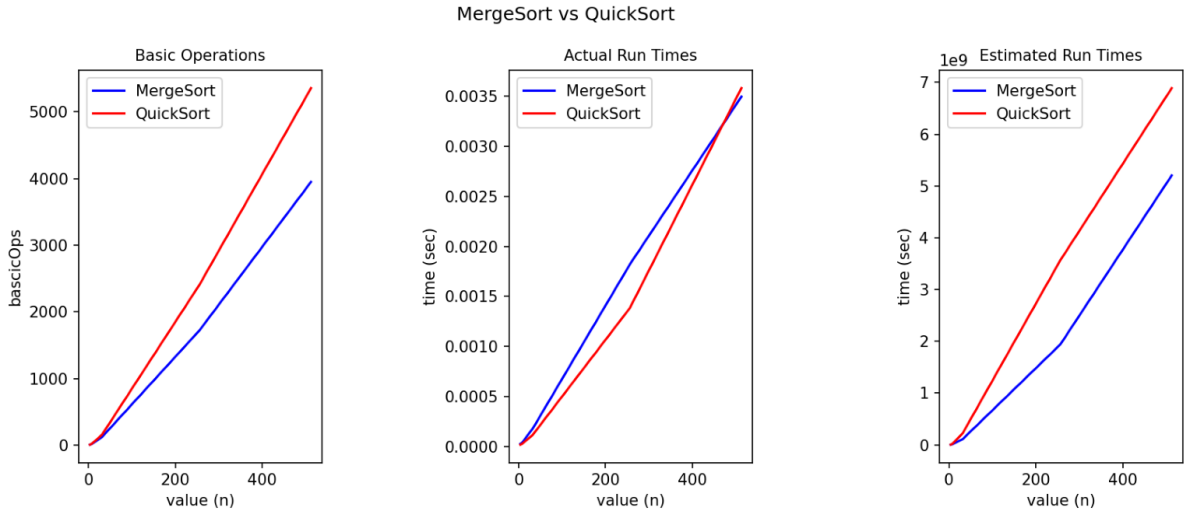
Project 1 Divide and Conquer Writeup

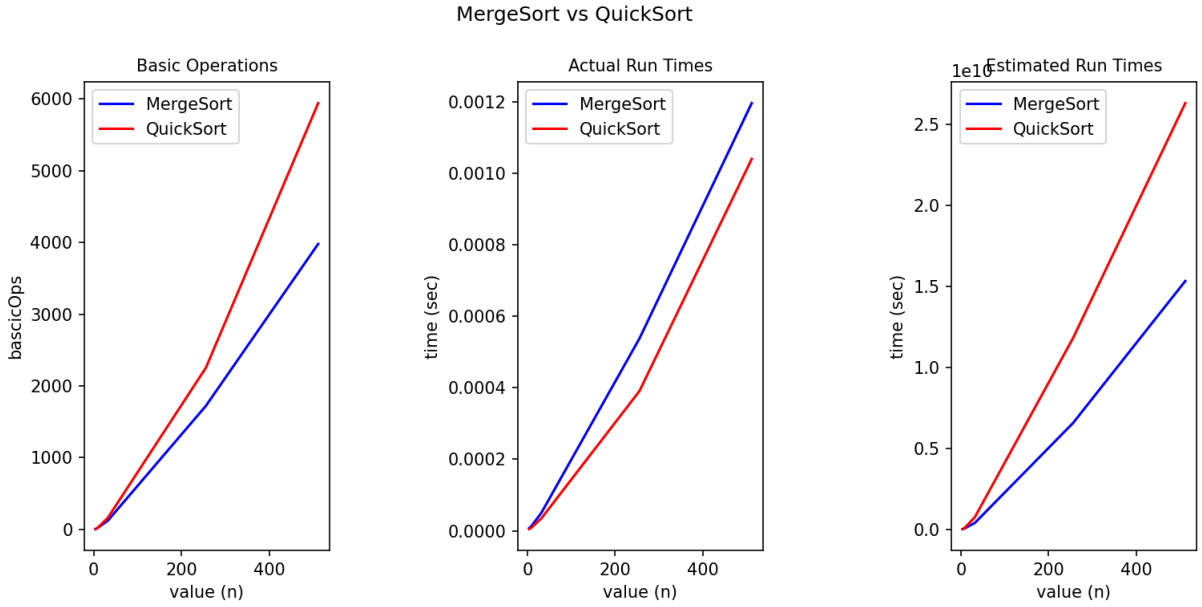
Skyler Ballard and Michael Shipley

The Experiment:

Run 1: this is an example where mergesort was faster than quicksort when n=512, this happens less than 5% of the time

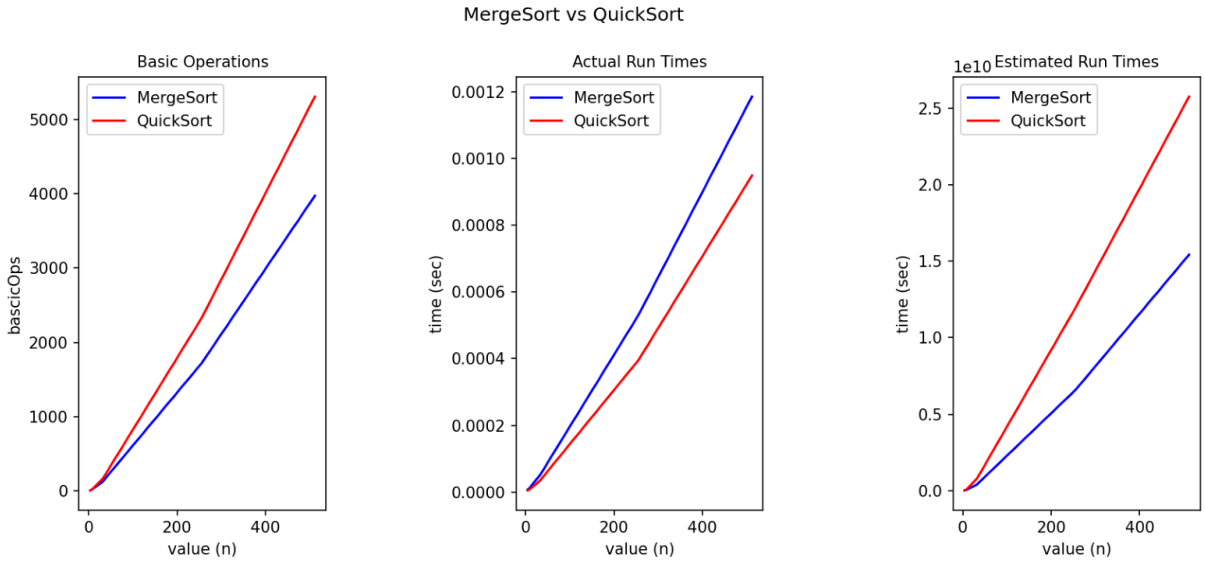


Run 2:



Page Break

Run 3:



The Analysis:

This result surprised me because: even though quicksort did more comparisons (basicOps) and the time estimations for quicksort predicted that it would take longer, it was in fact faster than mergesort. Mergesort was actual slower with less comparisons, and having the better estimated runtime(s). I did run the test over many times and did find every once in a while mergesort would

win a single round during the actual runtimes, but on average 90% + quicksort would win every time.

Regarding the research conducted on QuickSort vs MergeSort, Skyler and Michael have found the following observations regarding the two algorithms aforementioned.

QuickSort is generally faster than MergeSort. In practice, Quicksort performs less allocation than MergeSort, a result of Quicksort being in place. Quicksort also only performs faster when the data is truly random, more on this later.

Mergesort generally has fewer operations than QuickSort. This is due to Mergesort consistently sorting. Quicksort may move a piece of data more than twice. Mergesort only moves a piece of data's order once to Merge. Less basic operations did not translate to a faster algorithm.

Our estimated times were often much lower and inverted to actual times when presented with datasets not of 2^n size. When presented with 2^n size datasets the actual time was magnitudes lower than estimated times. This is likely due to the algorithms being drastically more efficient at datasets in 2^n restriction. Merge sort splits in half and Quicksort compares two at a time, making both naturally more inclined to even data sets. This is shown in our collected data that it does take more time on odd datasets than even datasets. However it seems strange that the difference is in a matter of x1000 and not a few nanoseconds.

Our initial estimates were based on datasets that were multiples of 5. I attempted to reduce the inaccuracy by trying data sets that were of powers of two. The results were better than I hoped to reduce the time, but made our estimates largely inaccurate! The error with datasets as multiples of 5 were off by a few seconds at most, but in sets of powers of 2 the estimate was off by a magnitude of 1000. This is because sets of non powers of 2 are average case, and sets of powers of 2 are best case, for both algorithms. In practice it would be advantageous to fill sets to powers

of two for sorting purposes. The deltas however are fairly similar to the actual result. This is shown by our side by side data of various tests. The powerset of 2^n is remarkably more accurate to the individual algorithms, but not wholly so as it predicts incorrectly which will be more efficient. With a notable expectation discussed after the proof.

Worst Case QuickSort

Proof by induction: Using the Worst-Case Time Complexity of Quicksort

- Using a list/array of size n
- n is a power of 2 (n=2^k)

We have a recurrence relation:

- W(1) = 0
- W(n) = W(n - 1) + n - 1

Hypothesis: T(n) = (n * n - n) / 2

Base: T(1) = (1*1 - 1) / 2 = 0/2 = 0 which is true

If it is true for n, then it is also true for (n + 1), substitute (n + 1) for n

Step: T(n + 1) = (((n + 1) * (n + 1)) – (n + 1)) / 2

Left Side:

= T(n + 1)

= T(n) + n And b/c T(n) = (n * n - n) / 2 We sub it into the equation

= ((n*n - n) / 2) + n

= (n*n - n + 2n) / 2

= (n^2 + 2n - n) / 2

= (n^2 + n) / 2

= (n(n + 1)) / 2

Right Side:

= (((n + 1) * (n + 1)) - (n + 1)) / 2

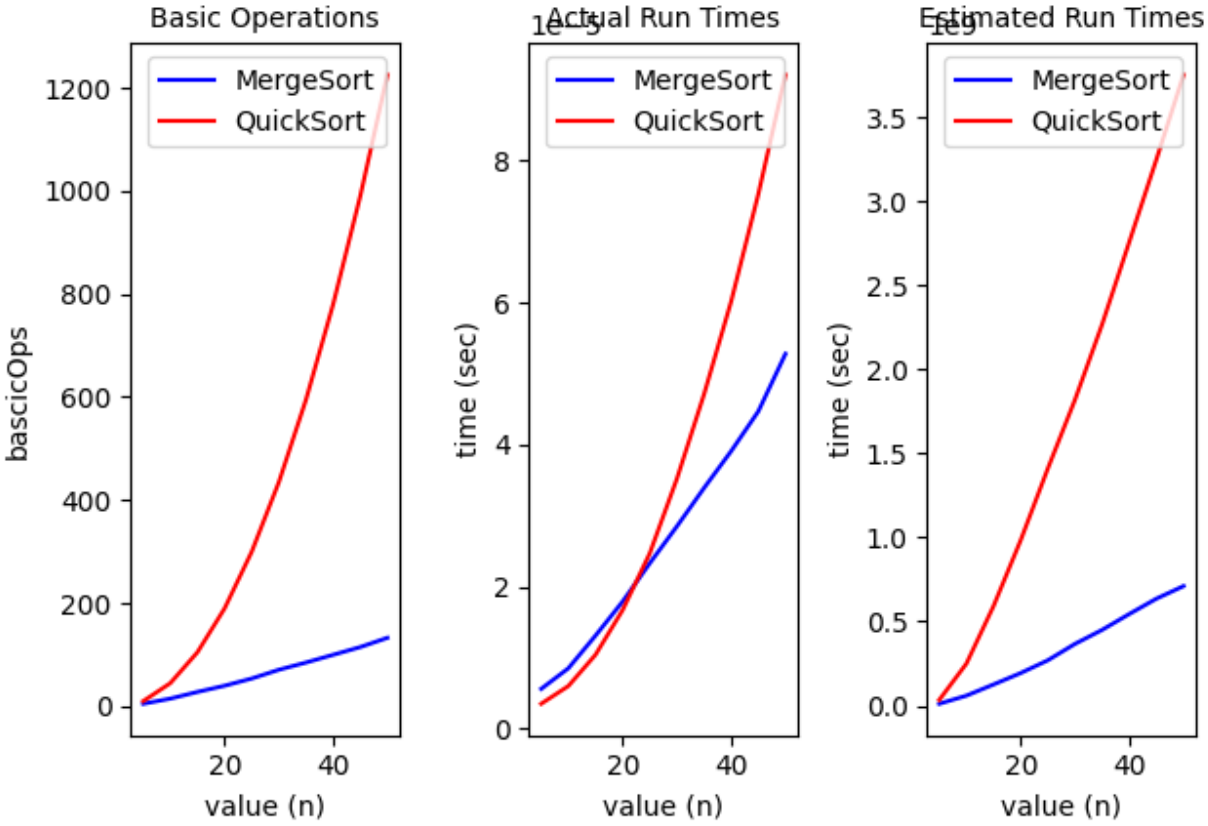
= ((n^2 + 2n + 1) - (n + 1)) / 2

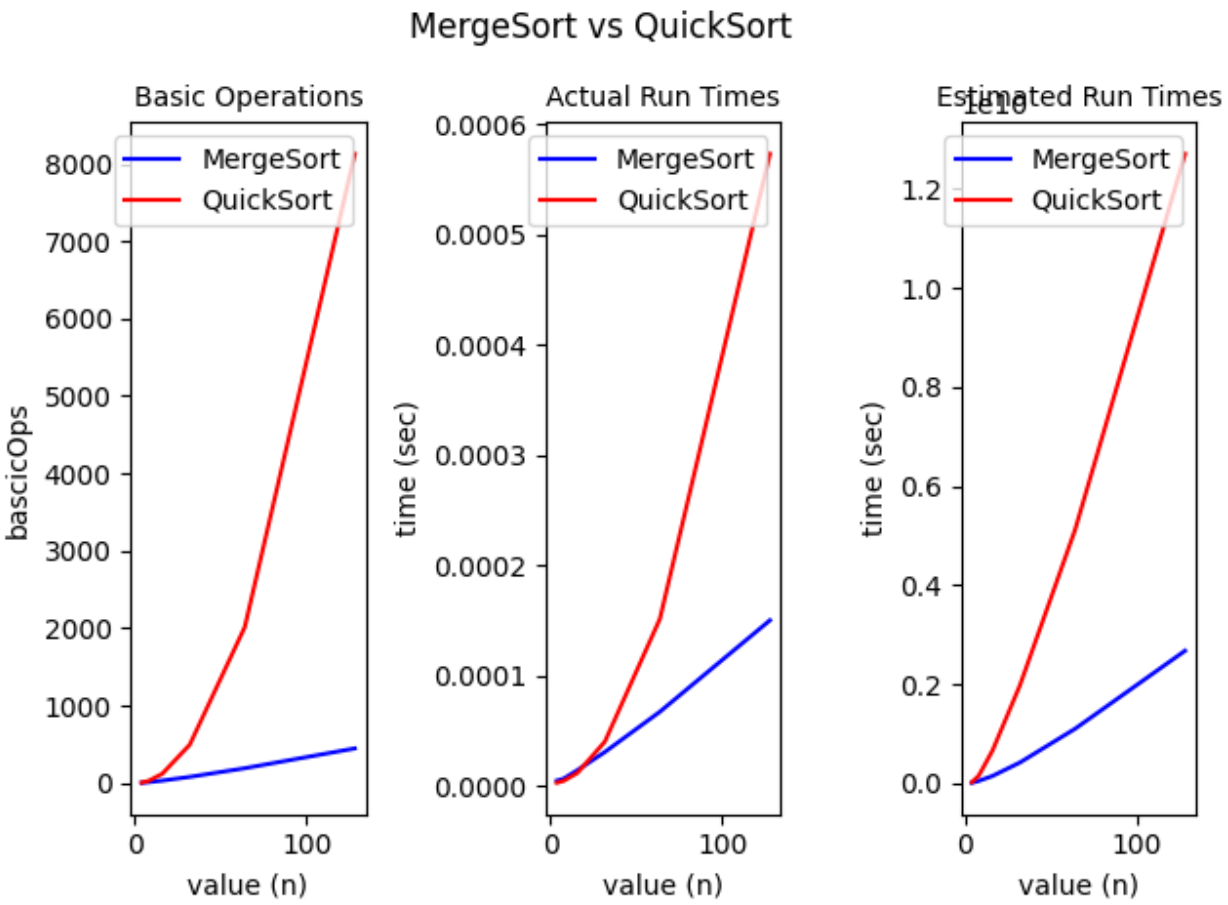
= (n^2 + 2n + 1 - n - 1) / 2

= (n^2 + n) / 2

= (n(n + 1)) / 2

We see that they are equal, therefore the recurrence equals the candidate solution


MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−5
1e9
0
200
400
600
800
1000
1200
2
4
6
8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
20
40

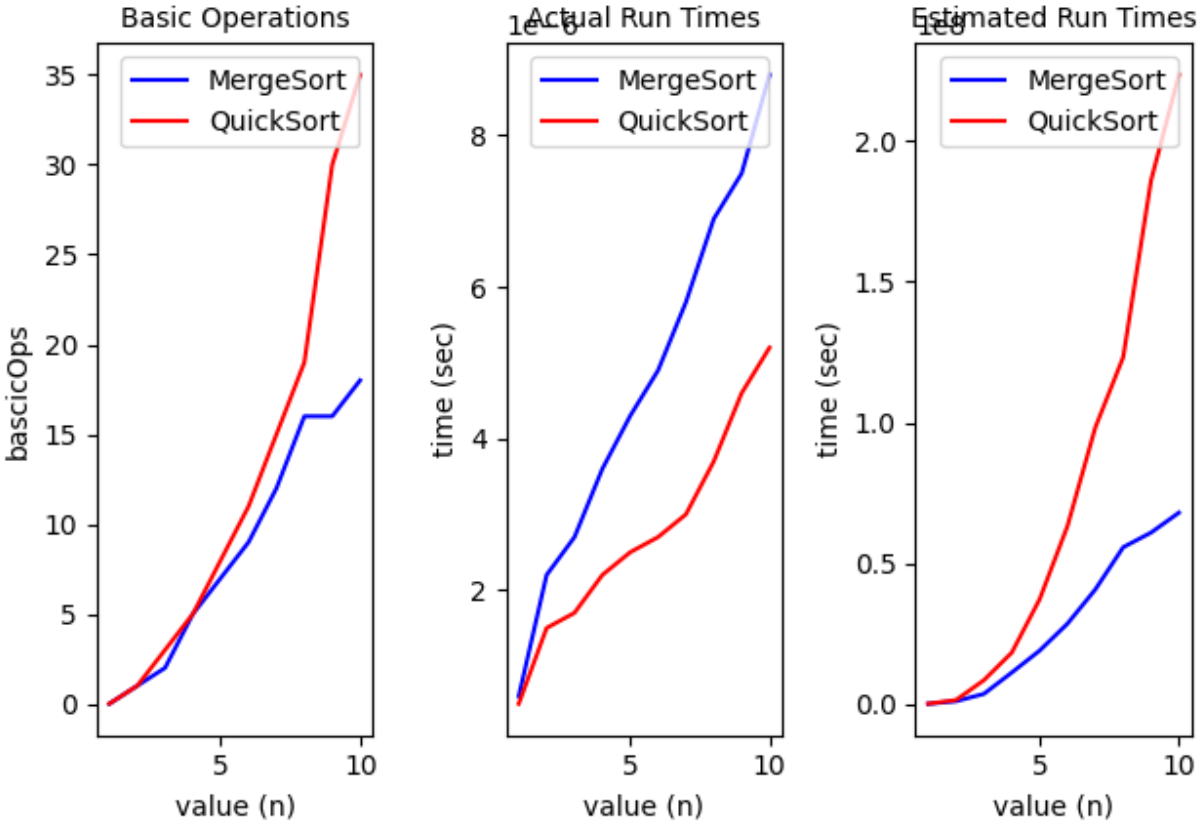


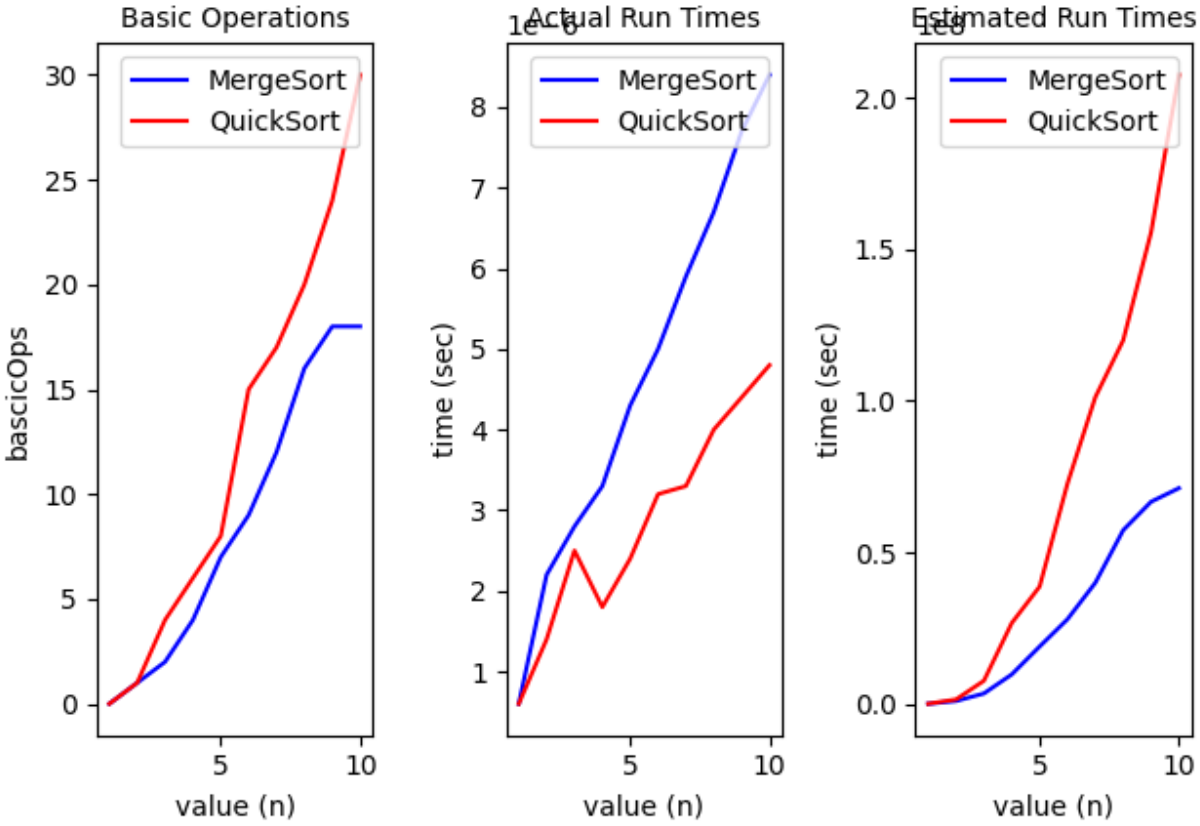
A note regarding the worst case of QuickSort: Quicksort has a fatal flaw. When the data is already sorted, or well ordered, it performs far worse, as estimated, than MergeSort. Also, in a later project we will make multithreaded MergeSort, which performs significantly better on its time quantum than QuickSort in large sets. Look forward to that data in those projects.

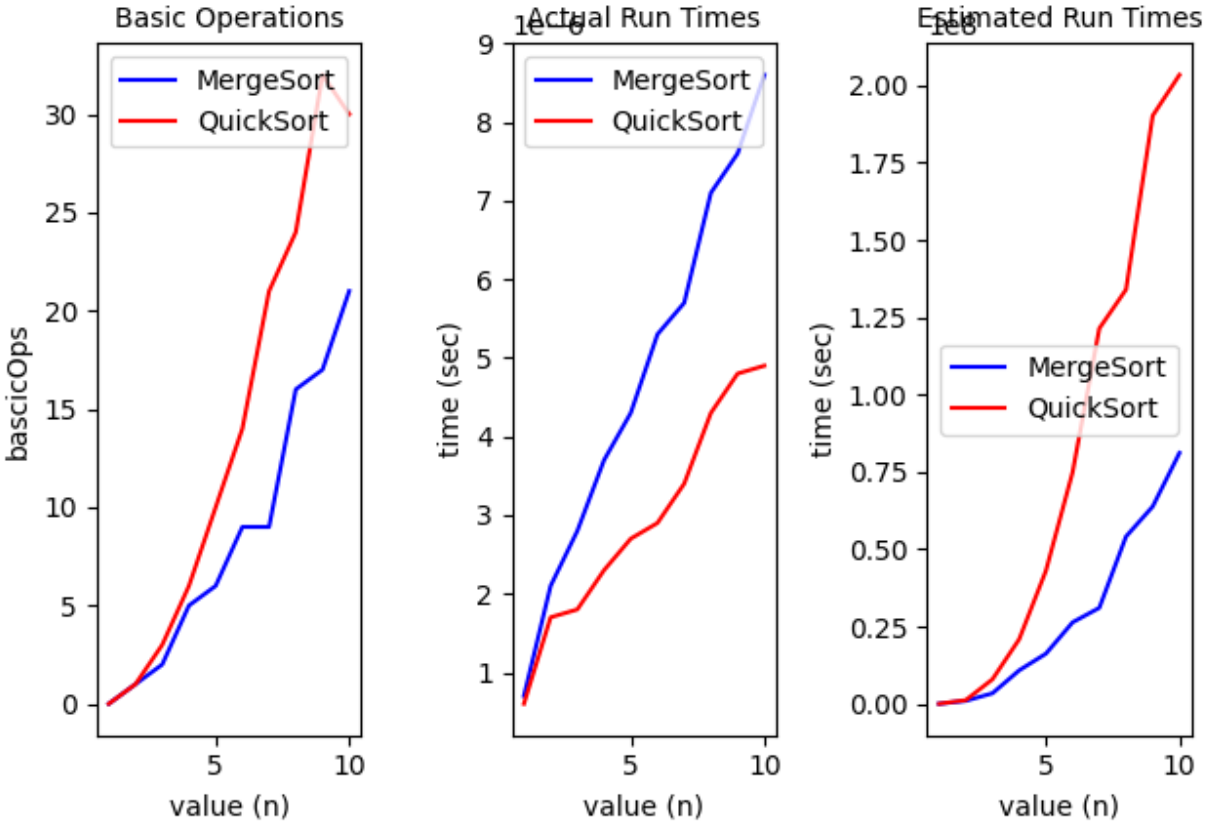
To test the well sorted theory, I implemented a random set that chose its element based on its index, meaning it was always sorted. This showed Mergesort to always perform better than Quicksort except in small datasets around size 4 to 32. This where mergesort still has an efficiency of nlogn and quicksort has an efficiency of n^2. This isn't an uncommon dataset either. Partially ordered sets fail to be fast on Quicksort. This should be considered as it can affect usage uptime of MergeSort, which does no worse on a partially ordered set. In the long run

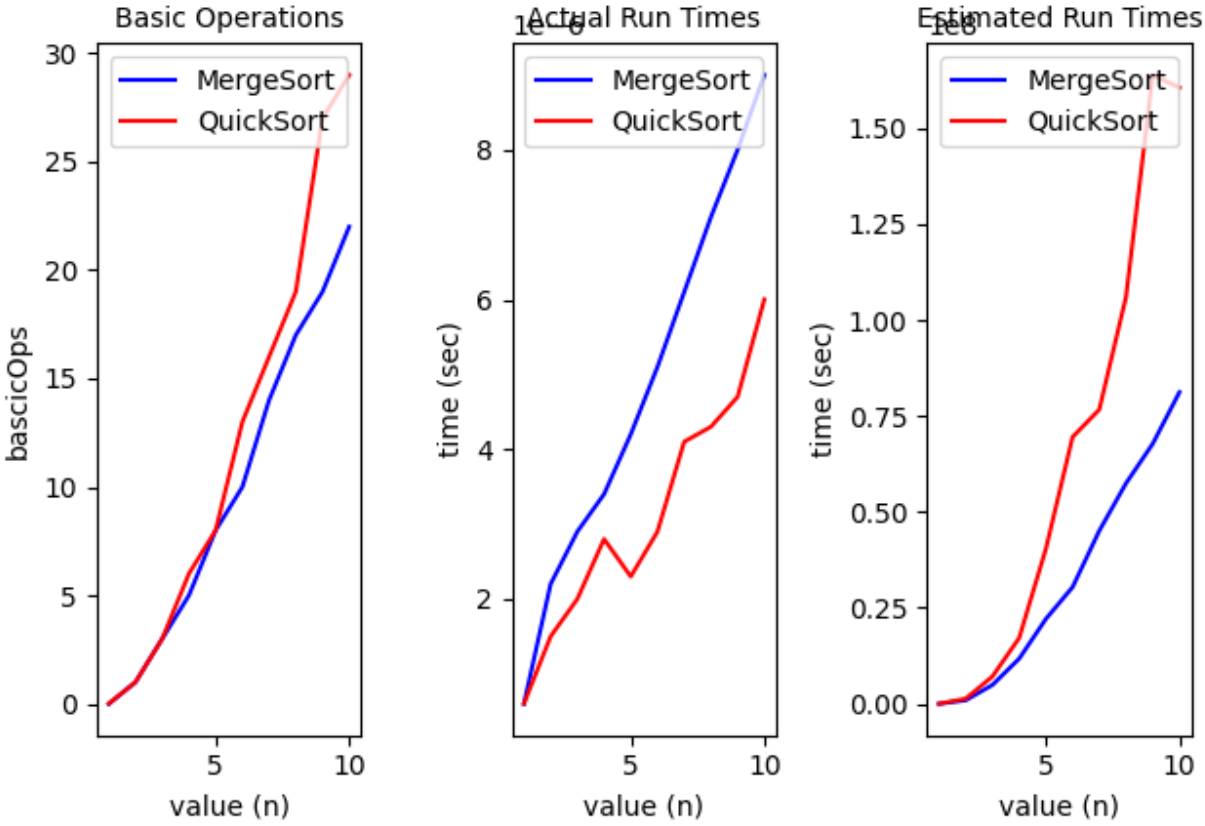
MergeSort should be used on kept data and Quicksort on new data. This is a lesson in knowing when to use an algorithm. It however does require knowledge regarding the set.

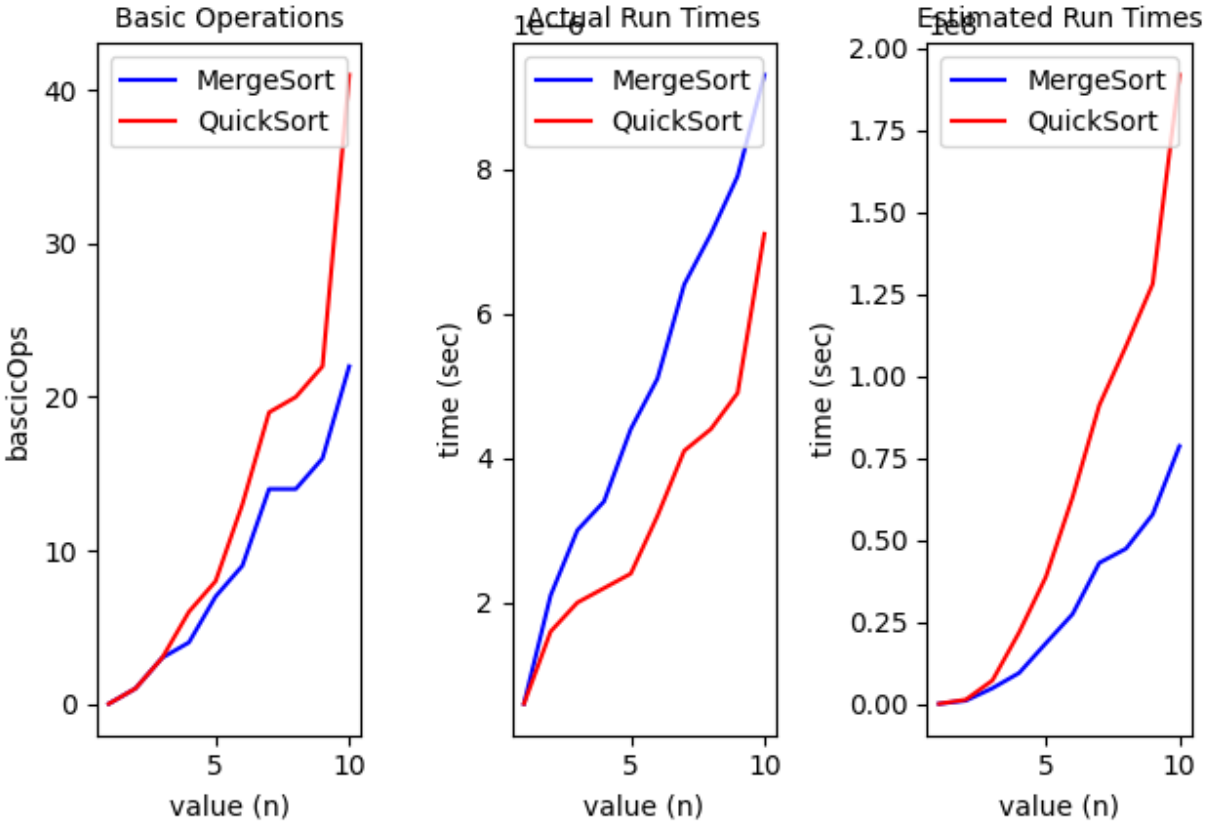
Another note, I did extensive testing and will include the folder of the results of each of the tests. The names on the tests are useful, which is why it had to be in a separate document.

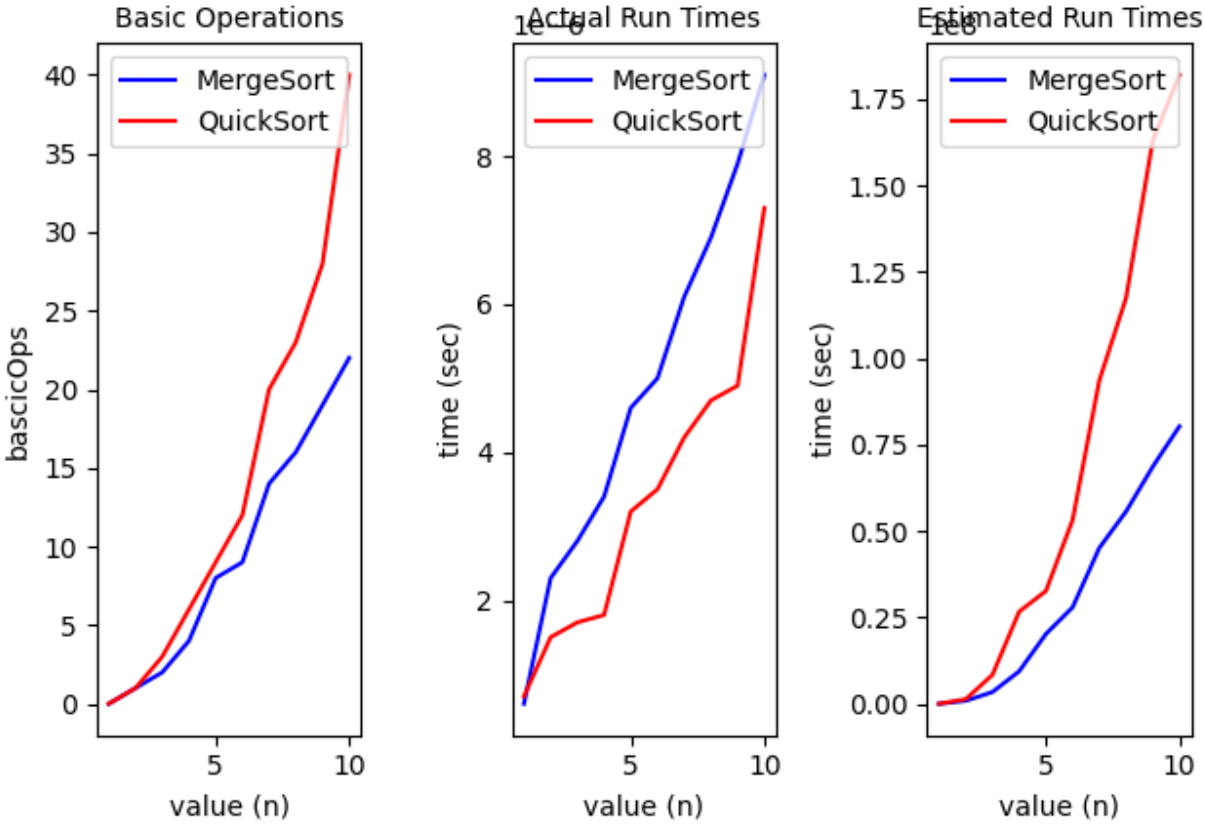

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
time (sec)
value (n)
value (n)
value (n)
1e−6
1e8
0
5
10
15
20
25
30
35
2
4
6
8
0.0
0.5
1.0
1.5
2.0

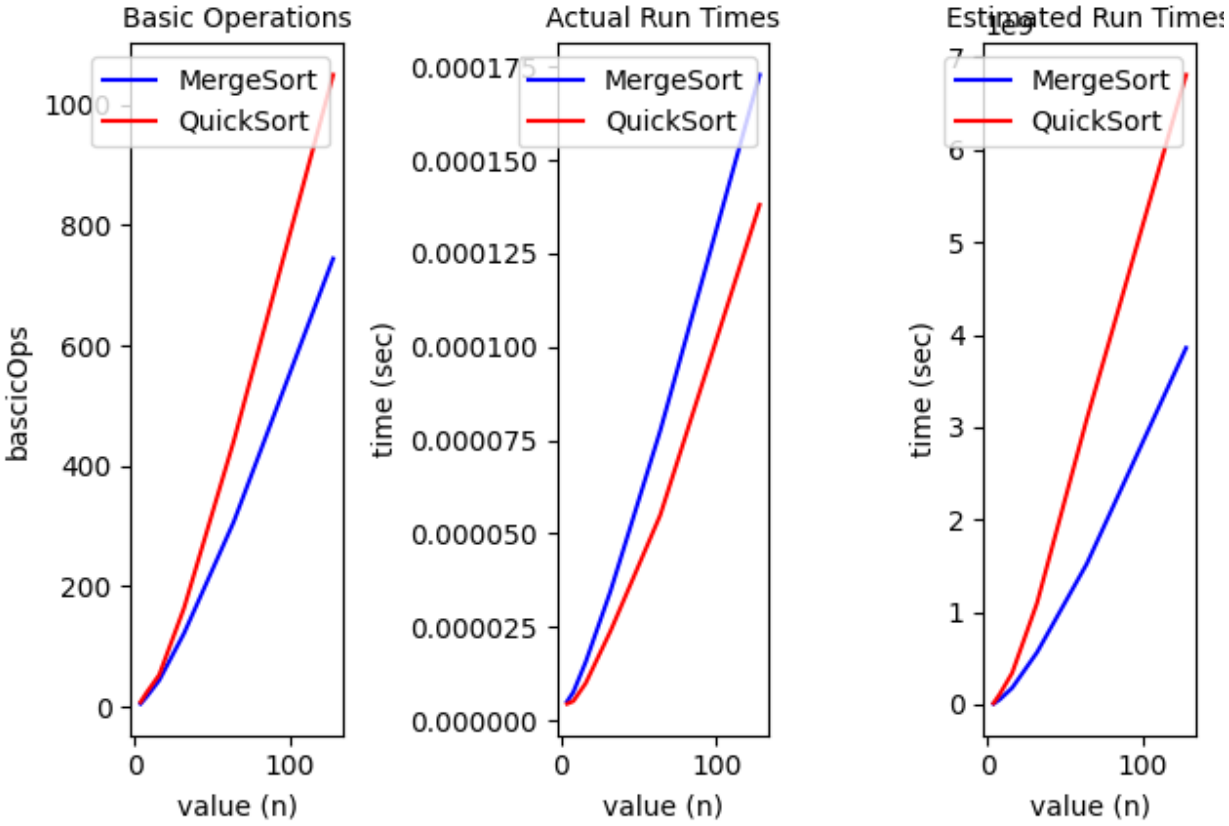

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−6
1e8

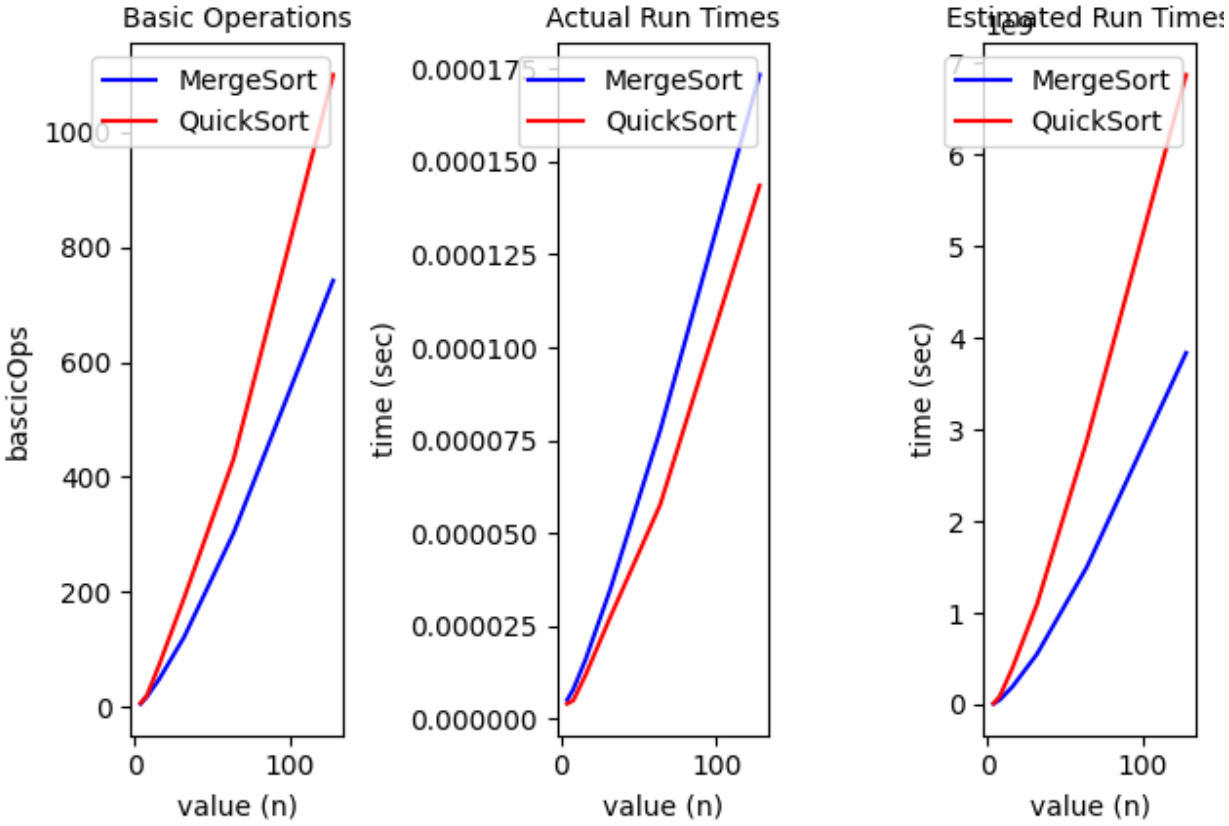

MergeSort vs QuickSort
Basic Operations
bascicOps
value (n)
MergeSort
QuickSort
Actual Run Times
1e−6
time (sec)
value (n)
MergeSort
QuickSort
Estimated Run Times
1e8
time (sec)
value (n)
MergeSort
QuickSort

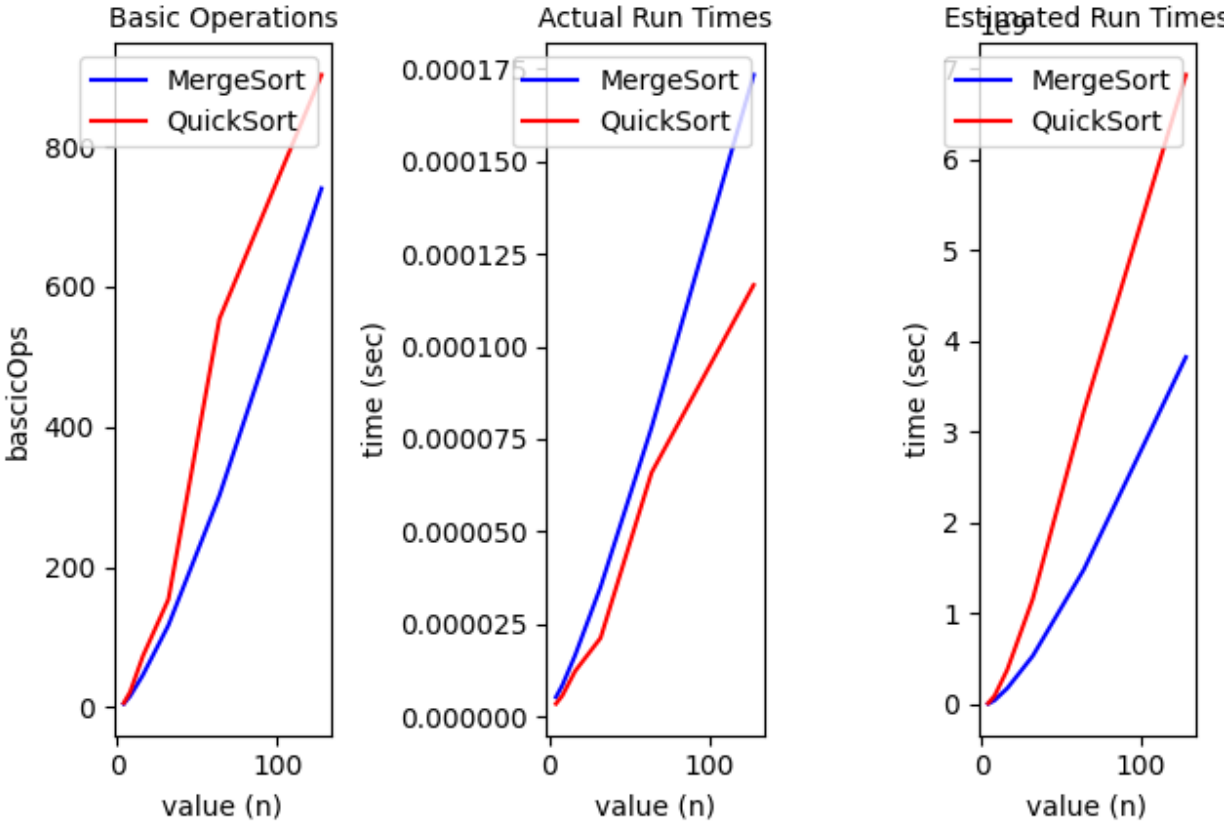

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−6
1e8

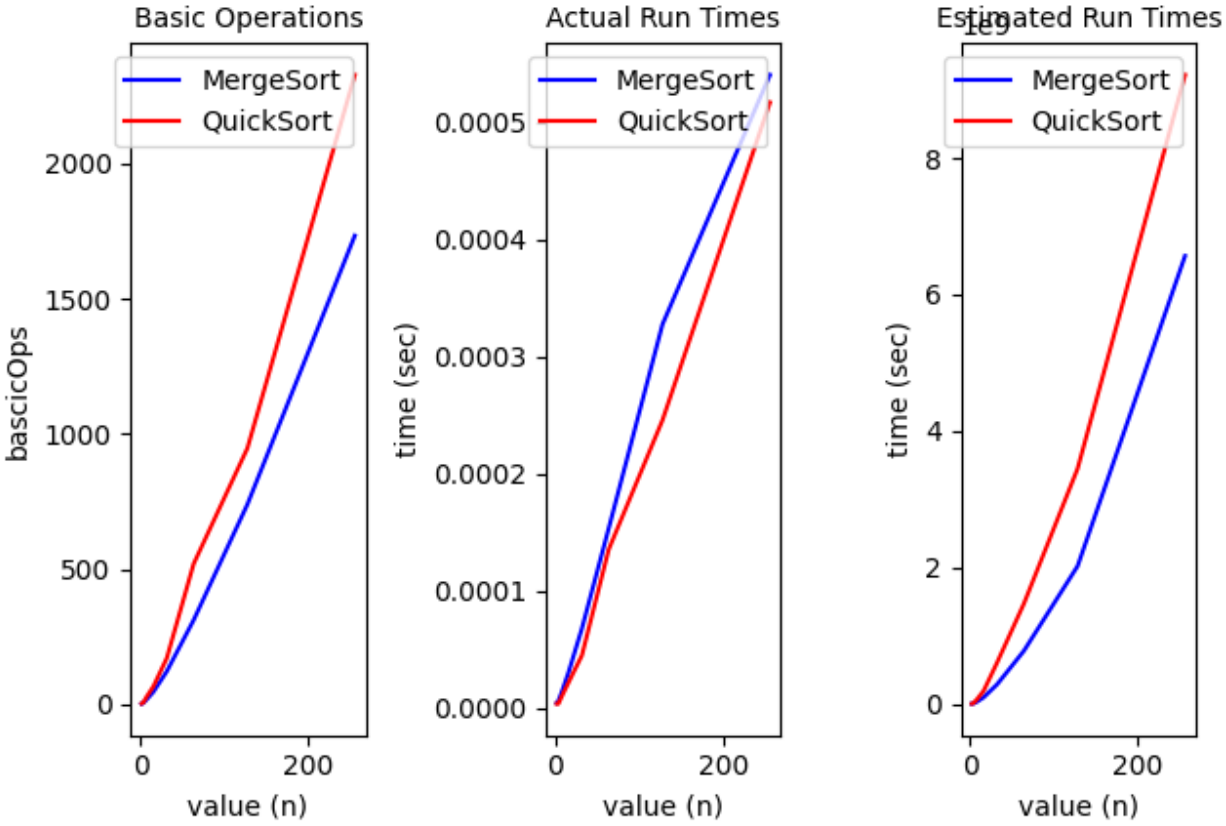

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−6
1e8
0
10
20
30
40
2
4
6
8
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
5
10

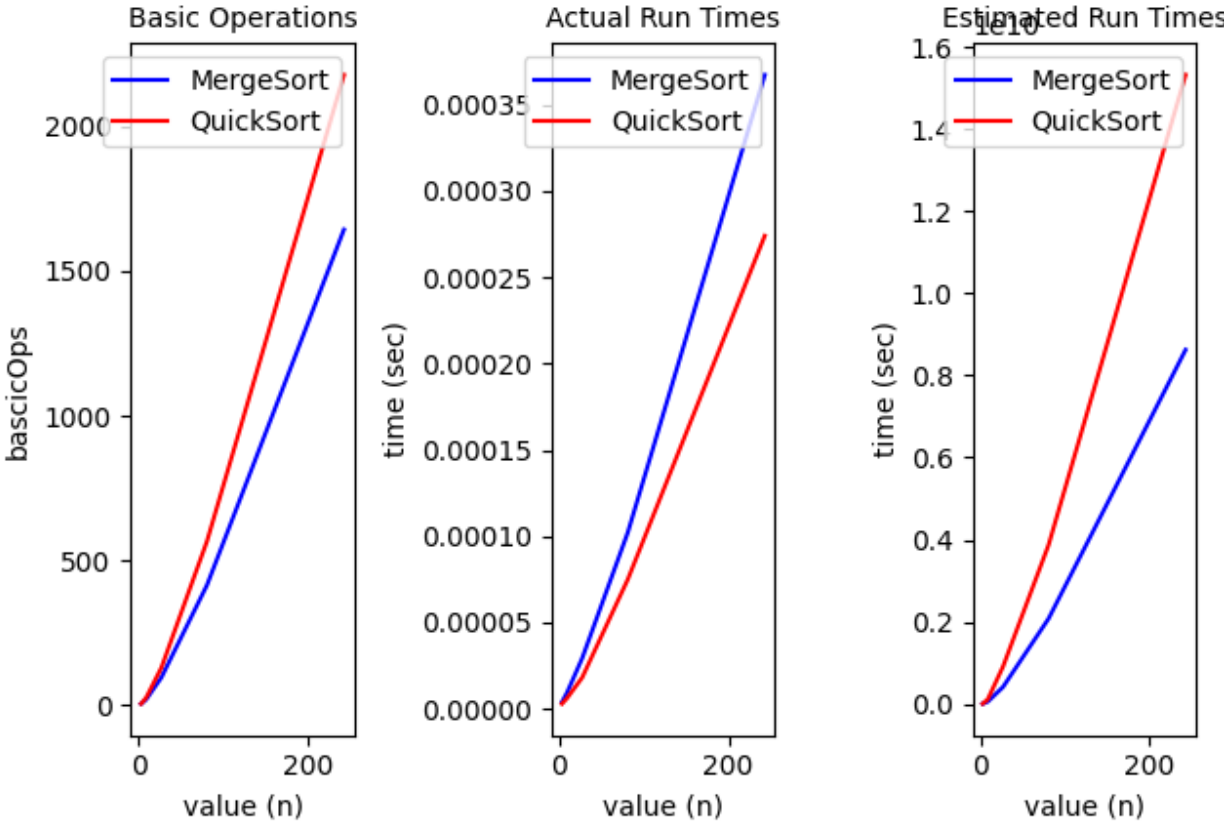

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−6
1e8
0
5
10
15
20
25
30
35
40
2
4
6
8
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75

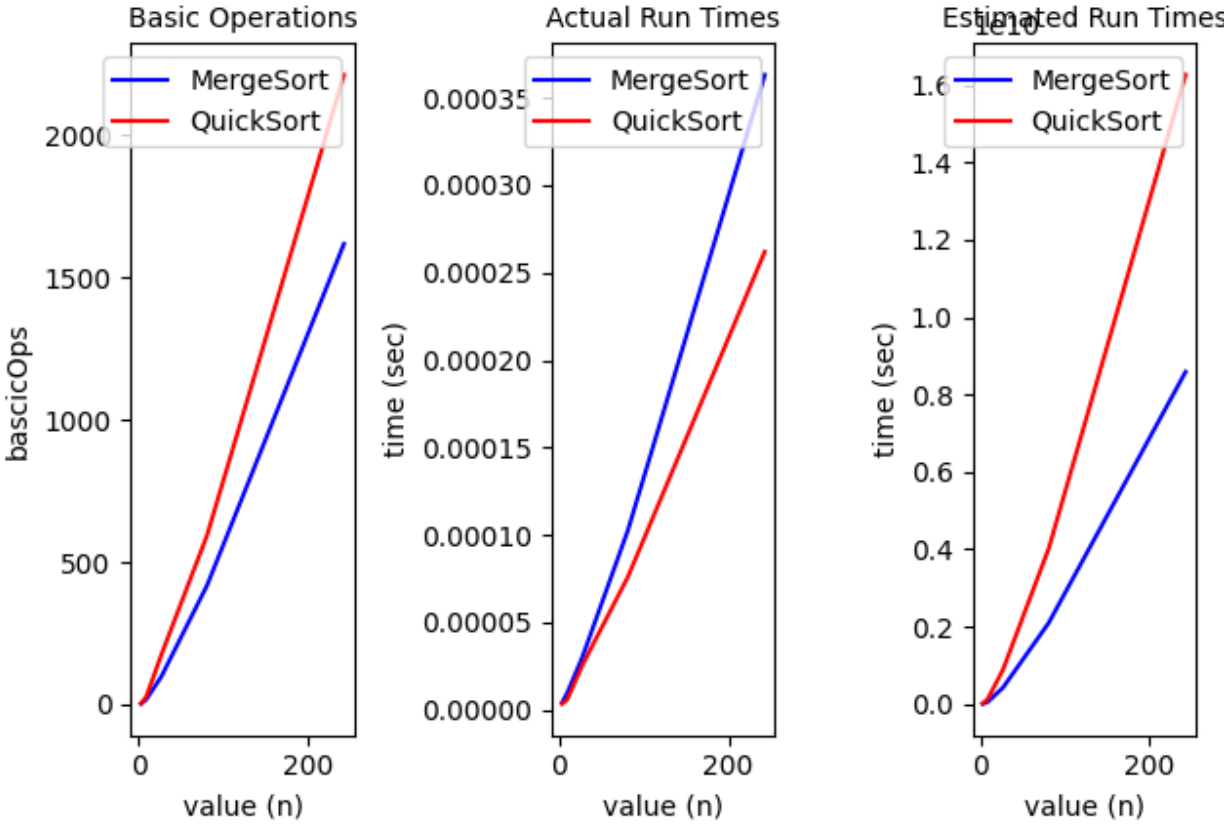

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
0
100
200
400
600
800
1000
0.000000
0.000025
0.000050
0.000075
0.000100
0.000125
0.000150
0.000175
1e9
1
2
3
4
5
6
7

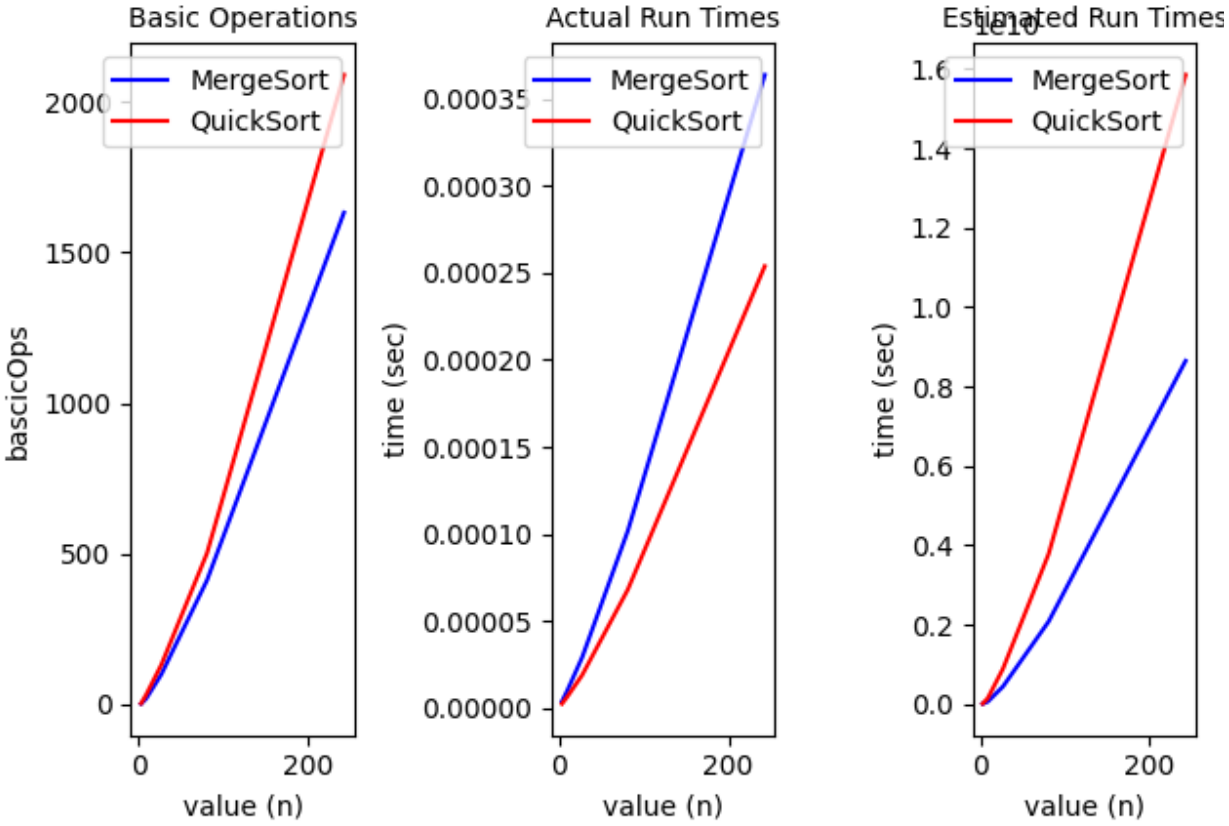

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
0
100
200
400
600
800
1000
0.000000
0.000025
0.000050
0.000075
0.000100
0.000125
0.000150
0.000175
1e9
1
2
3
4
5
6
7

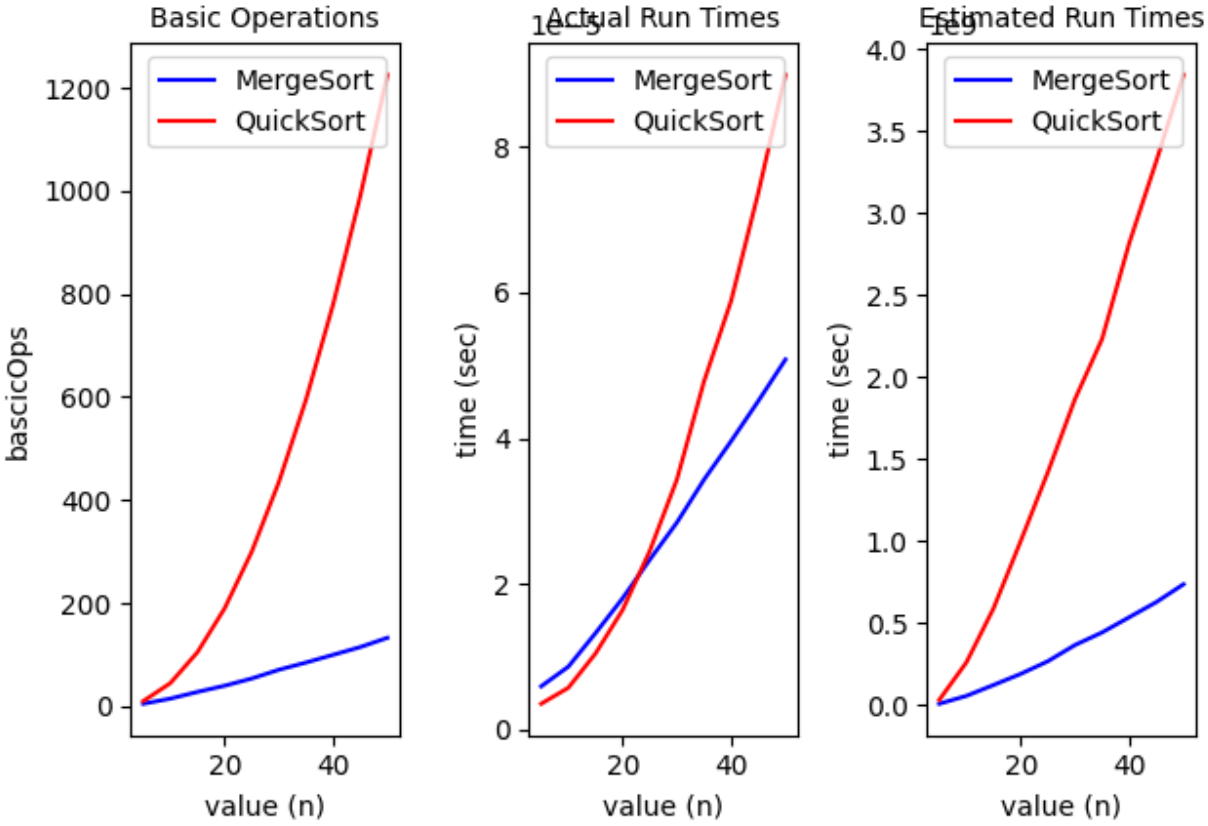

MergeSort vs QuickSort
Basic Operations
MergeSort
QuickSort
bascicOps
800
600
400
200
0
0
100
value (n)
Actual Run Times
MergeSort
QuickSort
time (sec)
0.000175
0.000150
0.000125
0.000100
0.000075
0.000050
0.000025
0.000000
0
100
value (n)
Estimated Run Times
1e9
MergeSort
QuickSort
time (sec)
6
5
4
3
2
1
0
0
100
value (n)

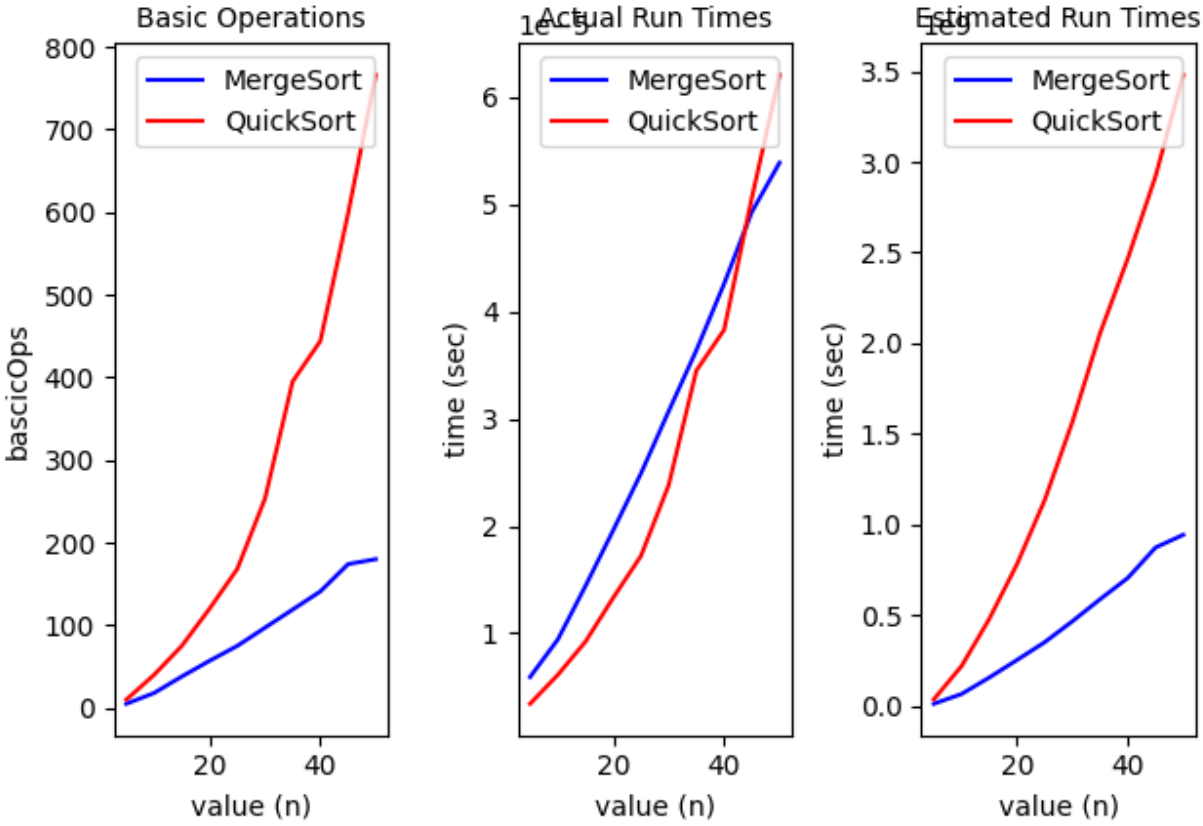

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e9
0
500
1000
1500
2000
0.0000
0.0001
0.0002
0.0003
0.0004
0.0005
2
4
6
8
200

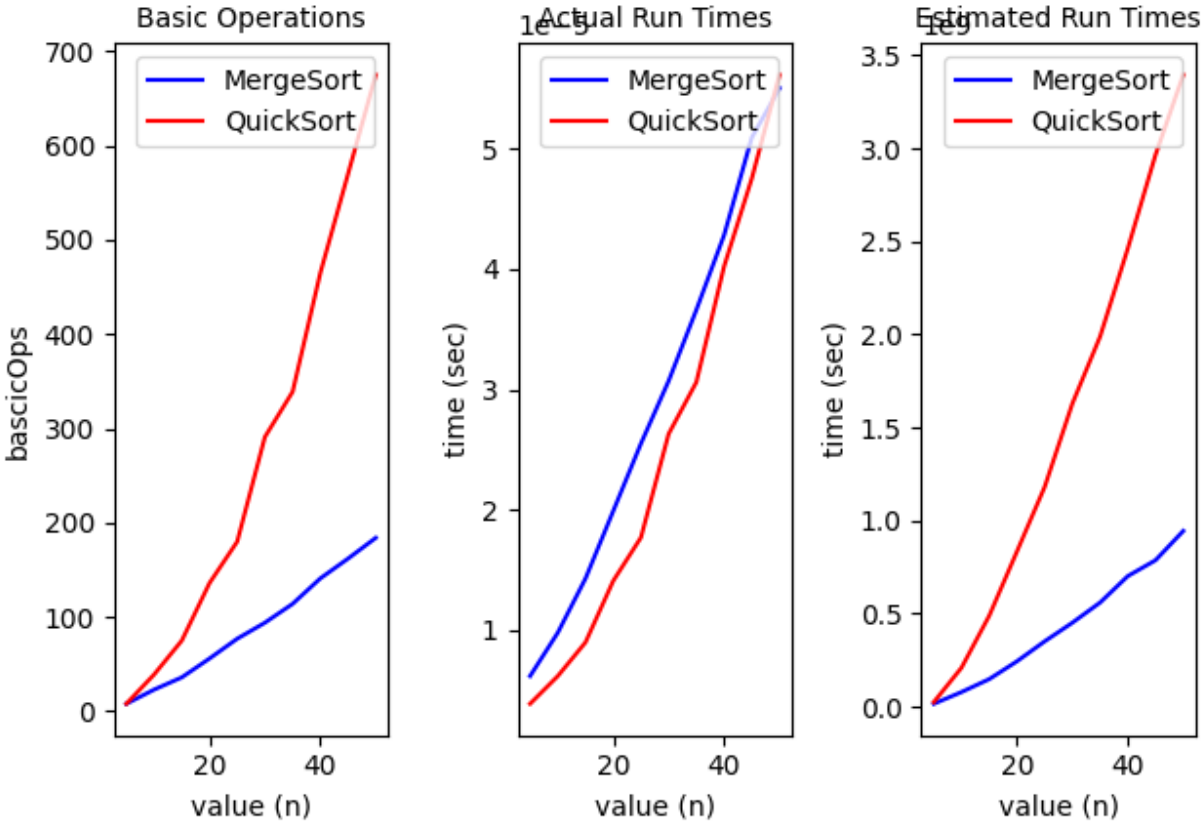

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e10
0
200
500
1000
1500
2000
0.00000
0.00005
0.00010
0.00015
0.00020
0.00025
0.00030
0.00035
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6

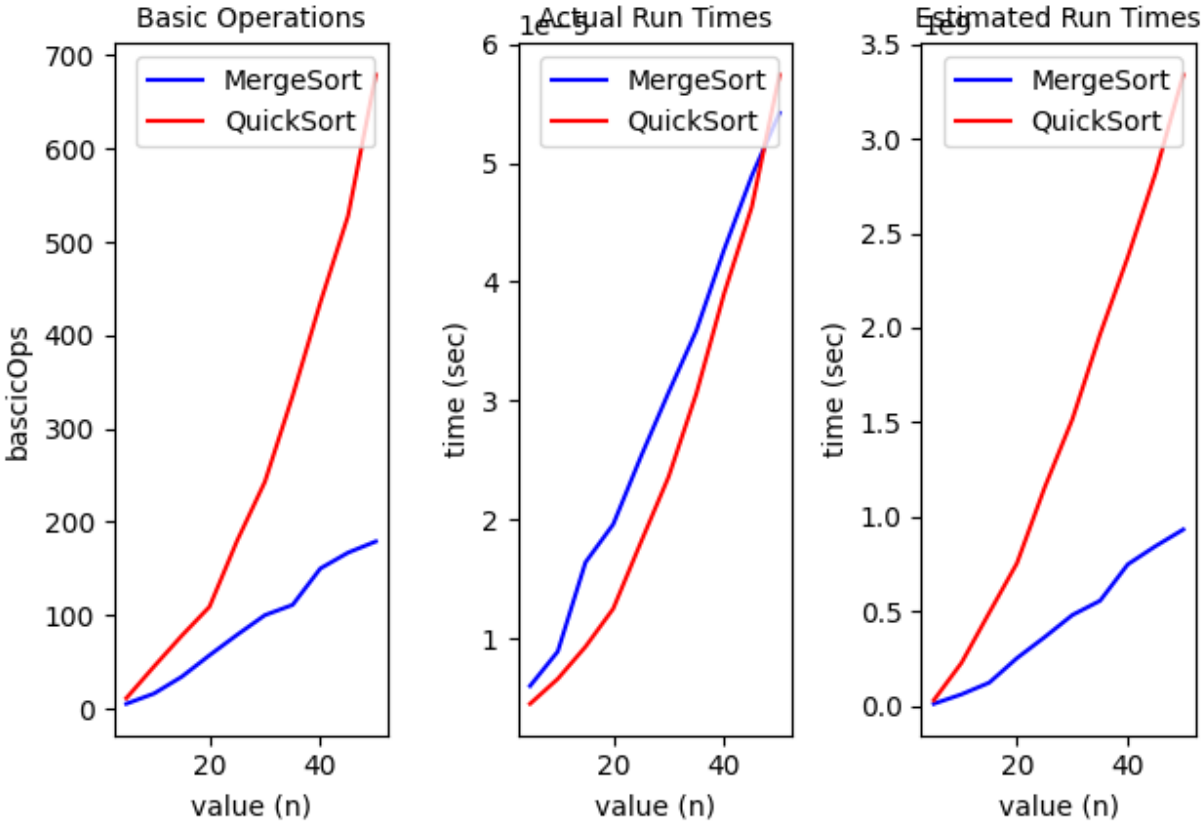

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
0
200
500
1000
1500
2000
0.00000
0.00005
0.00010
0.00015
0.00020
0.00025
0.00030
0.00035
1e10
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6

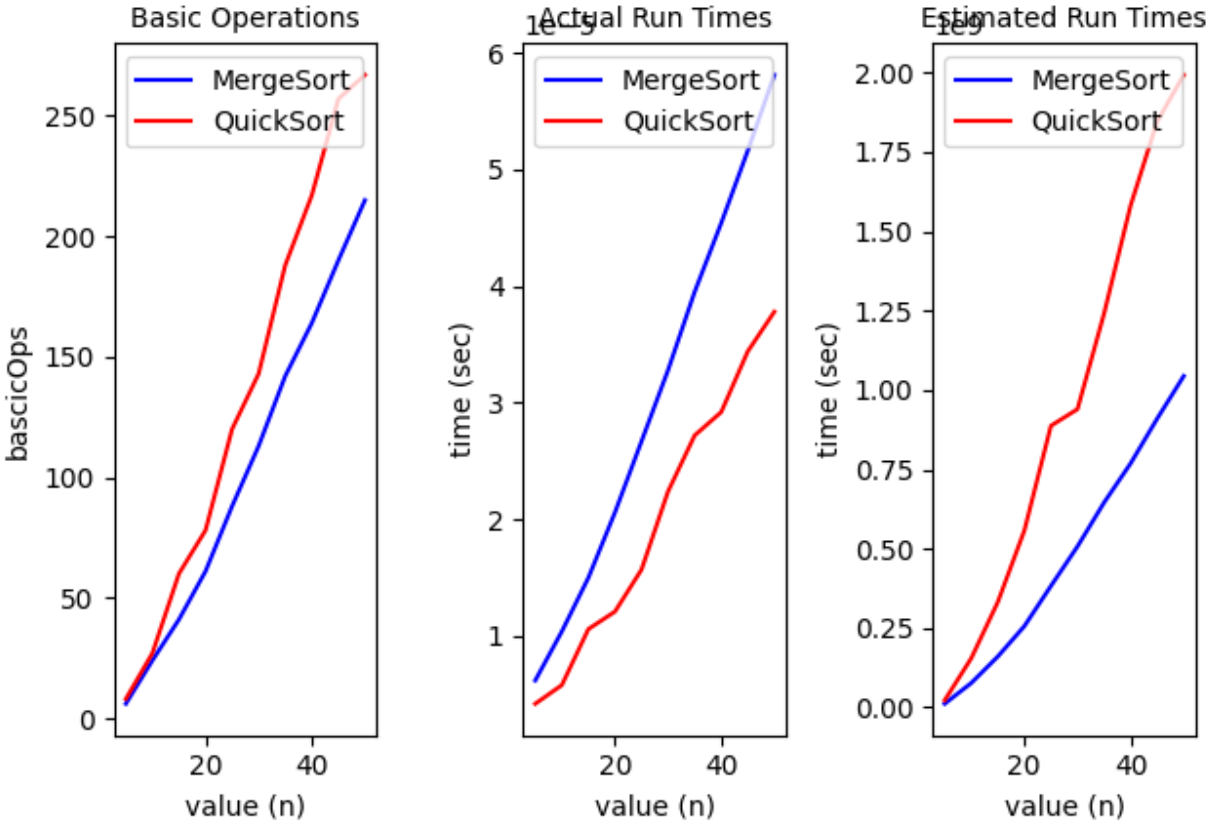

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e10
0
200
500
1000
1500
2000
0.00000
0.00005
0.00010
0.00015
0.00020
0.00025
0.00030
0.00035
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6

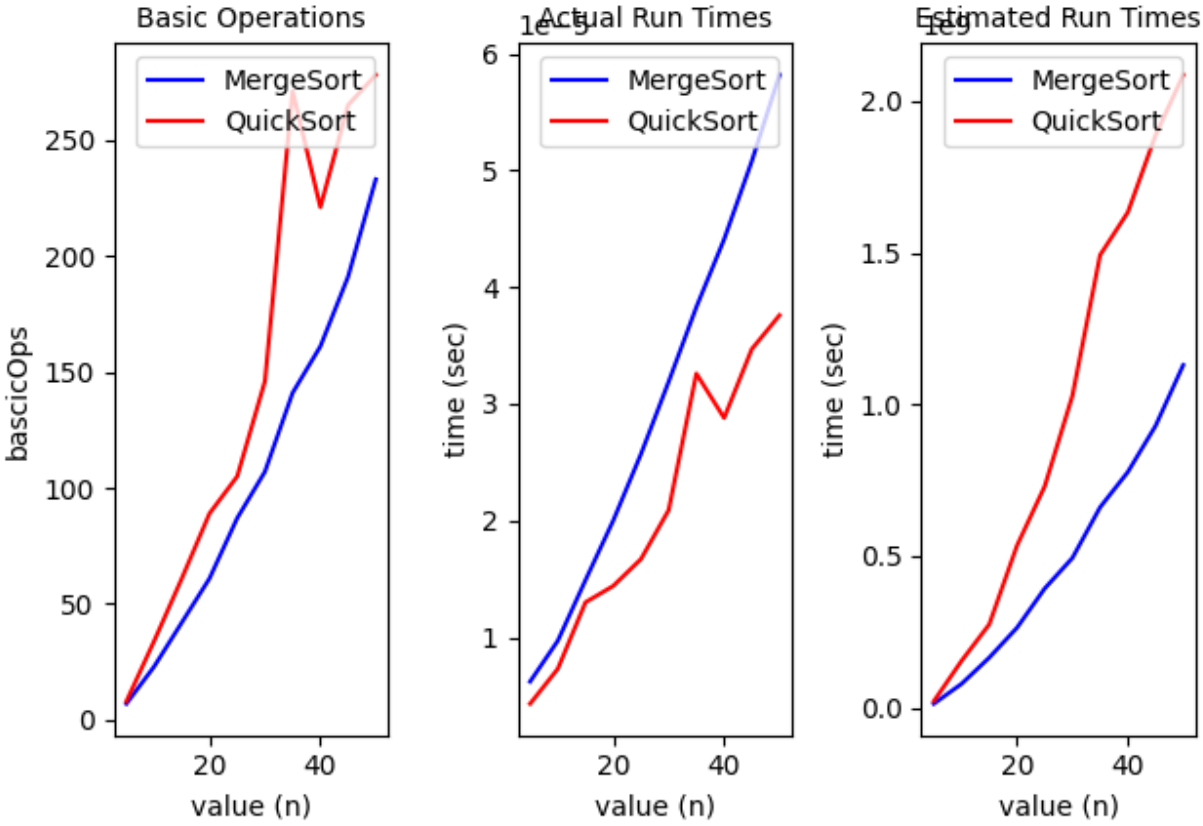

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−5
1e9
0
200
400
600
800
1000
1200
2
4
6
8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
20
40

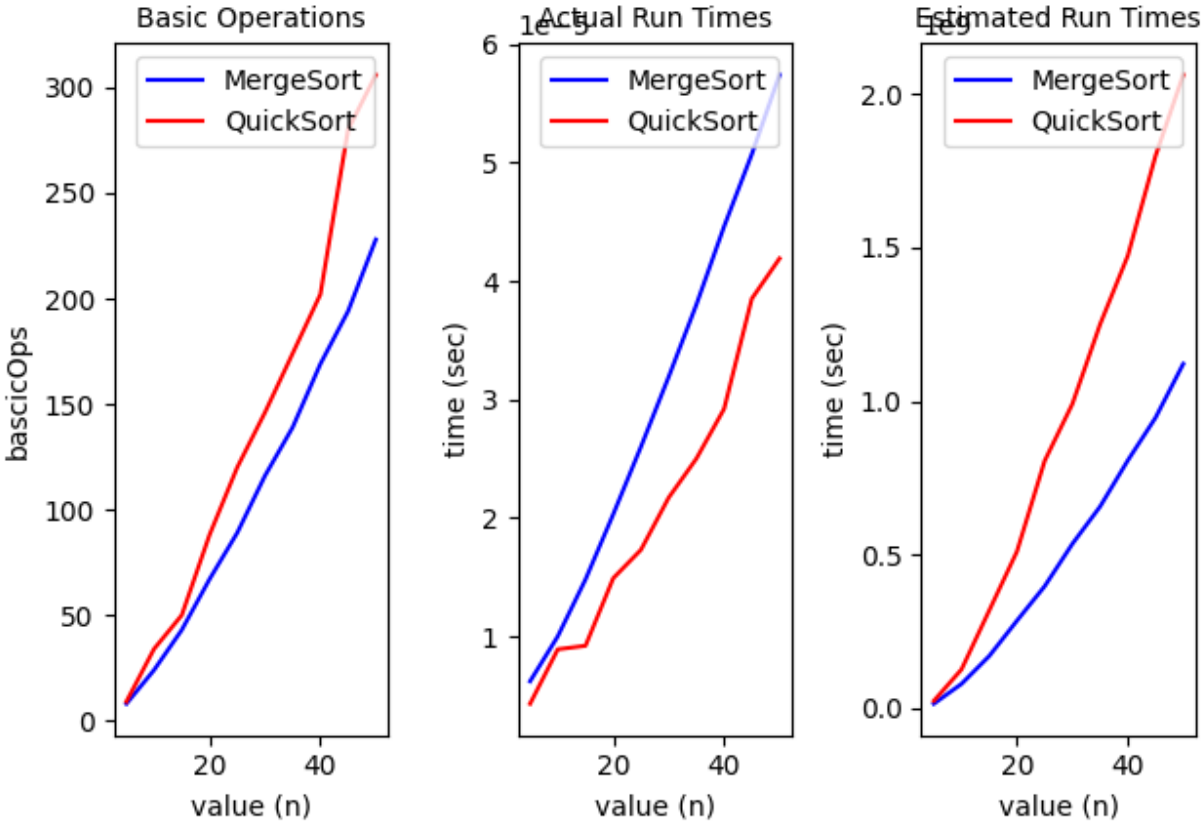

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−5
1e9
0
100
200
300
400
500
600
700
800
1
2
3
4
5
6
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
20
40

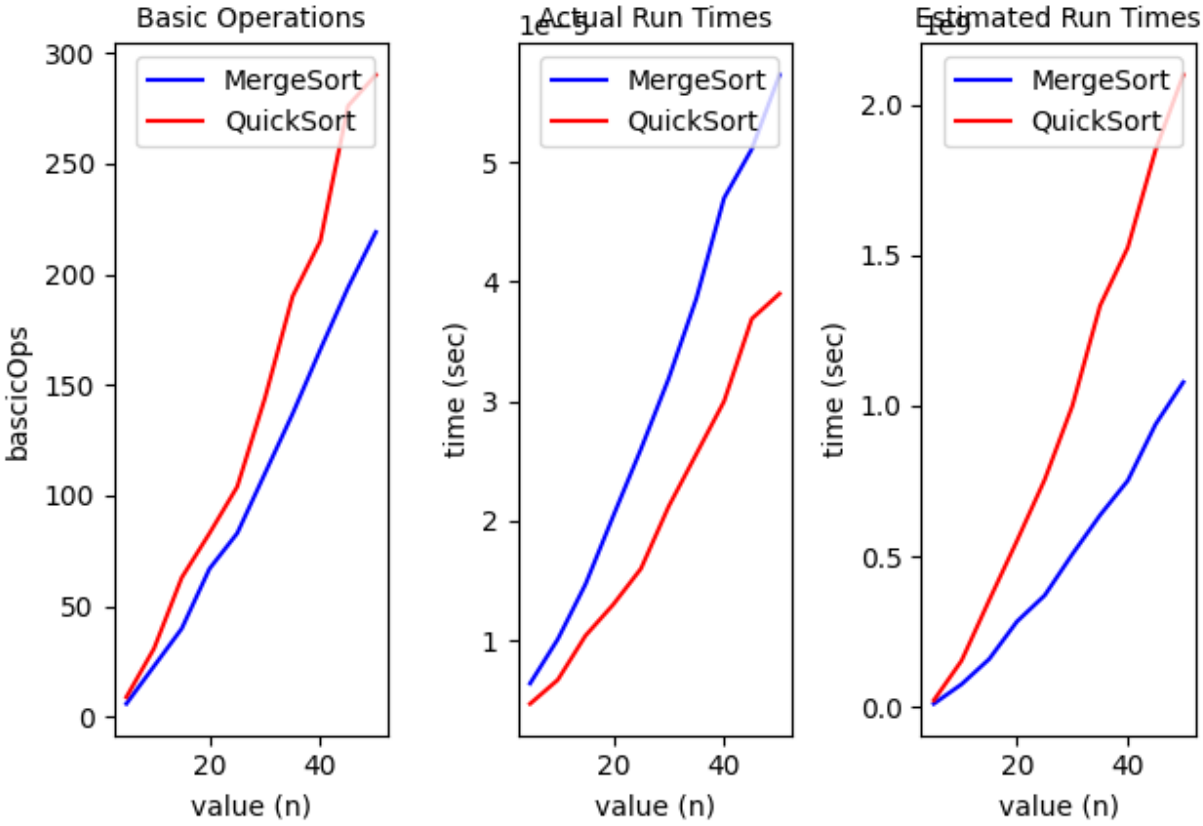

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−5
1e9

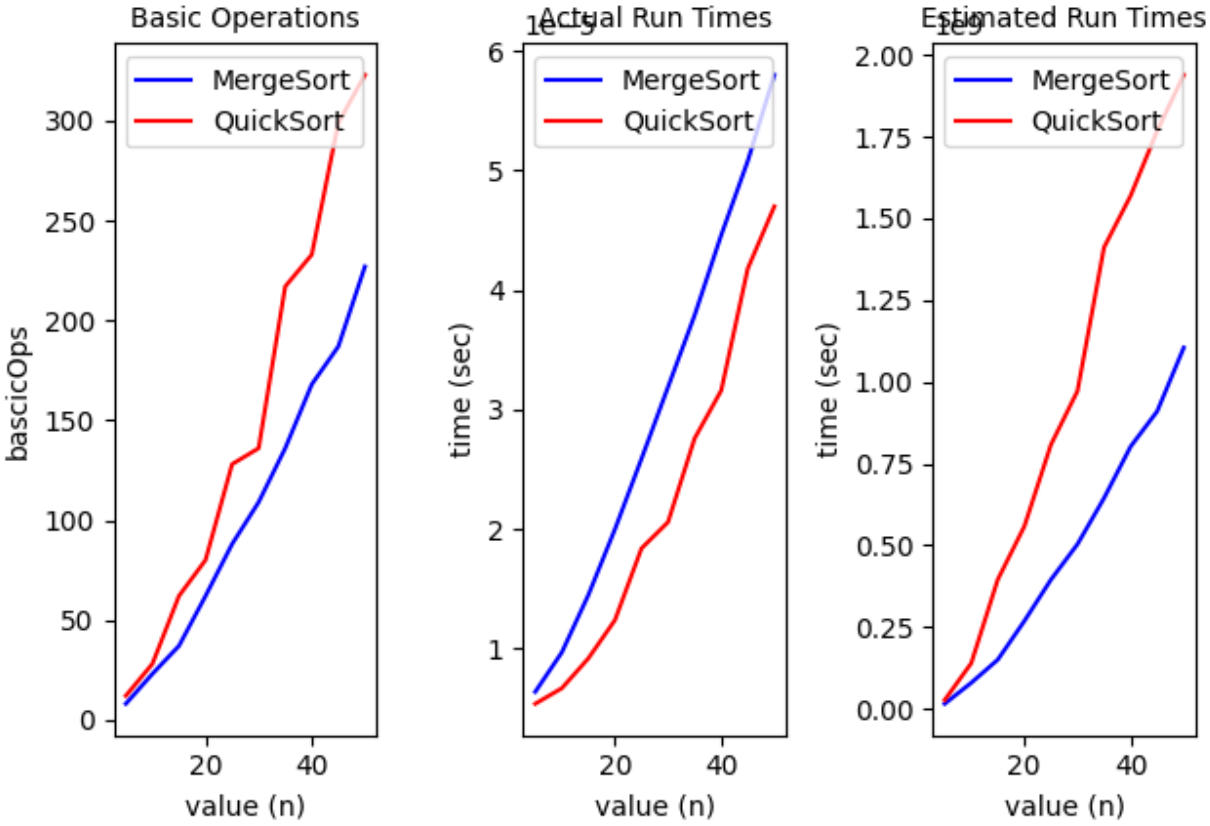

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−5
1e9
0
100
200
300
400
500
600
700
1
2
3
4
5
6
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
20
40

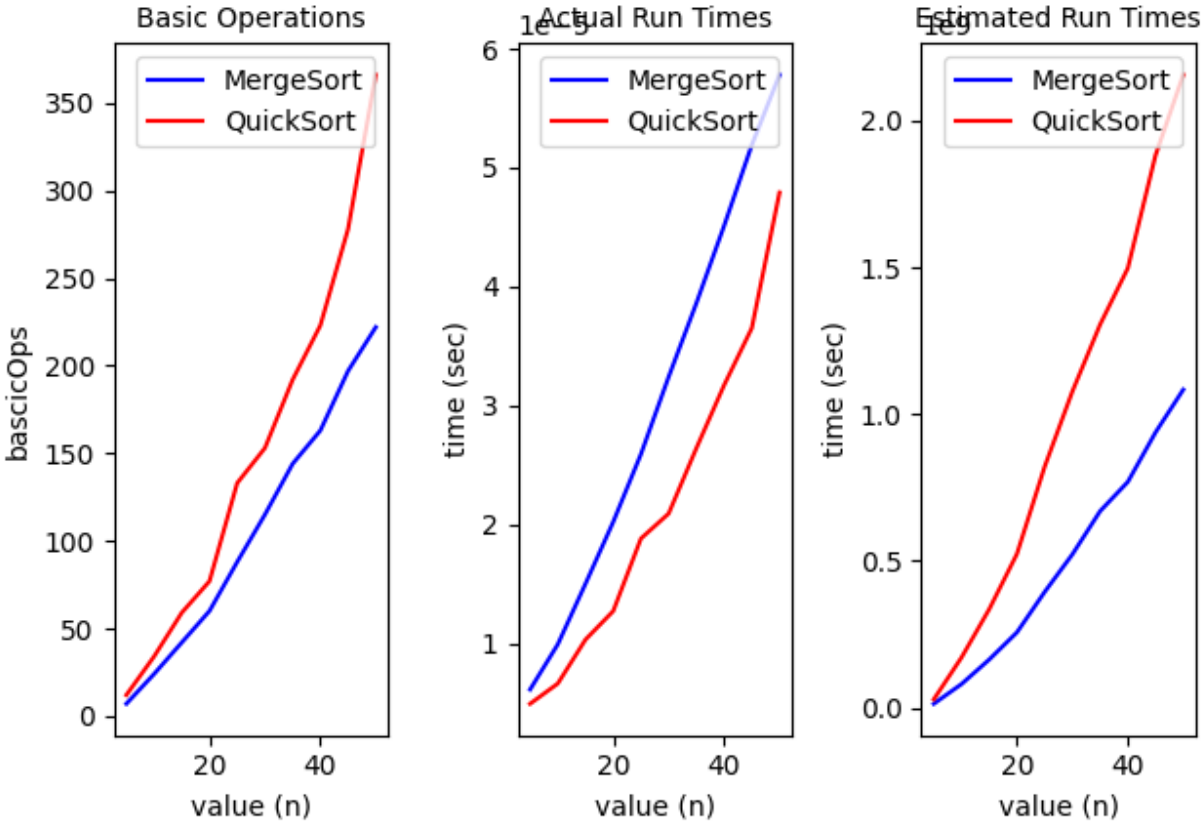

MergeSort vs QuickSort
Basic Operations
bascicOps
value (n)
MergeSort
QuickSort
Actual Run Times
1e−5
time (sec)
value (n)
MergeSort
QuickSort
Estimated Run Times
1e9
time (sec)
value (n)
MergeSort
QuickSort

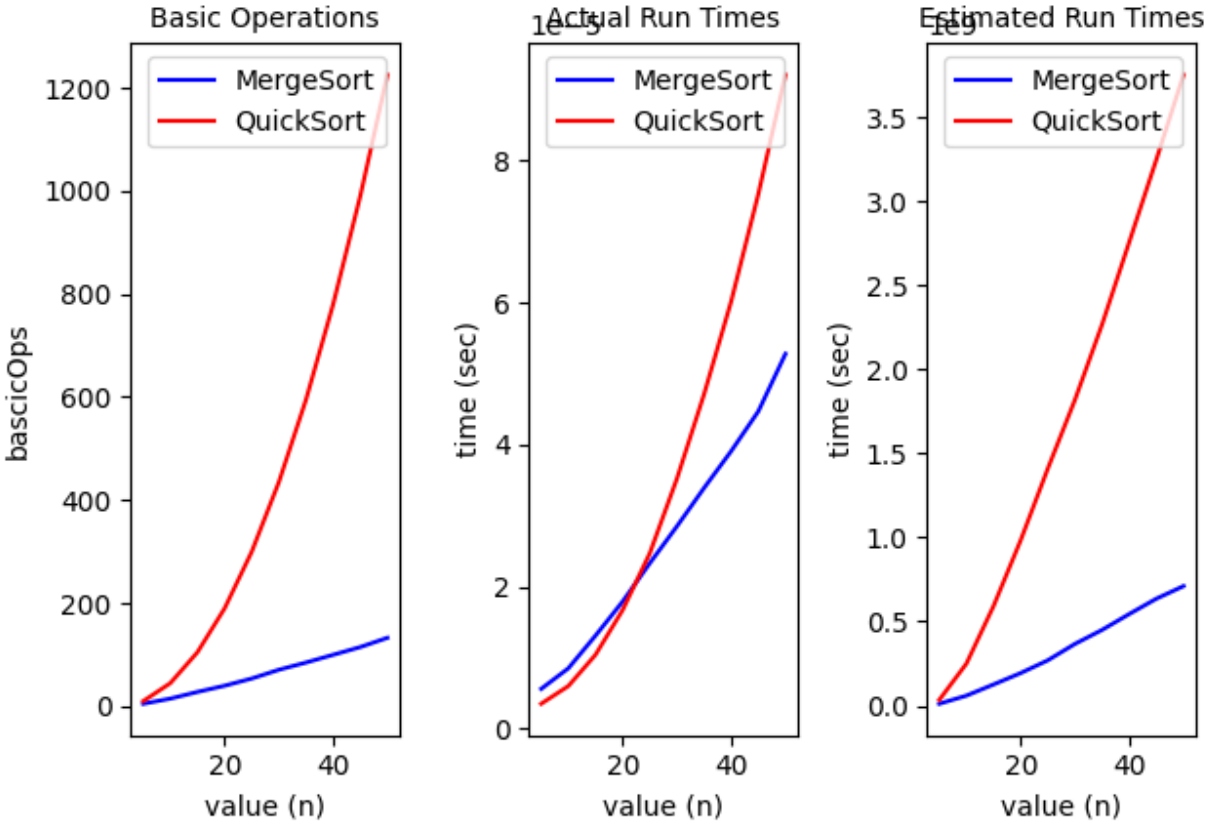

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e-5
1e9
0
50
100
150
200
250
1
2
3
4
5
6
0.0
0.5
1.0
1.5
2.0
20
40

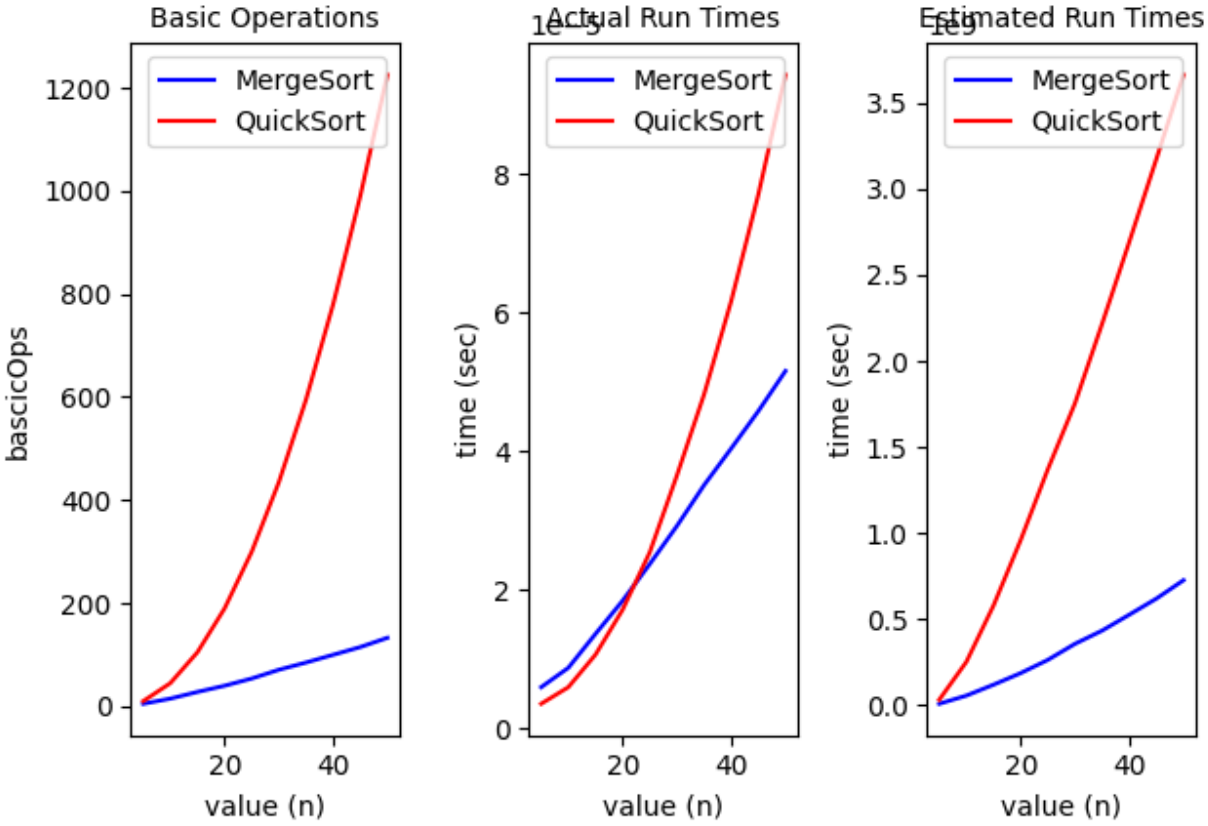

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−5
1e9
0
50
100
150
200
250
300
1
2
3
4
5
6
0.0
0.5
1.0
1.5
2.0
20
40

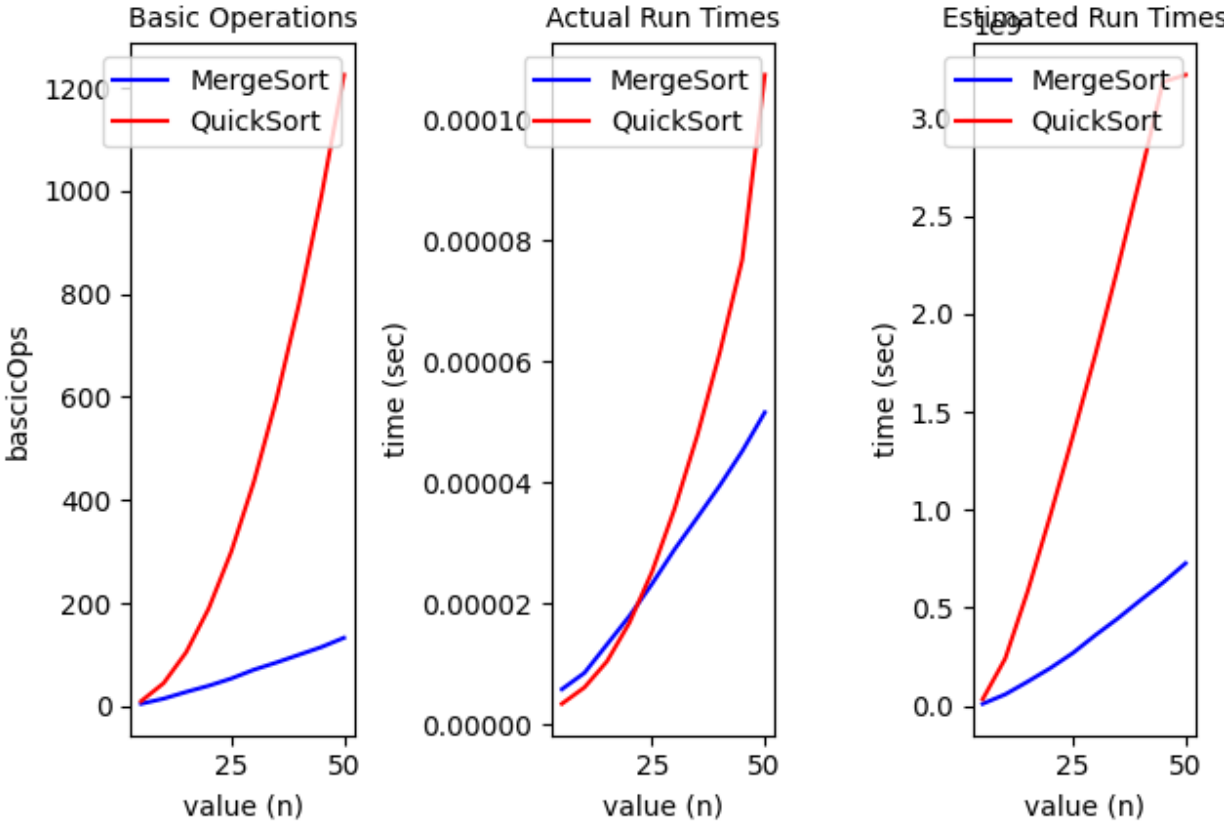

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−5
1e9
0
50
100
150
200
250
300
1
2
3
4
5
0.0
0.5
1.0
1.5
2.0
20
40

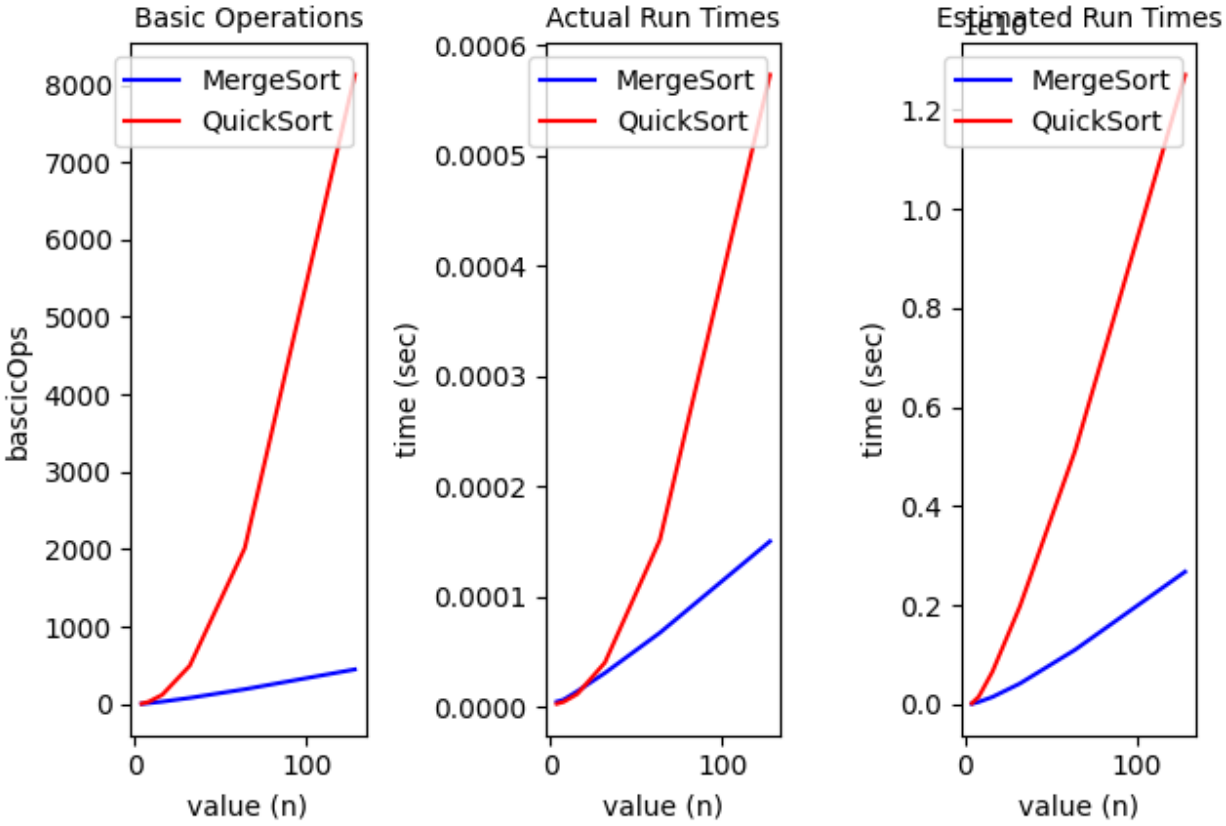

MergeSort vs QuickSort
Basic Operations
bascicOps
value (n)
MergeSort
QuickSort
Actual Run Times
1e−5
time (sec)
value (n)
MergeSort
QuickSort
Estimated Run Times
1e9
time (sec)
value (n)
MergeSort
QuickSort

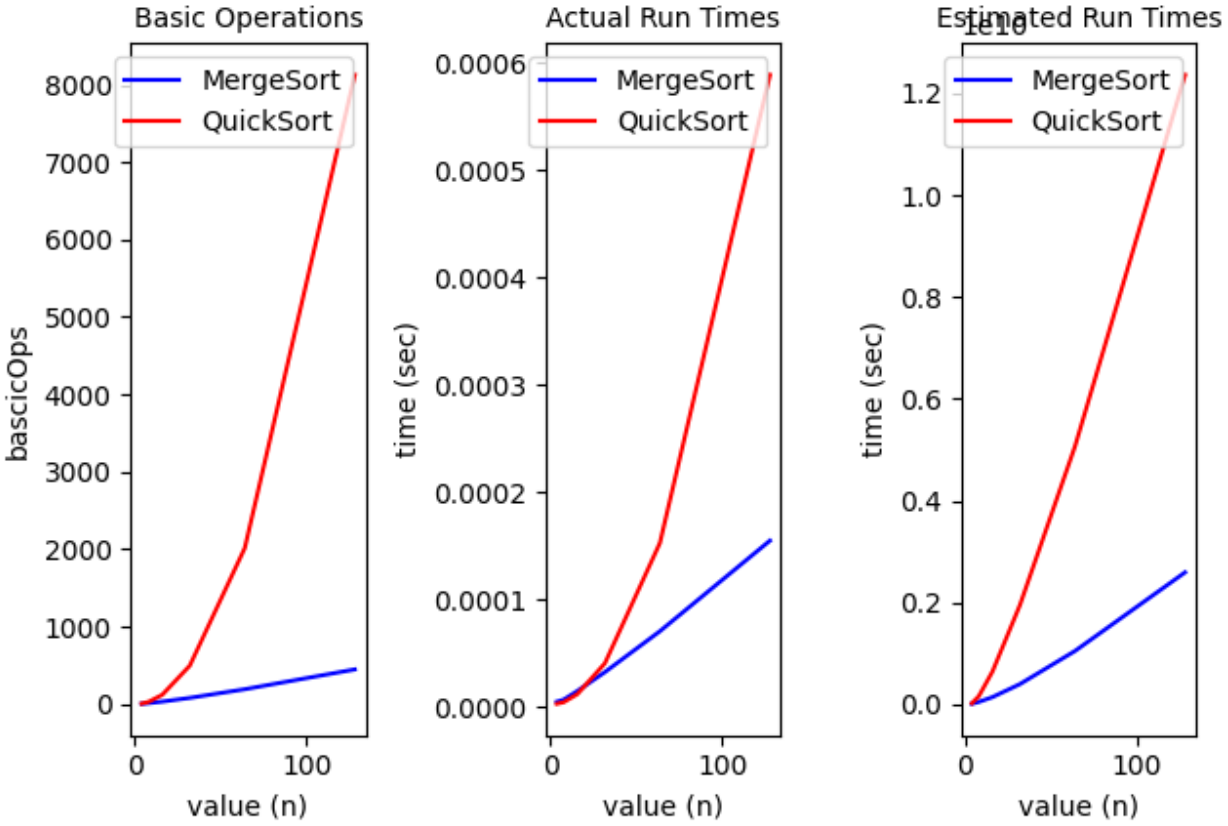

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−5
1e9
0
50
100
150
200
250
300
350
1
2
3
4
5
6
0.0
0.5
1.0
1.5
2.0
20
40

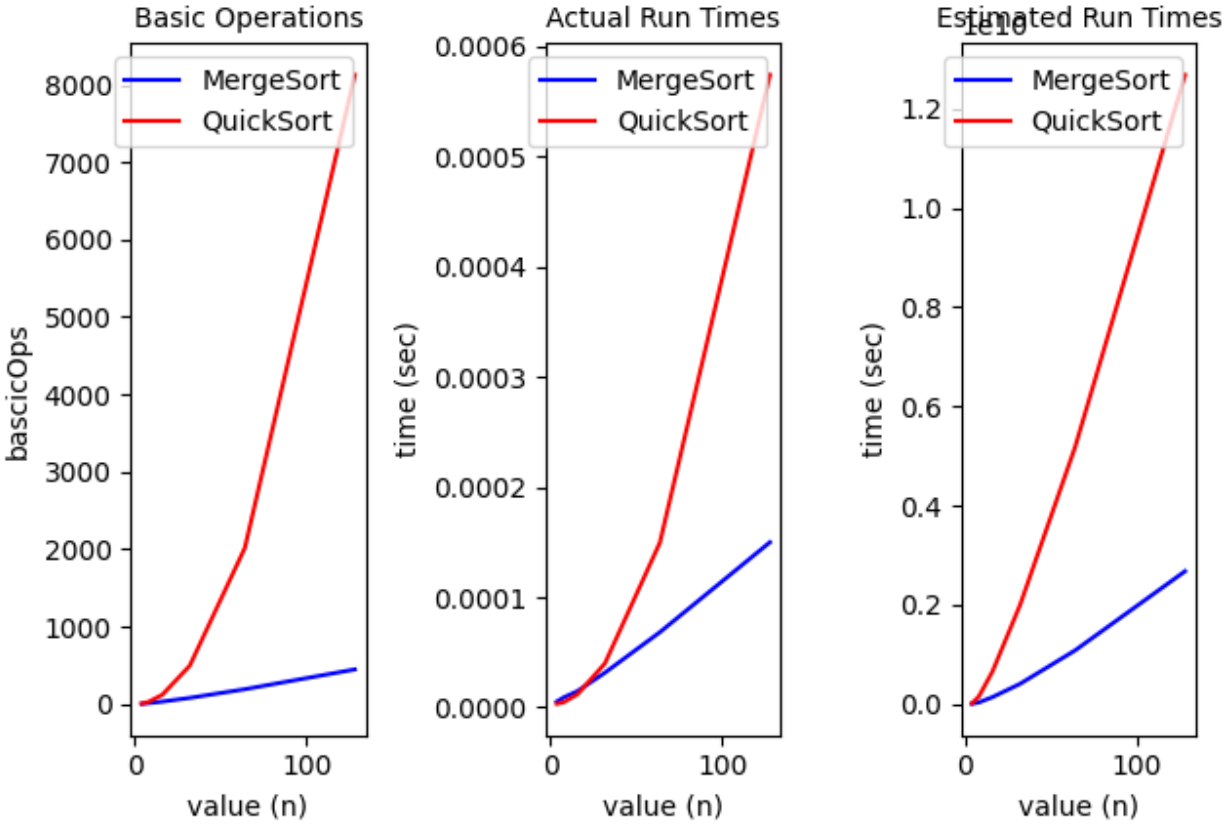

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−5
1e9
0
200
400
600
800
1000
1200
2
4
6
8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
20
40

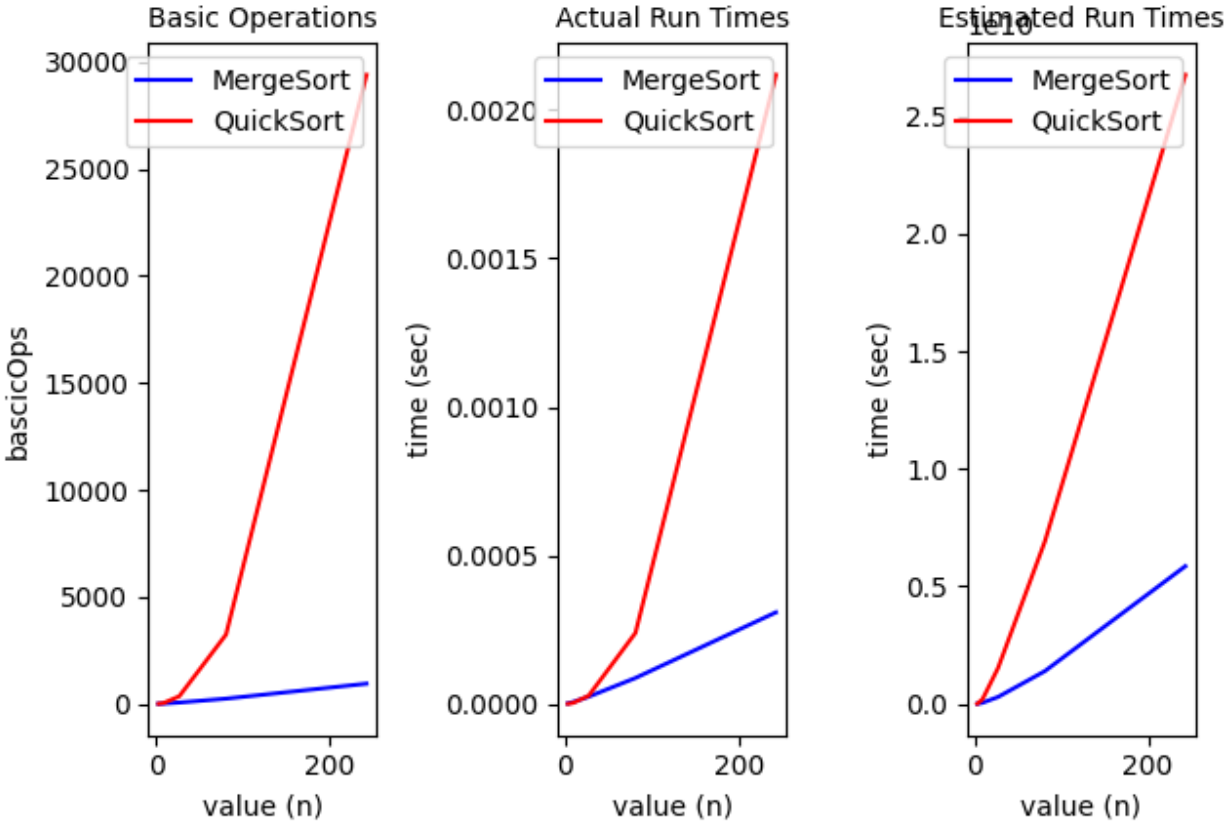

MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e−5
1e9
0
200
400
600
800
1000
1200
2
4
6
8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
20
40


MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
1e9
value (n)
0
200
400
600
800
1000
1200
0.00000
0.00002
0.00004
0.00006
0.00008
0.0
0.5
1.0
1.5
2.0
2.5
3.0
25
50


MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e10


MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
1e10
value (n)
0
100
1000
2000
3000
4000
5000
6000
7000
8000
0.0000
0.0001
0.0002
0.0003
0.0004
0.0005
0.0006
0.0
0.2
0.4
0.6
0.8
1.0
1.2


MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e10


MergeSort vs QuickSort
Basic Operations
Actual Run Times
Estimated Run Times
MergeSort
QuickSort
bascicOps
time (sec)
value (n)
1e10
30000
25000
20000
15000
10000
5000
0
0.0020
0.0015
0.0010
0.0005
0.0000
2.5
2.0
1.5
1.0
0.5
0.0
200