Analyses of various algorithms

Term paper for INF221 2021

Mats Hoem Olsen, Roy Erling Granheim, Sindre Stokke

DataSci
Norwegian university of life science
2021 December 3

Contents

Abstract

There exist many algorithms used for sorting data, some better than others for small or big types of data. This paper will go through many different algorithms and test them to see which is faster depending on size, complexity and sortedness. We will benchmark by using identically randomly generated data.

1 Introduction

When we talk about a sorting algorithm we mean a method or a set of instructions for rearranging an array in a way that can be considered sorted, either by index, value or alphabetical. We will be analysing various sorting algorithms and see how they perform, measured by how long it takes to fully sort an array or by how much of an array they can sort in a given time. The algorithms we will be analysing are: Bubble sort, Insertion sort, Quicksort, Quicksort using median of three as pivot, quicksort insertion sort hybrid, merge sort, merge sort insertion sort hybrid, built in python sort and built in numpy sort. We are most interested in the time complexity these sorting algorithms give out by using big O(n) notation where n represents the number of elements in an array. Another factor that is notable when it comes to sorting algorithms is the amount of memory it can take to run one. In this paper we will be ignoring this factor and only be looking at the time complexity.

2 Theory

Provide a brief description of the algorithms you will be investigating, including pseudocode for the algorithms. Describe in particular the expected runtime of algorithms in terms of problem size. Use a separate subsection for each algorithm.

2.1 Algo 1: Bubble sort

Algorithm 1 Insertion sort algorithm from Cormen et al., *Introduction to Algorithms*, Ch. 2.1.

```
BUBBLESORT($A$)
1  for $i = 1$ to $A.length - 1$
2      for $j = A.length$ downto $i + 1$
3          if $A[j] < A[j-1]$
4              exchange $A[j]$ with $A[j-1]$
```

Pseudocode for the first algorithm is shown in Listing 1. Best case runtime for this algorithm is

$$T(n) = \Theta(n) \ . \tag{1}$$

It is achieved for correctly sorted input data. Worst case runtime for this algorithm is

$$T(n) = O(n^2) \ . \tag{2}$$

It is achieve for a reversed sorted input data. Bubblesort is in place and stable.

2.2 Algo 2: Insertion sort

Algorithm 2 Insertion sort algorithm from Cormen et al., *Introduction to Algorithms*, Ch. 2.1.

```
INSERTION-SORT($A$)
for $j = 2$ to $A.length$
    $key = A[j]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

Insertion sort iterates through the list's numbers from left to right, moving each number as far to the left as needed for there not to be any lower number to the left of it. It is an in-place algorithm that only stores one auxiliary variable called the key. This conatins the number that is being moved to the left and compared to it's next. Pseudocode for the second algorithm is shown in Listing 2. Best case runtime for this algorithm is

$$T(n) = \Theta(n)\ . \tag{3}$$

It is achieved for correctly sorted input data. Worst case runtime for this algorithm is

$$T(n) = O(n^2)\ . \tag{4}$$

It is achieve for a reversed sorted input data. Insertion sort is in place and stable.

2.3 Algo 3: Quicksort

Algorithm 3 Quicksort algorithm from Cormen et al., *Introduction to Algorithms*, Ch. 2.1.

```
QUICKSORT($A, p, r$)
if $p < r$
    $q = $ PARTITION($A, p, r$)
    QUICKSORT($A, p, q - 1$)
    QUICKSORT($A, q + 1, r$)
```

Algorithm 4 Partition from Cormen et al., *Introduction to Algorithms*, Ch. 2.1.

```
PARTITION($A, p, r$)
$x = A[r]$
$i = p - 1$
for $j = p$ to $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
return $i + 1$
```

Quicksort is a "divide and conquer" algorithm. Pseudocode for the third algorithm is shown in Listing 3 and 4. Best case runtime for this algorithm is

$$T(n) = \Theta(n * log(n))\ . \tag{5}$$

It is achieved for where it chooses pivot always in the middle and average case for when the array is random. Worst case runtime for this algorithm is

$$T(n) = O(n^2)\ . \tag{6}$$

It is achieved for when the input data is both reversed sorted and sorted.

2.4 Algo 4: Quicksort Insertion sort hybrid

Algorithm 5 Quicksort insertion sort hybrid from GeeksforGeeks advanced algorithm

```
QUICKINSERTION-SORT($A, p, r$)
while $p < r$
    if $r - p + 1 < 10$
        INSERTION-SORT(A)
        End loop
    else
        $x = $ PARTITION($A, p, r$)
        if $x - p < r - x$
            QUICKINSERTION-SORT($A, p, x - 1$)
            $p = x + 1$
        else
            QUICKINSERTION-SORT($A, x + 1, r$)
            $r = x - 1$
```

Pseudocode for the fourth algorithm is shown in Listing 5. Best case runtime for this algorithm is

$$T(n) = \Theta(n)\ . \qquad (7)$$

It is achieved for correctly sorted input data.

2.5 Algo 5: Mergesort

Algorithm 6 Mergesort algorithm from Cormen et al., *Introduction to Algorithms*, Ch. 2.1.

```
MERGE-SORT($A, p, r$)
1  if $p < r$
2      $q = \lfloor (p+r)/2 \rfloor$
3      MERGE-SORT($A, p, q$)
4      MERGE-SORT($A, q+1, r$)
5      MERGE($A, p, q, r$)
```

Algorithm 7 Merge from Cormen et al., *Introduction to Algorithms*, Ch. 2.1.

```
MERGE($A, p, q, r$)
 1  $n_1 = q - p + 1$
 2  $n_2 = r - q$
 3  let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
 4  for $i = 1$ to $n_1$
 5      $L[i] = A[p + i - 1]$
 6  for $j = 1$ to $n_2$
 7      $R[j] = A[q + j]$
 8  $L[n_1 + 1] = \infty$
 9  $R[n_2 + 1] = \infty$
10  $i = 1$
11  $j = 1$
12  for $k = p$ to $r$
13      if $L[i] \leq R[j]$
14          $A[k] = L[i]$
15          $i = i + 1$
16      else $A[k] = R[j]$
17          $j = j + 1$
```

Pseudocode for the fifth algorithm is shown in Listing **??**. The runtime for this algorithm in all cases is

$$T(n) = \Theta(n * log(n))\ . \qquad (8)$$

2.6 Algo 6: Mergesort Insertion sort hybrid

Algorithm 8 Mergesort Insertion sort hybrid from GeeksforGeeks timsort

```
MERGEINSERTION-SORT($A$)
 1  $minrun = 32$
 2  for $s = 0$ to $A.length$ by $minrun$
 3      $e =$ lesser of $s + minrun - 1$ and $A.length - 1$
 4      INSERTION-SORT($A, s, e$)
 5  $size = minrun$
 6  while $h < A.length$
 7      for $p = 0$ to $A.length$ by $2 * size$
 8          $q =$ lesser of $A.length - 1$ and $p + size - 1$
 9          $r =$ lesser of $p + 2 * size - 1$ and $A.length - 1$
10          if $q < r$
11              MERGE($A, p, q, r$)
12      $size = 2 * size$
```

Algorithm 9 Insertion sort from GeeksforGeeks timsort

```
INSERTION-SORT($A, p, r$)
1  for $i = p + 1$ to $r + 1$
2      $j = i$
3      while $j > p$ and $A[j] < A[j - 1]$
4          exchange $A[j]$ with $A[j - 1]$
```

Pseudocode for the sixth algorithm is shown in Listing **??**. Best case runtime for this algorithm is

$$T(n) = \Theta(n)\ . \qquad (9)$$

It is achieved for correctly sorted input data.

2.7 Algo 7: Python sort

Built in sorting algorithm standard in the python programming language

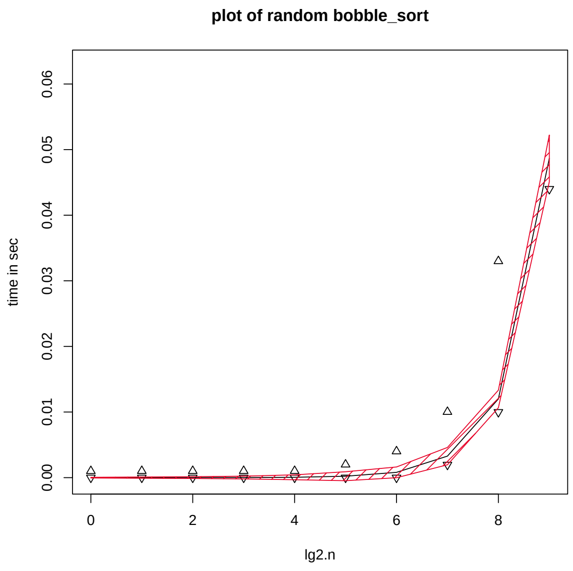
2.8 Algo 8: Numpy sort

Built in sorting algortihm from Numpy a python package.

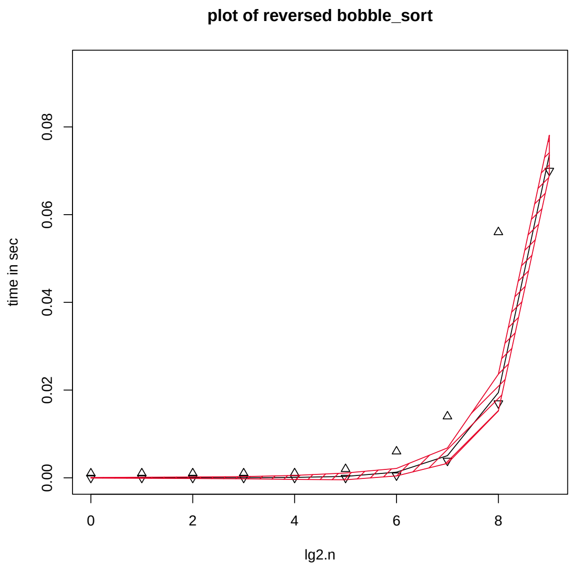
3 Method used

In this section we will show our workflow and automation of tests.(**brady__haran__death__2020**)

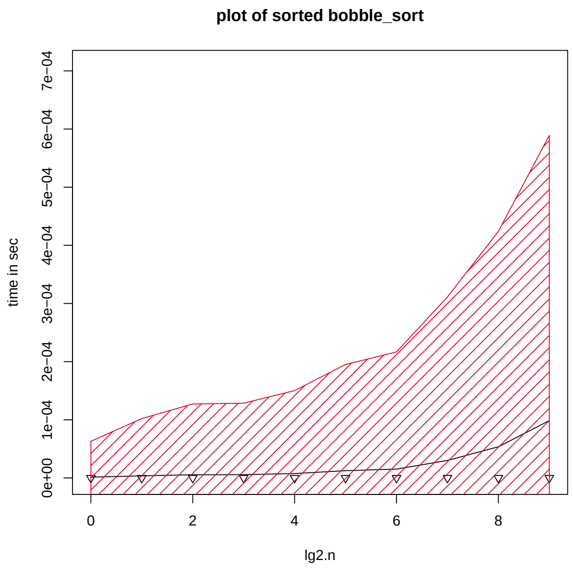
4 Results



(a) bubble sort random input data



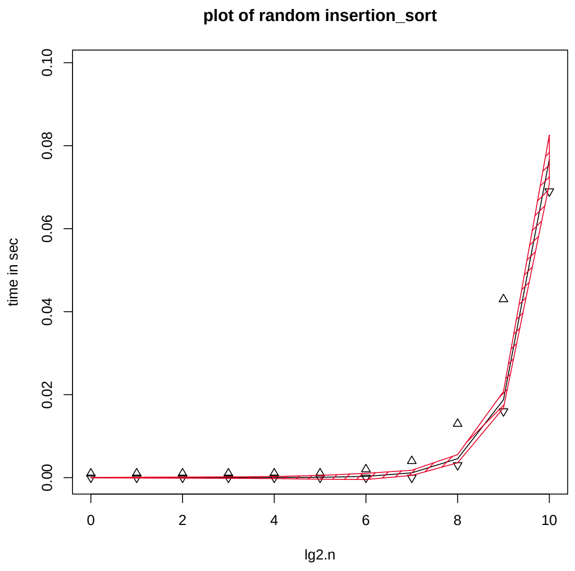
(b) bubble sort reversed sorted data



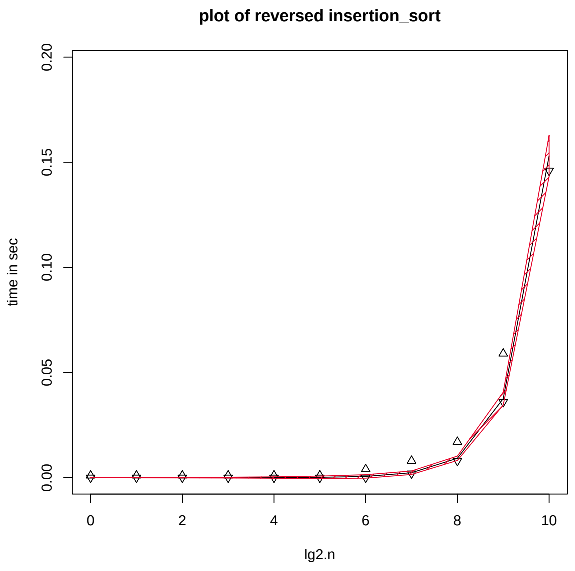
(c) bubble sort sorted data

Bubble sort results: From the plot we can see that the average running time on the unsorted data and the reverse order data both follow the same curve. Although the running time on the reverse sorted data is a little bit slower. The average running time on the sorted data is way less and also follows a less steep curve.

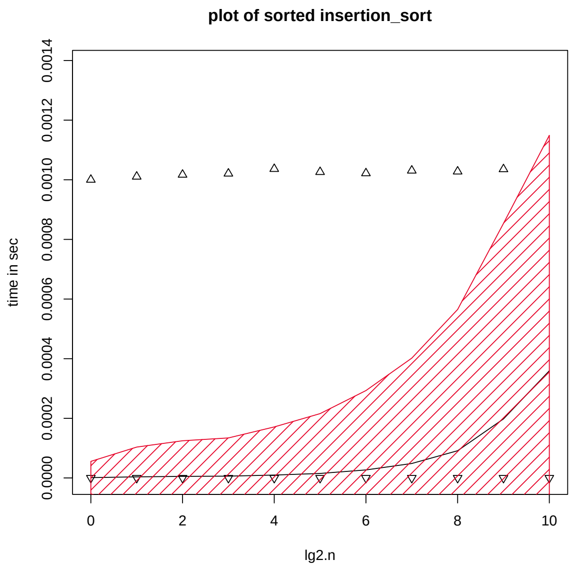
Insertion sort results: As we can se from the plots, the sorted order takes much way less time on average than the random order and reverse order.



(a) Insertion sort random input data

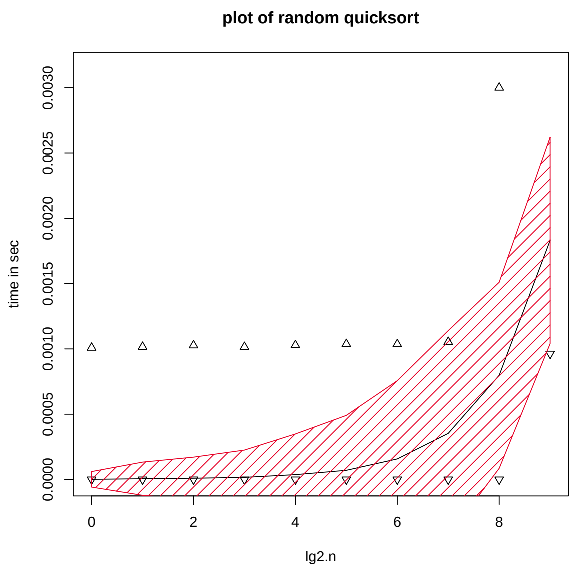


(b) Insertion sort reversed sorted data

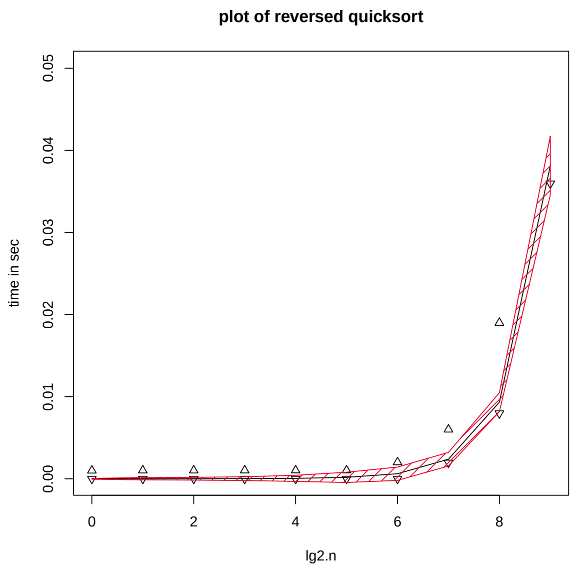


(c) Insertion sort sorted data

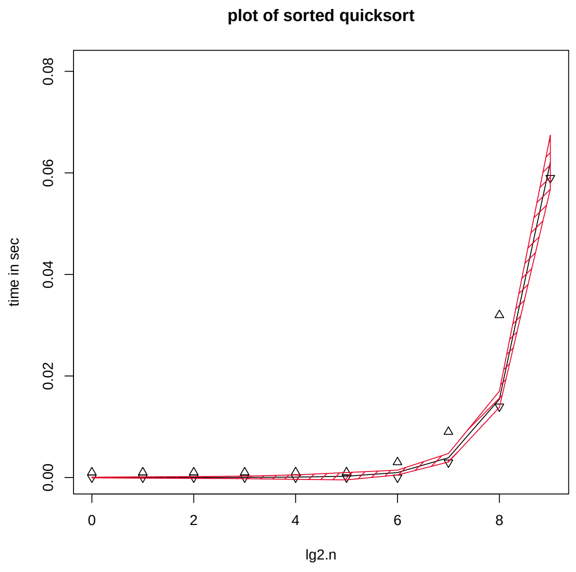
Quicksort results: From the plot we can see that the average running time on sorted data and the reverse order data both follow the same curve. Although the running time on the sorted data is a little bit slower than the reverse sorted. The average running time on the unsorted data is way less and also follows a less steep curve.



(a) Quicksort random input data

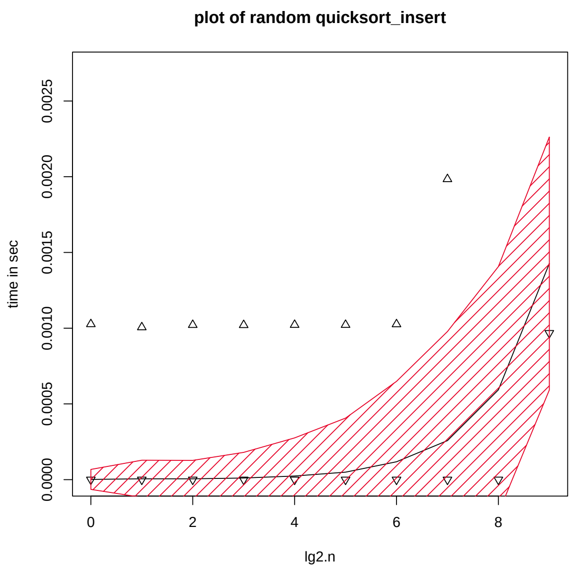


(b) Quicksort reversed sorted data

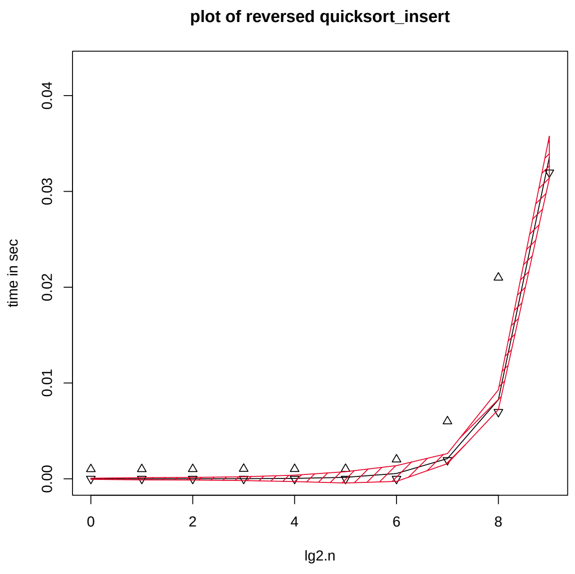


(c) Quicksort sorted data

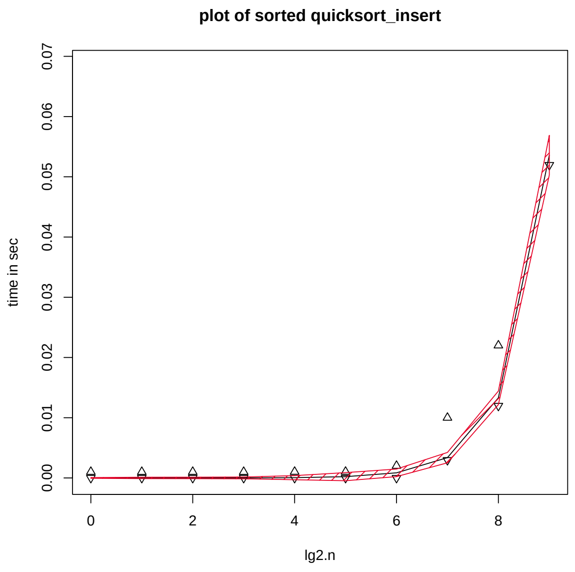
Quicksort Insertion sort hybrid results: The Quicksort insertion sort hybrid outperformes both quicksort and insertionsort on the unsorted data and the reverse sorted data. In fact, on the unsorted data, it is the best performing algorithm of all tested. On the sorted data it is beaten by insertionsort though.



(a) Quicksort Insertion sort random input data

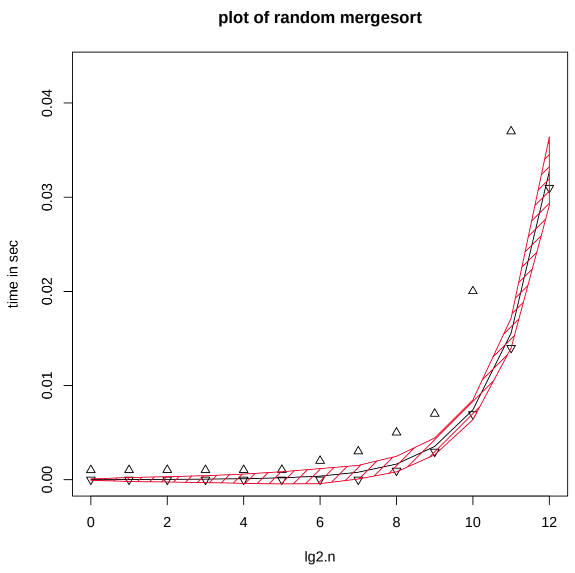


(b) Quicksort Insertion sort reversed sorted data

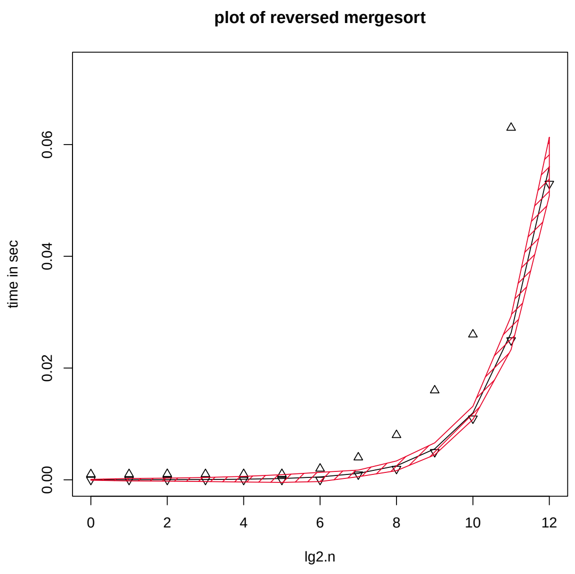


(c) Quicksort Insertion sort sorted data

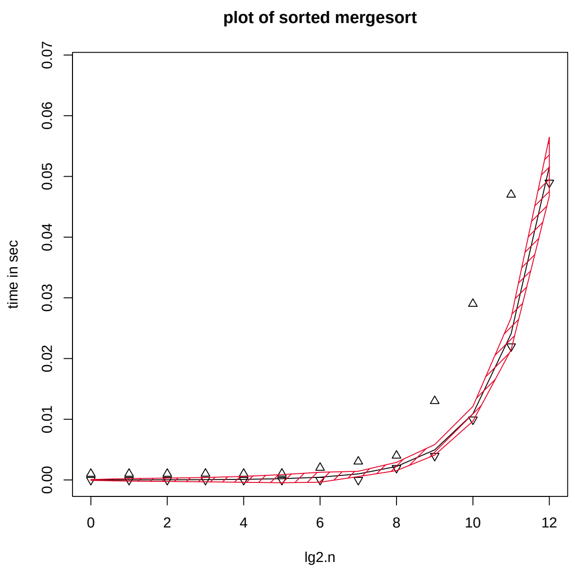
Mergesort results: Both the unsorted data, the sorted and the reverse sorted follows the same curve for the average running time. The unsorted data hade the shortest running time, followed by the reverse ordered and the ordered data.



(a) mergesort random input data

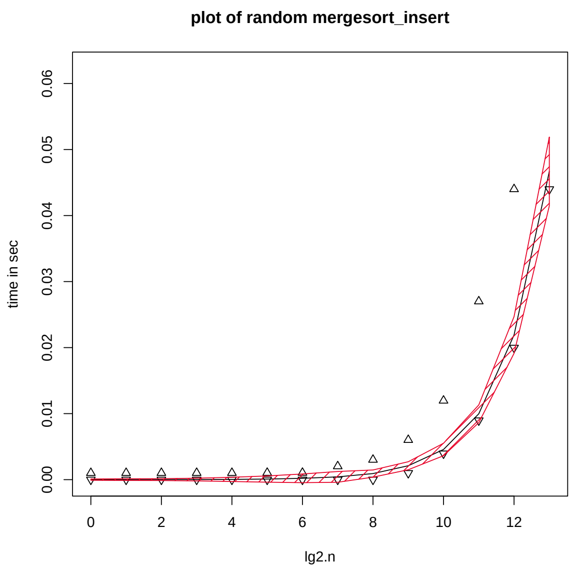


(b) mergesort reversed sorted data

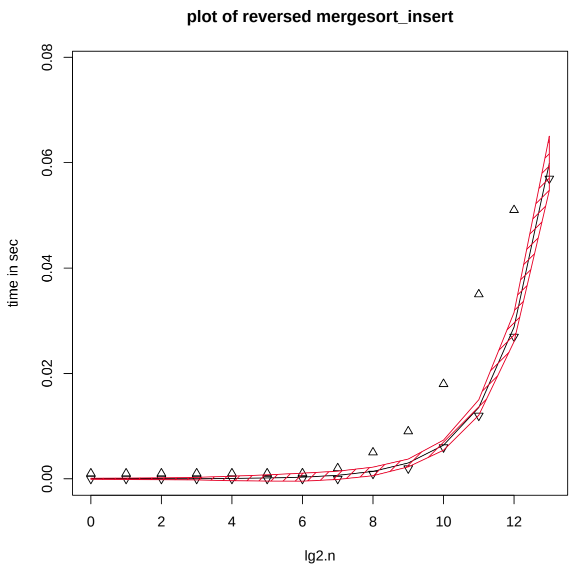


(c) mergesort sorted data

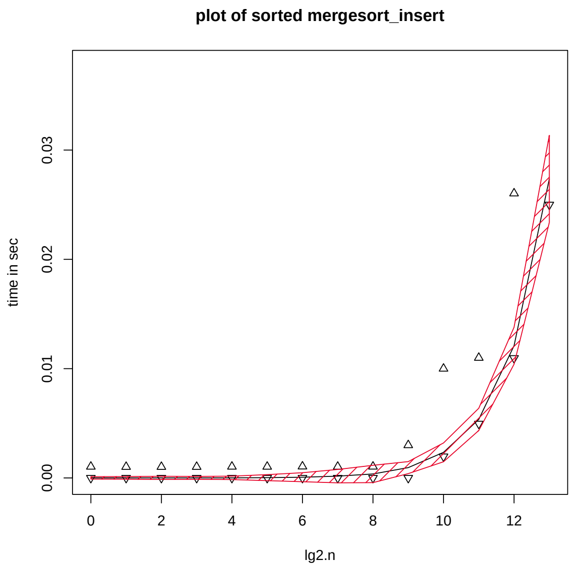
Mergesort Insertion sort hybrid results: The mergesort insertionsort hybrid outperformed mergesort slightly on the unsorted data and significantly on the reverse sorted data. It took way longer time than insertionsort though.



(a) Mergesort Insertion sort random input data

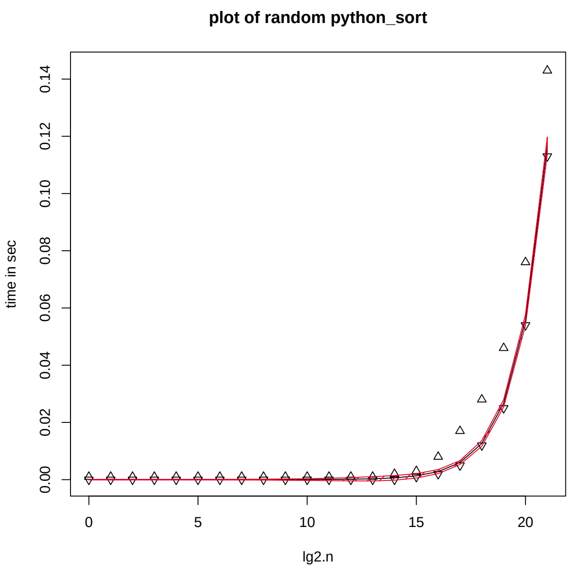


(b) Mergesort Insertion sort reversed sorted data

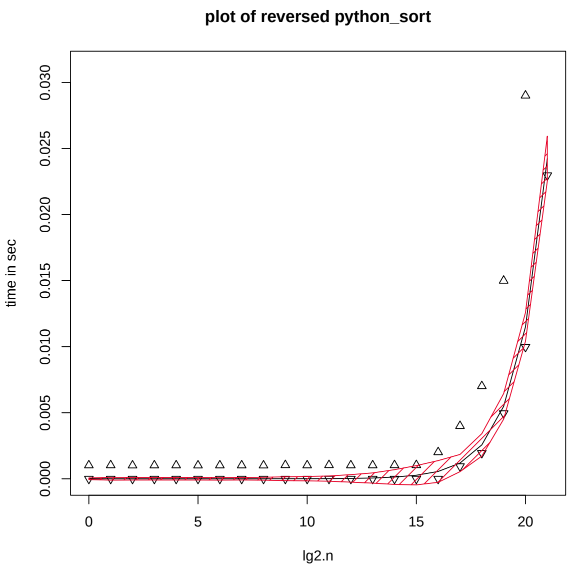


(c) Mergesort Insertion sort sorted data

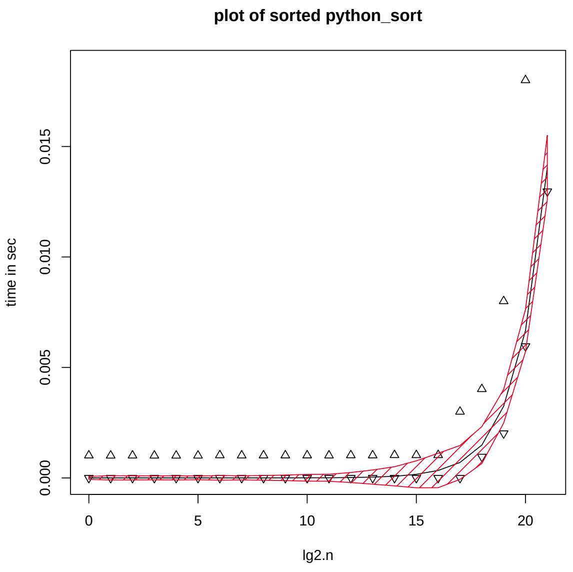
Python sort results: Pythonsort is the best performing algorithm on the reverse sorted data, of all tested.



(a) python sort random input data

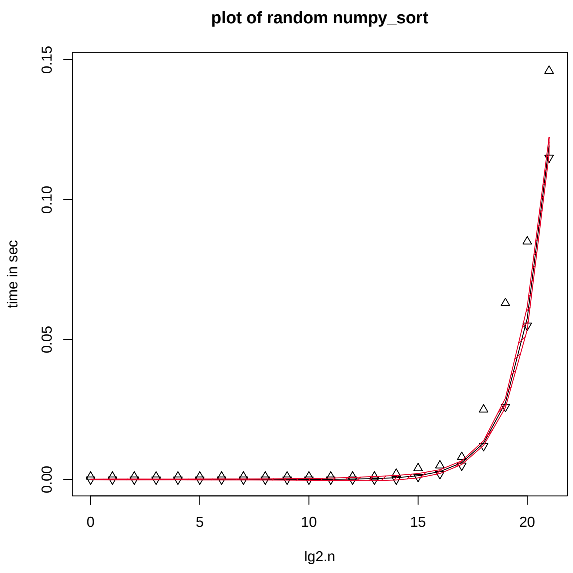


(b) python sort reversed sorted data

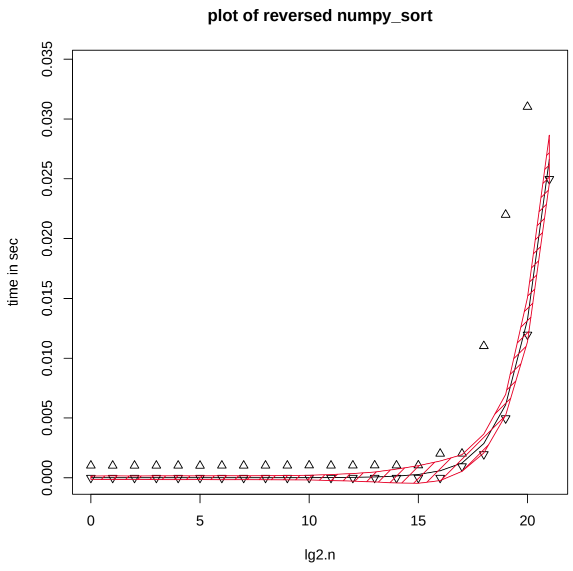


(c) python sort sorted data

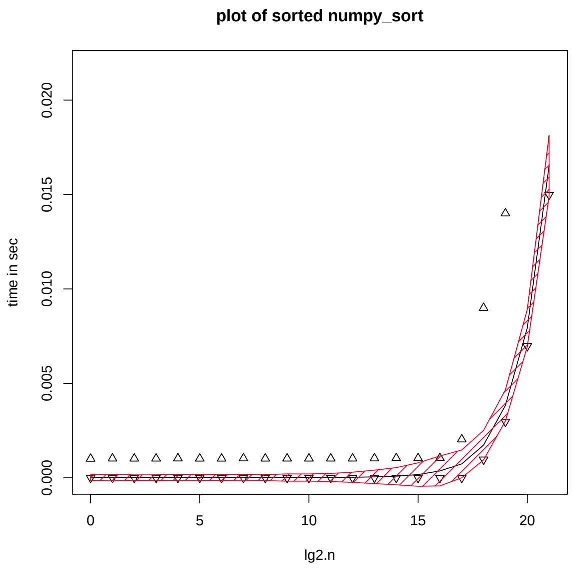
Numpy sort results:



(a) numpy sort random input data



(b) numpy sort reversed sorted data



(c) numpy sort sorted data

5 Discussion

Based on the known time complexities, what we saw matched our expectations.

6 Acknowledgements

We acknowledge that we may, or may not be made out of bread.

References

Cormen, Thomas H. et al. *Introduction to Algorithms*. 3rd. Computer science. USA: McGraw-Hill, 2009. 1292 pp. ISBN: 978-0-262-03384-8. URL: `https://books.google.no/books?id=aefUBQAAQBAJ`.