Benchmarking sorting algorithms in Python

INF221 Term Paper, NMBU, Autumn 2019

Anders Mølmen Høst and Ashesh Raj Gnawali
Data Science Section, Faculty of Science and Technology, Norwegian University of Life Sciences
anderhos@nmbu.no,asgn@nmbu.no

ABSTRACT

In this paper, we analyze different sorting algorithms and apply them to randomly generated data. We bench-mark the sorting algorithms on data consisting of up to a million numbers, the numbers ordered in four different ways to simulate best-case, average-case and worst-case scenarios. Our motivation for this paper was to test and investigate further the expected runtimes of sorting algorithms presented in the theory.

The results of our benchmarking show that NumPy sort performs best for large randomized data followed by Python sort. Among the three algorithms, we implemented in Python, Quicksort performs best for random data. Random data is usually used for representing the average case. The constant factors in Quicksort is small and has advantage of sorting in place. It also make good use of memory taking advantage of virtual memory and available caches [Cormen et al. 2009]

1 INTRODUCTION

According to [Cormen et al. 2009] "An algorithm is a sequence of computational steps that transform the input into the output. " Algorithms are used to typically solve a class of problems or to perform computations. There are various kinds of algorithms, namely, a searching algorithm that looks for an item in a list, a sorting algorithm that orders a list in numeric or alphabetical order and compression algorithms which reduces the size of a file.

In this paper, we deal with sorting algorithms that are extensively used in the computer science world. When the lists are ordered it is easier to find the item which we are looking for as a smaller number of comparisons are required. We generated our test data in four different orders consisting of random data, sorted data, reverse sorted data and identical data. Then we benchmarked five different sorting algorithms for data consisting of up to a million numbers.

2 THEORY

2.1 Merge sort

Merge sort encompasses the divide-and-conquer technique as it repeatedly divides the given problem size into half. If the list is empty or has one item it is sorted by definition. If the list has more than one item, we split the list and recursively invoke a Merge sort on both halves. When the two halves are sorted, a merge procedure is performed which combines the two halves into a new single, sorted list. A downside of this algorithm is that it requires extra memory to copy the elements when sorting. The average and the worst-case performance of merge sort is $O(nlgn)$.

Listing 1 Merge sort algorithm from Cormen et al. [2009, Ch. 2.3. p 31].

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

Listing 2 Merge algorithm from Cormen et al. [2009, Ch. 2.3. p 34].

```
MERGE($A, p, q, r$)
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to r
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
            $j = j + 1$
```

2.2 Quick sort

Quick sort also uses the divide-and-conquer technique but does not require additional memory which was necessary for merge sort. The algorithm uses a pivot element to separate the array in two subarrays. After the partition procedure is done, all elements placed in the left subarray are less or equal the elements placed in the right subarray. The efficiency of quick sort depends on how we choose the pivot element. The pivot element is used to divide the list for consequent recursive calls to Quicksort.

The average runtime of Quicksort is $O(nlogn)$ but runs in $O(n^2)$ in the worst case when the input data is in ascending order or when the leftmost element is chosen as the pivot, because in these cases the Quicksort has most unbalanced partitions possible.

[Cormen et al. 2009]

Listing 3 Quicksort algorithm from Cormen et al. [2009, Ch. 7.1. p 171].

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

Listing 4 Partition algorithm from Cormen et al. [2009, Ch. 7.1. p 171].

```
PARTITION($A, p, r$)
1  $x = A[r]$
2  $i = p - 1$
3  for $j = p$ to $r - 1$
4      if $A[j] \leq x$
5          $i = i + 1$
6          exchange $A[i] with A[j]$
7  exchange $A[i + 1]$ with $A[r]$
8  return $i + 1$
```

2.3 Heap sort

The heapsort algorithm builds a heap from a list of unordered elements and then extracts the elements from the heap to create a sorted list. A heap is a complete binary tree that obeys the heap ordering property. We can extract, remove or insert a new value into a heap. The sort procedure removes the largest element from the root of a max heap and replaces it with the rightmost leaf. The largest element is now stored in an array. Now, we re-establish the max heap and repeat the process until there are no elements left in the heap. The best case runtime is $\Omega(nlgn)$ while the worst and the average runtime is order of $O(nlgn)$.

Listing 5 Buildig a Max Heap algorithm from Cormen et al. [2009, Ch. 6.3. p 157].

```
BUILD-MAX-HEAP($A$)
$A.heap\text{-}size = A.length$
for $i = \lfloor (A.length/2) \rfloor$
    MAX-HEAPIFY($A, i$)
```

2.4 Python sort and NumPy sort

In addition to the previous sorting algorithms, we have also benchmarked the built-in Python sort function as well as NumPy sort.

Python sort is a method for the list data type and part of the python standard library of Python [Library [n. d.]]. It sorts the data in place if the sort() method is used. The mentioned method only works on lists. However, Python also has the command sorted() which is a function to sort any iterable by copying elements in a new list. In this paper we will be using the sorted() command.

NumPy sort by copying the data in a new array. By default, NumPy sort is based on Quicksort, but it is able to switch to heapsort if it does not make enough progress. However, the user may specify

Listing 6 Max-Heapify algorithm from Cormen et al. [2009, Ch. 6.3 p 154].

```
MAX-HEAPIFY($A, i$)
$l = $ LEFT($i$)
$r = $ RIGHT($i$)
if $l \leq A.heap\text{-}size$ and $A[l] > A[i]$
    $largest = l$
else $largest = i$
if $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$
    $largest = r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    MAX-HEAPIFY($A, largest$)
```

Listing 7 Heapsort algorithm from Cormen et al. [2009, Ch. 6.4 p 160].

```
HEAPSORT($A$)
BUILD-MAX-HEAP$A$
for $i = A.length$ downto 2
    exchange $A[1]$ with $A[i]$
    $A.HeapsizeA.Heapsize - 1$
    MAX-HEAPIFY($A$, 1)
```

if it will use e.g NumPy based on merge-sort instead. ["NumPy" 2019]

3 METHODS

We implemented the algorithms in Python using Jupyter notebook. We tested the correctness by generating small data sizes and verifying it to our own knowledge.

We used timeit module to record the time for benchmarking by creating a Timer object named clock. We provided a fresh copy of data to be sorted on every call to the `sort_func` parameter so that when sorting all the excecutions of the sorting function would sort the test data. We excecuted five repetitions to sort the same amount of data and used the minimum time value to plot the graphs. We also used $to_pickle()$ method to store the benchmark results so that we would not have to run the tests over again.

We generated the random data using the random module in Python. We used the sorted function in Python to sort the generated random data to use in our experiment. For various kinds of input data which were sorted, random, reversed and identical, the resulting times were plotted and analyzed by using seaborn, matplotlib and pandas library in Python.

We encountered a Python recursion limit error while benchmarking the sorted and reversed data sizes greater than 1000 for Quicksort. This is because, in case of sorted data, the Partition algorithm in the for-loop of lines 3-6 compares element[j] from p to $r - 1$ with the pivot-element. As we are dealing with sorted data, the If-statement in line 4 will always be true. The number of required recursions will then be one less than the length of the data.

Thus, we have presented two different graphs. The first one includes data size up to 1000 numbers plotted for all the sorting

Listing 8 Timer function from [Plesser 2019]

```
import numpy as np
import timeit
import copy

np.random.seed(12235)
test_data = np.random.random((1000,))

clock = timeit.Timer(stmt='sort_func(copy(data))',
         globals={'sort_func': sorted,
                  'data': test_data,
                  'copy': copy.copy})

n_ar, t_ar = clock.autorange()

t = clock.repeat(repeat=5, number=n_ar)
```

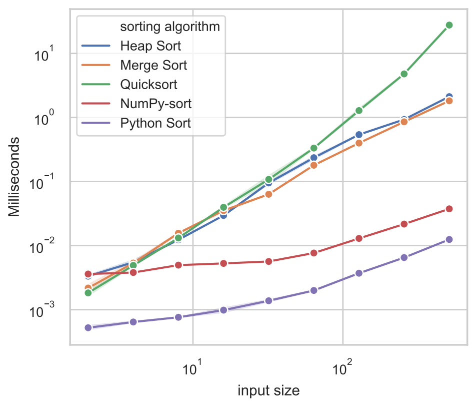

Figure 1: Benchmark results for small sorted data, log-log plot

algorithms and the second includes a million numbers without sorted and reversed data in case of Quicksort.

The benchmarks were run on a computer with processor specifications: $IntelCorei5 - 8250U@1.6Ghz(8CPU) \sim 1.8Ghz$ with memory 8192 MB of RAM. Python version 3.7.3 with Pandas 0.24.2 and NumPy 1.16.4 was used. The results from the benchmarking can be found in the tables in the Appendix. The source code can be found in the GitHub link https://github.com/ anderhos/termpaper

4 RESULTS

4.1 Benchmark results for a thousand data

Starting with the bench-marking of sorted data in Figure 1. Python sort is the fastest with around 20 μs and then NumPy sort at around 70 μs followed by Merge Sort at 3 ms and Heapsort and Quicksort at 6 and 77 ms respectively.

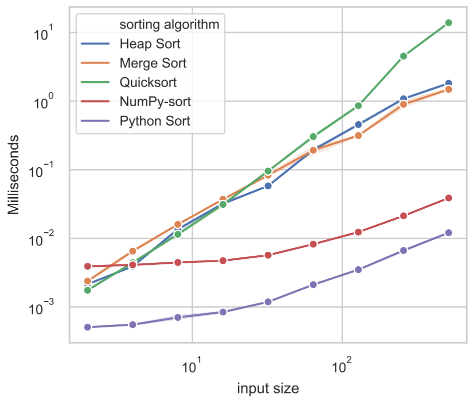

Figure 2: Benchmark results for small reversed data, log-log plot

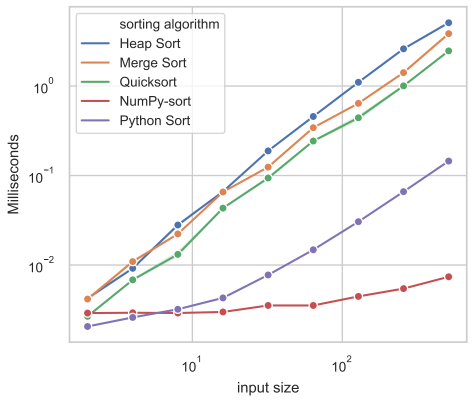

Figure 3: Benchmark results for small random data, log-log plot

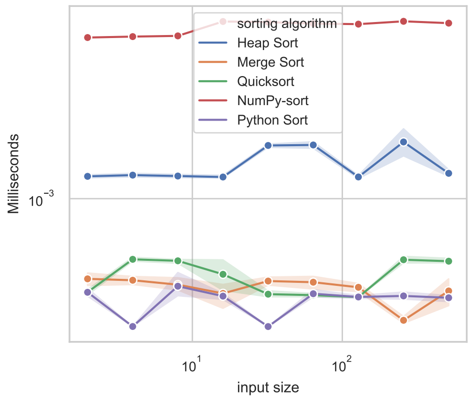

Figure 4: Benchmark results for a thousand identical data, log-log plot

For reverse sorted data Python sort was quickest with a runtime of 23 μs followed by NumPy sort at around 73 μs. The worst performer was Quicksort with a runtime of 62 ms. Merge sort and heap sort performed averagely at around 4 ms.

For random data, NumPy sort was the quickest of all at 30 μs, followed by Python sort at 307 μs. The third algorithm was Quicksort with a runtime of 6 ms being slightly faster than merge sort which took 7 ms. The heap sort took the longest time at 111 ms. Surprisingly, the Numpy sort took the longest time at 3 μs for identical data, followed by heapsort at 1 μs. Python sort and Quicksort had a similar performance at 0.5 μs. The quickest one was Merge sort at 0.4 μs.

4.2 Benchmarking results for one million data

	Types of Data	Quicksort	Heapsort	Mergesort	Pythonsort	Numpysort
0	Sorted	NaN	9.862467	6.460021e+00	1.269479e-01	0.247409
1	Reversed	NaN	9.474967	6.752326e+00	1.283044e-01	0.270596
2	Random	1.020480e+01	23.684537	1.327605e+01	7.630761e-01	0.081780
3	Identical	4.920000e-07	0.000001	4.070000e-07	3.950000e-07	0.000003

Table 1: Sorting of data consisting of one million numbers measured in seconds.

We can see in Table 1 that for random ordering NumPy sort has the lowest runtime with 0.08 seconds followed by Python sort with 0.76 seconds. Quicksort is the third fastest sorting algorithm and has a runtime 10.20 seconds. Then follows Merge sort with 13.27 seconds and Heapsort performs the worst at 23.68 seconds. The ordering is the same for one million data as for a thousand data. However, Pyhon sort and Numpy sort are performing relatively better than the Python implemented algorithms. When comparing to Python sort which is ranked as second in runtime for random data. Then for a thousand data the runtime of Quicksort is around five times the runtime of Python sort, while for a million data the factor is around ten.

When looking at reversed sorted data Quicksort is excluded because of the recursion limit of a thousand as mentioned earlier. The ordering of runtime for sorting algorithms for a million data is the same as for a thousand data. The best performing algorithm is Python sort which uses 0.13 seconds followed by NumPy sort at 0.27 seconds, then Merge sort 6.75 seconds and Heap sort as the worst performer with a runtime of 9.47 seconds. NumPy sort performs relatively better for a million data than for a thousand compared to Python sort with the runtime of NumPy being a factor of close to three times the runtime of Python sort for a thousand data, while for a million data this factor has reduced to approximately two times the runtime of Python sort. Compared to merge sort which is ranked as the third-fastest algorithm in both cases merge sort runs at a factor of 25 times the runtime of NumPy sort for a million data while for a thousand data this number is almost 50. When comparing the three fastest algorithms merge sort is performing relatively better for a million data than for a thousand data.

As for reversed sorted data, when using Quicksort on ordinary sorted data it reaches the same recursion limit and is therefore excluded. Python sort is the fastest running on 0.13 seconds followed by NumPy sort on 0.25 seconds, then Merge sort 6.75 seconds and Heapsort 9.47 seconds. NumPy sort performing better on sorted data for a million data than for a thousand data when comparing to

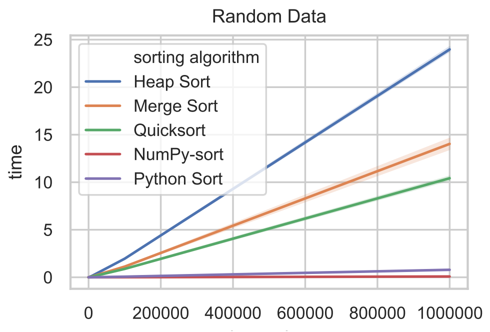

Figure 5: Benchmark results for a million random data

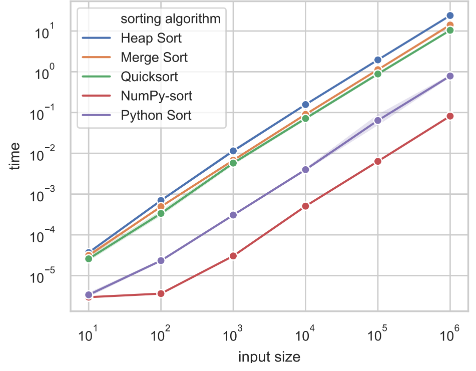

Figure 6: Benchmark results for a million random data

Python sort. For a thousand data, the runtime of NumPy is almost three times as high as for Python, while for a million data NumPy sort uses less than double the amount of time.

The times are almost the same for sorting a million data identical data as for a thousand data. In the graph, we can see that Heap sort increases by a small amount while NumPy decreases by an even smaller amount. The ordering of the performance of the Algorithms is the same as for a thousand data and the runtime stays close to zero.

5 DISCUSSION

As discussed in the theory, Quicksort took more time for sorting sorted data (worst case) than it took for sorting the randomly generated data. It took less time for sorting reversed data as compared to sorted data.

Merge sort took the same time for sorting reversed as well as sorted data as the ordering of data does not affect the sorting procedure in merge sort.

Heapsort takes less time for sorting reversed data when compared to the sorting time of sorted data for a thousand data. Heapsort is the slowest of the $O(nlgn)$. However it has the advantage that it do less comparisons compared to Quicksort so that on already sorted data it is faster. Merge sort is slightly faster than heap sort but requires twice the memory because of the second array.

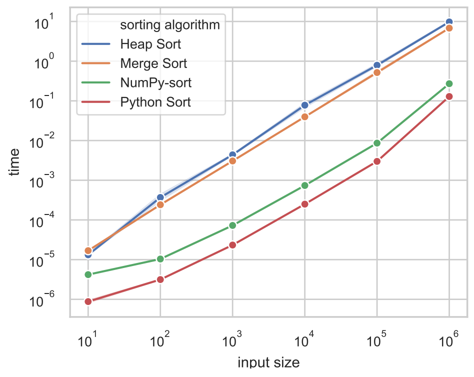

Figure 7: Benchmark results for a million reverse sorted data, log-log plot

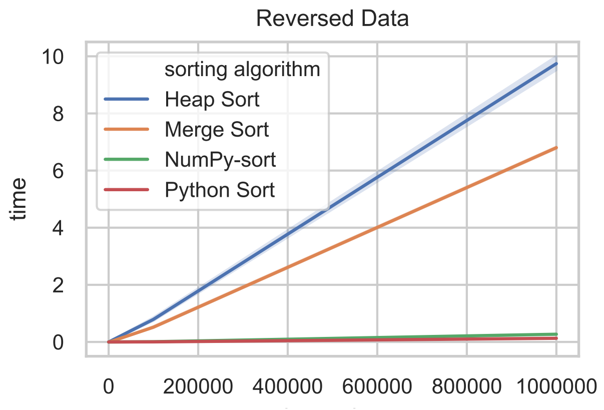

Figure 8: Benchmark results for a million reverse sorted data

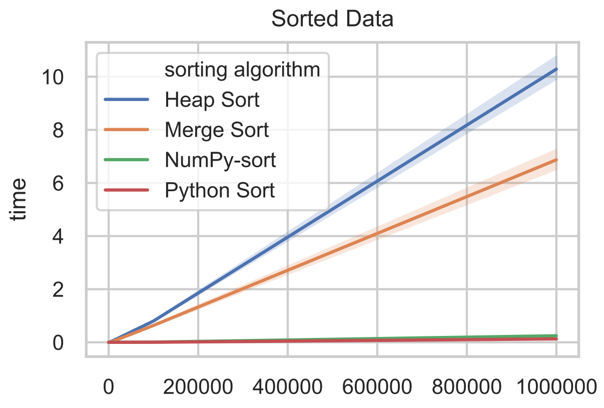

Figure 9: Benchmark results for a million sorted data

Comparing Heapsort and Merge sort we can see, that for sorted, reversed and random data Merge sort performs best. The greatest difference in our data between the performance of Heap sort and

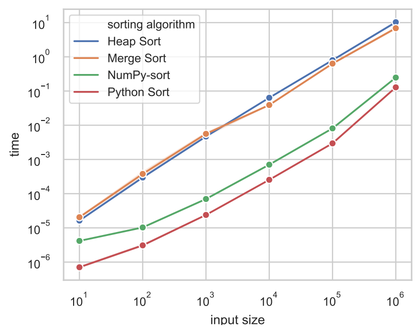

Figure 10: Benchmark results for a million sorted data, log-log plot

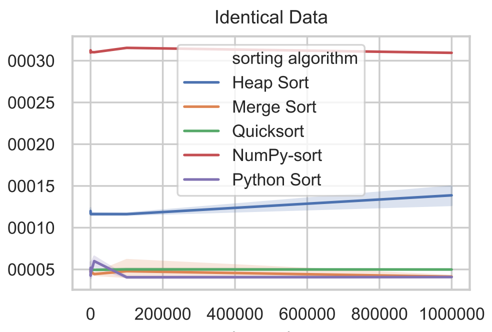

Figure 11: Benchmark results for a million identical data

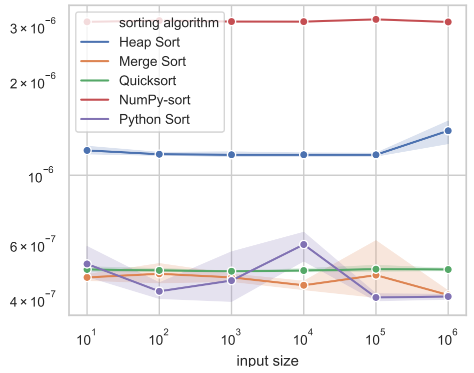

Figure 12: Benchmark results for a million identical data, log-log plot

Merge sort is when sorting already sorted data for a data including 512 numbers. In this case Merge sort is twice as fast as Heap sort. While for reversed data Merge sort and Heapsort performs almost the same. In the case of reverse sorted data, when building the heap it is already a max heap meaning we do not need to swap any parent with child nodes when we call Max-Heapify. One possible explanation for the overall differences between Merge sort and

Heapsort is that Merge sort has what is called better locality of references compared to Heapsort. [Wikipedia [n. d.]] Meaning it frequently accesses memory locations in closer space. Another way to look at Heapsort and Merge sort is that Heapsort swaps data, that is two reads and two writes per swap while Merge sort moves data, one read and one write per move. [Stackoverflow [n. d.]]

Quick sort is faster than both Heapsort and Merge sort. Python sort is most efficient in runtime when looking at sorted data for a million data, while for random data NumPy sort is fastest. Both Python sort and NumPy sort is significantly faster than the three Python implemented algorithms of Quicksort, Heap sort and Merge sort. Both NumPy and Python is partially programmed in the lower lever C-language which is known for its high speed. This might be one of the important reasons for why they are outperforming the other sorting algorithms which is implemented entirely in Python. ["NumPy" 2019]

Among the three python implemented algorithms Quicksort is fastest for randomly ordered numbers but is slow when sorting already sorted data and reverse sorted data. This is because of the pivoting element separating the data in a subset with the pivot element itself and a subset with the rest of the data as mentioned earlier in this paper. Then we will have the worst-case scenario of Quicksort with a runtime of $\Theta(n^2)$. Therefore, it seems to be important to compare Quicksort with other sorting algorithms, in particular in cases when sorting large datasets which might be partially or entirely sorted beforehand.

6 ACKNOWLEDGEMENTS

We would like to express our thanks to our teaching assistant Krista Gilman for her continuous guidance and support in completing this project.

Furthermore, we would also like to extend our gratitude to professor Hans Ekkehard Plesser who is responsible for the course, Computer Science for Data scientists, and also providing valuable aid with the term paper format and Latex code.

7 REFERENCES

REFERENCES

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press, Cambridge, MA.
The Python Standard Library. [n. d.]. Built-in types. https://docs.python.org/3/library/stdtypes.html?highlight=sort#list.sort
"NumPy". "2019". "numpy.sort". "https://numpy.org/doc/1.16/reference/generated/numpy.sort.html"
Hans Ekkehard Plesser. 2019. Sample term paper. (Oktober 2019).
Stackoverflow. [n. d.]. Heap sort vs Merge Sort in Speed. stackoverflow.com/questions/53269004/heap-sort-vs-merge-sort-in-speed
Wikipedia. [n. d.].

8 APPENDIX

	Types of Data	Quicksort	Heapsort	Mergesort	Pythonsort	Numpysort
0	Sorted	7.706708e-02	0.006292	2.923120e-03	2.390000e-05	0.000070
1	Reversed	6.199364e-02	0.003974	3.823736e-03	2.310000e-05	0.000073
2	Random	5.959418e-03	0.011186	7.484258e-03	3.072950e-04	0.000030
3	Identical	4.850000e-07	0.000001	4.000000e-07	4.860000e-07	0.000003

Table 2: Measured times in seconds sorting data consisting of one thousand numbers.