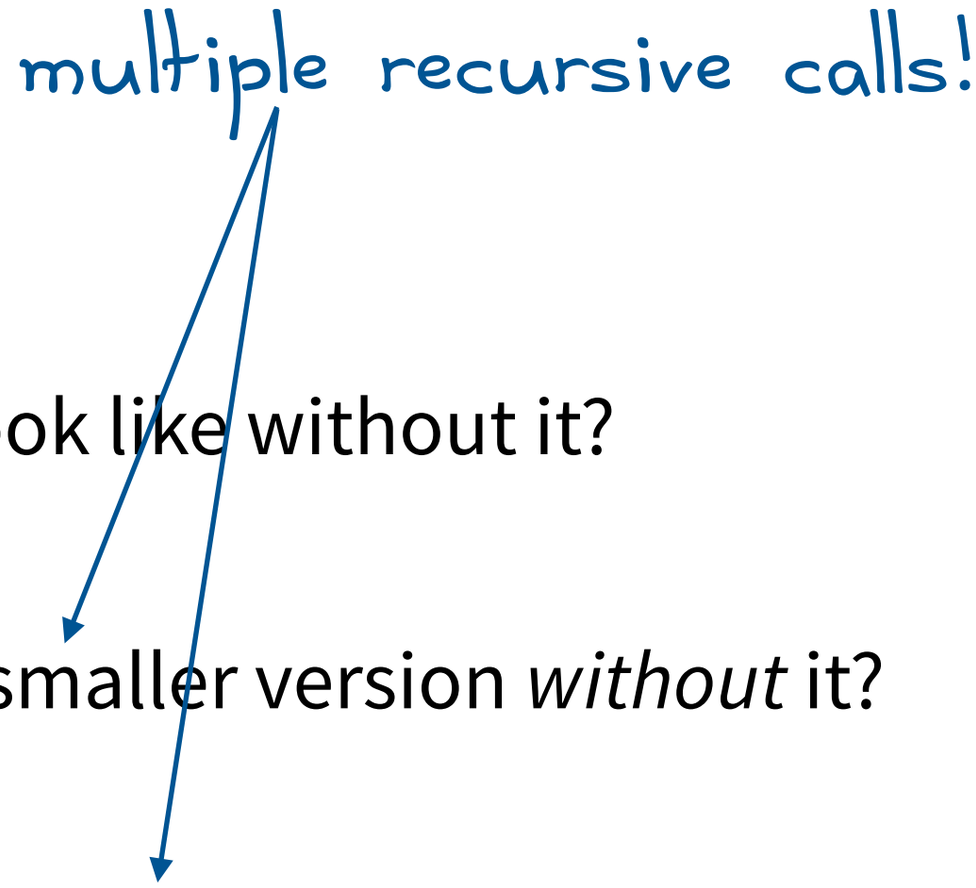
use it or lose it

a recursive strategy — fill in the **pieces**

Base case(s)

Recursive case(s)

1. **It:** a piece of the problem
2. **Smaller version:** What does the input look like without it?
3. **Lose-it solution:** How could we solve a smaller version *without* it?
4. **Use-it solution:** How could we solve a smaller version *with* it?
5. How would we **combine** the solutions to solve the full problem?



```
> (uniq '(c a l i f o r n i a))
'(c l f o r n i a)

> (uniq '(m u d d))
'(m u d)

> (uniq '(m i s s i p p i))
'(m s p i)
```

`unique`: use it or lose it

a recursive strategy — fill in the **pieces**

```
> (uniq '(c a l i f o r n i a))
'(c l f o r n i a)

> (uniq '(m u d d))
'(m u d)

> (uniq '(m i s s i p p i))
'(m s p i)
```

Base case(s)

empty list → empty list

Recursive case(s)

1. **It:** a piece of the problem

   the first element

2. **Smaller version:** What does the input look like without it?

   the rest of the list

3. **Lose-it solution:** How could we solve a smaller version *without* it?

   unique-ify the rest of the list

4. **Use-it solution:** How could we solve a smaller version *with* it?

   pre-pend "it" to the "lose-it" solution

5. How would we **combine** the solutions to solve the full problem?

   if "it" is in "lose-it", then "lose-it"; else "use-it"

`unique`: use it or lose it

a recursive strategy

```
(define (uniq L)
```

```
> (member 1 '(1))
'(1)

> (member 2 '(1))
#f
```

`unique`: use it or lose it

a recursive strategy

```
(define (uniq L)
  (if (empty? L)
      '()
      (let* ([it       (first L)]
             [lose-it (uniq (rest L))]
             [use-it  (cons it lose-it)])
        (if (member it lose-it)
            lose-it
            use-it))))
```

http://i.imgur.com/Gx58aJ5.jpg

```
> (sublists '())
'(())

> (sublists '(1))
'((1) ())

> (sublists '(1 2))
'((1 2) (1) (2) ())
```

`sublists`: use it or lose it

a recursive strategy — fill in the **pieces**

```
> (sublists '())
'(())

> (sublists '(1))
'((1) ())

> (sublists '(1 2))
'((1 2) (1) (2) ())
```

Base case(s)

empty list → list of empty list

Recursive case(s)

1. **It:** a piece of the problem

   the first element

2. **Smaller version:** What does the input look like without it?

   the rest of the list

3. **Lose-it solution:** How could we solve a smaller version *without* it?

   find sublists the rest of the list

4. **Use-it solution:** How could we solve a smaller version *with* it?

   prepend "it" to each element of the "lose-it" solution

5. How would we **combine** the solutions to solve the full problem?

   append the "lose-it" solution to the "use-it" solution

Solution technique: “use it or lose it”

a recursive search strategy

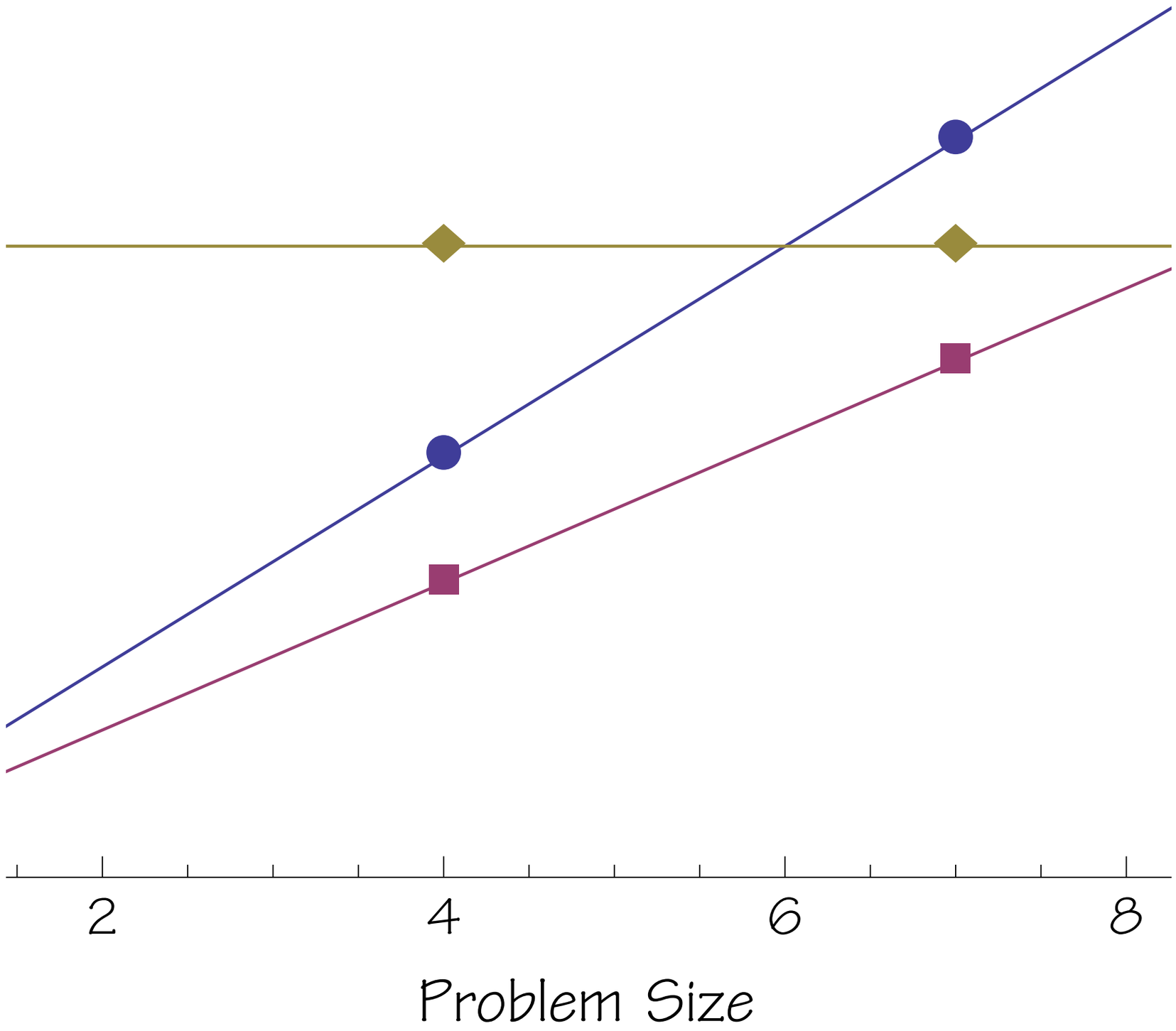
```
(define (sublists L)
  (if (empty? L)
      '(())
      (let* ([it       (first L)]
             [lose-it (sublists (rest L))]
             [use-it  (map (lambda (l) (cons it l)) lose-it)])
        (append use-it lose-it))))
```

How “good” are these solutions?

Are they efficient?
Do they “cost” more than they should?

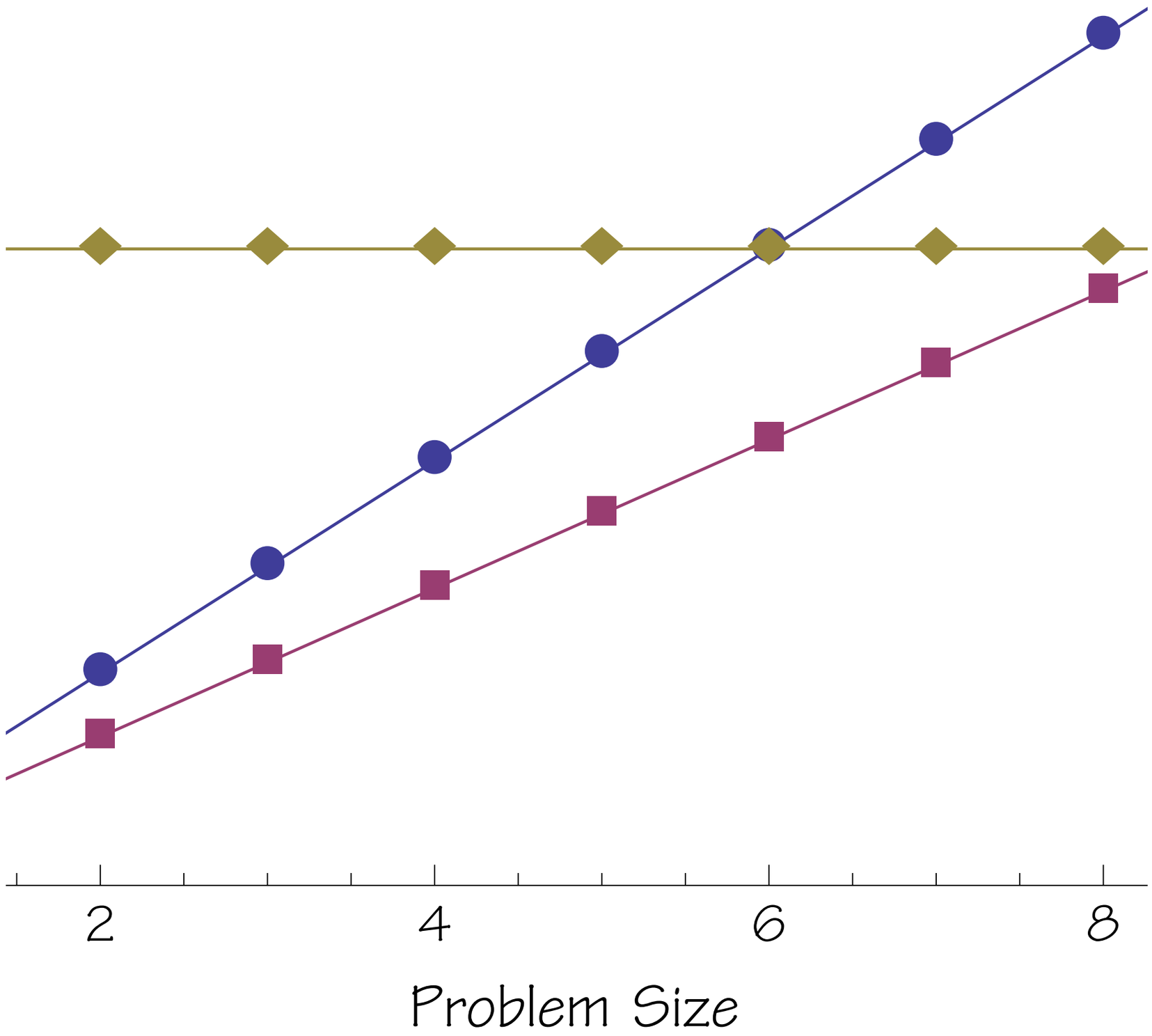
Data: which algorithm is best?

Lower is better



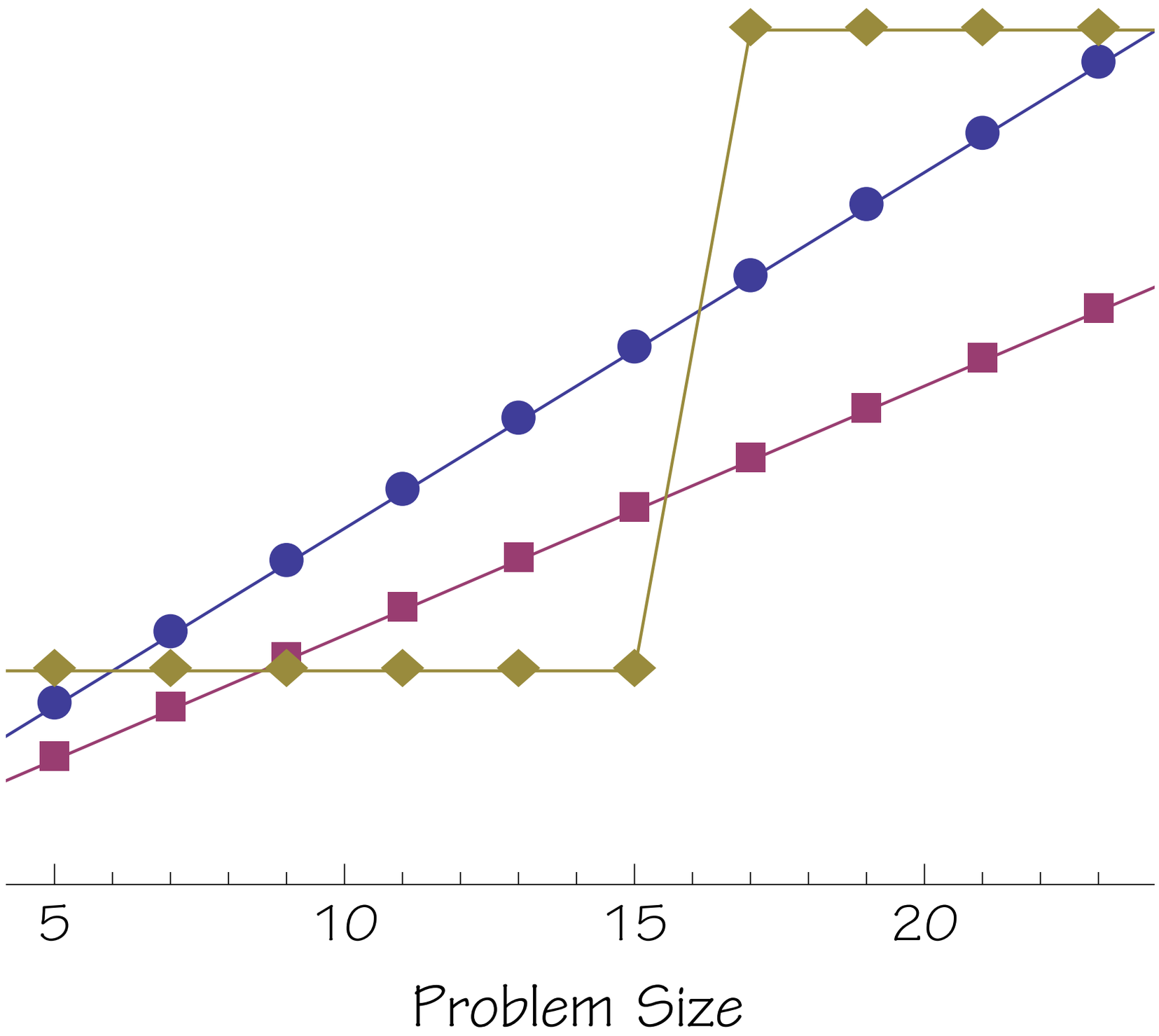
Data: which algorithm is best?

Lower is better



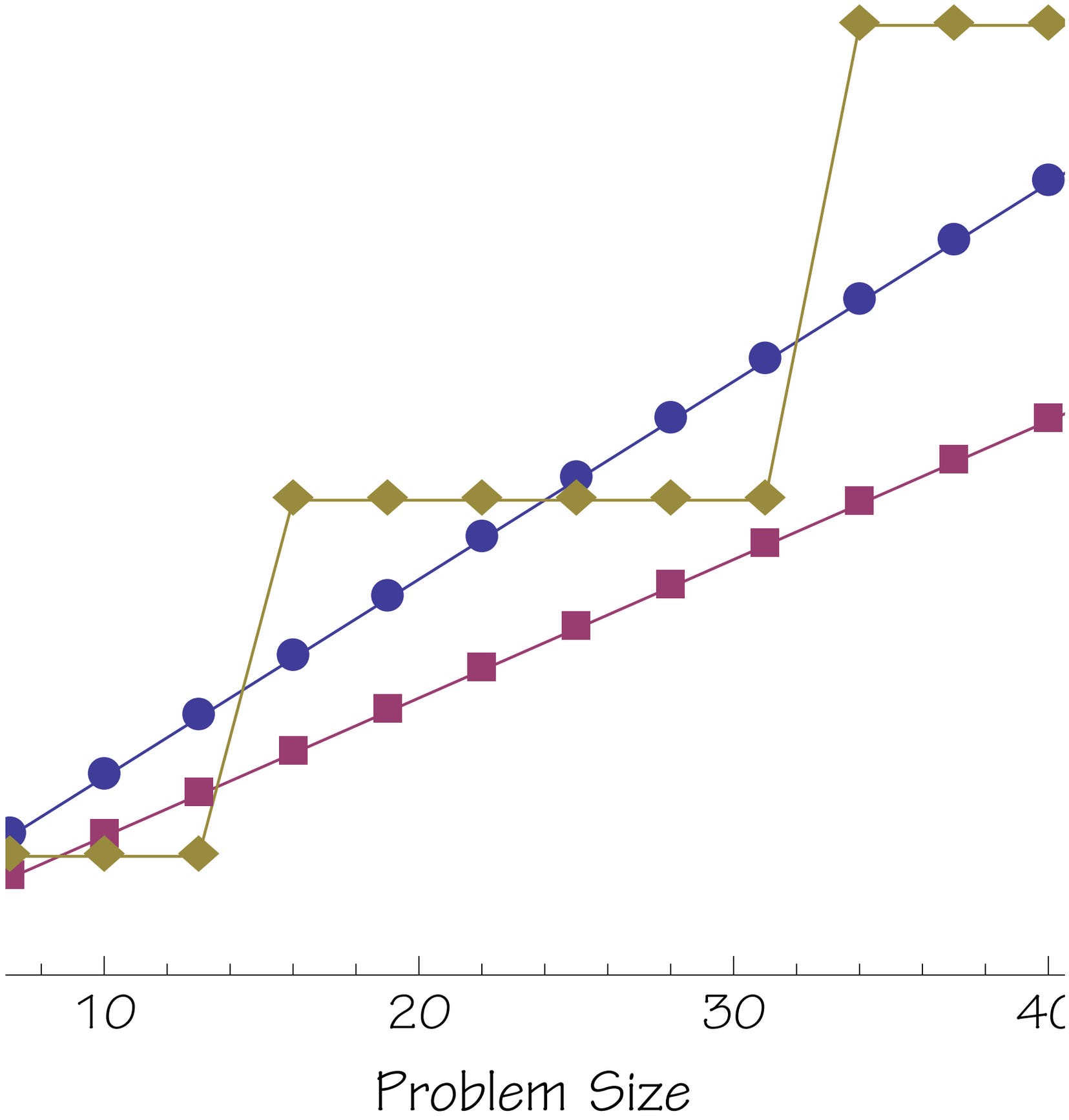
Data: which algorithm is best?

Lower is better



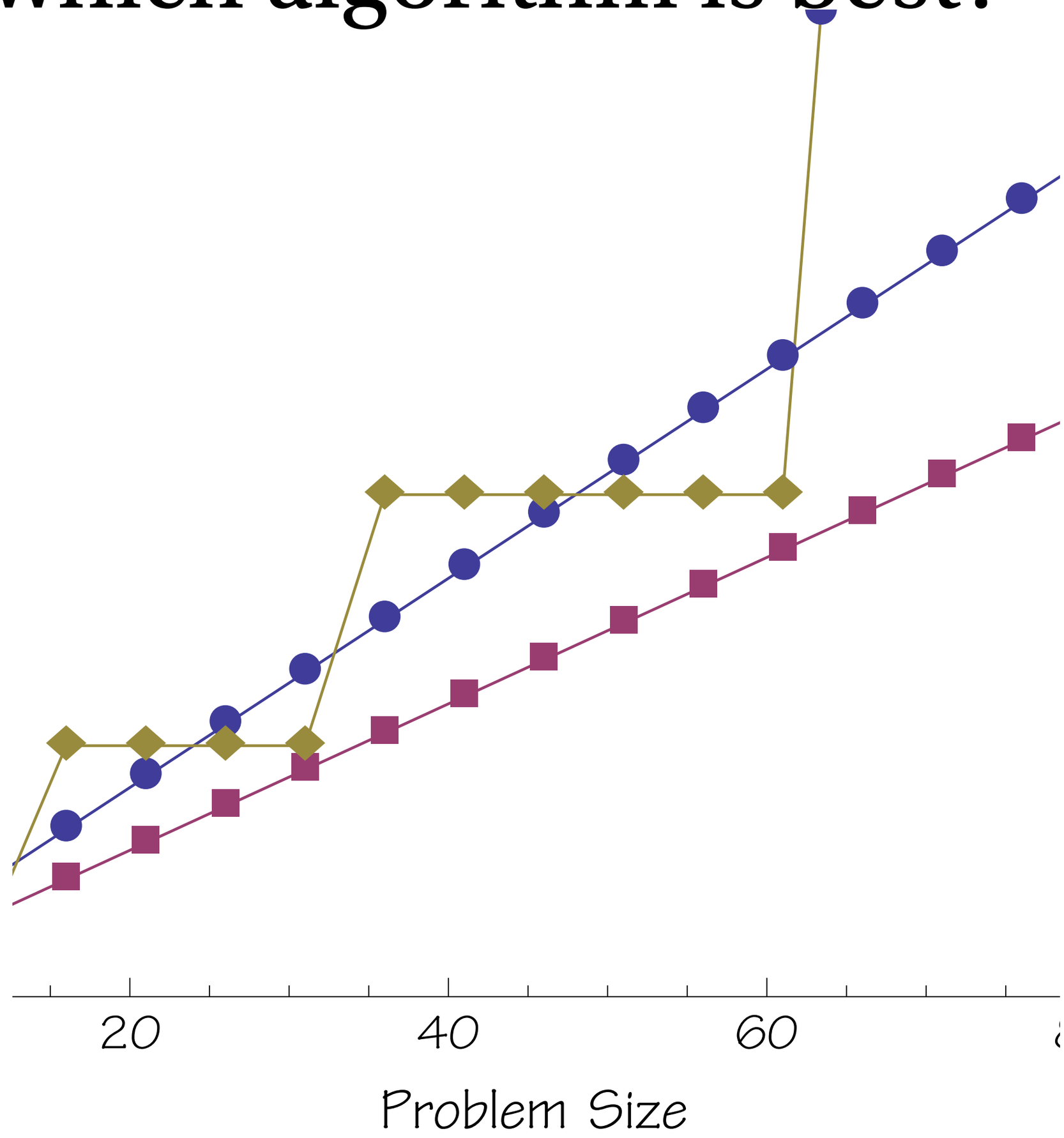
Data: which algorithm is best?

Lower is better



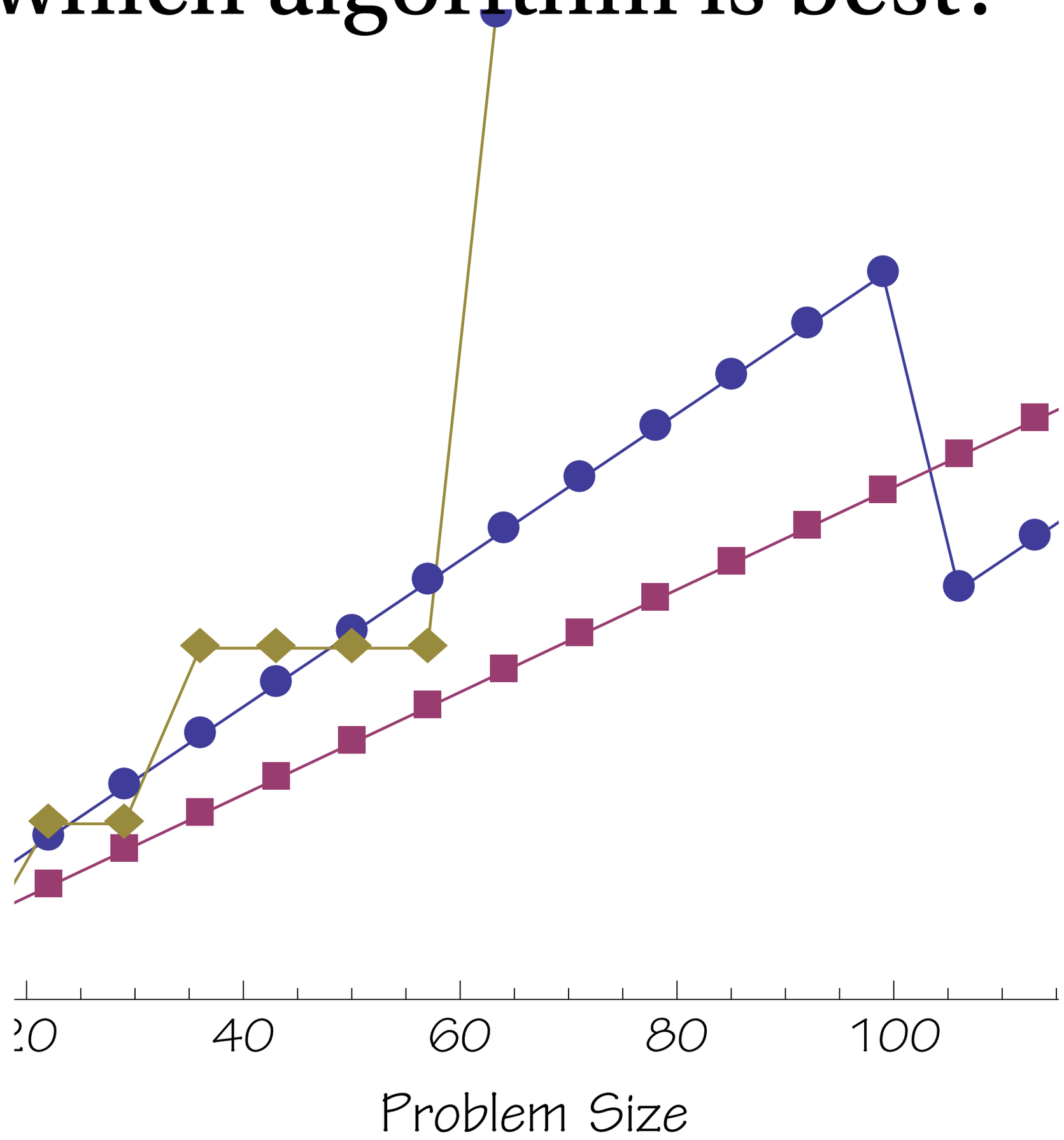
Data: which algorithm is best?

Lower is better



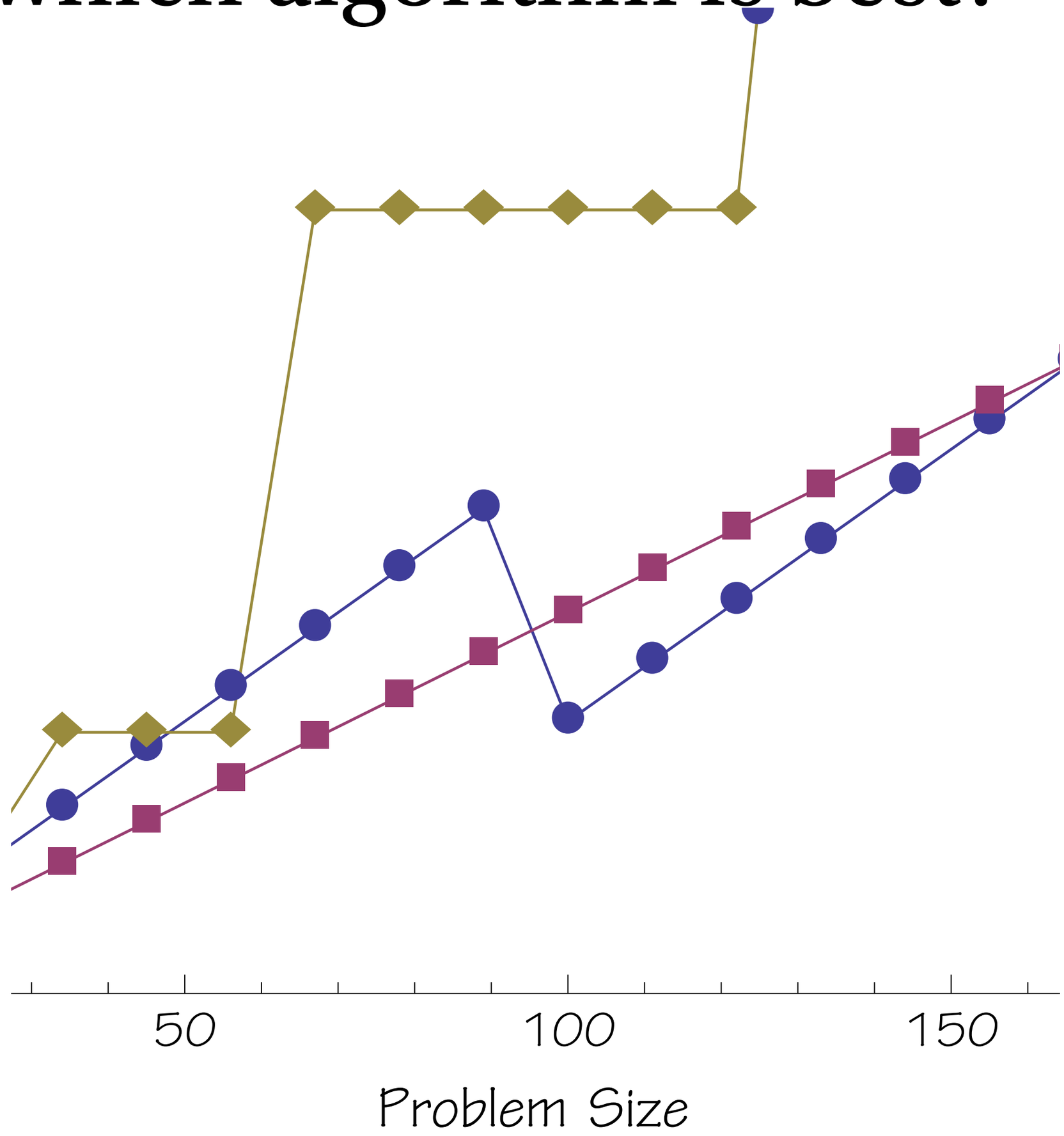
Data: which algorithm is best?

Lower is better



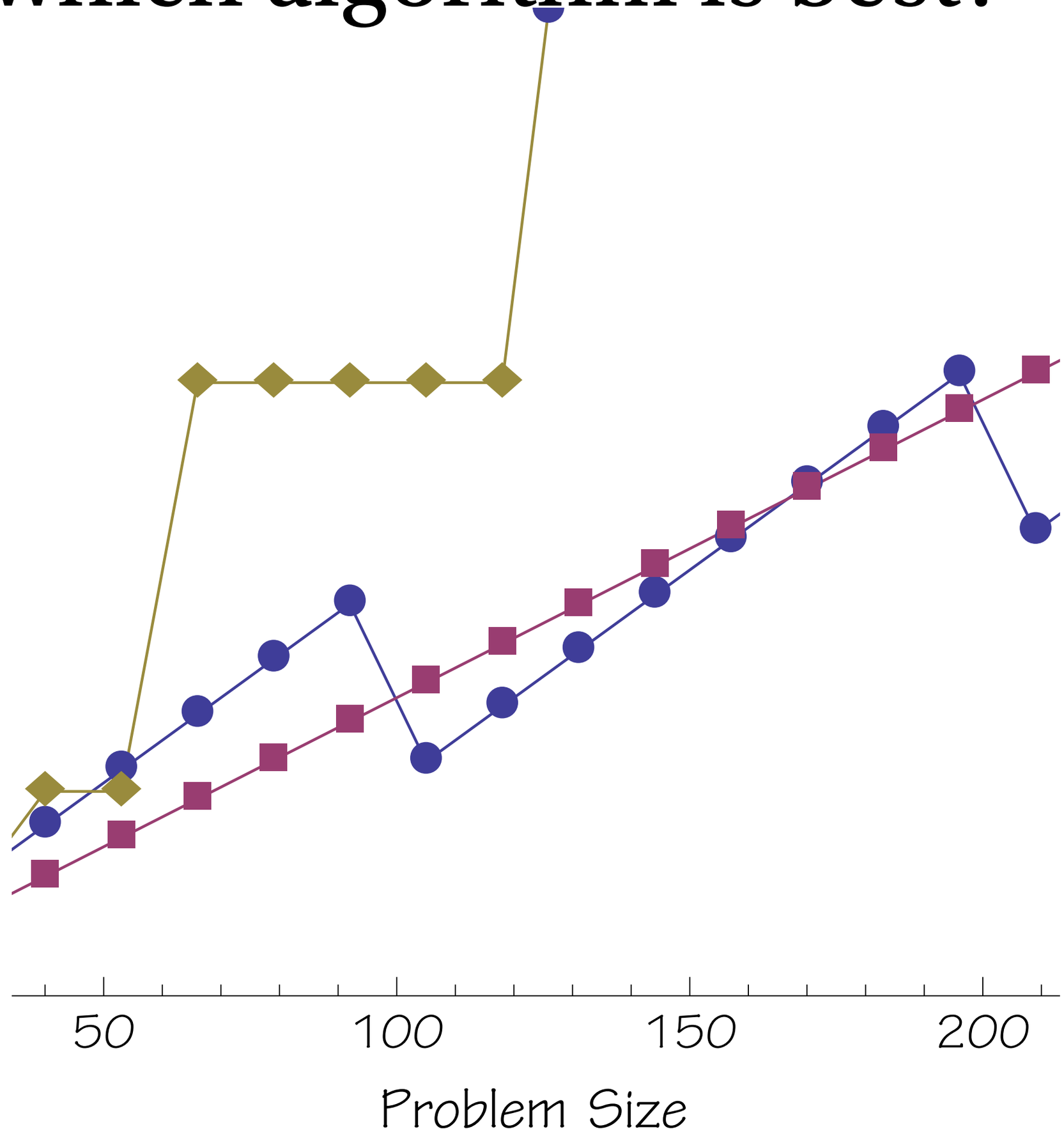
Data: which algorithm is best?

Lower is better



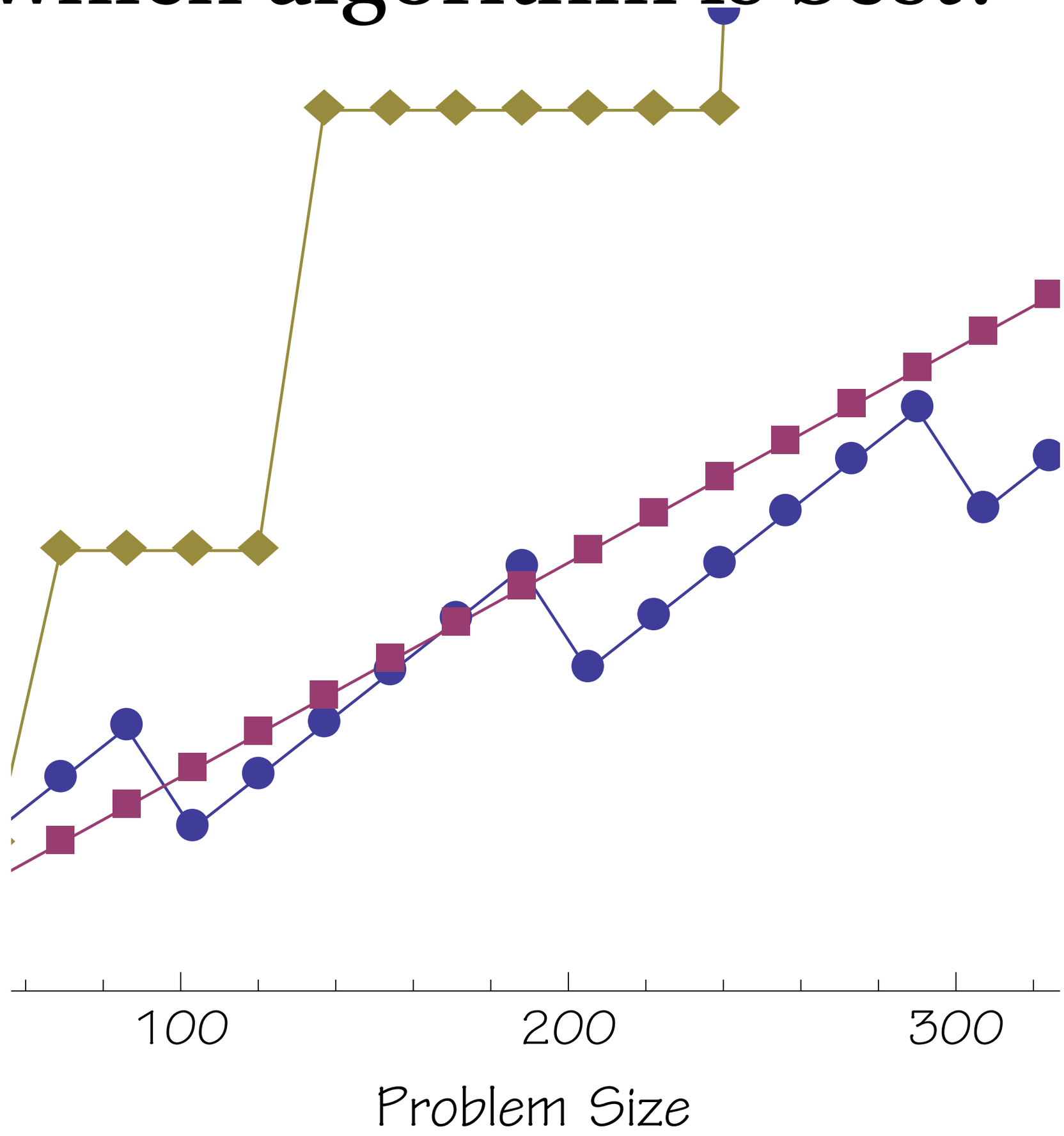
Data: which algorithm is best?

Lower is better



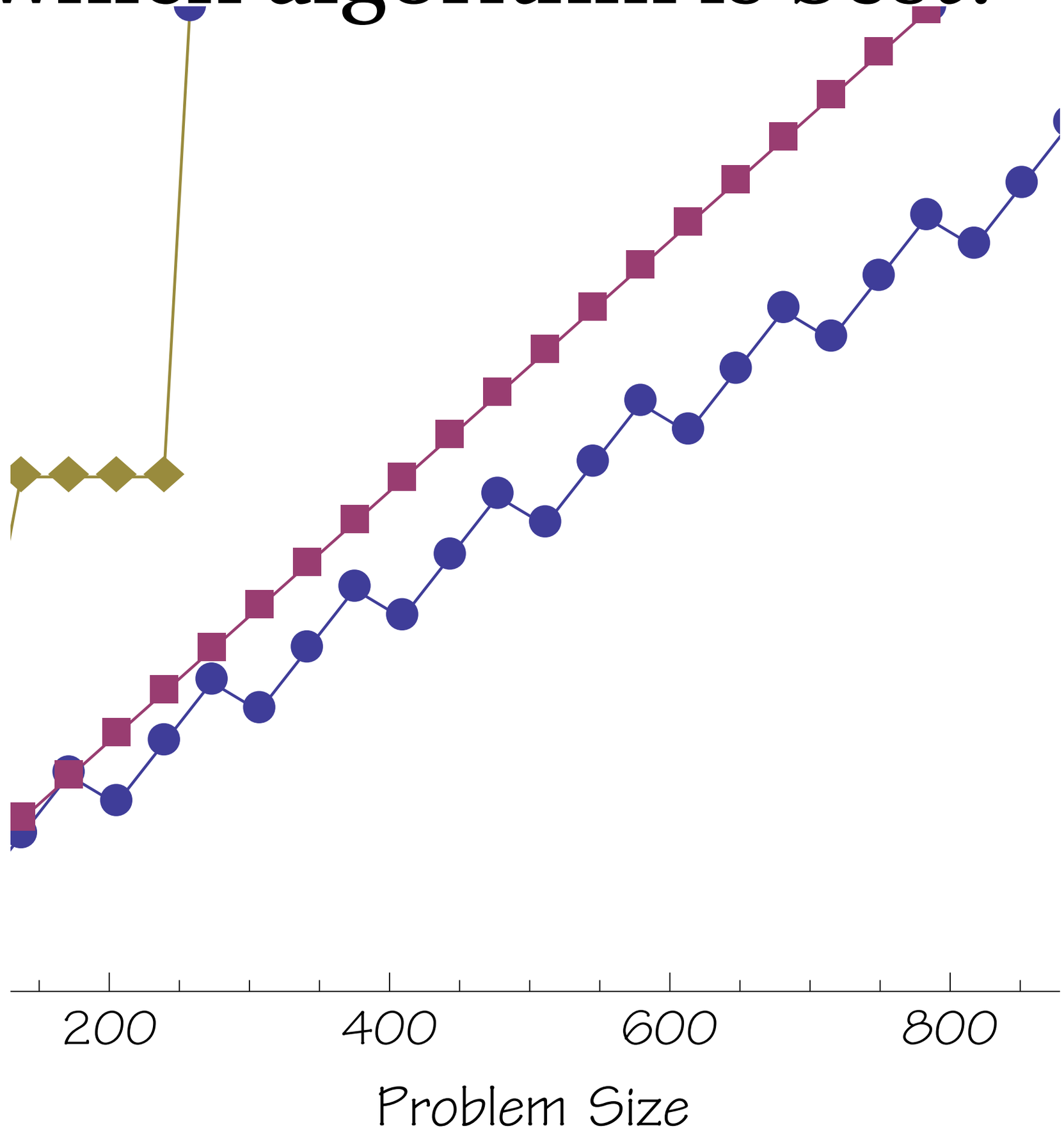
Data: which algorithm is best?

Lower is better



Data: which algorithm is best?

Lower is better



Interpreting empirical data

Key take-away: it's **messy** and **incomplete**!

We can measure

- a particular **algorithm**
- written in a particular **language**
- as a particular **program**
- compiled using a particular version of a particular **compiler**
- with particular **settings** (e.g., enabling / disabling optimizations)
- running on a particular **data set**, of a particular **size**
- on a particular **computer**
- with particular **resources** (CPUs, memory, hard drive, …)
- under a particular version of a particular **operating system**
- in a particular **environment**

  e.g., with other programs running in the background

Interpreting a theoretical model

Key take-away: it's **lossy**!

A theory abstracts away certain details.

cost metric:

- corresponds to one "step"
- highlights the essence of the work
  e.g., multiplications, comparisons, function calls…
- serves as a proxy for an empirical measurement

Instead of measuring time, we count steps.

e.g., "This algorithm costs n^2 multiplications."

good data + good theory

=

good science

we can make predictions and
we can communicate with other scientists

Asymptotic Analysis
(Big O)

Asymptotic analysis

We're always answering the same question:

How does the cost ***scale***
(when we try larger and larger inputs)?

Not:

- Exactly how many steps will it execute?
- How many seconds will it take?
- How many megabytes of memory will it need?

The informal definition of “Big O”

A *reasonable* upper bound on
(an abstraction of)
a problem’s difficulty or
a solution’s performance,
for *reasonably* large input sizes.