NTI Gymnasiet
Teknikprogrammet
Gymnasiearbete 100 p
HT 2023-VT 2024

Sorting algorithms

Comparison of sorting algorithms' performance

Axel Thornberg
Handledare: Filip Vallin
Examinator: Beatrice Larsson

Abstract

Sorting algorithms are an important piece of any decently sized modern digital system and are easily bottlenecks due to the sheer complexity of their task. Therefore it is of great importance that knowing which algorithms are the best in certain situations is crucial for creating efficient software. Therefore this paper attempts to answer the question “How does the efficiency of sorting algorithms compare when subjected to different sizes of data with different standard deviations of a Gaussian distribution?” in a concrete and precise manner to give a good source of information regarding algorithms and their applicability in different scenarios. The survey will be conducted through a comprehensive Python program that will repeatedly test different algorithms. To do this it will use 5 libraries (random, pickle, time, os and plotly) as well as high-quality hardware (i9-12900KF and 32 GB of 4400 MHz RAM). The results of the survey showed that Quicksort by far had the fastest mean time when compared to the other algorithms. Still, it is not always the best as the other’s proved to be more consistent and a hybrid algorithm based on Quicksort will most likely outperform it.

Table of contents

1. Introduction

Sorting is not as easy as most people believe it to be. When manually sorting ten elements it does not matter much what way you do it as you get the same results. But when done with datasets with orders of magnitude larger, those methods quickly fall apart. For a computer, the time it takes to sort said dataset with an efficient algorithm and an inefficient algorithm will take milliseconds respectively minutes. Certain algorithms can be quite easy to filter out as slower or faster than others, but when it comes to comparing the fastest algorithms with each other there is no clear indication of which is faster. Here, the only method to find the faster algorithm is to test and compare them to each other.

2. Purpose and question

The purpose of this survey is to find out how different sorting algorithms compare to each other. This information is somewhat hard to find due to the nature of the question as the efficiency of each algorithm naturally varies with the input data. This survey will address this by using different sets of data to produce comprehensive data for which algorithm is best in different scenarios. How does the efficiency of sorting algorithms compare when subjected to different sizes of data with different standard deviations of a Gaussian distribution?

3. Theory

Algorithm: An algorithm is a sequence of instructions, typically used to solve problems or to perform a computation.[1]

Sorting algorithm: A sorting algorithm is an algorithm that orders the elements of a list.[2]

Big O notation: Big O notation is a notation that shows the correlation between input data and the growth rate of the run time.[10]

Time complexity: Time complexity describes the number of elemental operations an algorithm will compute. It is measured in Big O notation.[3]

Function: A function is a sequence of program instructions that performs a specific task, packaged as a unit.[4]

Recursive function: A function which calls upon itself from within its code.[5]

Gaussian distribution: Gaussian distribution is a type of continuous probability distribution for a real-valued random variable.[6]

Comparison sort: A comparison sort is a sorting algorithm that sorts a list by comparing the elements of a list with other elements.[7]

Computational overhead: Computational overhead represents an amount of time unrelated to the task itself but necessary to run it.[8]

Hybrid algorithm: An algorithm that uses different algorithms depending on the data therefore being able to use the optimal algorithm depending on the circumstances but losing performance to overhead.[9] These algorithms can be some of the fastest available.[2]

Quicksort: Quicksort is a recursive comparison sorting algorithm that selects a pivot before iterating through the array and recursively sorting all values smaller or greater than the selected pivot. It has an average time complexity of O(n log n).[11]

Mergesort: Mergesort is a recursive comparison sorting algorithm that recursively splits the array in two until it reaches a length of 1. Then it recursively merges the list while comparing and moving the values to give a sorted list. It has an average time complexity of O(n log n).[12]

Heapsort: Heapsort is a comparison sorting algorithm that splits the array into a sorted and unsorted region. Then it extracts the largest value of the unsorted region and inserts it into the correct position of the sorted region. It has an average time complexity of O(n log n).[13]

Insertion sort: Insertion sort is a comparison sorting algorithm that iterates through the array and moves each item to the correct position. It has an average time complexity of $O(n^2)$.[14]

Timsort: Timsort is a comparison sorting algorithm that creates small sets of sorted data called *runs*. Considering the smaller size of these, insertion sort is fast enough even considering its worse time complexity. After that, it merges these runs in the same manner as Mergesort does.[15]

Uniform distribution: Uniform distribution is a probability distribution where each value has an equal probability.[16]

All sorting algorithms are affected by the input data. Therefore, most concrete studies have mainly focused on their time complexity, which is the same for an algorithm no matter the input.[2] Heapsort, Mergesort, and Quicksort are listed as the most efficient algorithms in different situations.[2] All of the tested algorithms are comparison sorts meaning that they are all structurally similar and that Python as a programming language will not benefit either in terms of performance.

1. Wikipedia. 2023. "Algorithm." Wikimedia Foundation. Last modified September 24, 2023. https://en.wikipedia.org/wiki/Algorithm.
2. Wikipedia. 2023. "Sorting Algorithm." Wikimedia Foundation. Last modified September 26, 2023. https://en.wikipedia.org/wiki/Sorting_algorithm.
3. Wikipedia. 2023. "Time Complexity." Wikimedia Foundation. Last modified September 22, 2023. https://en.wikipedia.org/wiki/Time_complexity.
4. Wikipedia. 2023. "Function (Computer Programming)." Wikimedia Foundation. Last modified September 27, 2023. https://en.wikipedia.org/wiki/Function_(computer_programming).
5. Wikipedia. 2023. "Recursion (Computer Science)." Wikimedia Foundation. Last modified August 21, 2023. https://en.wikipedia.org/wiki/Recursion_(computer_science).
6. Wikipedia. 2023. "Normal Distribution." Wikimedia Foundation. Last modified September 27, 2023. https://en.wikipedia.org/wiki/Normal_distribution.
7. Wikipedia. 2023. "Comparison Sort." Wikimedia Foundation. Last modified July 31, 2023. https://en.wikipedia.org/wiki/Comparison_sort.
8. Wikipedia. 2023. "Overhead (Computing)." Wikimedia Foundation. Last modified May 21, 2023. https://en.wikipedia.org/wiki/Overhead_(computing).
9. Wikipedia. 2023. "Hybrid Algorithm." Wikimedia Foundation. Last modified February 3, 2023. https://en.wikipedia.org/wiki/Hybrid_algorithm.
10. Wikipedia. 2024. "Big O notation." Wikimedia Foundation. Last modified February 4, 2024. https://en.wikipedia.org/wiki/Big_O_notation.
11. Wikipedia. 2024. "Quicksort." Wikimedia Foundation. Last modified January 3, 2024. https://en.wikipedia.org/wiki/Quicksort.
12. Wikipedia. 2024. "Merge Sort." Wikimedia Foundation. Last modified January 27, 2024. https://en.wikipedia.org/wiki/Merge_sort.
13. Wikipedia. 2024. "Heapsort." Wikimedia Foundation. Last modified February 12, 2024. https://en.wikipedia.org/wiki/Heapsort.
14. Wikipedia. 2023. "Insertion Sort." Wikimedia Foundation. Last modified December 29, 2023. https://en.wikipedia.org/wiki/Insertion_sort.
15. Wikipedia. 2024. "Timsort." Wikimedia Foundation. Last modified February 13, 2024. https://en.wikipedia.org/wiki/Timsort.

16. Wikipedia. 2024. "Continuous Uniform Distribution." Wikimedia Foundation. Last modified January 26, 2024. https://en.wikipedia.org/wiki/Continuous_uniform_distribution.

4. Method and material

4.1. Method

This project consists of three Python programs. The first contains the algorithms as efficient Python functions that take input in the form of an unsorted array and then return the sorted permutation. These algorithms are used in the second program.

The second generates the data necessary for this project. It creates 9 different random normal distributed arrays using the Gauss function in the random library. These are a combination of three different sizes and three different densities of standard deviation. The lists as mentioned earlier are tested against four different sorting algorithms. Using the time library, the time is measured. This yields 36 different times, all saved to an external file using the pickle library. The program then loops this 25,000 times over.

The third program takes the data from the second and processes it into readable and useful formats. Some of these are the mean time in each category as well as the overall mean time for each algorithm. Other formats are graphs to show the distribution of times.

4.2. Material

This program has been written in Python (3.11.6) alongside 5 libraries. These are random, pickle, time, os and plotly. The relevant hardware is an i9-12900KF and 32 GB of 4400 MHz RAM.

5. Results

The survey yielded that Quicksort was the fastest algorithm for all categories. It had a mean time of 23.051756841333308 milliseconds to sort an array. Mergesort was the second fastest across all categories taking a mean of 46.04735194177758 milliseconds to sort an array. Heapsort had the third fastest mean time of 66.34182150711104 milliseconds. Timsort was the slowest at a mean time of 67.54026761244455 milliseconds. Timsort was faster than Heapsort in 4 categories while Heapsort was faster than Timsort in 5 categories. Timsort was generally faster than Heapsort when it came to smaller arrays while Heapsort was faster for larger arrays. The distribution of times is available in a series of graphs near the end of the paper. These graphs showed that although Quicksort was the fastest algorithm it was less consistent in the times with a larger deviation than other algorithms.

Quicksort	1 000	10 000	100 000
0.1	0.5115081320000007	7.095698688000023	86.94529332400013
0.01	0.2867567160000012	4.655413507999978	64.56828979199977
0.001	0.12861147999999917	2.5994345640000054	40.67480536799985

Table 1: Quicksort's mean time to sort for different data sizes and deviations

Mergesort	1 000	10 000	100 000
0.1	0.834567356000006	11.253142395999983	129.0017180359995
0.01	0.8418935760000105	10.862672968000004	126.90739609199957
0.001	0.8310639239999923	10.615920836000026	123.27779229199915

Table 2: Mergesort's mean time to sort for different data sizes and deviations

Heapsort	1 000	10 000	100 000
0.1	0.9804551359999946	14.634866380000087	186.42501203199836ö
0.01	0.9979233439999945	13.847970408000087	184.75011486800096
0.001	0.833101831999997	14.040838683999912	180.56611087999985

Table 3: Heapsort's mean time to sort for different data sizes and deviations

Timsort	1 000	10 000	100 000
0.1	0.9692688480000061	14.552908247999987	191.6550402720008
0.01	0.9542062319999943	14.023030819999951	188.94938102800057
0.001	0.8773404840000044	13.911564695999983	181.96966788399968

Table 4: Timsort's mean time to sort for different data sizes and deviations

6. Discussion

This survey is by no means free from biases and errors. The fact that Quicksort is the fastest across all of my categories does not by any means mean that it is the fastest in all situations. The Quicksort algorithm was written first and therefore had more time than the others to be optimized. I do on the other hand not believe that to be the sole reason for Quicksort's good performance. Quicksort is generally considered to be one of the fastest sorting algorithms as stated by Wikipedia. I also believe the datasets to be in favour of Quicksort as it was made to take advantage of duplicate instances of items. This is of course not wrong as there are plenty of real-world scenarios where the majority of values are duplicates such as a leaderboard. There are on the other hand also scenarios where there would be no duplicates such as a database with user IDs.

The fact that there was no significant difference between the sizes used is also no surprise considering they all have a time complexity of O(n log n). This combined with the fact that the sorting of larger lists took up most of the time I believe that any future survey regarding this topic should not use array sizes beyond 1,000 considering that there was no notable difference. Considering that the time complexity and big O notation only measure the largest factor and not the smaller factors, such as constants and smaller exponents, which will become less negligible as the size of the data set decreases. Therefore, there may be a significant difference between the algorithms when the data set is small enough, This was partly shown in the survey for Timsort and Heapsort as Heapsort was faster for larger arrays and Timsort faster for smaller arrays. Therefore, I recommend any future survey in this field to survey arrays as small as 10 items. The smaller arrays would free up much time that would be better used in different sets of data, These could include but not be limited to other standard deviations of a Gaussian distribution, a shuffled range of numbers representing instances where no duplicates are present, a uniform distribution of numbers representing a set of data with duplicates each of which with an equal chance of occurring, and any other distribution or set of data with practical real-world equivalents.

As shown in the graphs, Quicksort may be the fastest overall but this survey revealed that it was less consistent in its time when compared to the other algorithms. This is because the other algorithms have a worst time complexity of O(n log n) while Quicksort has a worst time

complexity of $O(n^2)$ which makes it a lot slower in some circumstances. This does limit the applicability of Quicksort as some fields require a consistent time rather than a fast average time. Quicksort will still be useful in applications such as computer science, but when sorted information needs to be presented to a user the other algorithms may be a better choice due to the lower chance of it taking too long.

Naturally, Quicksort will rarely be the best sorting algorithm, and a hybrid algorithm will almost always be the fastest. Still, Quicksort will probably be best suited as a base for said hybrid algorithms. Those algorithms can then be fine-tuned and experimented on for individual applications to get the best performance for the relevant data.

7. Conclusion

Therefore, it is apparent that Quicksort is generally the most applicable sorting algorithm for most purposes as it has a mean time that is much faster than any other algorithm although these other algorithms might be better in certain circumstances where there are different requirements.

List of sources

Wikipedia. 2023. "Algorithm." Wikimedia Foundation. Last modified September 24, 2023. https://en.wikipedia.org/wiki/Algorithm.

Wikipedia. 2024. "Big O notation." Wikimedia Foundation. Last modified February 4, 2024. https://en.wikipedia.org/wiki/Big_O_notation.

Wikipedia. 2023. "Comparison Sort." Wikimedia Foundation. Last modified July 31, 2023. https://en.wikipedia.org/wiki/Comparison_sort.

Wikipedia. 2024. "Continuous Uniform Distribution." Wikimedia Foundation. Last modified January 26, 2024. https://en.wikipedia.org/wiki/Continuous_uniform_distribution.

Wikipedia. 2023. "Function (Computer Programming)." Wikimedia Foundation. Last modified September 27, 2023. https://en.wikipedia.org/wiki/Function_(computer_programming).

Wikipedia. 2024. "Heapsort." Wikimedia Foundation. Last modified February 12, 2024. https://en.wikipedia.org/wiki/Heapsort.

Wikipedia. 2023. "Hybrid Algorithm." Wikimedia Foundation. Last modified February 3, 2023. https://en.wikipedia.org/wiki/Hybrid_algorithm.

Wikipedia. 2023. "Insertion Sort." Wikimedia Foundation. Last modified December 29, 2023. https://en.wikipedia.org/wiki/Insertion_sort.

Wikipedia. 2024. "Merge Sort." Wikimedia Foundation. Last modified January 27, 2024. https://en.wikipedia.org/wiki/Merge_sort.

Wikipedia. 2023. "Normal Distribution." Wikimedia Foundation. Last modified September 27, 2023. https://en.wikipedia.org/wiki/Normal_distribution.

Wikipedia. 2023. "Overhead (Computing)." Wikimedia Foundation. Last modified May 21, 2023. https://en.wikipedia.org/wiki/Overhead_(computing).

Wikipedia. 2024. "Quicksort." Wikimedia Foundation. Last modified January 3, 2024. https://en.wikipedia.org/wiki/Quicksort.

Wikipedia. 2023. "Recursion (Computer Science)." Wikimedia Foundation. Last modified August 21, 2023. https://en.wikipedia.org/wiki/Recursion_(computer_science).

Wikipedia. 2023. "Sorting Algorithm." Wikimedia Foundation. Last modified September 26, 2023. https://en.wikipedia.org/wiki/Sorting_algorithm.

Wikipedia. 2023. "Time Complexity." Wikimedia Foundation. Last modified September 22, 2023. https://en.wikipedia.org/wiki/Time_complexity.

Wikipedia. 2024. "Timsort." Wikimedia Foundation. Last modified February 13, 2024. https://en.wikipedia.org/wiki/Timsort.

Attachments

Link to GitHub Repository: https://github.com/axelNTI/Gymnasiearbete

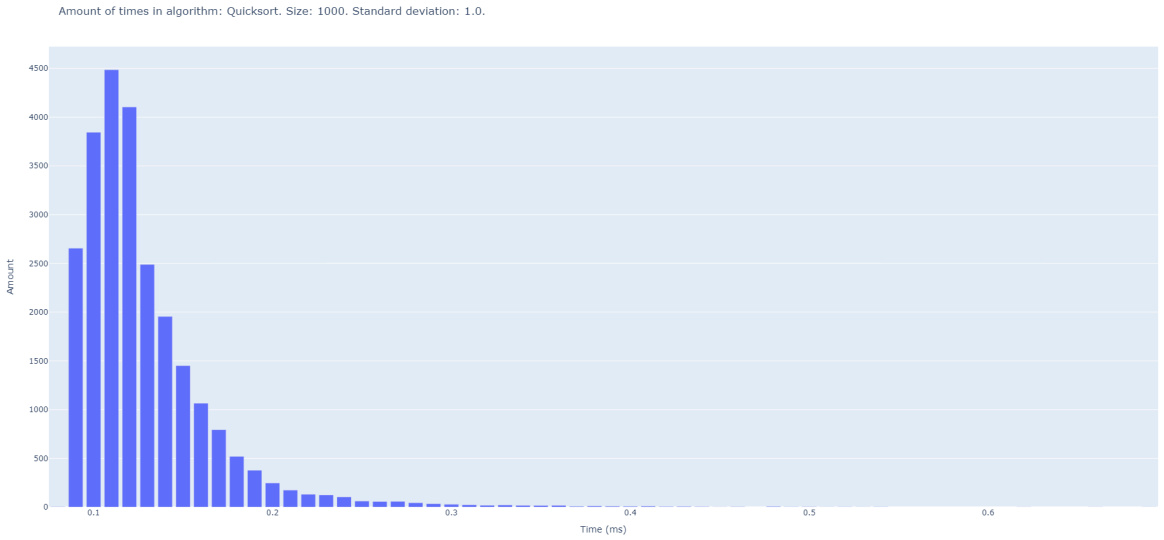

Chart 1: Quicksort's distribution of times sorting 1,000 elements with a standard deviation of 1.

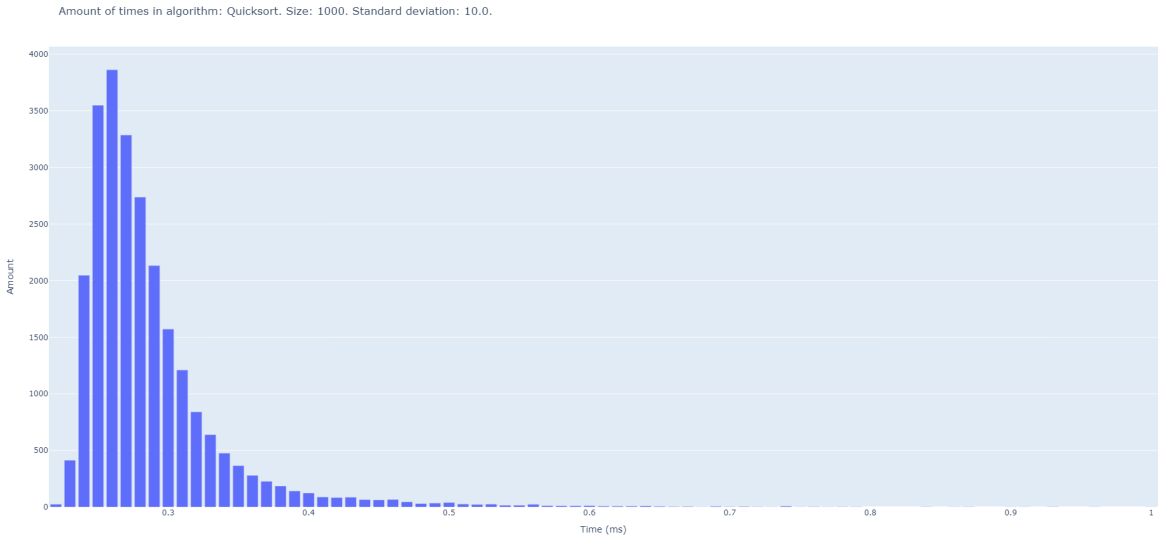

Chart 2: Quicksort's distribution of times sorting 1,000 elements with a standard deviation of 10.

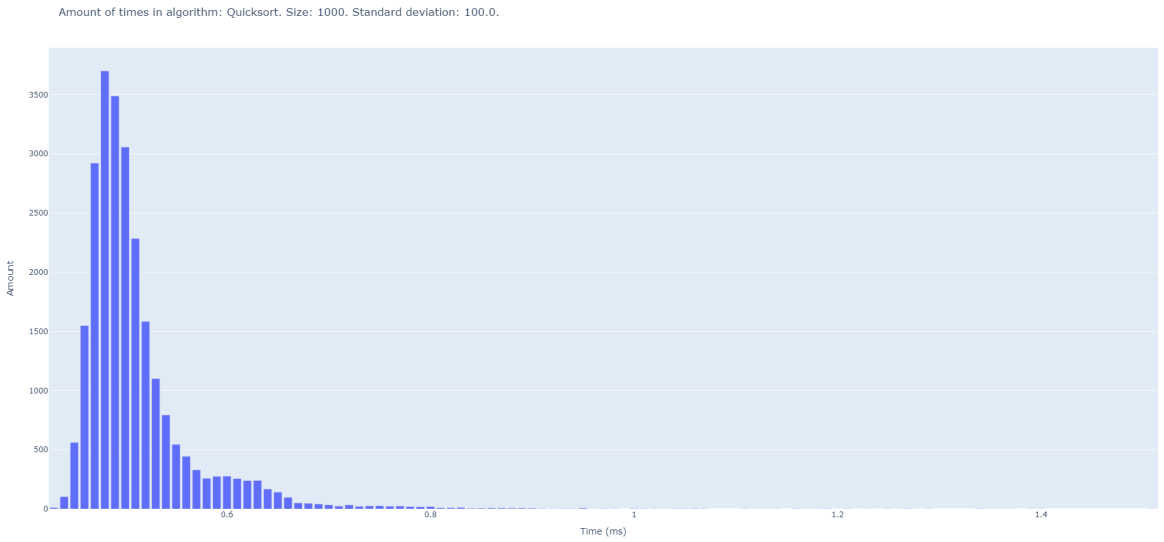

Chart 3: Quicksort's distribution of times sorting 1,000 elements with a standard deviation of 100.

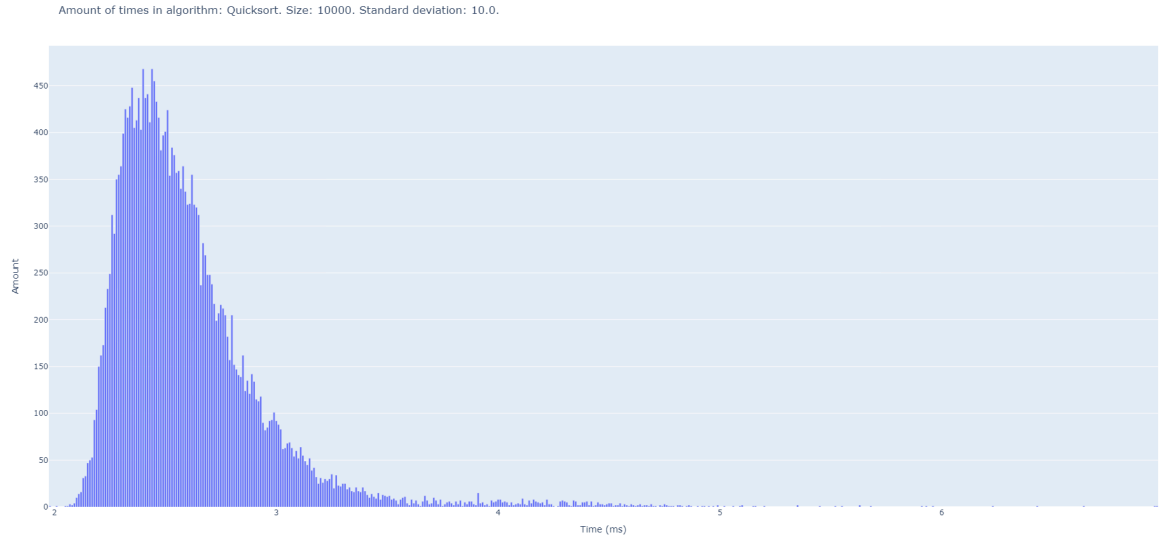

Chart 4: Quicksort's distribution of times sorting 10,000 elements with a standard deviation of 10.

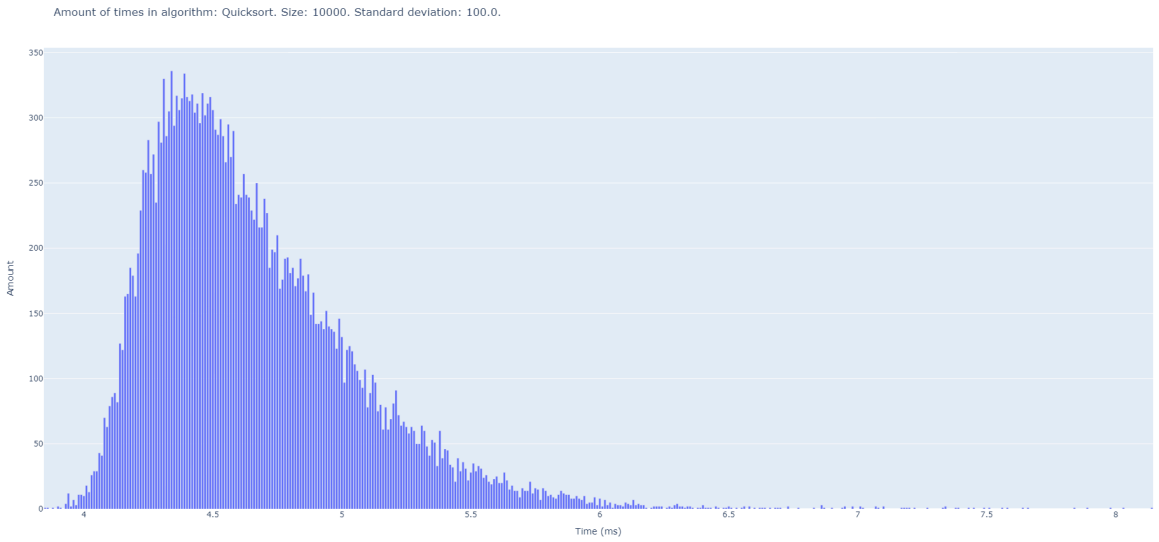

Chart 5: Quicksort's distribution of times sorting 10,000 elements with a standard deviation of 100.

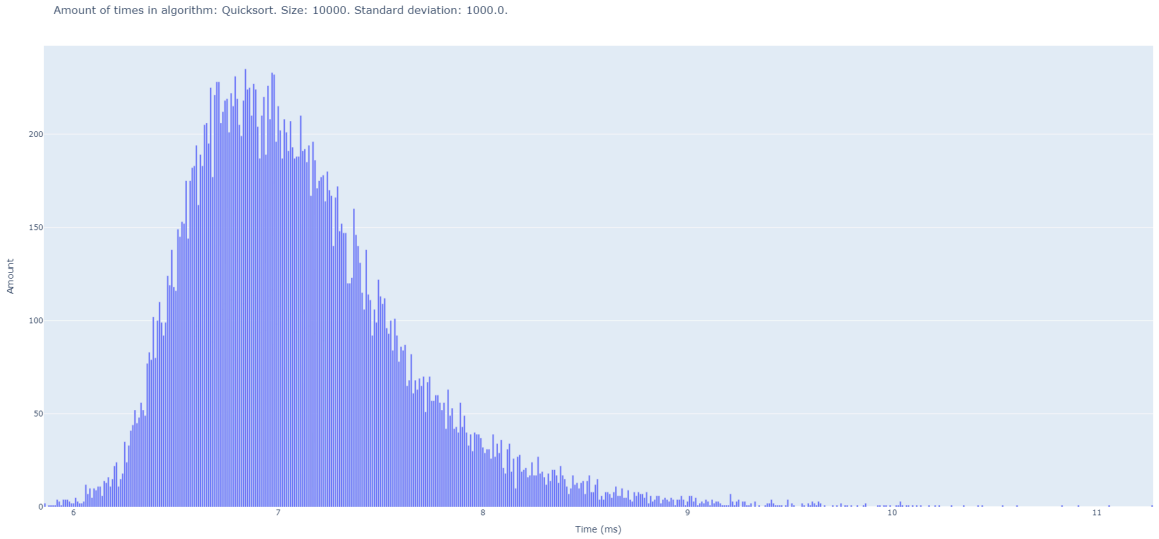

Chart 6: Quicksort's distribution of times sorting 10,000 elements with a standard deviation of 1,000.

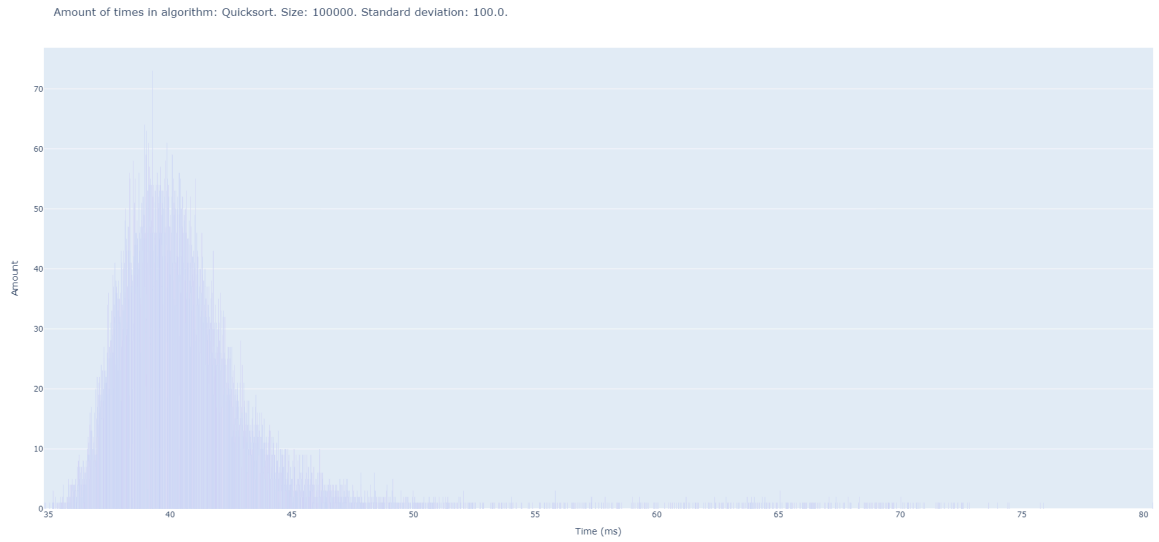

Chart 7: Quicksort's distribution of times sorting 100,000 elements with a standard deviation of 100.

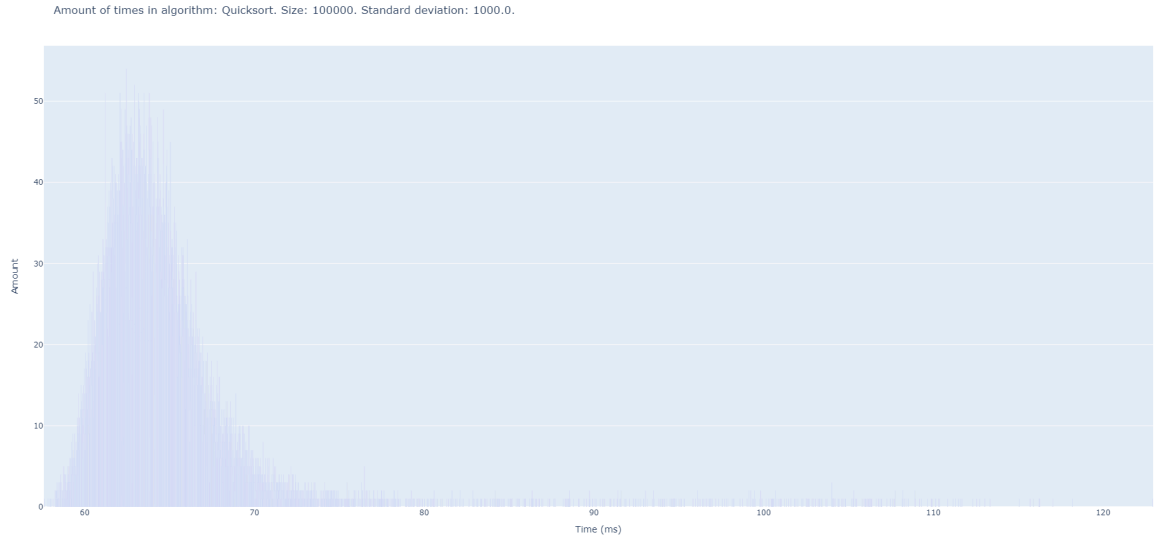

Chart 8: Quicksort's distribution of times sorting 100,000 elements with a standard deviation of 1,000.

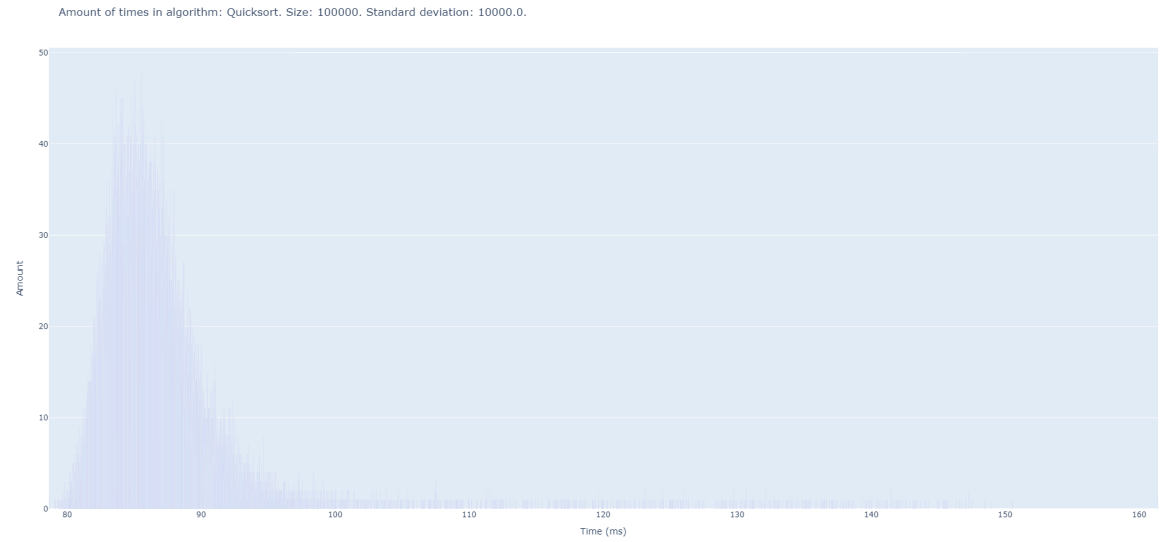

Chart 9: Quicksort's distribution of times sorting 100,000 elements with a standard deviation of 10,000.

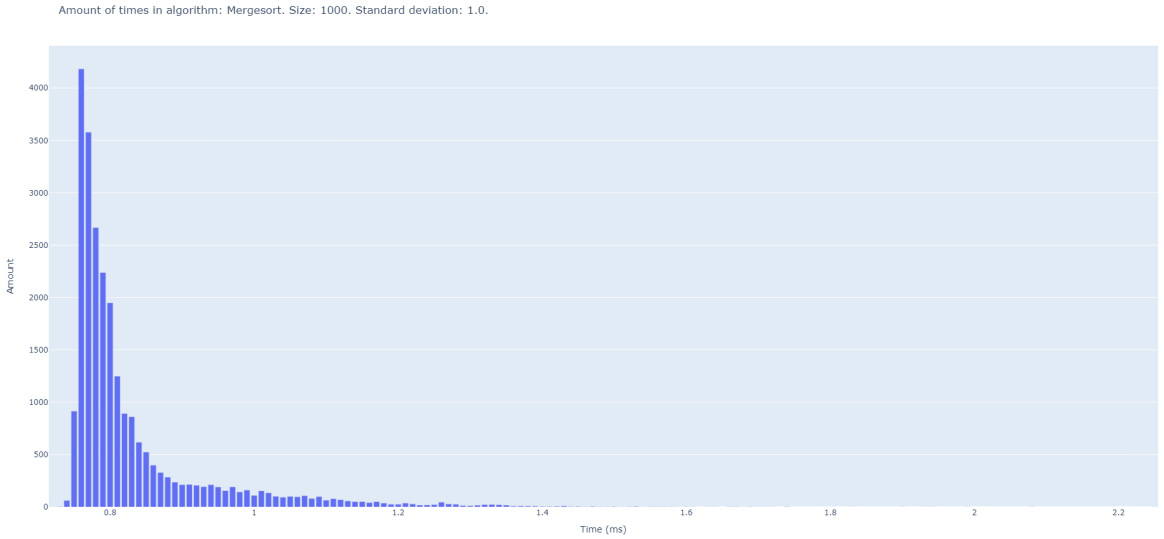

Chart 10: Mergesort's distribution of times sorting 1,000 elements with a standard deviation of 1.

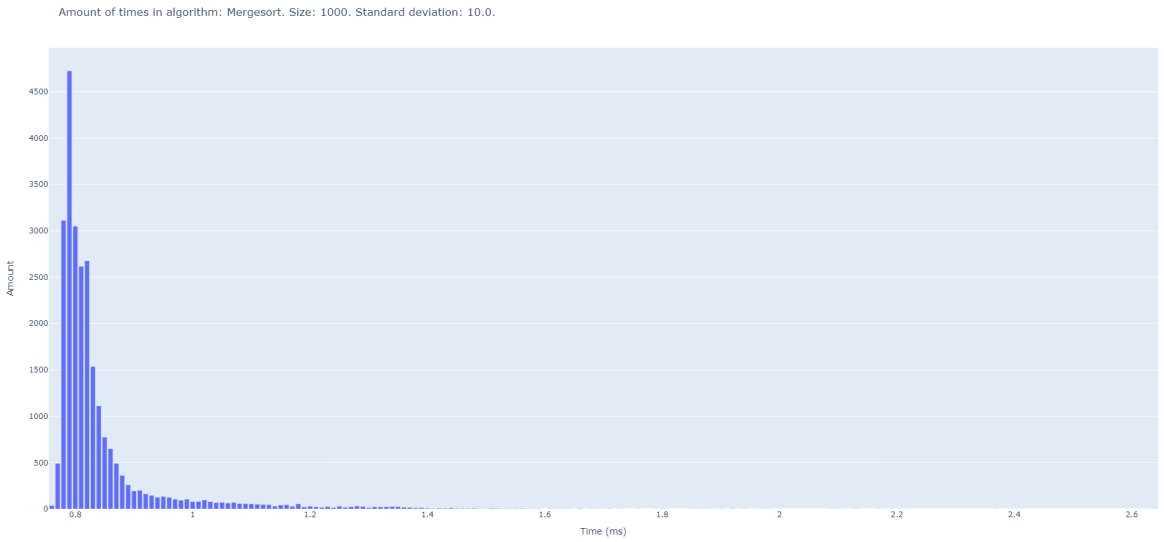

Chart 11: Mergesort's distribution of times sorting 1,000 elements with a standard deviation of 10.

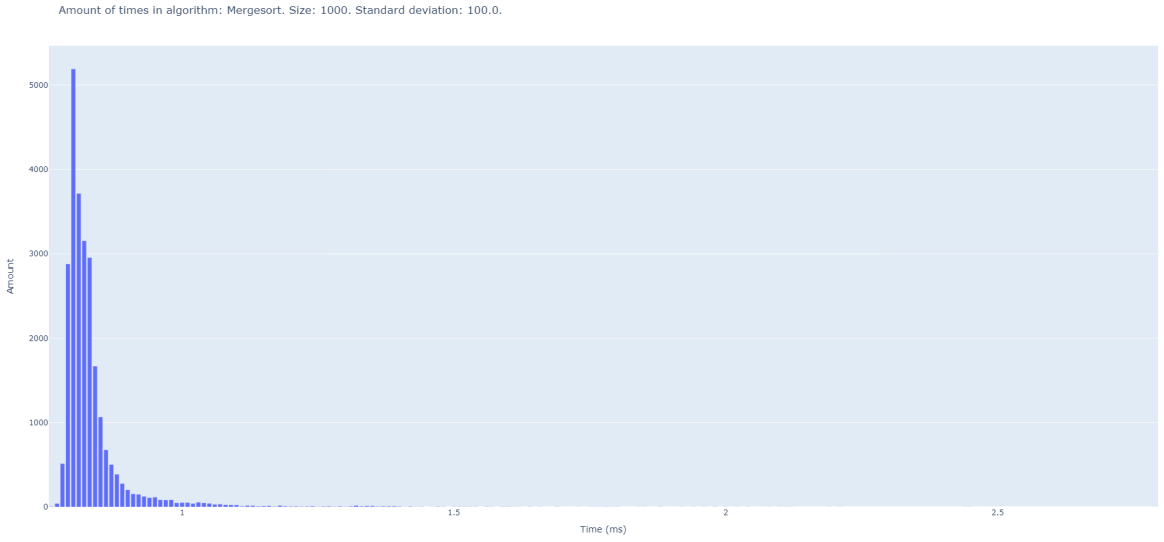

Chart 12: Mergesort's distribution of times sorting 1,000 elements with a standard deviation of 100.

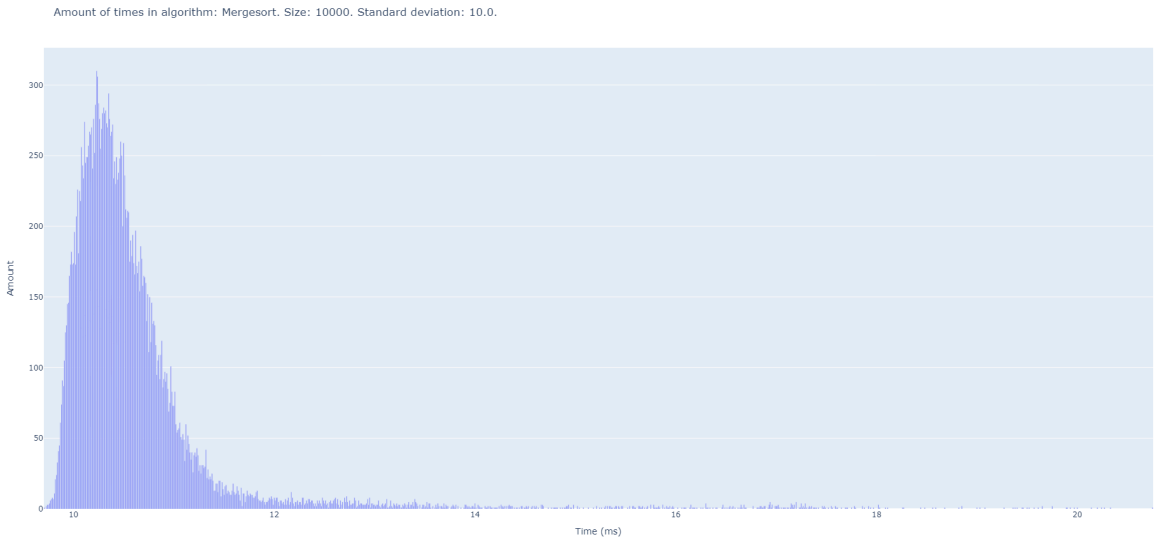

Chart 13: Mergesort's distribution of times sorting 10,000 elements with a standard deviation of 10.

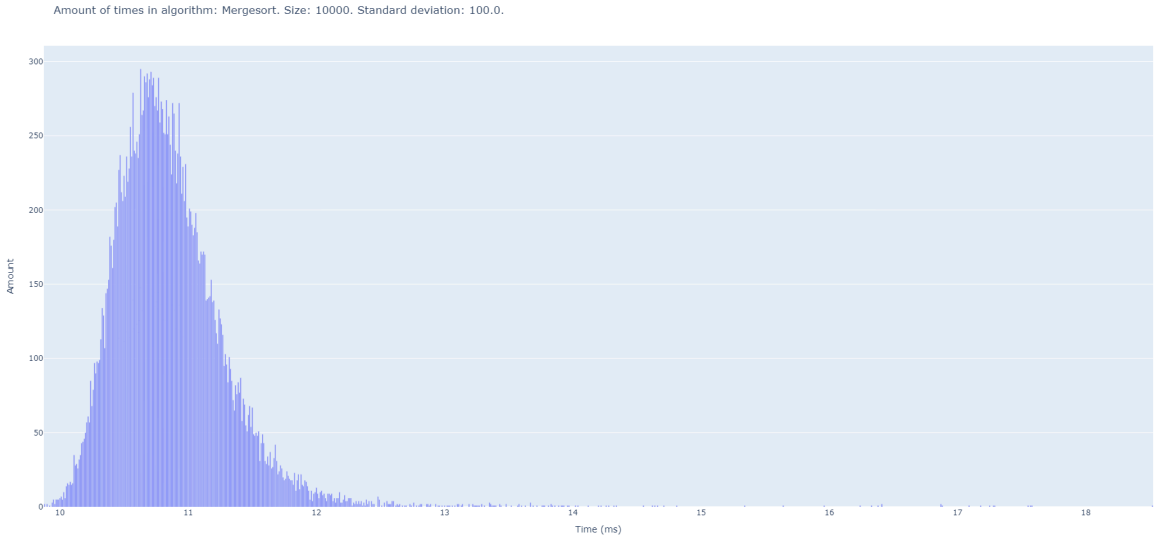

Chart 14: Mergesort's distribution of times sorting 10,000 elements with a standard deviation of 100.

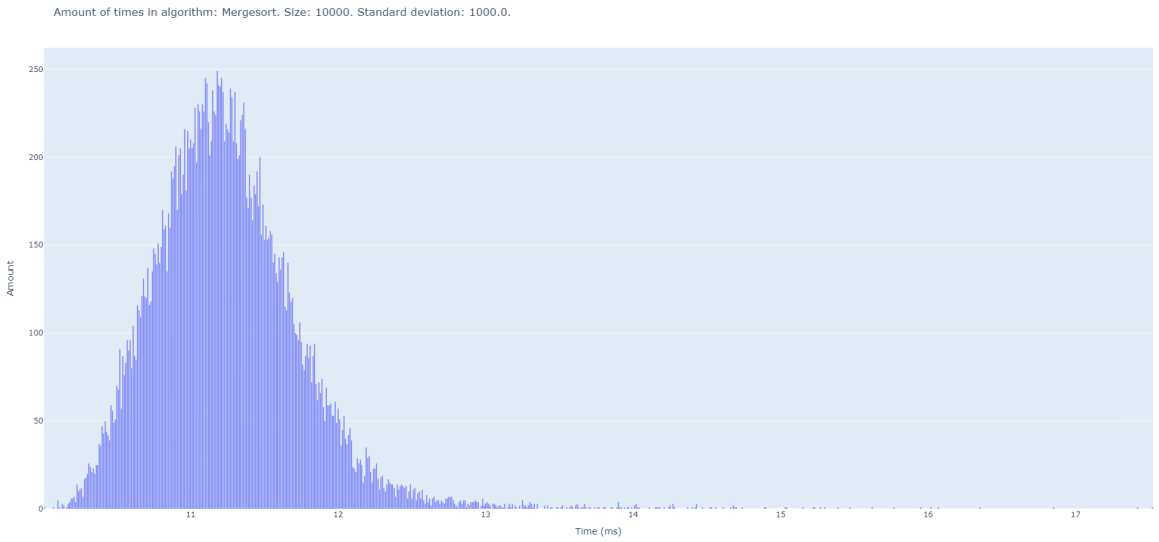

Chart 15: Mergesort's distribution of times sorting 10,000 elements with a standard deviation of 1,000.

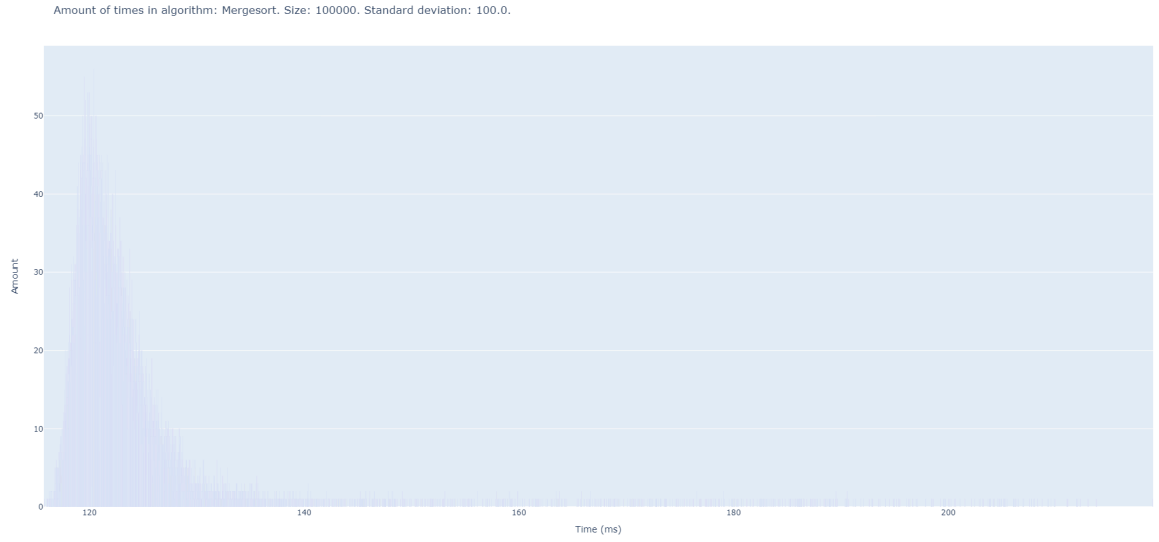

Chart 16: Mergesort's distribution of times sorting 100,000 elements with a standard deviation of 100.

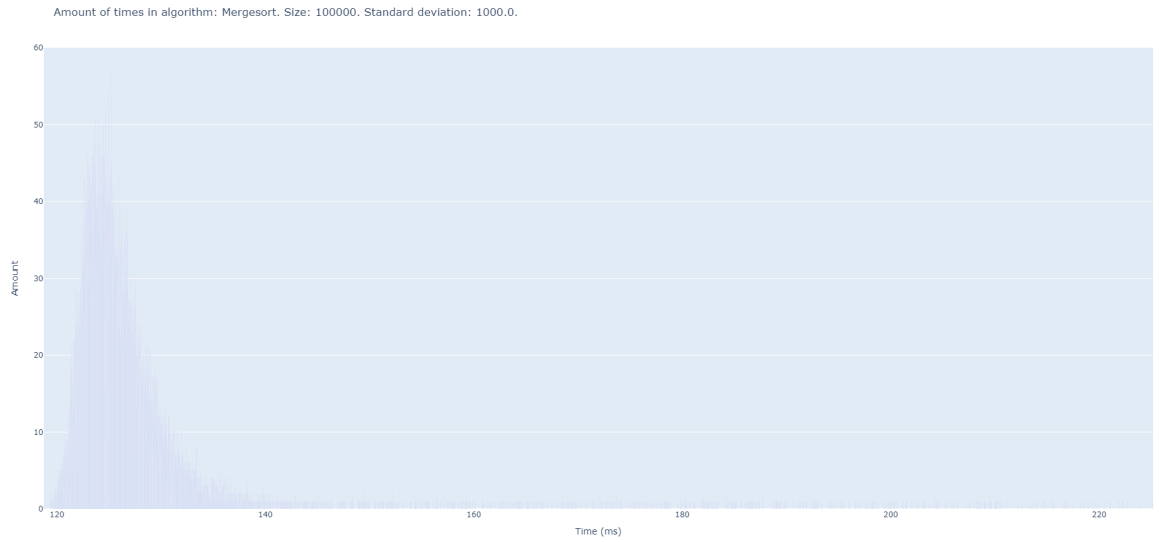

Chart 17: Mergesort's distribution of times sorting 100,000 elements with a standard deviation of 1,000.

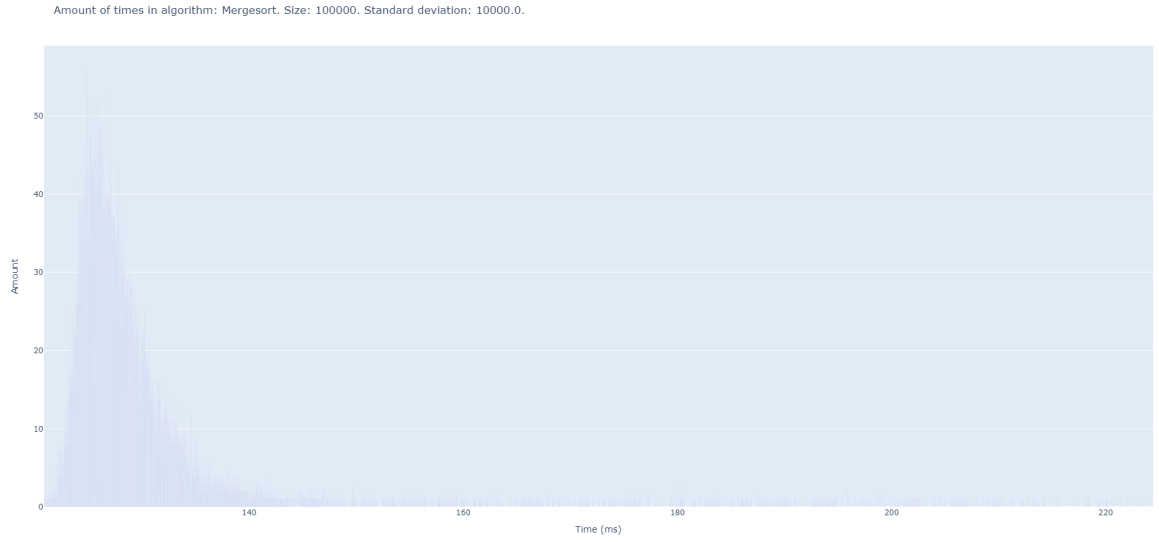

Chart 18: Mergesort's distribution of times sorting 100,000 elements with a standard deviation of 10,000.

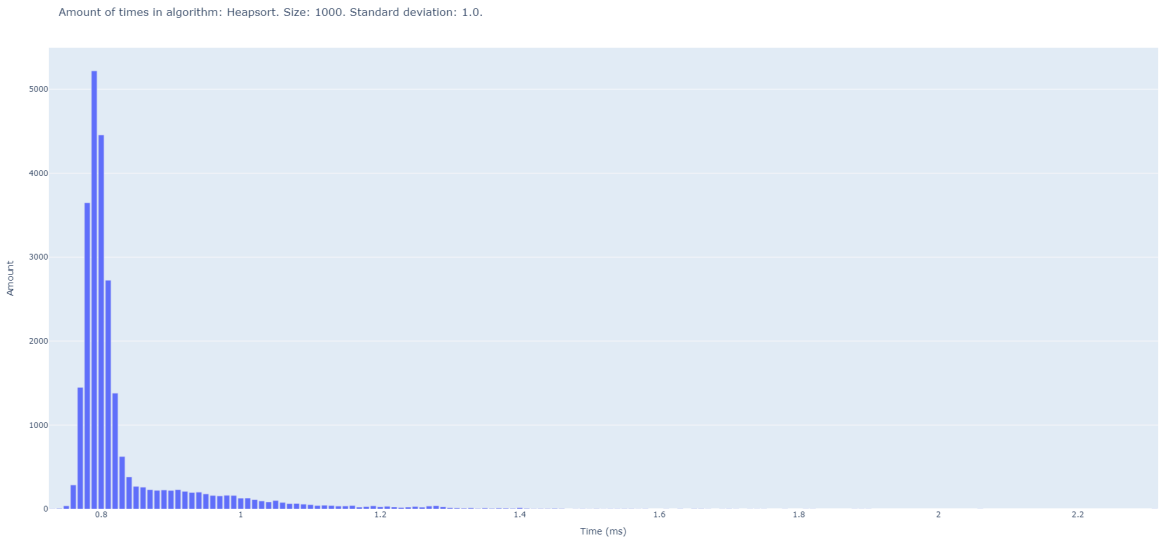

Chart 19: Heapsort's distribution of times sorting 1,000 elements with a standard deviation of 1.

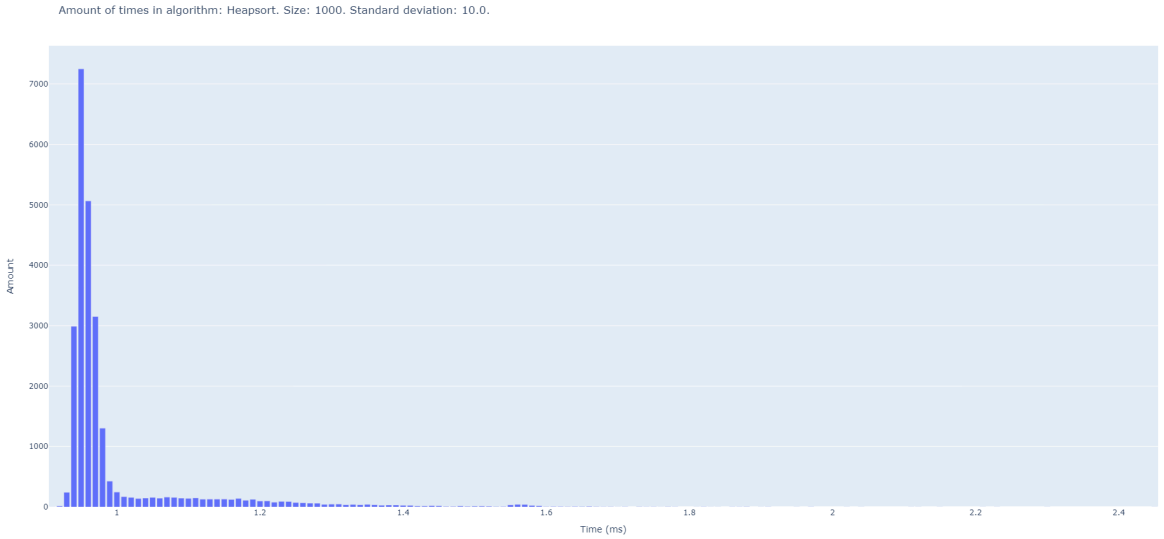

Chart 20: Heapsort's distribution of times sorting 1,000 elements with a standard deviation of 10.

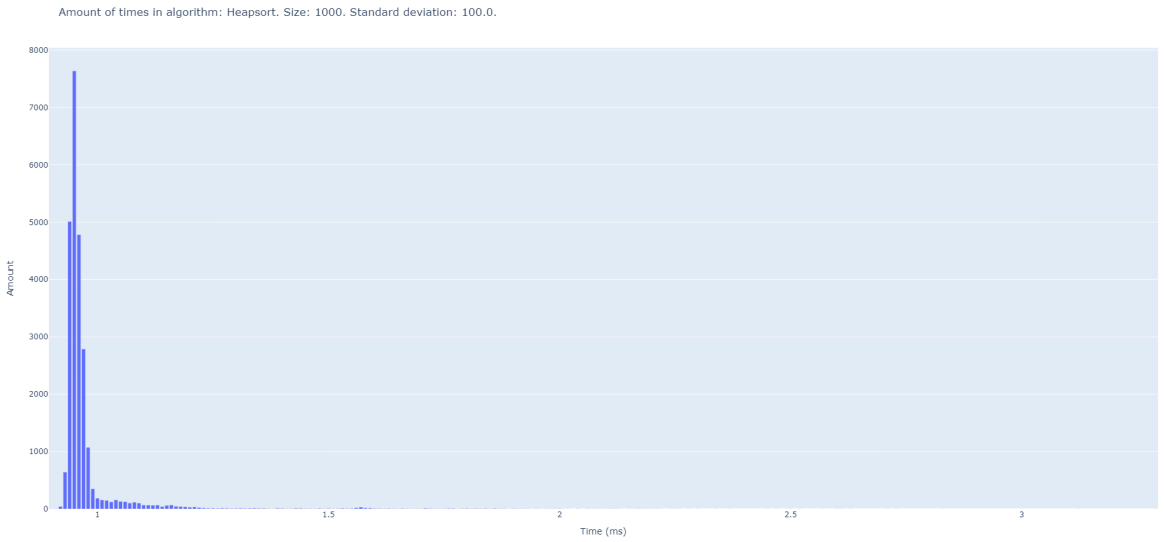

Chart 21: Heapsort's distribution of times sorting 1,000 elements with a standard deviation of 100.

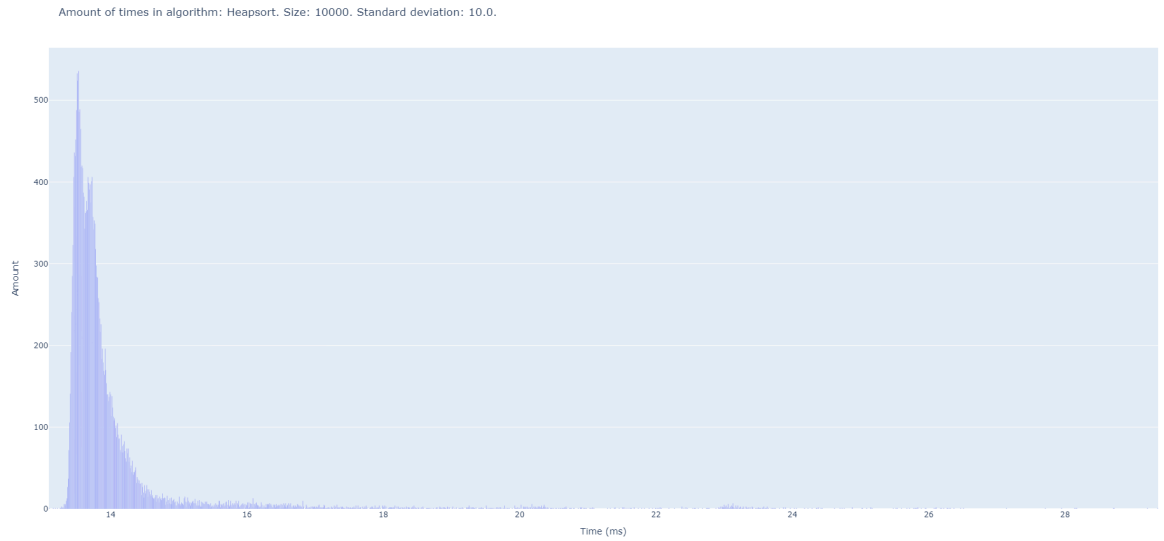

Chart 22: Heapsort's distribution of times sorting 10,000 elements with a standard deviation of 10.

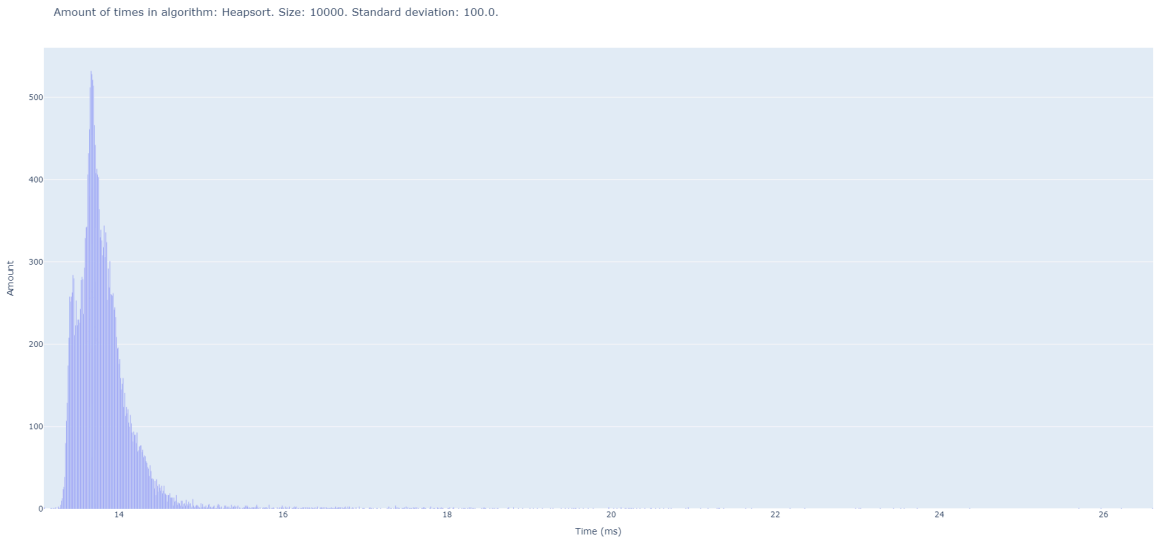

Chart 23: Heapsort's distribution of times sorting 10,000 elements with a standard deviation of 100.

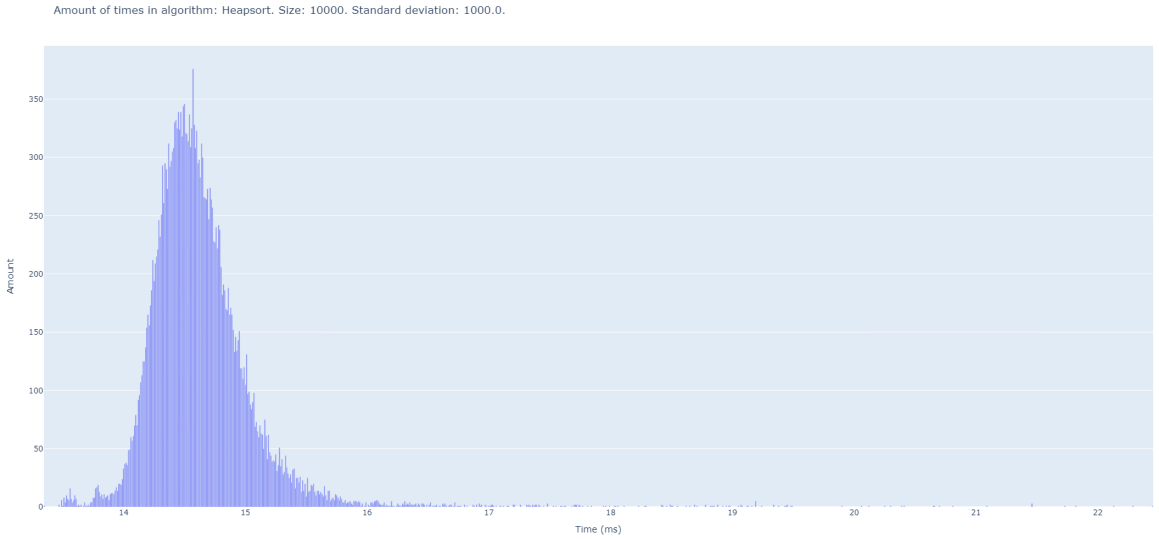

Chart 24: Heapsort's distribution of times sorting 10,000 elements with a standard deviation of 1,000.

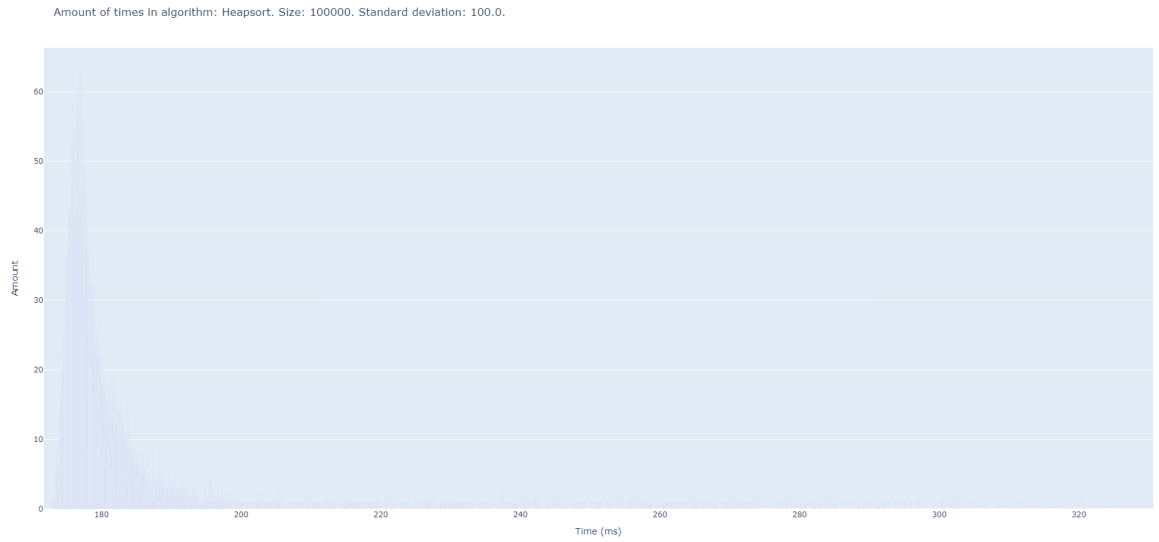

Chart 25: Heapsort's distribution of times sorting 100,000 elements with a standard deviation of 100.

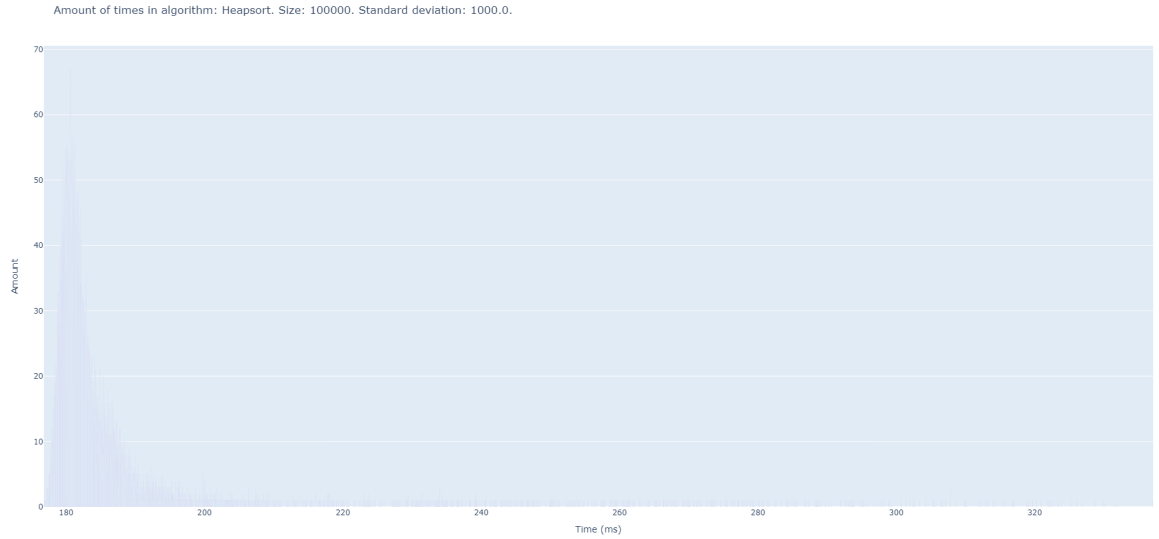

Chart 26: Heapsort's distribution of times sorting 100,000 elements with a standard deviation of 1,000.

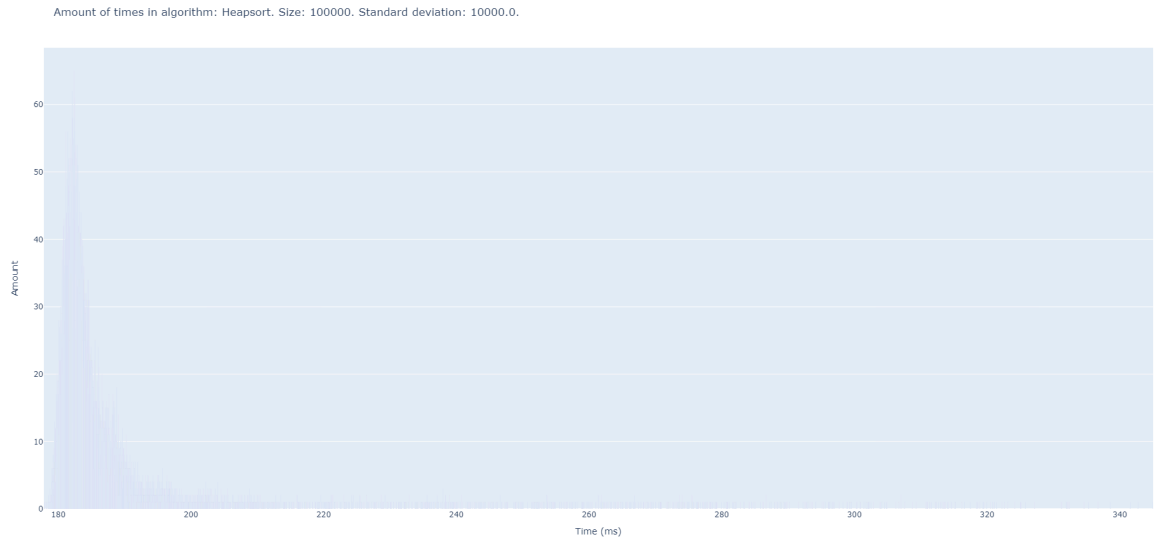

Chart 27: Heapsort's distribution of times sorting 100,000 elements with a standard deviation of 10,000.

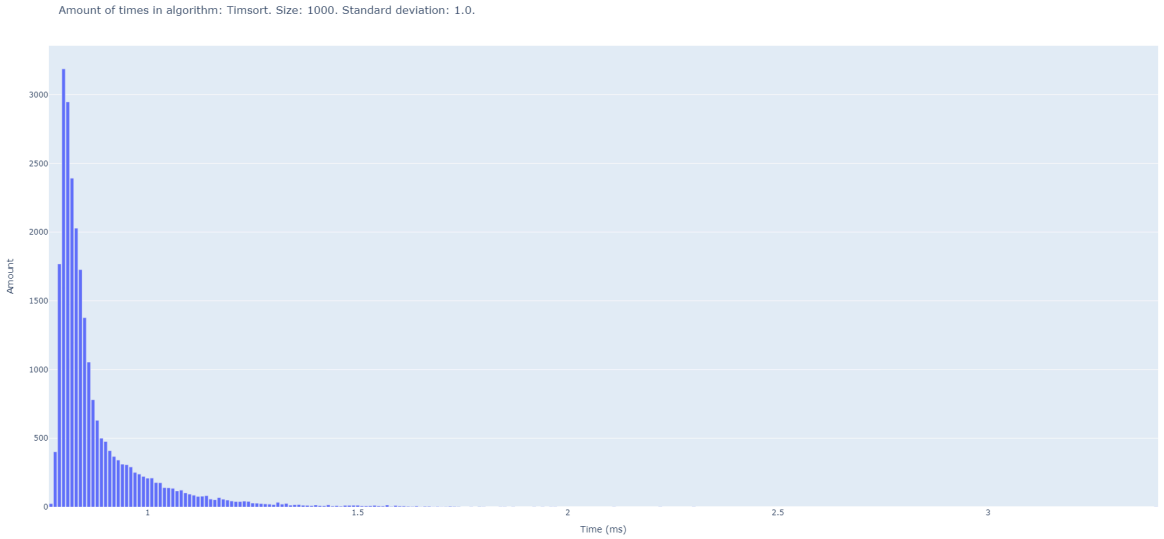

Chart 28: Timsort's distribution of times sorting 1,000 elements with a standard deviation of 1.

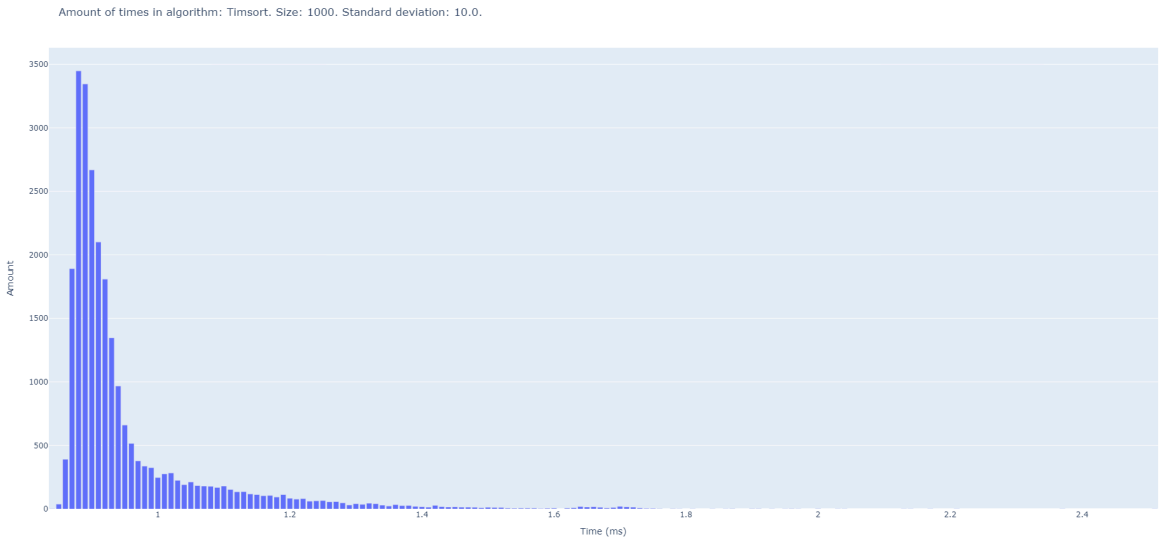

Chart 29: Timsort's distribution of times sorting 1,000 elements with a standard deviation of 10.

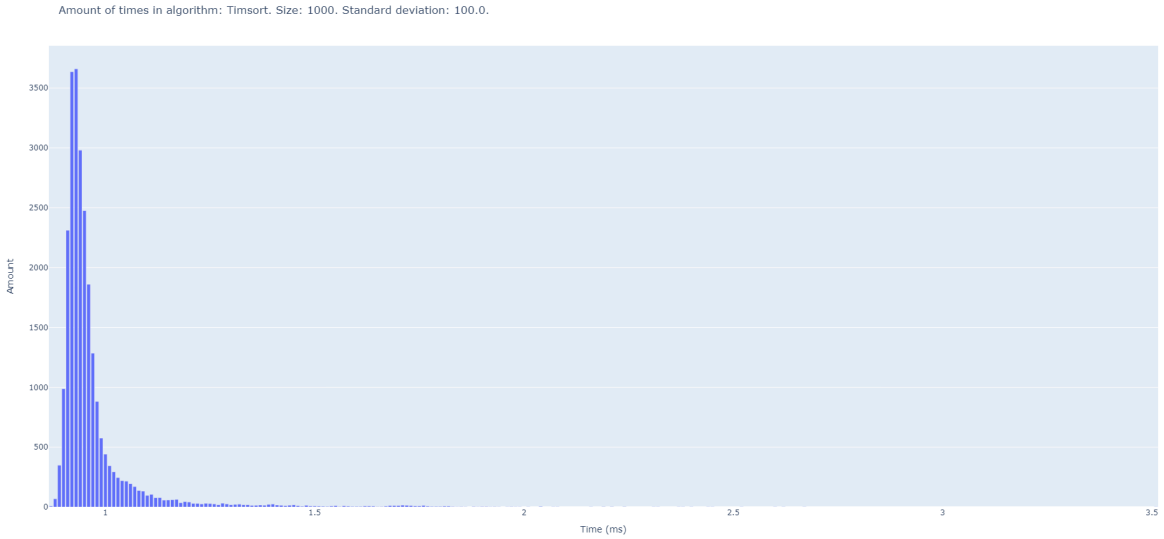

Chart 30: Timsort's distribution of times sorting 1,000 elements with a standard deviation of 100.

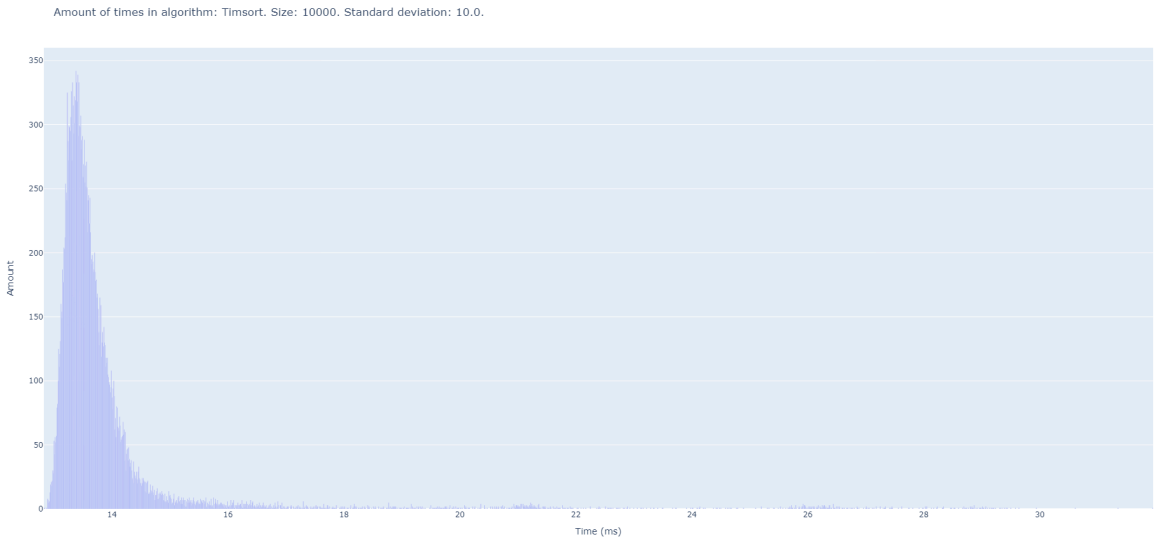

Chart 31: Timsort's distribution of times sorting 10,000 elements with a standard deviation of 10.

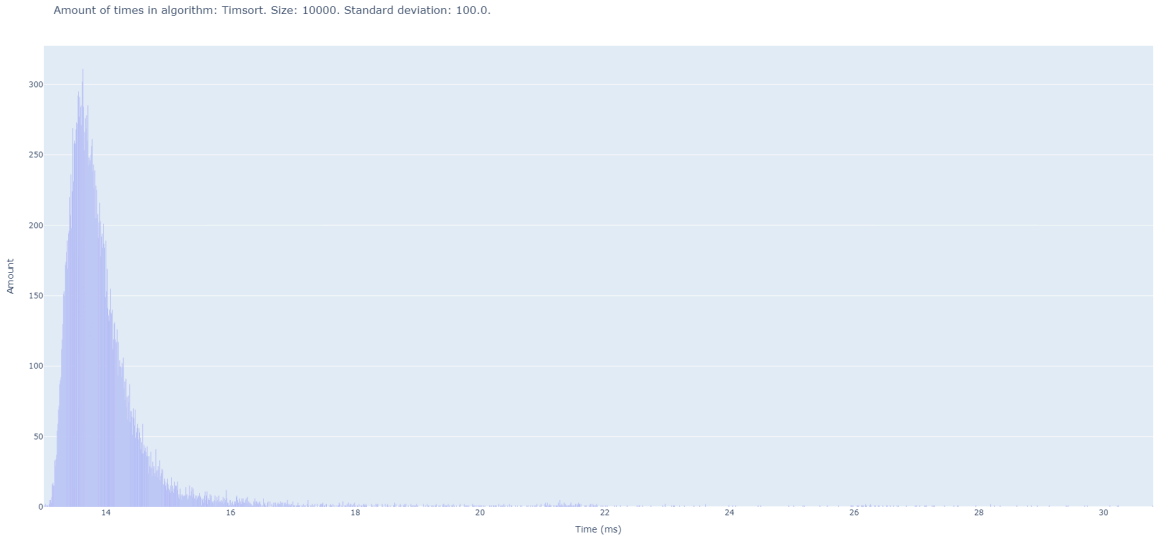

Chart 32: Timsort's distribution of times sorting 10,000 elements with a standard deviation of 100.

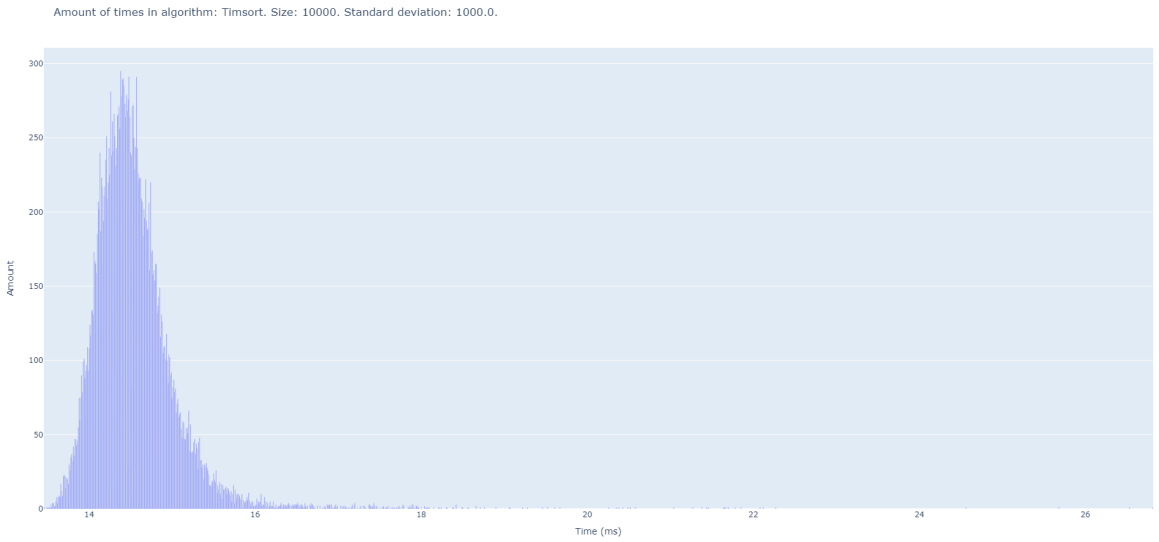

Chart 33: Timsort's distribution of times sorting 10,000 elements with a standard deviation of 1,000.

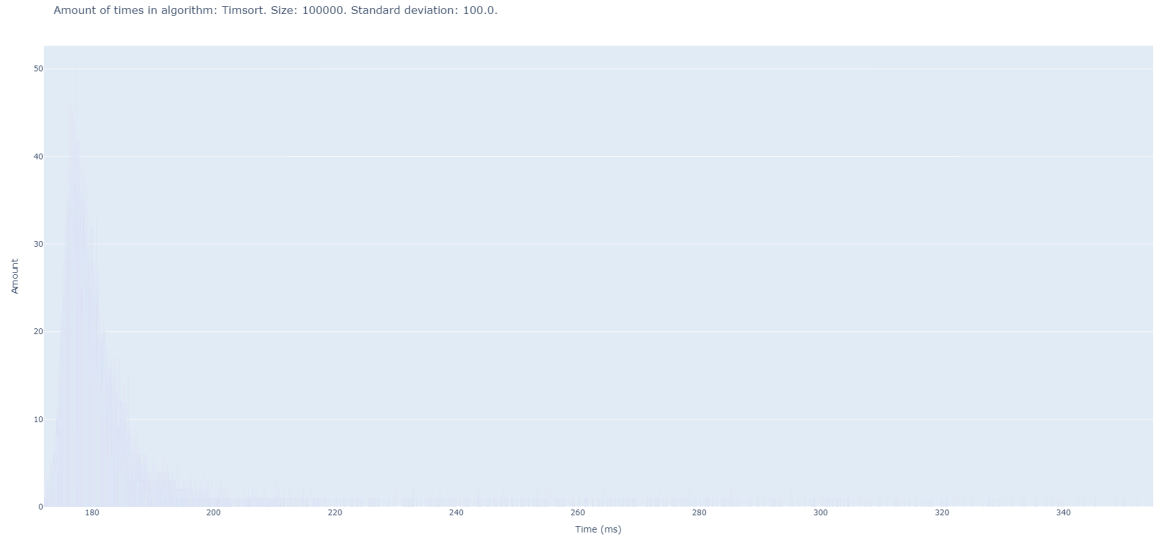

Chart 34: Timsort's distribution of times sorting 100,000 elements with a standard deviation of 100.

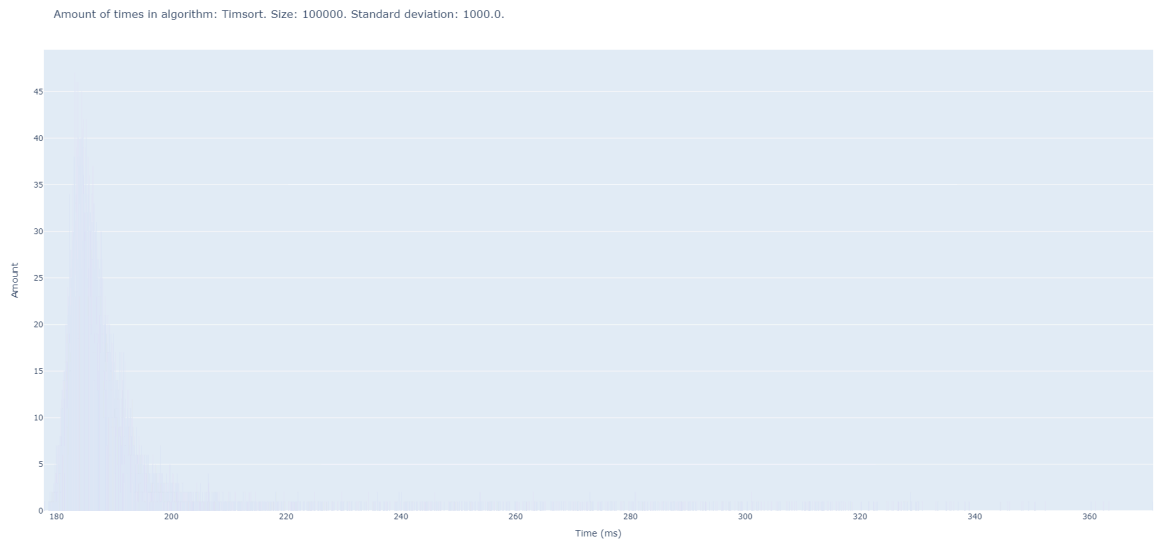

Chart 35: Timsort's distribution of times sorting 100,000 elements with a standard deviation of 1,000.

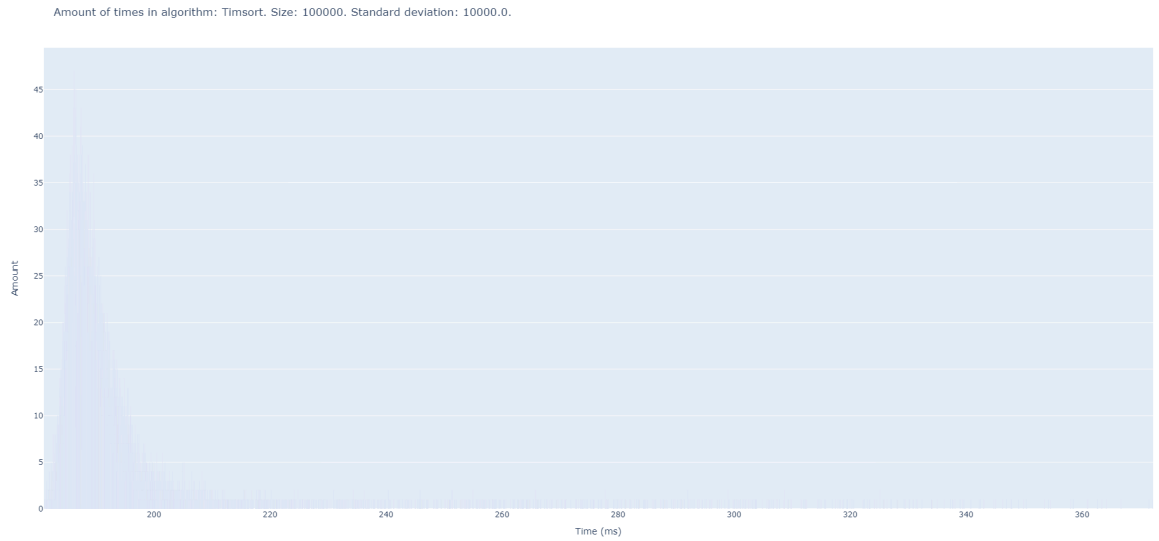

Chart 36: Timsort's distribution of times sorting 100,000 elements with a standard deviation of 10,000.

main.py:

```python
import random
import pickle
import time
import os
from algorithms import *
```

```
class Result:
  def __init__(self, time_ns, algorithm, size, deviation) -> None:
    self.time_ns = time_ns
    self.algorithm = algorithm
    self.size = size
    self.deviation = deviation


random.seed(os.urandom(255))
sizes = (1000, 10000, 100000)
deviations = (0.1, 0.01, 0.001)
iterations = 25000
results_list = []
for iteration in range(iterations):
  print(iteration)
  for size in sizes:
    for deviation in deviations:
      to_be_sorted = [
        int(round(random.gauss(0, deviation * size), 0)) for _ in range(size)
      ]

      quick_start = time.perf_counter_ns()
      quicksort(to_be_sorted.copy())
      quick_end = time.perf_counter_ns()
      results_list.append(
        Result(quick_end - quick_start, "Quicksort", size, deviation)
      )

      merge_start = time.perf_counter_ns()
      mergesort(to_be_sorted.copy())
      merge_end = time.perf_counter_ns()
      results_list.append(
        Result(merge_end - merge_start, "Mergesort", size, deviation)
      )

      heap_start = time.perf_counter_ns()
```

```
      heapsort(to_be_sorted.copy())
      heap_end = time.perf_counter_ns()
      results_list.append(
        Result(heap_end - heap_start, "Heapsort", size, deviation)
      )

      tim_start = time.perf_counter_ns()
      timsort(to_be_sorted.copy())
      tim_end = time.perf_counter_ns()
              results_list.append(Result(tim_end - tim_start, "Timsort", size,
deviation))


with open("data.pkl", "wb") as file:
  pickle.dump(results_list, file)
```

Program 1: The code that runs and tests all of the algorithms and saves the data.

algorithms.py:

```python
def quicksort(arr: list) -> list:
  if len(arr) < 2:
    return arr
  mid = len(arr) // 2
  if arr[0] > arr[mid]:
    arr[0], arr[mid] = arr[mid], arr[0]
  if arr[0] > arr[-1]:
    arr[0], arr[-1] = arr[-1], arr[0]
  if arr[mid] > arr[-1]:
    arr[mid], arr[-1] = arr[-1], arr[mid]
  pivot = arr[mid]
  less, equal, greater = [], [], []
  for i in arr:
    if i < pivot:
      less.append(i)
    elif i == pivot:
      equal.append(i)
```

```
    else:
      greater.append(i)
  return quicksort(less) + equal + quicksort(greater)


def mergesort(arr: list) -> list:
  if len(arr) < 2:
    return arr
  left = mergesort(arr[: len(arr) // 2])
  right = mergesort(arr[len(arr) // 2 :])
  sorted_list = []
  left_index = 0
  right_index = 0
  left_len = len(left)
  right_len = len(right)
  while left_len > left_index and right_len > right_index:
    left_item = left[left_index]
    right_item = right[right_index]
    if left_item <= right_item:
      sorted_list.append(left_item)
      left_index += 1
    else:
      sorted_list.append(right_item)
      right_index += 1
  while left_len > left_index:
    sorted_list.append(left[left_index])
    left_index += 1
  while right_len > right_index:
    sorted_list.append(right[right_index])
    right_index += 1
  return sorted_list


def heapsort(arr: list) -> list:
  arr = arr.copy()
  start = len(arr) // 2
  end = len(arr)
```

```
  while end > 1:
    if start > 0:
      start -= 1
    else:
      end -= 1
      arr[end], arr[0] = arr[0], arr[end]
    root = start
    while 2 * root + 1 < end:
      child = 2 * root + 1
      if child + 1 < end and arr[child] < arr[child + 1]:
        child += 1
      if arr[root] < arr[child]:
        arr[root], arr[child] = arr[child], arr[root]
        root = child
      else:
        break
  return arr


def timsort(arr: list) -> list:
  min_run = 32
  n = len(arr)
  for i in range(0, n, min_run):
    left = i
    right = min((i + min_run - 1), n - 1)
    if right is None:
      right = len(arr) - 1
    for i in range(left + 1, right + 1):
      key_item = arr[i]
      j = i - 1
      while j >= left and arr[j] > key_item:
        arr[j + 1] = arr[j]
        j -= 1
      arr[j + 1] = key_item

  size = min_run
  while size < n:
```

```
    for start in range(0, n, size * 2):
      midpoint = start + size - 1
      end = min((start + size * 2 - 1), (n - 1))
      left = arr[start : midpoint + 1]
      right = arr[midpoint + 1 : end + 1]
      if len(left) == 0:
        merged_array = right
      elif len(right) == 0:
        merged_array = left
      else:
        arr2 = []
        index_left = index_right = 0
        while len(arr2) < len(left) + len(right):
          if left[index_left] <= right[index_right]:
            arr2.append(left[index_left])
            index_left += 1
          else:
            arr2.append(right[index_right])
            index_right += 1
          if index_right == len(right):
            arr2 += left[index_left:]
            break

          if index_left == len(left):
            arr2 += right[index_right:]
            break
        merged_array = arr2
      arr[start : start + len(merged_array)] = merged_array
    size *= 2

  return arr
```

Program 2: The code that contains all of the algorithms.

plot.py:

Python

```python
import pickle
import plotly.express as px


class Result:
  def __init__(self, time_ns, algorithm, size, deviation) -> None:
    self.time_ns = time_ns
    self.algorithm = algorithm
    self.size = size
    self.deviation = deviation


with open("data.pkl", "rb") as file:
  results_list = pickle.load(file)


def average(arr: list) -> float:
  return sum(arr) / len(arr)


def plot_list_as_graph(data, name, size, deviation):
  count_dict = {}
  for item in data:
    count_dict[item] = count_dict.get(item, 0) + 1
  x_values = list(count_dict.keys())
  y_values = list(count_dict.values())
  fig = px.bar(
    x=x_values,
    y=y_values,
    labels={"x": "Time (ms)", "y": "Amount"},
       title=f"Amount of times in algorithm: {name}. Size: {size}. Standard
deviation: {size*deviation}.",
  )
  fig.show()


algorithms = ("Quicksort", "Mergesort", "Heapsort", "Timsort")
sizes = (1000, 10000, 100000)
```

```
deviations = (0.1, 0.01, 0.001)
list_of_averages = {
  f"{i}_{j}": {
    k: average(
      [
        l.time_ns * 1e-6
        for l in results_list
        if l.size == i and l.deviation == j and l.algorithm == k
      ]
    )
    for k in algorithms
  }
  for i in sizes
  for j in deviations
}
list_of_values = {
  f"{i}_{j}": {
    k: [
      round(l.time_ns * 1e-6, 2)
      for l in results_list
      if l.size == i and l.deviation == j and l.algorithm == k
    ]
    for k in algorithms
  }
  for i in sizes
  for j in deviations
}
categories = [f"{i}_{j}" for i in sizes for j in deviations]
for i in categories:
   list_of_averages[i] = dict(sorted(list_of_averages[i].items(), key=lambda x:
x[1]))
  print(i)
  print(list_of_averages[i])
quicksort_times = []
mergesort_times = []
heapsort_times = []
timsort_times = []
```

```
for i in list_of_averages:
  quicksort_times.append(list_of_averages[i]["Quicksort"])
  mergesort_times.append(list_of_averages[i]["Mergesort"])
  heapsort_times.append(list_of_averages[i]["Heapsort"])
  timsort_times.append(list_of_averages[i]["Timsort"])
print(average(quicksort_times))
print(average(mergesort_times))
print(average(heapsort_times))
print(average(timsort_times))
[
  plot_list_as_graph(
    list_of_values[f"{size}_{deviation}"][algorithm],
    algorithm,
    size,
    deviation,
  )
  for algorithm in algorithms
  for size in sizes
  for deviation in deviations
]
```

Program 3: The code that takes the data and processes it into readable and useful formats.