Analysis of Various Sorting Algorithms

Deebakkarthi C R (CB.EN.U4CSE20613)

Pravin Sabari Bala (CB.EN.U4CSE20648)

October 7, 2022

Contents

1 Comparisons of various sorting algorithms

1.1 Testing Data Generation

The input data for this was using the `random.random()` function in python. The length of the input ranged from 100 to 10000.

1.2 Taking Measurements

Swaps, Comparisons and basic operations were calculated by updating a global variable. *Time taken* was measured using `perf_counter_ns()` and *memory* was measured using `tracemalloc()`

1.3 Plot Generation

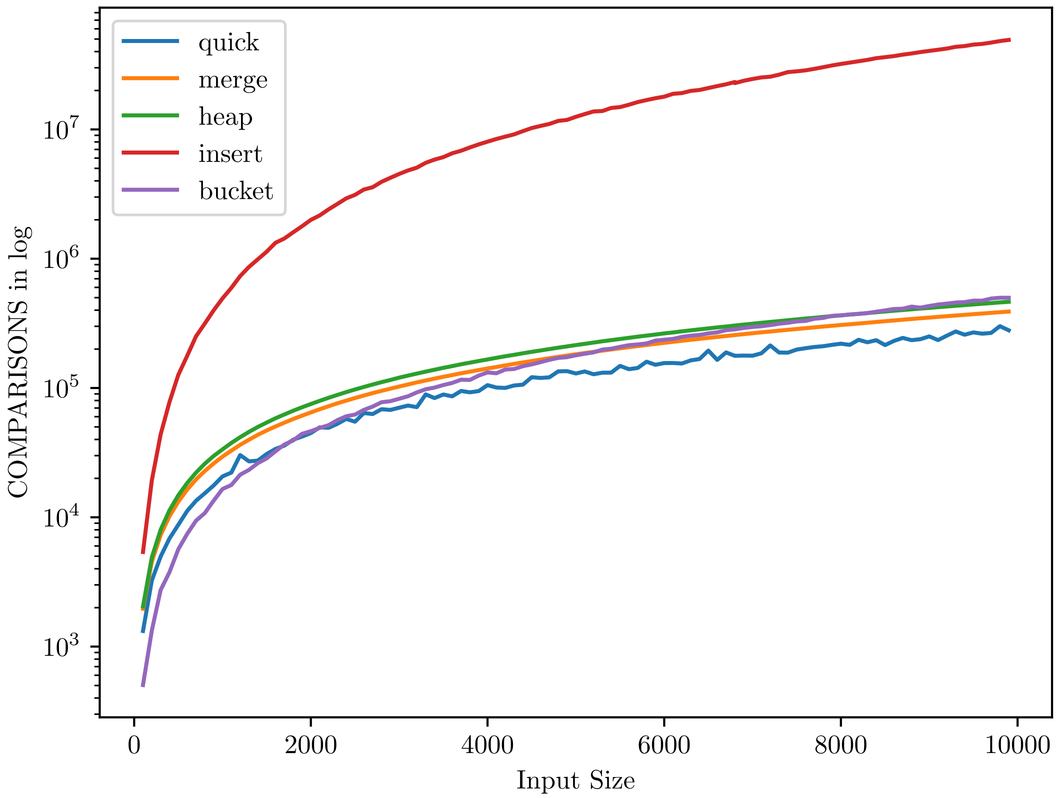
The measurements were store in a *csv* file which was then used by `matplotlib` to produce the plots. The y-scale of certain plots which were too large are presented in a *log* scale.

1.4 Theoretical time complexities

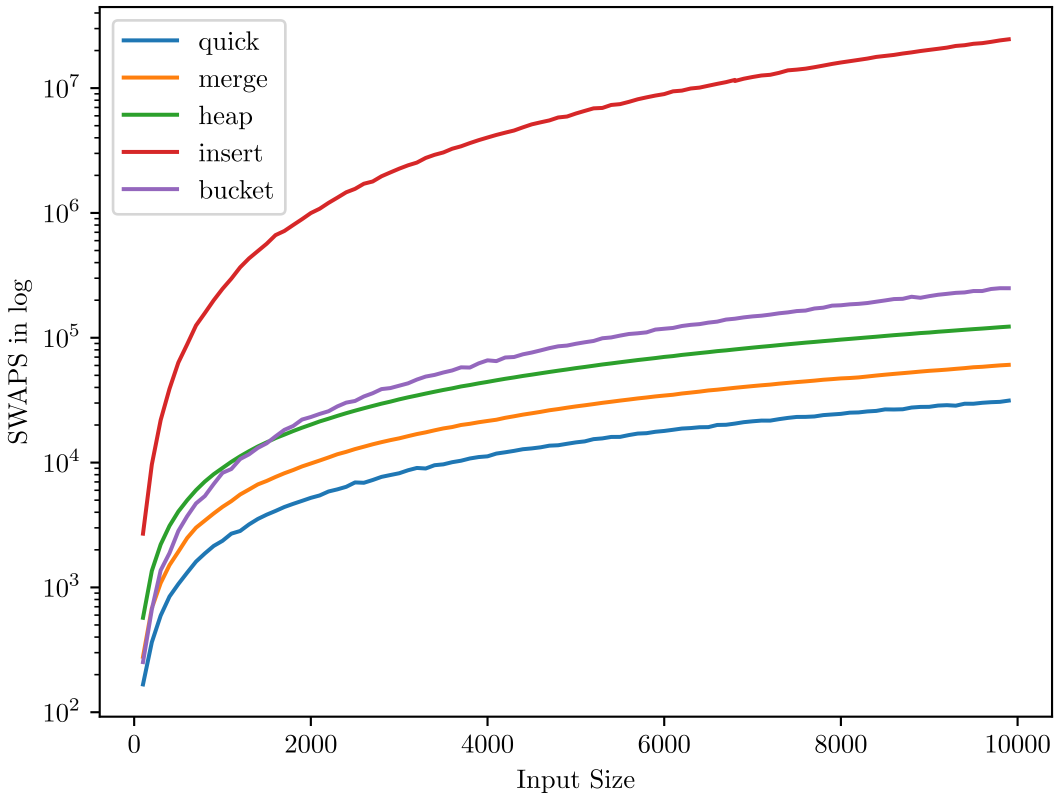
Algorithm	Best	Average	Worst	Worst(Space)
Quick	$\Omega(nlogn)$	$\Theta(nlogn)$	$O(n^2)$	$O(n)/O(1)$
Merge	$\Omega(nlogn)$	$\Theta(nlogn)$	$O(nlogn)$	$O(n)$
Heap	$\Omega(nlogn)$	$\Theta(nlogn)$	$O(nlogn)$	$O(n)/O(1)$
Insertion	$\Omega(n)$	$\Theta(nlogn)$	$O(n^2)$	$O(1)$
Bucket	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$

Here the space complexity of *Quick sort and Heap sort* varies depending upon the implementation

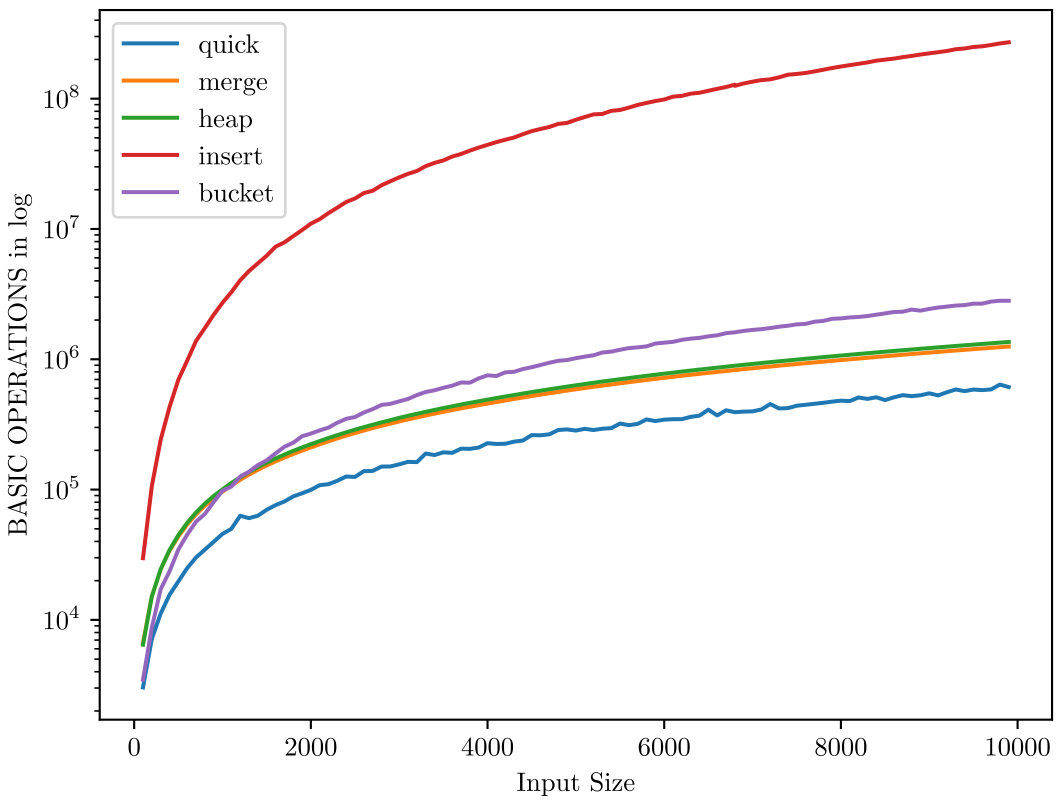
1.5 Plots

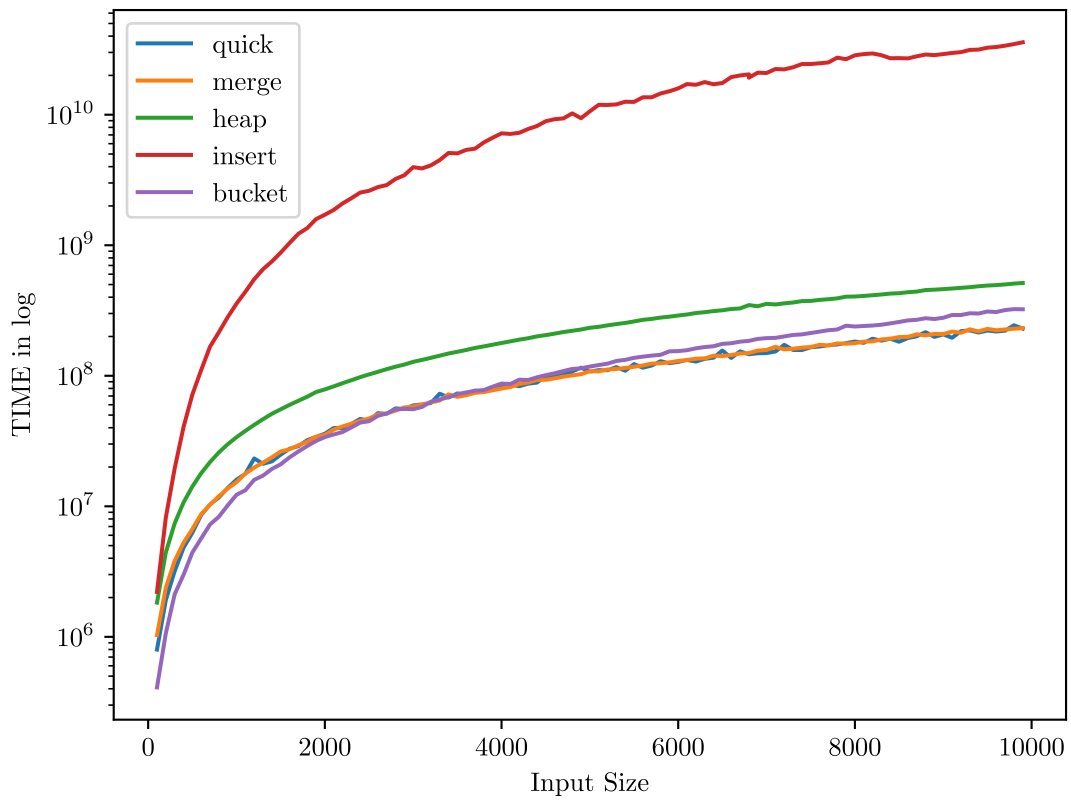


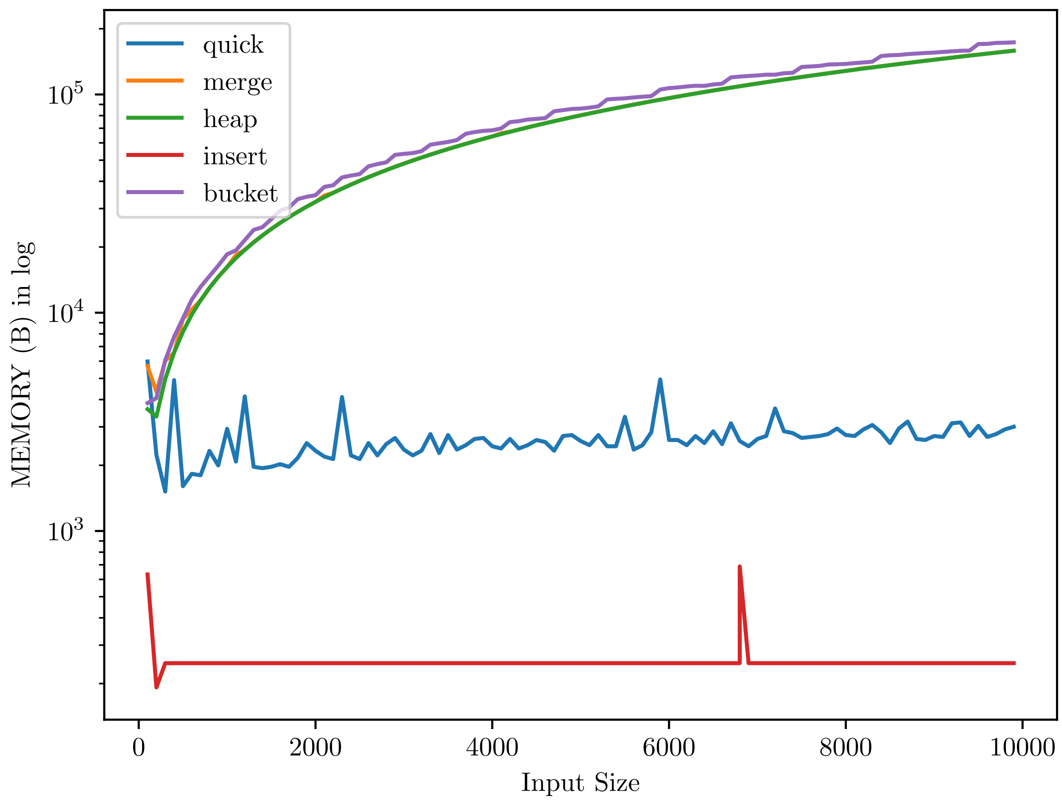
quick
merge
heap
insert
bucket
COMPARISONS in log
10^3
10^4
10^5
10^6
10^7
0
2000
4000
6000
8000
10000
Input Size



quick
merge
heap
insert
bucket
10^7
10^6
10^5
10^4
10^3
10^2
SWAPS in log
0
2000
4000
6000
8000
10000
Input Size


quick
merge
heap
insert
bucket
BASIC OPERATIONS in log
10^4
10^5
10^6
10^7
10^8
0
2000
4000
6000
8000
10000
Input Size


quick
merge
heap
insert
bucket
10^{10}
10^9
10^8
10^7
10^6
TIME in log
0
2000
4000
6000
8000
10000
Input Size



1.6 Analysis

As we can see the graphs match the theoretical complexities calculated. Let us focus on two most important measurements - *Time, Space.*

1.6.1 Time

Insertion sort with its $O(n^2)$ performs the worst. The other 4, which all have $O(nlogn)$, perform relatively the same. The reason why heapsort doesn't perform nearly as good as the other three is on the hardware level heapsort has **more instruction that the other three.**

1.6.2 Space

Here *Inplace quicksort and insertion sort* are the best as the have $O(1)$ space complexity. In-place heapsort also has $O(1)$ space complexity but the ADT used created a new heap from the given array. That is the reason it has a similar memory use of other $O(n)$ algorithms. But the heapsort itself **doesn't** need any auxiliary space.

2 Optimizing Quick Sort

2.1 Problem

The worst case performance of Quick sort is $O(n^2)$. This occurs when the *pivot chosen is the greatest or the smallest* causing one of the partitions to be of size $n - 1$.

2.2 Solution

This has an easy fix by picking a better pivot. The solution implemented here is called `Three median pivot`. Use *Insertion sort* for partitions **smaller than a certain threshold(Eg. 10)**

```
mid = (left + right) // 2
if arr[mid] < arr[left]:
    arr[left], arr[mid] = arr[mid], arr[left]
if arr[end] < arr[left]:
    arr[left], arr[end] = arr[end], arr[left]
if arr[mid] < arr[end]:
    arr[end], arr[mid] = arr[mid], arr[end]
pivot = arr[end]
pivot_i = end
```

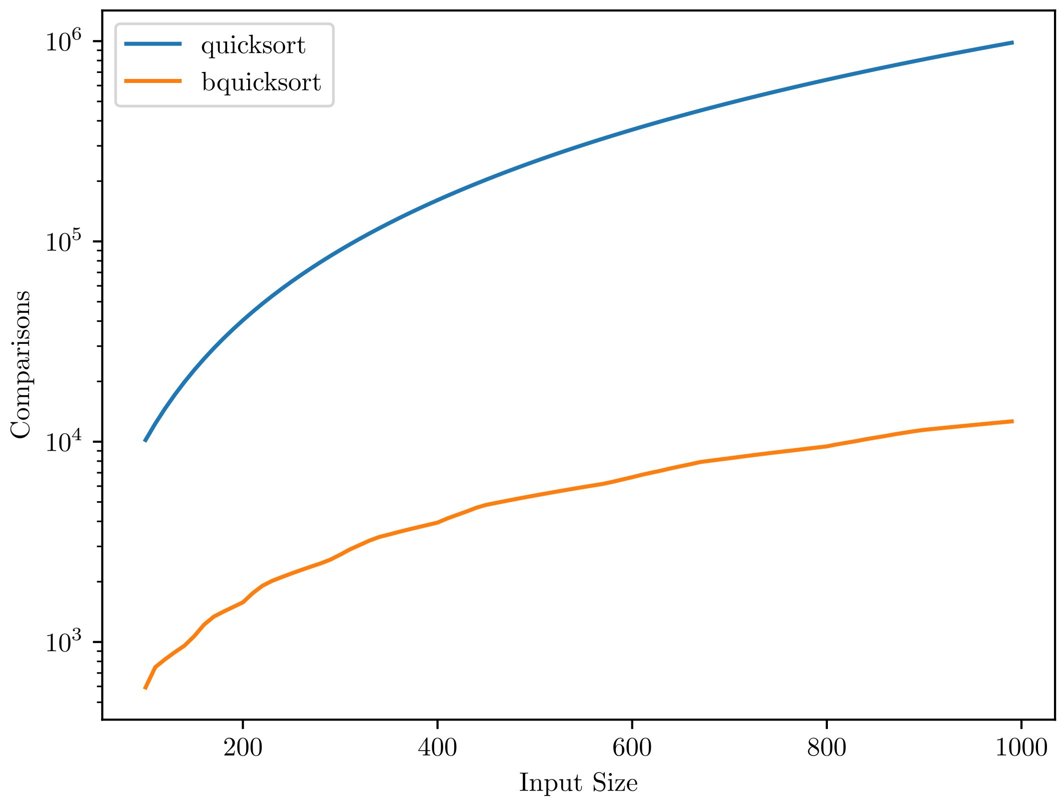
This makes it so that the pivot is never the greatest or the smallest.

2.3 Testing

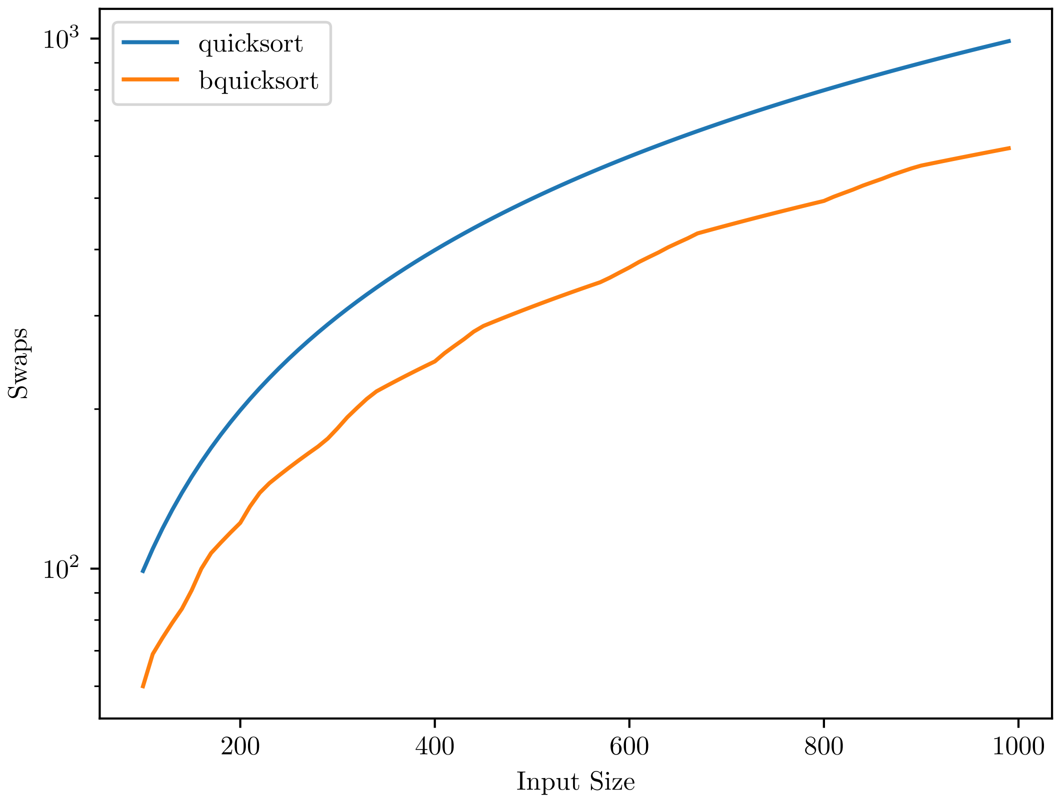
The data given to this particular comparison was sorted in the reverse order which is the worst case of normal quick sort.

2.4 Plots

2.4.1 Comparisons

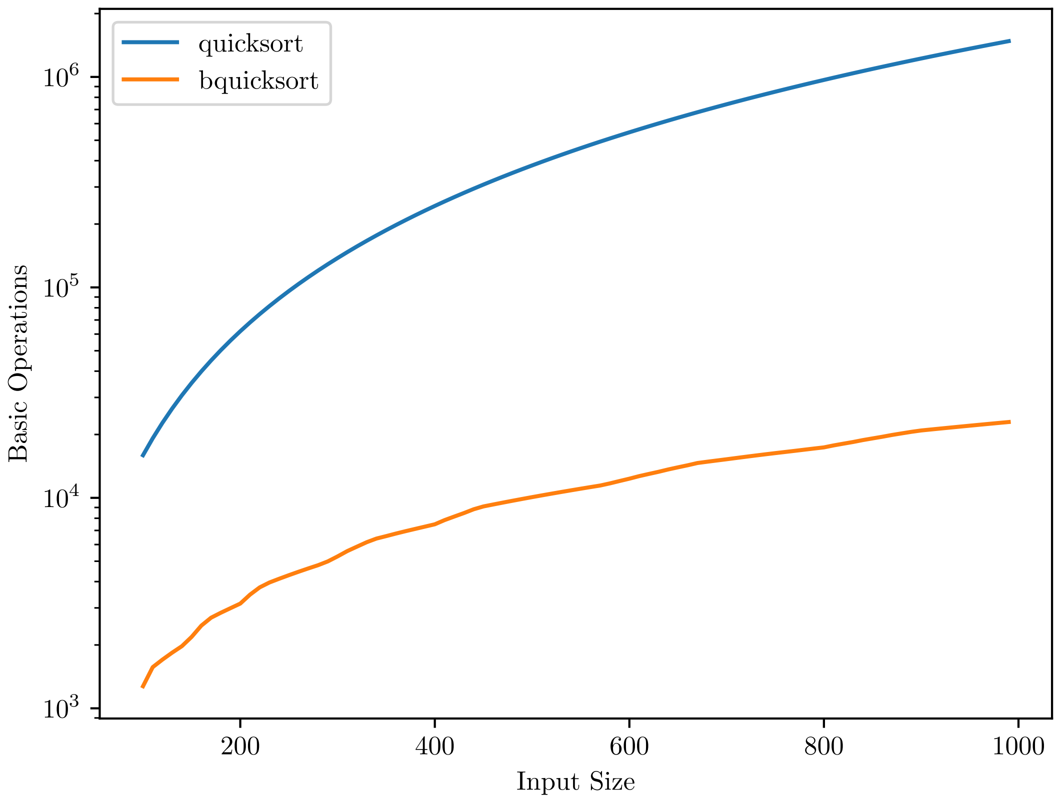


2.4.2 Swaps

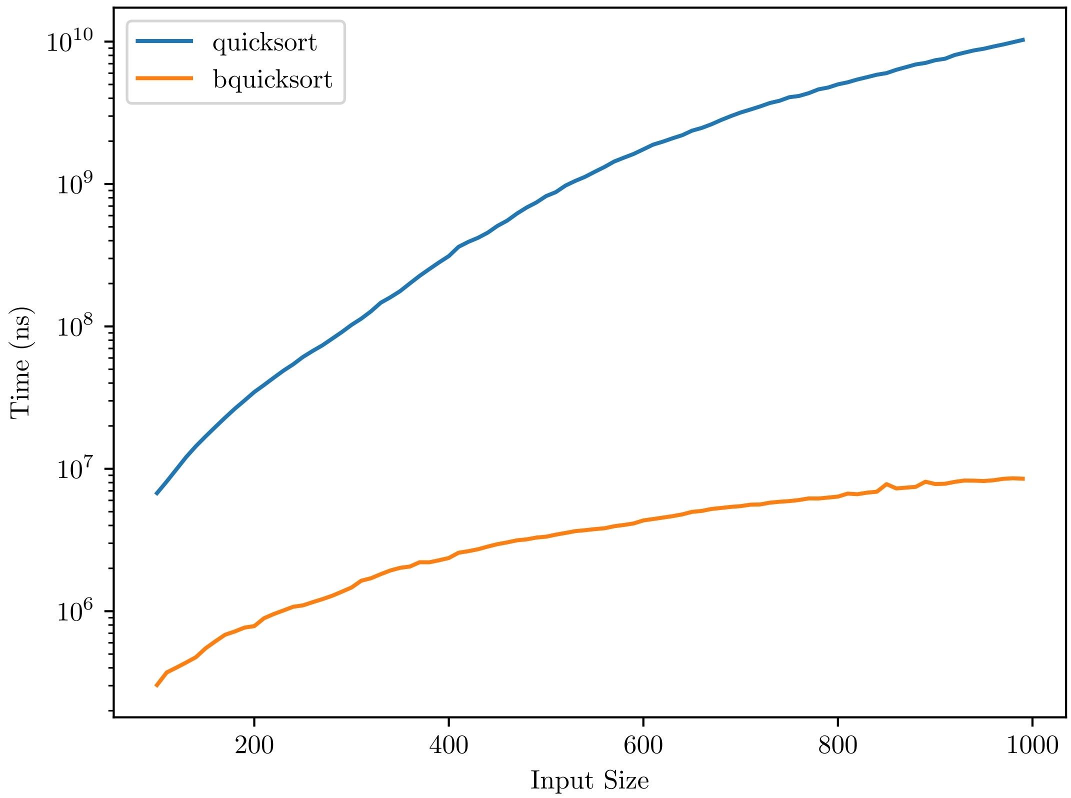


10^3
10^2
quicksort
bquicksort
Swaps
200
400
600
800
1000
Input Size

2.4.3 Basic Operations

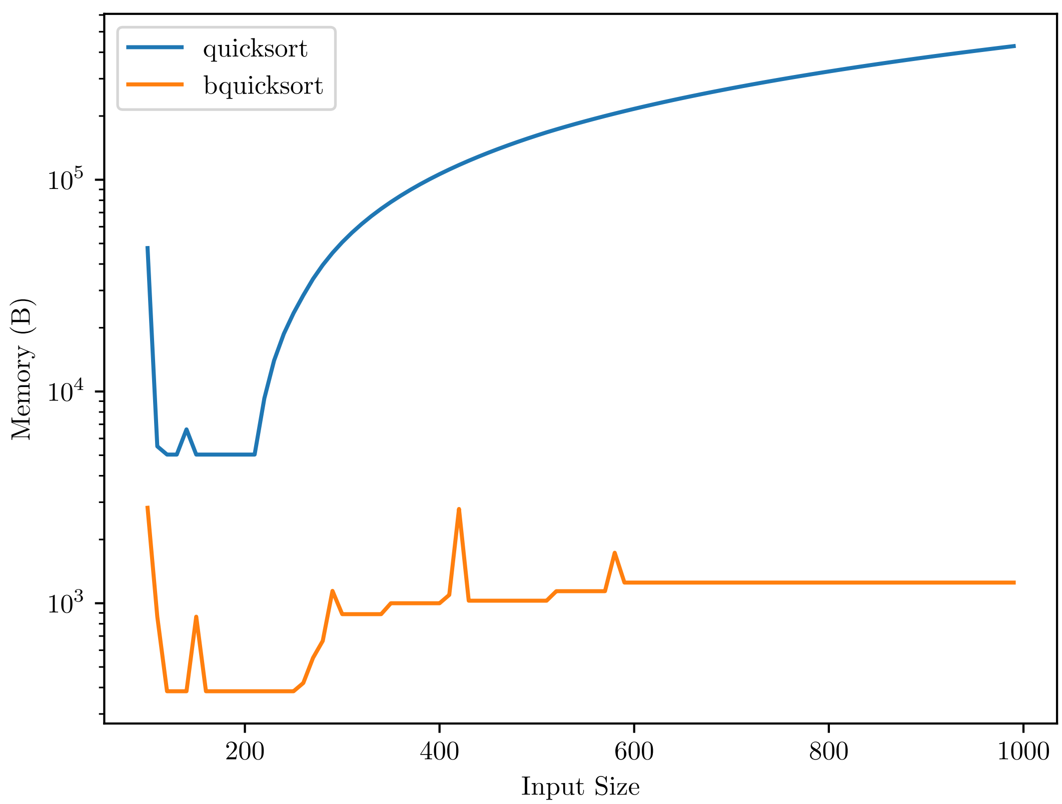


2.4.4 Time



quicksort
bquicksort
10^{10}
10^9
10^8
10^7
10^6
Time (ns)
200
400
600
800
1000
Input Size

2.4.5 Memory



2.5 Analysis

As we can see the time taken has *decreased significantly*. The number of comparison, swaps and basic operations has also shown a *slight decrease*. The decrease in memory is probably due to *lesser number of recursion calls*.

3 Optimizing Merge Sort

3.1 Problem

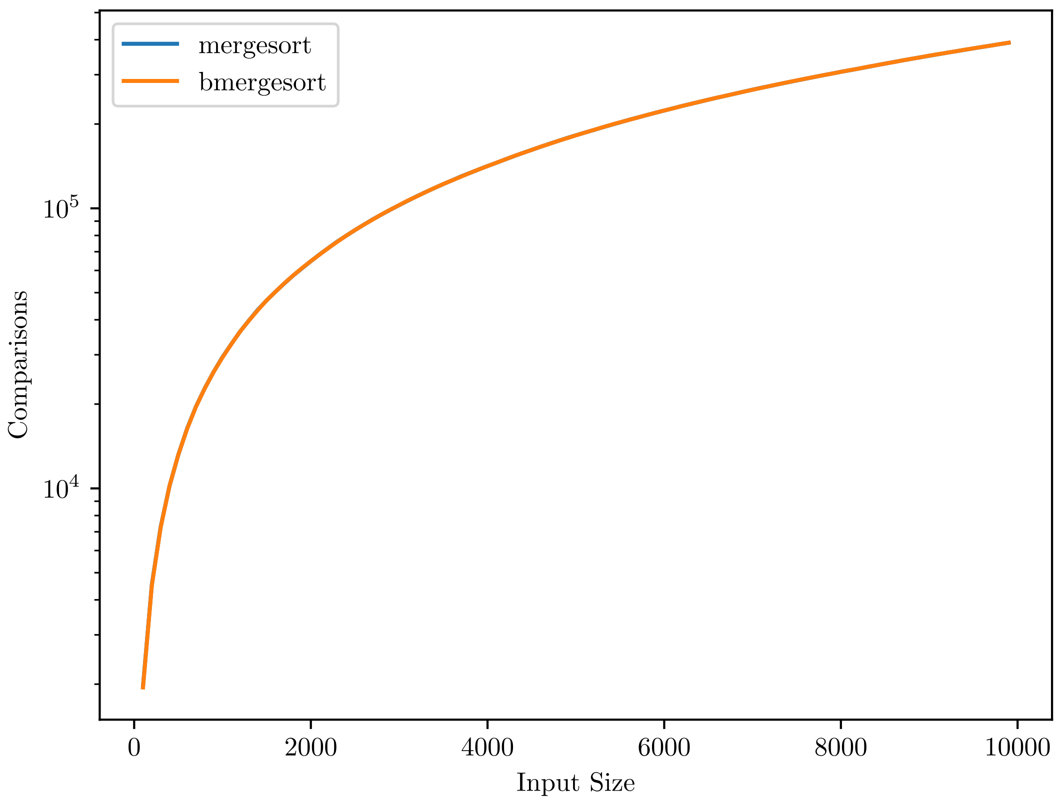
The main problem with merge sort is not the time complexity as it is one of the fastest but the *space overhead*. Implemented in its most basic form it requires $O(n)$ space to create *subarrays* each recursion.

3.2 Solution

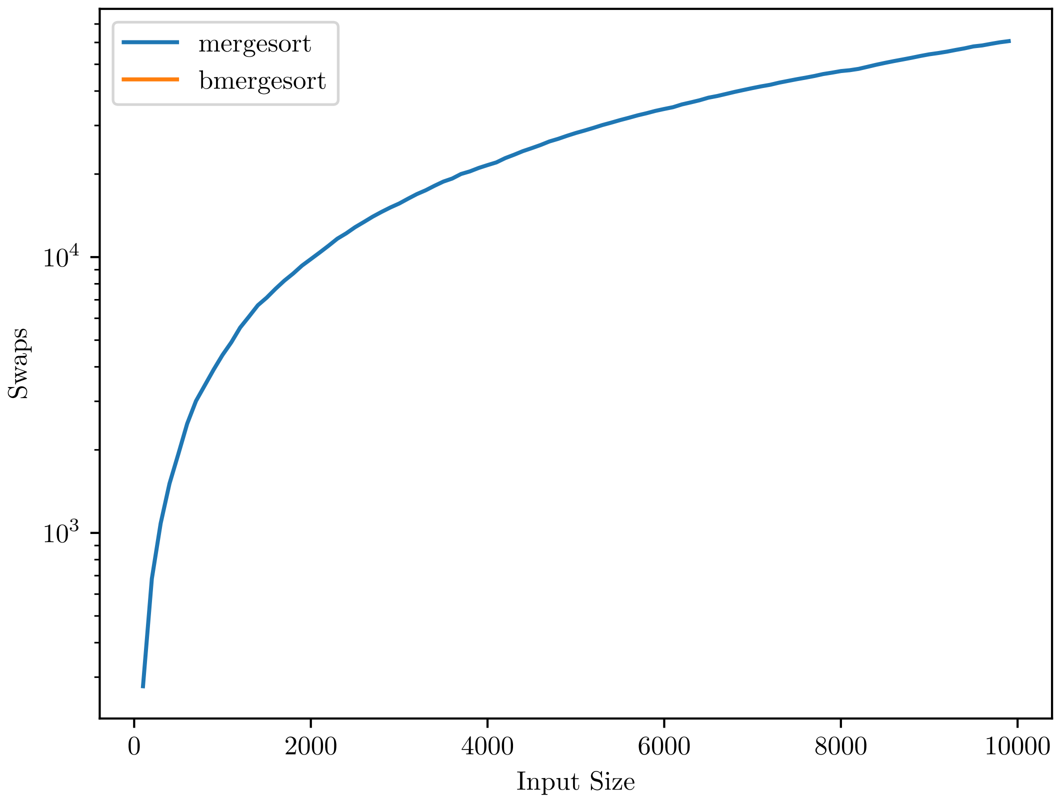
The solution implemented here is from *Katajainen, Jyrki; Pasanen, Tomi; Teuhola, Jukka (1996)*. Instead of creating subarray we pass the original array with index ranges. The only time subarrays are created is in the `merge()` routine. Here `merge()` creates a copy of the smaller subarray.

3.3 Plots

3.3.1 Comparisons

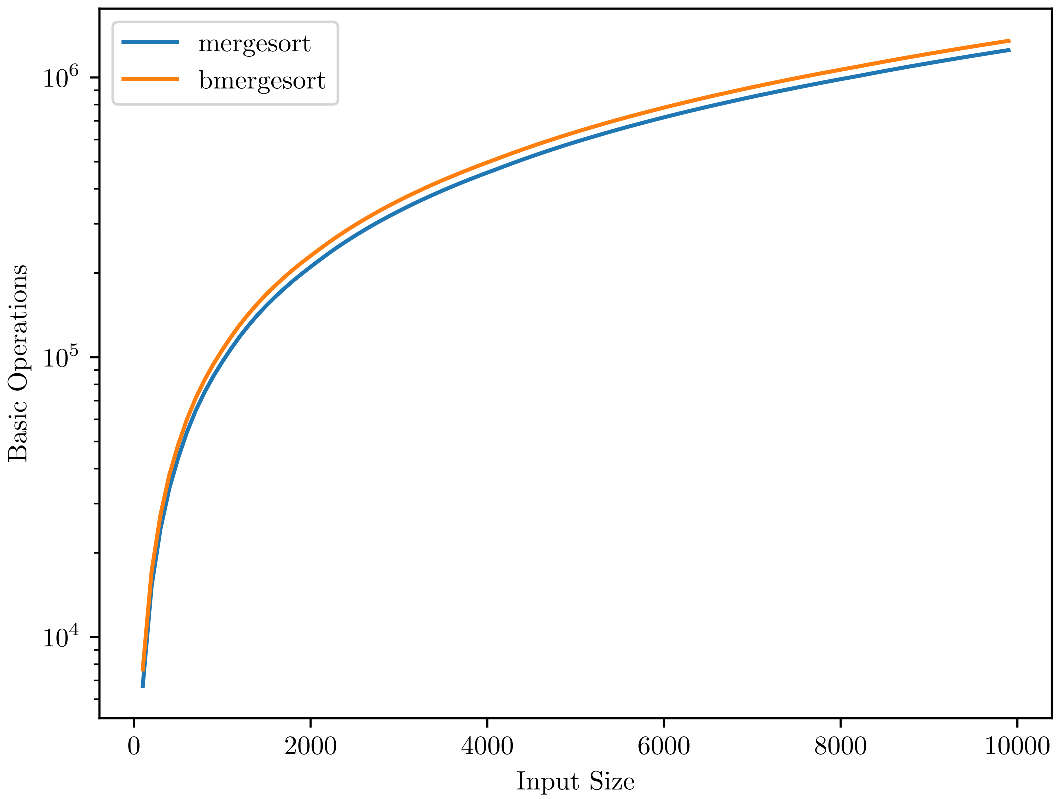


3.3.2 Swaps

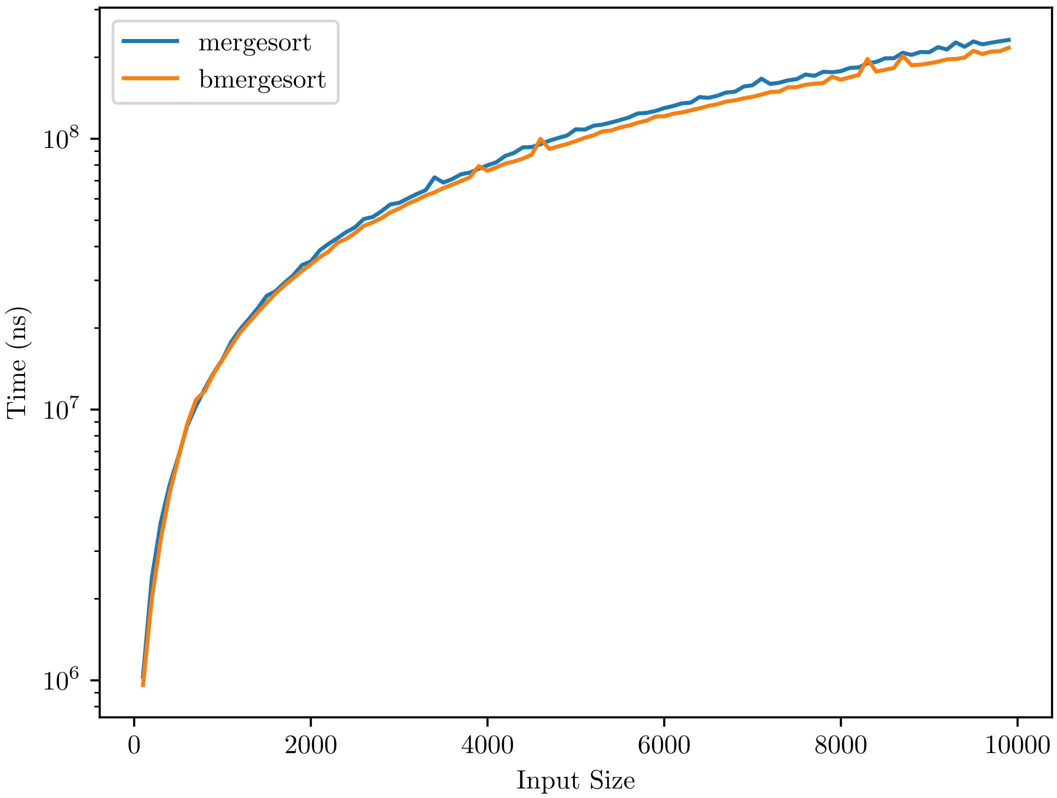


mergesort
bmergesort
Swaps
10^4
10^3
0
2000
4000
6000
8000
10000
Input Size

3.3.3 Basic Operations

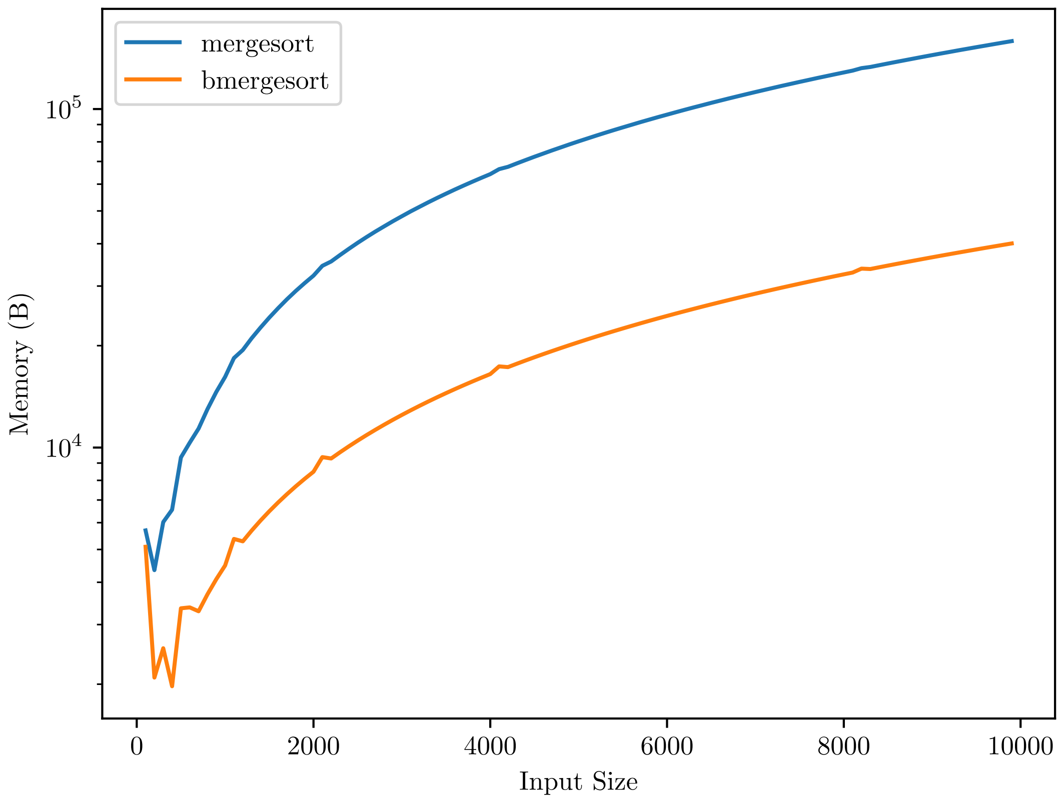


3.3.4 Time



mergesort
bmergesort
Time (ns)
10^8
10^7
10^6
0
2000
4000
6000
8000
10000
Input Size

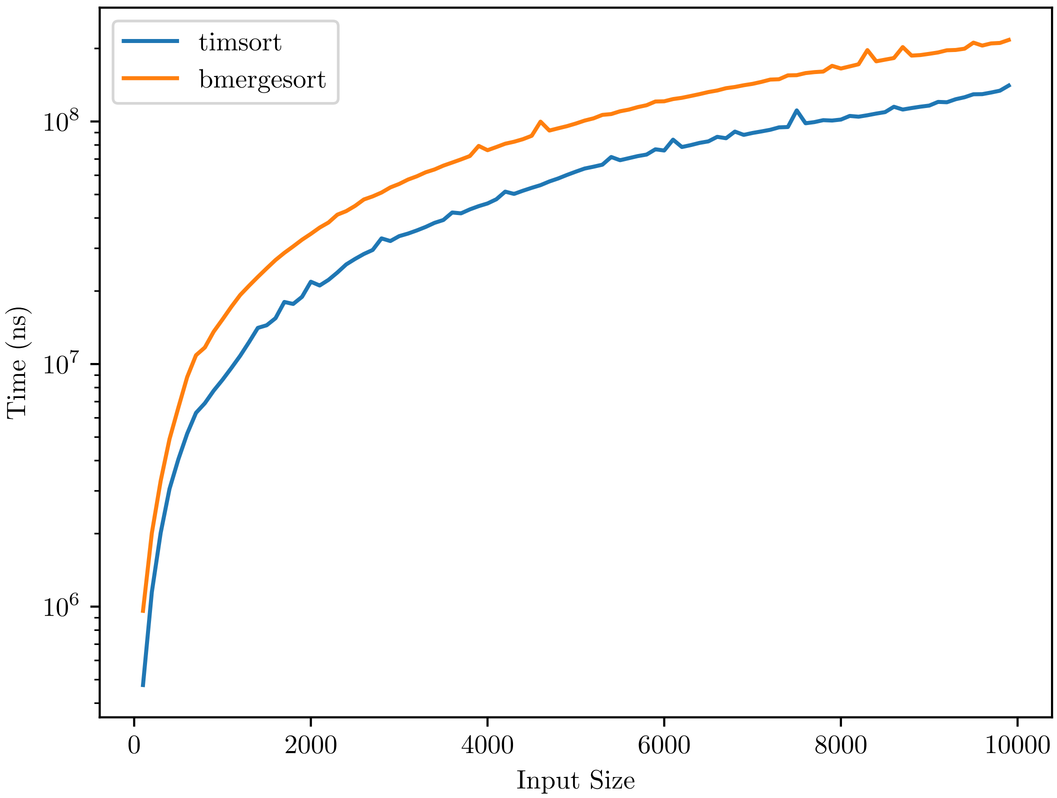
3.3.5 Memory



3.4 Analysis

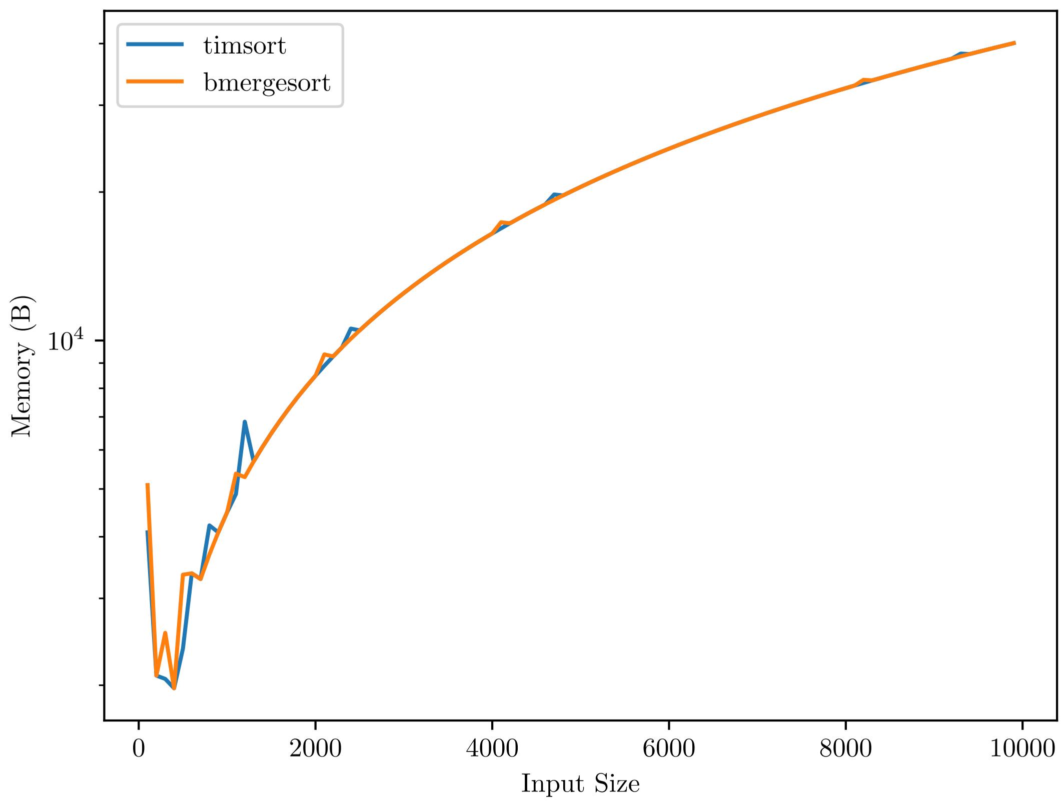
Since we are practically using only half the memory we see a huge **decrease**. The other metrics are very similar. To improve the time we can use **timsort** which runs *Insertion sort* upto a point and switches to merge sort.

3.4.1 Time



timsort
bmergesort
Time (ns)
10^8
10^7
10^6
0
2000
4000
6000
8000
10000
Input Size

3.4.2 Memory



timsort
bmergesort
Memory (B)
10^4
0
2000
4000
6000
8000
10000
Input Size