Analysis of sorting algorithms by benchmarking, Python

INF221 Term Paper, NMBU, Autumn 2021

Ole Christian Johnsrud
ole.christian.johnsrud@nmbu.no
NMBU

Tormod Høilo Ongstad
tormod.hoilo.ongstad@nmbu.no
NMBU

Kristian Møll Stegane
kristian.moll.stegane@nmbu.no
NMBU

Abstract

This paper has the purpose to highlight differences in performance for multiple sorting algorithms on sets of data that range from short to long, with a weighted empashis on the shorter set. Essentially, we will compare real-life-experiences with theoretical expectations. The algorithms under investigation can be sorted into four categories: quadratic (insertion and bubble sort), sub-quadratic (merge sort and quicksort), combined algorithms (merge sort and quicksort switching over to insertion sort for small data) and the built-in sorting functions (NumPy-sort and Python-sort). The input data used were lists of integer numbers with sizes of input base 2 and input power 14. There were no surprising findings from this study. Every algorithm performed according to theory, with the integrated sorting algorithms being significantly faster than our implementations. Insertion sort excelled on the sorted list and singled itself out as the main contender with regards to runtime for the sorted list. Bubble sort was the slowest overall and is not recommended for use in large scale sorting. Our hybrids, merge-insertion and quick-insertion, proved to utilize the quicker runtime of insertion sort for smaller lists and proved to be useful upgrades of the original pure algorithms. A notice to be made for further testing and tuning is that there is room for improvement and fine tuning with regards to the threshold value in the hybrid algorithms.

1 Introduction

As an ever changing world with continuous new problems, adaptation is the key. The same principle goes for optimizing our algorithms to best solve our problems. This study aimed to test theory against actual real life tests in regards to the running time of the different sorting algorithms presented in the "Theory" section. In this section each algorithm, and which family it belongs to, are presented along with what the theoretical running times are expected to be. In section 3, the "Methods" section, the testing methods for each sorting algorithm is presented. Here hardware and software specifications are presented alongside methods for the benchmarking with the purpose of making future reproduction of this experiment possible. Section 4 presents the results from each test illustrated by graphs and a short description of the test data. Lastly, the "Discussion" section combines the theoretical expected results with our actual output and compare the findings before a summary is presented.

2 Theory

2.1 Quadratic algorithms

This section analyse insertion sort and bubble sort as our selection of quadratic algorithms. Quadratic algorithms follow a quadratic pattern with the big theta notation being $\Theta(n^2)$. As stated in Cormen et al. [2009], both algorithms are in-place sorting algorithms, meaning that the elements are moved within the original array while sorted.

2.1.1 Insertion sort

Insertion sort starts its sorting process, as shown in listing 1, by defining the first element in a list as sorted. It then moves from the end of the sorted elements, comparing the new value against the sorted list elements until the new element is greater than the next element. As the program iterates through the list, it updates the sorted part so that the cursor (marking line between sorted and unsorted elements) moves one step to the right for each new element compared. This happens $i - 1$ times , where i is the range of numbers, until i reaches -1 which is decreased by one as the while-loop is activated. The swaps will continue until the entire list, including the element at index j, is sorted.

In the average and worst case scenarios, the insertion sort will have a time complexity of $\Theta(n^2)$, shown in equation 1, due to the fact that both loops needs to be activated in order if the list elements arent sorted.

$$T(n) = \Theta(n^2)\ . \tag{1}$$

The best case scenario has a time complexity of $\Omega(n)$ as the while loop is not activated due to the condition, as seen in line 4 of the pseudocode, not being satisfied. This happens as the list is already sorted and all the elements are on the right place. According to Cormen et al. [2009], "Insertion sort is an effective algorithm for smaller number of elements."

Listing 1 Insertion sort from Cormen et al. [2009, Ch. 2.1].

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      i = j − 1
4      while i > 0 and A[i] > key
5          A[i + 1] = A[i]
6          i = i − 1
7      A[i + 1] = key
```

2.1.2 Bubble sort

Bubble sort, as seen in listing 2, works by comparing pairs of elements in a list from left to right. If the right number of the pair is smaller than the left element, they swap places. This operation runs for every pair until the rightmost element is the largest and the leftmost element is the smallest. This means that during the operation of swapping, the algorithm will be sorted from the last position first and work its way backwards. The algorithm now knows that

the last element is sorted and repeats this process minus the sorted element. Since the bubble sort algorithm contains a nested for-loop, the runtime for this sorting algorithm is quadratic, as shown in equation 2.

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

Listing 2 Bubble sort algorithm from Cormen et al. [2009, Ch. 2].

```
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]
```

2.2 Sub-quadratic algorithms

The family of sub-quadratic algorithms share a logarithmic running time of $\Theta(n \cdot \lg n)$. They have in common that they use a divide-and-conquer method where the problems are divided into smaller sub-problems that are easier to handle. These smaller sub-problems, when broken down into single elements, are then recursively put back together giving an output, in form of a solution, to the problem of origin.

2.2.1 Merge sort

Using recursion, the merge sort algorithm, shown in listing 3, divides a list of n elements into two lists, given that $n > 1$, and sorts the two lists separately. Then, with the merge function presented in the pseudocode in listing 3, the two lists are combined to form a sorted list. What is unique for the merge sort algorithm, according to Cormen et al. [2009], is that all the different instances of time complexity, as seen in equation 3, are equal. This means that the function does not care whether the list is sorted, reversed or random.

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

2.2.2 Quicksort

Quicksort is, according to Cormen et al. [2009], another algorithm that sorts in place. The key aspect to this algorithm is the Partition function, as seen in listing 4, which selects a pivot element from the array and rearranges it in place. The pivot element is then being used recursively using the quicksort algorithm by sorting two new subarrays. The algorithm continues to divide and sort the subarrays until the entire list is sorted. As with the merge sort algorithm, the best and average case scenarios for quicksort will have an expected run time complexity of $\Theta(n \cdot \lg n)$. According to Cormen et al. [2009], the worst case runtime for this algorithm, as seen in equation 4, occurs when the partitioning becomes unbalanced and creates one empty subarray and one subarray with $n - 1$ elements.

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

Listing 3 Merge sort algorithm from Cormen et al. [2009, Ch. 2.3].

```
MERGE(A, p, q, r)
1   n1 = q − p + 1
2   n2 = r − q
3   let L[1 . . . n1 + 1 and R[1 . . . n2 + 1 be new arrays
4   for i = 1 to n1
5       L[i] = A[p + i − 1]
6   for j = 1 to n2
7       R[j] = A[q + j]
8   L[n1 + 1] = ∞
9   R[n2 + 1 = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Listing 4 Quicksort algorithm from Cormen et al. [2009, Ch. 7.1].

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ 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 Combined algorithms

The concept behind hybrid algorithms is utilizing the strengths of different sorting algorithms on different problems in order, choosing the right algorithm in order to make the total time spent shorter. As we presented earlier in the theory section, both merge sort and quicksort are slower than insertion sort on smaller lists, but they both excel, though with different results, when the lists becomes larger. Examples on the theoretical advantages will be presented in the according subsections 2.3.1 and 2.3.2.

2.3.1 Merge sort switching to insertion sort for small data

As seen in listing 5, there is no new code for any sorting algorithm in the pseudocode. The threshold value represents the crossing of the two algorithms, with regards to the running time for sorting the given list, and will naturally vary based on the length of each test set. As the threshold value is reached, the algorithm switches over and continues sorting with the more effective algorithm. By implementing this hybrid, the aim is to take advantage of a faster running time for insertion sort for smaller lists, $\Omega(n)$, before switching over as insertion sort reaches its running time of $O(n^2)$ and merge sort operates in $\Theta(n \cdot \lg n)$ for all cases.

Listing 5 Merge sort switching to insertion sort

```
MERGE-INSERTION(A, p, r)
1  threshold = Crossing between Insertion and Merge
2  length = r − p
3  if length ≤ threshold
4        A = MERGE-SORT(A, p, r)
5  else
6        A = INSERTION-SORT(A)
```

2.3.2 Quicksort switching to insertion sort for small data

The implementation of the quicksort-insertion sort hybrid can be seen in listing 6. It follows the same principles and implementation as the hybrid in subsection 2.3.1. However, while the running times of insertion sort still are $O(n)$ (best case) and $O(n^2)$ (avg and worst case), quicksort operates at $O(n \cdot \lg n)$ (avg and best case) until it reaches $O(n^2)$ (worst case). The treshold value is the crossing between insertion sort and quicksort, where the actual value is based on the length of the list that is sorted.

Listing 6 Quicksort switching to insertion sort

```
QUICK-INSERTION(A, p, r)
1  threshold = Crossing between Insertion and Quick
2  length = r − p
3  if length ≤ threshold
4        A = QUICKSORT(A, p, r)
5  else
6        A = INSERTION-SORT(A)
```

2.4 Built-in sorting functions

2.4.1 Python 'sort()'

The integrated .sorted() function in Python uses timsort in its sorting process. According to **?** and **?**, timsort uses a combination of merge sort and insertion sort where, if the amount of numbers to sort exceeds a certain threshold, it switches from insertion sort to merge sort. The algorithm was created for Python in 2002 by Tim Peters as its own sorting algorithm. The running time of this complex implementation of timsort is, according to Auger et al. [2018], $O(n \cdot \lg n)$ or $O(n \cdot \lg \rho)$ where ρ is the number of runs explained as maximal monotonic sequences, whereas the best case is $\Omega(n)$.

2.4.2 NumPy 'sort()'

The built in NumPy.sort() in Python gives the user multiple choices as to which sorting algorithm can be used. According to Numpy-community [2021], the official documentation site for NumPy, the user can choose between quicksort, heapsort, merge sort, and timsort where the quicksort option is the default one. Depending on which algorithm the user chooses, there will also be different running times. The worst case scenario for quicksort is given in equation 5, while the worst case running time for the remaining three is given in equation 6. According to the same documentation, NumPy.sort() in Python switches from quicksort to heapsort if the quicksort algorithm does not make enough progress with sorting the list it is given, regardless of the nature of the list.

$$T(n) = O(n^2)\ . \tag{5}$$

$$T(n) = O(n \cdot \lg n)\ . \tag{6}$$

3 Methods

3.1 Test data

To explore and test the sorting algorithms presented in the "Theory" section, each algorithm ran on a sorted list (lowest number at leftmost position), a reversed list, and a random list. In other words, to look at how the different sorting algorithms performed, all sorted, random, and reversed lists were essential for testing the theoretical purposes.

The different test sets were generated with the random module in Python, along with a seed, to assure the possibility of an exact replication of the project, should that be wanted. All the test sets were generated with the same numbers, and accordingly then sorted, reversed or randomly put together before the tests commenced.

3.2 Measuring runtimes

Runtime was measured using the timeit.Timer module. Small datasets are more often influenced by variance caused by variance in the dataset composition, but also by external noise. This could be caused by other processes and conditions slowing down the execution. The impact of this was averaged out by sorting the same section of the dataset multiple times before averaging the time. Since the smaller datasets were more prone to variance, they needed more repetitions in order to aggregate a reliable result. We used the timeit.autorange module to allocate a time frame of 0.2 seconds and ran the sorting functions on the data set until the time restriction was broken. The total time expended was then recorded with the total number of repetitions. The average time of the measurements was then calculated by dividing the accumulated time on the number of executions.

With this approach, smaller datasets had the possibility to be repeated more in the allocated timeframe, while the time consuming process of repeating longer datasets was reduced. In order to obtain

accurate results, this process was repeated 7 times and the minimum, average, maximum, and variance in sorting time between the repetitions were recorded. We chose to mainly utilize the minimum value of the repetitions. Since the data tested stayed the same there was no variance in the dataset and any variance present was therefore caused by hardware performance. This made the shortest recorded time the most accurate result.

3.3 Execution of benchmarks

Some of the of the algorithms were, as previously mentioned, already implemented. Sorted() is native to Python and NumPy.sort() is from the NumPy library. All of the other algorithms were adapted from the pseudocodes specified earlier in the paper and implemented using Python.

3.4 Hardware and relevant files

The hardware specifications and software versions for the testing are presented in table 1 and 2 along with git hashes from our GitLab repository in table 3.

Table 1: Hardware specifications

Machine	AMD64
Processor	Intel64 Family 6 Model 142 Stepping 12, Genuin...
Physical cores	4
Virtual cores	8
GB Ram	(8,)

Table 2: Software versions

System	Windows-10-10.0.19041-SP0
Release	10
Python_version	3.8.8
Python_build	Apr 13 2021 15:08:03
Python_compiler	MSC v.1916 64 bit (AMD64)
Python_implementation	CPython
Numpy_version	1.20.1

Table 3: Versions of files used for this report; GitLab repository https://gitlab.com/nmbu.no/emner/inf221/INF221_2021/student-term-papers_21/group3112/inf221-term-paper-group39/.

File	Git hash
`bubble_sort.py`	`ef7939aa`
`insertion_sort.py`	`5420c844`
`merge_sort.py`	`9392f16a`
`quick_sort.py`	`5c1b42ce`
`threshold_sorts.py`	`1676d840`
`main.py`	`1fc86741`
`test.py`	`1fc86741`

4 Results

This section is divided into the same subsections as the "Theory" section with the results presented using the same order as the theory was explained. In order to get a data distribution that is weighted towards shorter datasets, the length of elements sorted in each test iteration is scaled by the power of two. Since they do not increase linearly, but exponentially, both the x-axis data points and the y-axis results will be skewed on a linear graph. To increase readability we have chosen to present the results using logarithmic scale along both axis. A logarithmic representation spaces the data points out better and increases the readability of differences between the resulting graph, somewhat at the cost of intuitively displaying the shape of the graphs. For reference this warping is applied graphically to commonly used growth rates in figure 1.

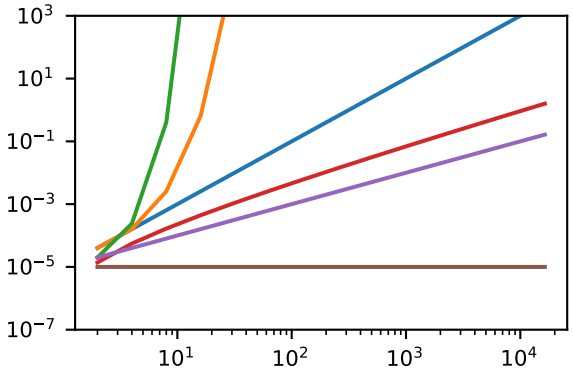

Figure 1: Commonly referenced growth rates (in seconds) with 10^{-5} (brown), n (purple), $n \cdot log(n)$ (red), n^2 (blue), 2^n (orange), and $n!$ (green)

4.1 Quadratic algorithms

The quadratic algorithms subsection covers subsections 4.1.1 and 4.1.2.

4.1.1 Insertion sort

Our testing indicates that insertion sort, as seen in figure 2, is best in its class of quadratic algorithms, being consistently faster than bubble sort. As seen in figure 2, the average and worst case scenarios follows each other closely. The random list averages out in slightly less time than the worst case scenario.

4.1.2 Bubble sort

The performance of bubble sort is presented in figure 3. Bubble sort performs similarly to insertion sort, however it is consistently the slower option. Both are faster on sorted data, however the difference for bubble sort is less stark. There is also a negligible difference between the reversed and the random dataset until n reaches values greater than 10^4. The reversed dataset is consistently the slowest until the larges value of n, where it is faster than the random list.

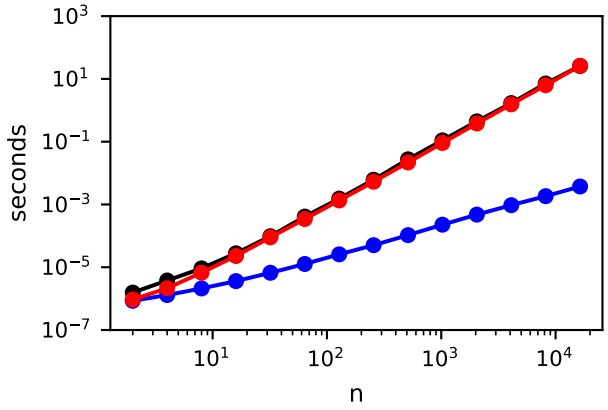

Figure 2: Benchmark results for the insertion sort algorithm (in seconds) with sorted input (blue), reversed input (red) and random input (black).

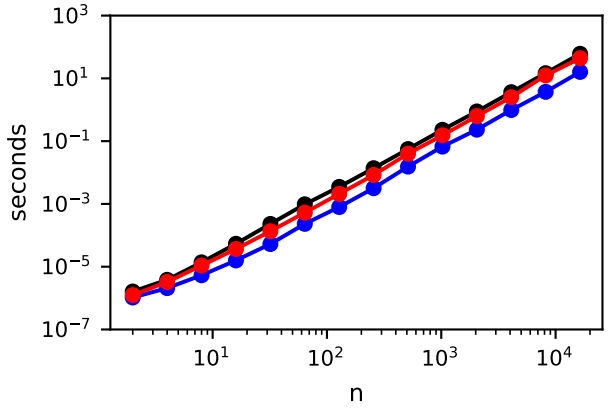

Figure 3: Benchmark results for the bubble sort algorithm (in seconds) with sorted input (blue), reversed input (red) and random input (black).

4.2 Sub-quadratic algorithms

The results of merge sort and quicksort are presented in the following subsections 4.2.1 and 4.2.2. Due to limits on recursion depth and the implementation of the algorithm, quicksort could only run up to 2^{11} numbers, relative to 2^{14} numbers that every algorithm, except for the quick-insertion hybrid along with quicksort, ran on. Any attempt to run larger lists (after the previous ones) resulted in a max recursion depth error that, according to ?, is of the value of 1000 calls. This means that Python automatically stops the current operation if the algorithm calls itself more than 1000 times. Therefore, there are differences in the figures 4 and 5, such as the number of elements sorted.

4.2.1 Merge sort

The results for the merge sort algorithm, as seen in figure 4, shows that there, for the sorted, random, and reversed lists, are minimal variance in runtimes for all values of n.

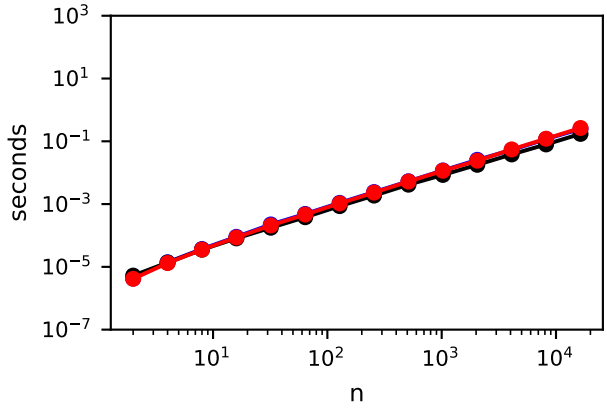

Figure 4: Benchmark results for the merge sort algorithm (in seconds) with sorted input (blue), reversed input (red) and random input (black).

4.2.2 Quicksort

The benchmark results for the quicksort algorithm can be seen in figure 5. Quicksort sorts the random list slower for values where $n < 10^1$ and faster for all values where $n > 10^1$. The sorted list and the reversed list have identical run times for all sizes of n.

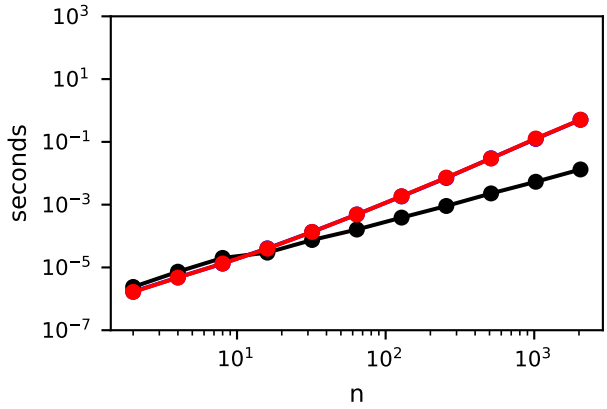

Figure 5: Benchmark results for the quicksort algorithm (in seconds) with sorted input (blue), reversed input (red) and random input (black).

4.3 Combined algorithms

Both implementations of the hybrid solutions, shown below in subsections 4.3.1 and 4.3.2, shows a distinct upgrade in run times for smaller values of n, compared to their respective competition in the original algorithms merge sort and quicksort.

4.3.1 Merge-Insertion

The sorting times for the hybrid merge-insertion, where insertion sort switches to merge sort at a certain threshold, can be seen in figure 6. The hybrid sorts the sorted list faster for sizes where $n < 10^{2.3}$ before it hits the threshold value and switches over to merge sort. There is a clear improvement in sorting time for the sorted list for values where $n < 10^{2.3}$. From that point on and for all sizes of $n > 10^{2.3}$ the hybrid shares the same pattern as the original merge sort. There is a minor improvement for both the reversed and the random list for values where $n < 10^{2.1}$, indicating that the threshold value, as expected, kicked in around that area.

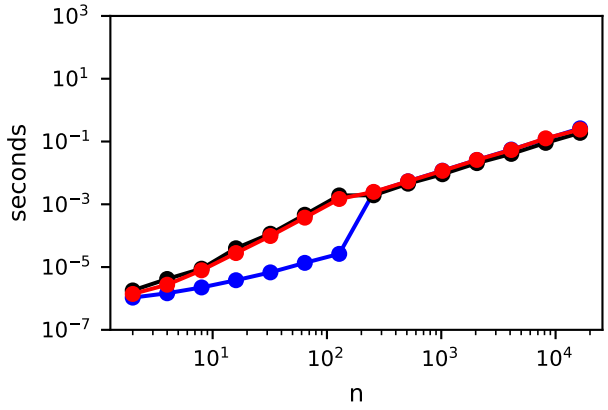

Figure 6: Benchmark results for the merge-insertion algorithm (in seconds) with sorted input (blue), reversed input (red) and random input (black).

4.3.2 Quick-Insertion

The sorting times for the hybrid quick-insertion, where insertion sort switches to quicksort at a certain threshold, can be seen in figure 7. The sorted and reversed lists both share the same run time for all sizes of n and are both sorted quicker than the random list when $n < 10^{1.1}$. For those same sizes, the random list is sorted quicker than with the original quicksort algorithm seen in subsection 4.2.2, indicating that the threshold value is set somewhere in that area. Where $n > 10^{1.1}$ all three lists share an identical pattern with the plot in figure 5, showing the development in run times for quicksort.

4.4 Built-in algorithms

4.4.1 NumPy sort

The run times the integrated module NumPy.sort performed at, can be seen in figure 8. For every instance of n, both the reversed and sorted list had identical run times. The random list had a marginally better sorting time that increased very little until the size of $10^{2.5}$ elements. After that point, the random list had a runtime that increased with a higher growth than both the reversed and sorted lists, resulting in a tiny runtime margin in favor of the random list for the final size n that was tested.

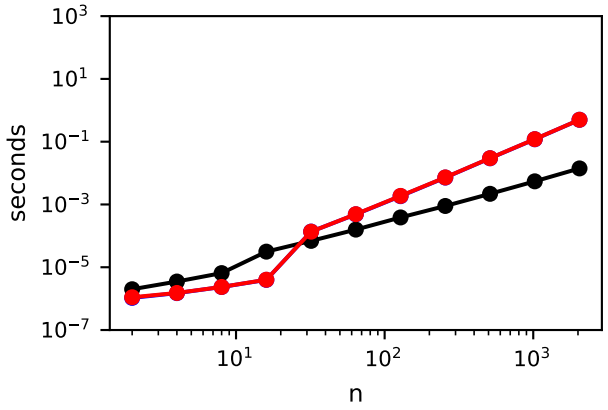

Figure 7: Benchmark results for the quick-insertion algorithm (in seconds) with sorted input (blue), reversed input (red) and random input (black).

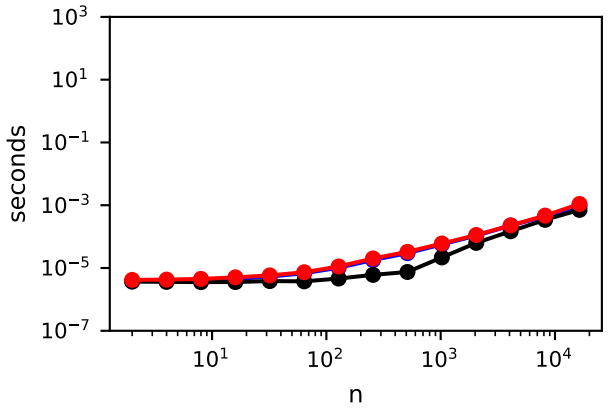

Figure 8: Benchmark results for NumPy sort (in seconds) with sorted input (blue), reversed input (red) and random input (black).

4.4.2 Python sort

The results for Python sort can be seen in figure 9. From the results, it is clear that the integrated Python sort function has the longest sorting time for the random list, regardless of the size of n. The reversed list and the sorted list have equal runtimes for sizes of $n < 10^2$. When n exceeds this size, Python sort handles the sorted list better than the reversed list and, naturally, ends up with the best sorting time for the sorted list at sizes where $n > 10^2$.

4.5 Comparisons

Now that the results have been presented group by group, we will look at the bigger picture and present the combined results to see which algorithm performed the best. To make the results easier to comprehend, each plot represents the sorting time for a specific list, whether it is random, sorted, or reversed. We have also chosen to exclude merge sort, merge-insertion and Python sort from the

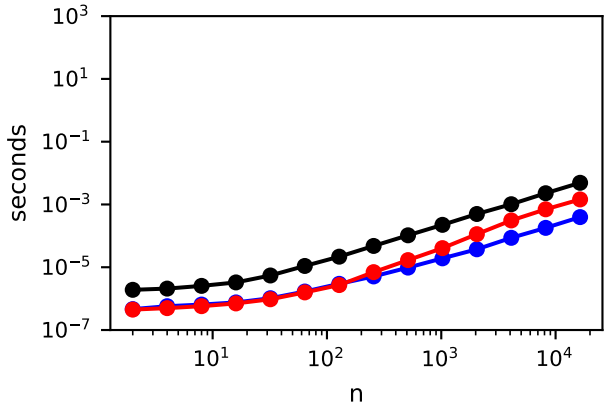

Figure 9: Benchmark results for Python sort (in seconds) with sorted input (blue), reversed input (red) and random input (black).

comparison plots in order to make the plots readable.

First out is the sorted list run times presented in figure 10. When comparing the sorted lists for the respective sorting algorithms, it is found that as the lists get larger, the insertion and NumPy sort are the most effective algorithms. Bubble sort, quicksort and quick-insertion are the least effective when sorting the larger lists and are found to follow the same trend.

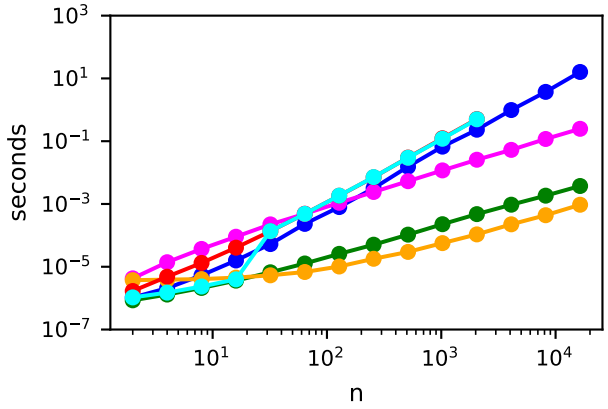

Figure 10: Comparison of the performance in sorting time (in seconds) for insertion sort (green), bubble sort (blue), merge sort(pink) quicksort (red), quick-insertion (cyan) and NumPy sort (orange) on sorted lists.

Next is the reversed list run times presented in figure 11. As with the sorted lists, the NumPy sort is the overall most effective sorting algorithm when sorting reversed lists. The insertion sort is the next most effective sorting algorithm until the number of elements to be sorted passes 103 elements. As with the sorted lists, the least effective sorting algorithms for the reversed lists are the bubble sort, quicksort and quick-insertion, in addition to insertion sort.

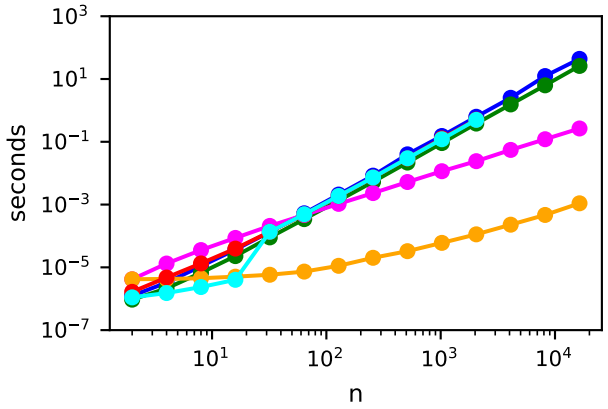

Figure 11: Comparison of the sorting time (in seconds) for insertion sort (green), bubble sort (blue), merge sort(pink) quicksort (red), quick-insertion (cyan) and NumPy sort (orange) on reversed lists.

Last is the random list run times presented in figure 12. As the lists get larger, the most effective sorting algorithm is the NumPy.sort and the least effective algorithm is the bubble sort. For small lists with less than 10 elements the insertion sort, along with the hybrids, perform the best.

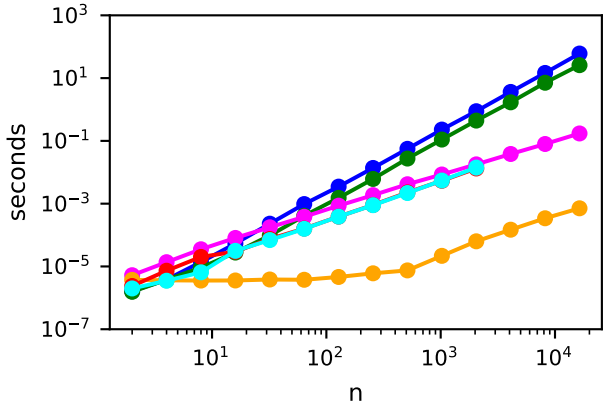

Figure 12: Comparison of the performance in sorting time (in seconds) for insertion sort (green), bubble sort (blue), merge sort(pink) quicksort (red), quick-insertion (cyan) and NumPy sort (orange) on random lists.

5 Discussion

5.1 Quadratic algorithms

As presented in the "Results" section, insertion sort is consistently faster than bubble sort, especially at sorting sorted data. Given that both algorithms have identical run time complexities for all the different instances, this result seems a little odd. Both are in-place sorting algorithms, so why is there such a difference in running time? This has to do with how the two algorithms are implemented. Bubble sort will continually compare the current number to its neighbors and make a swap based on which is greater. For each iteration, the algorithm knows that the last number in the list is sorted, and it reduces the size of the list by one. Here you can implement an early exit mechanism, which dictates that if no swaps were done the algorithm terminates. This has, however, not been done, so with our implementation every single iteration had to be done. Insertion sort, with its for-loop and while statement, iterates forward, picking up new values before moving that value backwards into the sorted list until it finds a lower value. Thus, if there are no higher values sorted already it will not be moved at all and the algorithm simply terminates when it reaches the end of the data. The main advantage in this comparison comes from the fact that insertion sort has a built in fast termination, while bubble sort does not.

Compared to other algorithms the quadratic algorithms perform well on shorter datasets and sorted data. In its current state, insertion sort could be an attractive option for sorting shorter datasets and data that has a nearly sorted initial order.

5.2 Sub-quadratic algorithms

The results for the sub-quadratic algorithms are following the theoretical standpoint presented in the "Theory" section. Merge sort performs better in total, even though the sorting time for the random list is slower than quicksort. This may be caused by an even number of elements in the lists, hindering quick sort from reaching its worst-case scenario where the running time no longer is $O(n \cdot lgn)$, but $O(n^2)$, as the partitioning retains its balance.

There are, however, lots of different implementations in order to improve quicksort to boost its performance. As a divide-and-conquer algorithm, quicksort depends on partitioning. This is an element in the sorting that can be improved by, for instance, dividing the sub parts of the list into not two, but three, parts. By doing so, the number of divisions are decreased, and depending on the length of the lists, to a larger extent. Another way to boost the performance is to handle repeated elements better. In most cases, the lists up for sorting contain one or more elements that are repeated. If quicksort uses its experience from a local memory versus running the same identical part multiple times (one for each duplicate), it only has to experience it once and can use that one experience on the other cases. A third way to improve quicksort is by making it a hybrid with insertion sort. With insertion sort being faster for smaller lists, the idea is to take advantage of the faster sorting time for insertion sort before switching over to quicksort when insertion sort slows down. This implementation has been done in this project and can be seen in subsection 2.3.2.

5.3 Combined algorithms

Despite the initial overhead of choosing an algorithm, the results for the combined algorithm approach show a significant improvement in overall runtime, leveraging the benefits of insertion sort on shorter datasets, while maintaining consistency of quicksort/merge sort on longer datasets. With insertion sort as our initial algorithm, we calculated the optimal threshold as the first datapoint where the alternative algorithm was faster than insertion sort on a randomized dataset, using that value for all three lists. This implementation is the most sensible for testing on blind data, however as previously mentioned, there may be fringe cases where it is sensible to weight the algorithm performance differently, such as when leveraging insertion sorts speed at nearly sorted or sorted data.

5.4 Built-in algorithms

While the quadratic algorithms are the fastest for datasets up to 4 elements long, from the results of the built-in algorithms, we can see that NumPy sort is the fastest algorithm among the once tested, when applied to random data across all other ranges. Python sort is a close second, but cannot reach the sorting speed of NumPy sort. However, when applied to sorted and reverse-sorted Python sort is significantly faster at reaching fast sorting speeds relative to its competition, outperforming NumPy sort. They are clearly different algorithms with python consistently achieving significantly higher efficacy on both sorted and reverse-sorted across all dataset lengths tested. This is possibly because NumPy is a scientific maths package and has been optimised in order to sort mathematical random data, while Pythons native package is optimised to leverage instances of non-randomness in the more naturally produced disorganised data that is presented to it in most if its cases of usage.

5.5 Summary

There were no surprises performance wise in this study. All algorithms prove to follow their theoretical runtime complexities, with the integrated NumPy sort and Python sort coming out on top as the quicker sorting algorithms. Bubble sort is the slowest algorithm overall despite having the same time complexity for its average and worst case as insertion sort.

Acknowledgments

We are grateful to PhD candidate Bao Ngoc Huynh for her suggestions, comments, and guidance.

References

Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. 2018. On the Worst-Case Complexity of TimSort. In *26th Annual European Symposium on Algorithms (ESA 2018) (LIPICS, Vol. 112)*. Helsinki, Finland, 4:1–4:13. https://doi.org/10.4230/LIPIcs.ESA.2018.4

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 Numpy-community. 2021. *numpy.sort.* https://numpy.org/doc/stable/reference/generated/numpy.sort.html

Python Software Foundation. 2021. *timer.Timeit.* https://docs.python.org/3/library/timeit.html