MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Adaptive Sorting Algorithms

Master's Thesis

DANIEL TIMKO

Brno, Spring 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Adaptive Sorting Algorithms

Master's Thesis

DANIEL TIMKO

Advisor: Prof. RNDr. Ivana Černá, CSc.

Brno, Spring 2025


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Daniel Timko

Advisor: Prof. RNDr. Ivana Černá, CSc.

Acknowledgements

I would like to thank my supervisor, prof. RNDr. Ivana Černá, CSc., for all her time, support, and advice throughout the development of this thesis.

Abstract

The general algorithmic problem of sorting has been solved for decades with asymptotically optimal algorithms like Heapsort or Merge Sort. However, much of real-world data is already partially ordered to some degree, and traditional algorithms may fail to take advantage of this. Adaptive sorting algorithms are designed to exploit this input property and improve the practical efficiency of sorting. The contribution of this thesis is threefold. First, a theoretical overview of adaptive sorting is presented, including descriptions and pseudocode for three selected algorithms: Natural Merge Sort, Timsort, and Powersort. Second, the work provides a practical implementation of these algorithms in Python. Finally, their performance is experimentally evaluated on inputs with varying degrees of presortedness.

Keywords

adaptive sorting, sorting algorithms, presortedness, Natural Merge Sort, Timsort, Powersort, benchmark, performance analysis, Python

Contents

Introduction **1**

1 Adaptive Sorting **2**
1.1 Merge Sort 2
1.2 Measures of Presortedness 4
1.3 Performance Measures 6

2 Algorithms **8**
2.1 Natural Merge Sort 8
2.2 Timsort 10
2.2.1 Optimizations 13
2.3 Powersort 17
2.3.1 Merge trees 18
2.3.2 Algorithm 18
2.3.3 Power 20

3 Benchmarks **22**
3.1 Random Input Generators 22
3.2 Scope Definition 28

4 Experimental evaluation **31**
4.1 Results on random arrays 32
4.2 Results based on the number of runs 33
4.3 Results based on the entropy of the run profile 35
4.4 Performance impact of the optimizations 39

5 Conclusion **41**

A Source Code Archive Structure **42**

Bibliography **43**

Introduction

Sorting data is a very common and fundamental algorithmic problem that is theoretically very well researched and it might seem there is little room left for improvement in this area.

Indeed, it was proven long ago that the asymptotically optimal time complexity for comparison-based sorting is $\mathcal{O}(n * log(n))$. However, what about practical considerations? We want our sorting algorithms to work well with real-world data and it appears that in practice, input data often contains some degree of inherent order, which raises intriguing questions. Is it possible to exploit this pre-existing partial order to save some time? How significant does this order need to be to make a meaningful difference in performance?

This work focuses on the analysis of three "natural merge" sorting algorithms (Natural Merge sort, Timsort, and Powersort), which is a subclass of adaptive sorting algorithms. They are called so because they aim to leverage the pre-existing order in the data and adapt to it, optimizing the number of performed operations. Additionally, natural merge algorithms, as the name suggests, are based on the merge operation as we know it from the traditional Merge Sort.

This thesis provides a uniform theoretical overview of these algorithms and experimentally evaluates their potential to reduce sorting time in real-world applications.

Chapter 1 introduces the concept of adaptive sorting and establishes the theoretical foundations necessary to comprehend the subsequent chapters and the thesis as a whole.

Chapter 2 explains the observed algorithms in detail, presenting their pseudocode and referencing the original publications.

The experimental part of the thesis starts with Chapter 3, which defines the scope of the benchmarks, characterizes their inputs, and explains the methods used to generate them.

Chapter 4 presents and interprets the obtained experimental results.

Finally, Chapter 5 concludes by summarizing the key findings and takeaways, and proposing potential future research and improvements.

1 Adaptive Sorting

1.1 Merge Sort

Before narrowing down on adaptive sorting, I consider it useful to review how the traditional top-down Merge Sort algorithm works, especially its `Merge` procedure that is also used in all the other sorting algorithms observed in this thesis.

Algorithm 1 presents the implementation of the algorithm. The `MergeSort` function itself employs a classic divide-and-conquer approach to split the passed array into two halves and then recursively call itself for both halves. The base condition (line 2) of the recursion is triggered when the size of the array received as an argument is 1, meaning it cannot be further divided. After line 7, both halves are sorted. Finally, the `Merge` procedure is called to combine them into a single sorted array.

This procedure receives two subarrays and maintains two pointers i and j, one for each of them, initialized to 0 (pointing to the first elements of their respective subarrays). Then it compares those two elements, appends the smaller one to the new array `arr`, and increments the pointer for the corresponding subarray. This process is repeated until one of the subarrays is exhausted. Finally, any remaining elements from the other subarray are copied over. This guarantees that at any point of the algorithm, the resulting array `arr` remains sorted.

The `Merge` procedure copies over one element in every step and every element is copied exactly once. Therefore, its time complexity is linear w.r.t the sizes n and m of the two subarrays: $\mathcal{O}(n + m)$. Consequently, because of the halving of the input with each call of `MergeSort`, the overall time complexity of the algorithm is $\mathcal{O}(n \log n)$, with n being the size of the array.

Adaptive Sorting

In this era of big data and data-driven decision making, the efficiency of data processing is more crucial than ever before. One of the most fundamental data operations is sorting. Of course, asymptotically optimal algorithms to sort the input data are well-known, assuming

Algorithm 1 Merge Sort implementation

```
1:  function MergeSort(arr)
2:      if arr.length() ≤ 1 then
3:          return arr
4:      end if
5:      mid ← arr.length() // 2
6:      left ← MergeSort(arr[: mid])
7:      right ← MergeSort(arr[mid :])
8:      return Merge(left, right)
9:  end function

10: function Merge(left, right)
11:     arr ← []
12:     i ← 0
13:     j ← 0
14:     while i < left.length() ∧ j < right.length() do
15:         if left[i] ≤ right[j] then
16:             arr.append(left[i])
17:             i += 1
18:         else
19:             arr.append(right[j])
20:             j += 1
21:         end if
22:     end while
23:     arr.extend(left[i :])
24:     arr.extend(right[j :])
25:     return arr
26: end function
```

limited memory and randomness of the data. However, many real-world applications work with data that is already partially presorted. That is where adaptive sorting steps in and tries to make the most of it.

Of course, even with significantly presorted data, the asymptotic time complexity based on input size does not improve, but as we'll see later, the constants involved can decrease enough to make a considerable difference.

Let's consider the scenario where we want to use the traditional Merge Sort to sort an input array of N elements with one half of it being completely sorted already. If we knew that a priori, we could avoid sorting that half of the array, which would save a significant amount of time. Adaptive sorting algorithms are all about detecting this pre-existing order and exploiting it.

This thesis focuses on algorithms belonging to a subclass of adaptive algorithms known as natural merge algorithms, which are based on the merge procedure identical to that of Merge Sort. The key distinction between them is how and when is this procedure invoked.

1.2 Measures of Presortedness

To be able to measure the impact of presortedness of the input data on the performance of sorting algorithms, it is necessary to define what that presortedness means and how to measure it. There are many different input characteristics that are suitable to be used to measure the degree of order in the data.

Runs

A **run** is a maximal increasing subarray – a contiguous part of the original array, which cannot be extended from either side by adding more elements while preserving its sortedness.

Every array has a run decomposition. This array is then a concatenation of all runs of its run decomposition. A **run profile** is an array of run lengths. Therefore, the length of a run profile is equal to the number of runs in the original array, and its sum is equal to the number of elements in the original array.

In some of the related literature, runs are defined only as increasing. It is completely valid to override this definition and introduce also **decreasing** runs. However, it also means that an array can have multiple correct run decompositions. For example, for array `[10,20,10]` both `[(10,20),(10)]` and `[(10),(20,10)]` are correct and minimal run decompositions. Another implication is that the minimal length of a run is 2.

An algorithm to find a run profile of an array is trivial and involves a single linear scan of the array. It simply iterates from left to right, keeping track of whether the current run is increasing or decreasing. When reaching an element that breaks this sequence, the current run is saved and a new run is started.

The number of runs of the array, or the length of its run profile, can be considered as a measure of presortedness. The fewer runs an array has, the more sorted subsequences it contains, and thus the more presorted it is. Later we will observe that all of the studied algorithms incorporate the process of finding the run profile of the input array. Therefore, this measure is very appropriate.

Entropy of the Run Profile

Apart from the number of runs, it also makes sense to look at their relative sizes. The question to ask is whether it helps when all the runs have similar sizes, or if it's better when their sizes are wildly different, or whether it has any impact at all. In other words, for an array with some fixed level of presortedness in terms of the number of runs, does it make any further difference whether this disorder is localized to a single long sorted subarray or distributed in smaller chunks across the entire array? This attribute is called **entropy of the run profile**.

Entropy is a concept stemming from information theory that is fundamental for probability and statistics. In general, it quantifies the level of disorder and uncertainty within a system. In its base form, it was first introduced in 1948 by Claude Shannon [1] and is also referred to as Shannon entropy. Given a discrete random variable X with probability distribution P, entropy of this random variable is defined as

$$H(X) = -\sum_{p \in P} p \, log_2 p$$

Higher entropy indicates a more uniform probability distribution (greater uncertainty), whereas lower entropy suggests a more predictable outcome. We can translate this concept to the context of measuring presortedness of an array, where instead of assessing the level of uncertainty in a probability distribution, the entropy shall assess the level of disorder in a run profile of the array. We shall call this specific usecase the **entropy of the run profile** P. With N being the sum of the run lengths, it uses the same definition as before but replaces probabilities p with run lengths normalized to sum up to 1 ($\frac{r}{N}$), same as probabilities do:

$$H(P) = -\sum_{r \in P} \frac{r}{N} log_2 \frac{r}{N}$$

Now let's return to the question posed at the beginning of this section. For example, for $N = 100$ and $k = 10$, does it make any performance difference whether the run profile is `[10,10,10,10,10,10,10,10,10,10]` or `[82,2,2,2,2,2,2,2,2,2]`? These two run profiles represent two extreme cases in terms of their entropy. The first one is completely uniform, thus having the highest entropy, whereas the second one is the most skewed, with the lowest entropy.

The answer to this question along with the underlying intuition is presented experimentally in Section 4.4.

1.3 Performance Measures

In the context of algorithm benchmarking, performance measure is a metric used to evaluate the gain or loss of performance.

For algorithms in general, the most common metric for benchmarking is time, whether it is the execution time or only the CPU time. This metric has its caveats. First of all, the resulting time can be affected by external unrelated processes running on the machine that is running the benchmarks. This can result in outliers and imprecise results. Secondly, different testing machines with different hardware characteristics will produce different results. Therefore, the benchmark results are tied to one specific machine and the resulting times only make sense when compared to other results from the same context. Their absolute values do not mean anything when observed separately.

That is why I opted for a more suitable and consistent measure of performance - **number of comparisons**. This is the number of times that any two elements of the input array are compared. It is especially suitable and natural for comparison-based sorting algorithms, in which comparing two elements is the most essential operation. At this point, a natural question arises whether copying or moving the elements is not equally important. Indeed, when using the number of comparison as the metric, any data-moving operations are completely disregarded. However, this is acceptable. In the real-world data, the elements are almost never plain integers or other primitive data types. They are usually much more complicated structures, e.g., database rows with hundreds of columns or measurement data with countless different variables. Comparing such elements might be a very expensive operation in comparison to the data moving operations that are very fast and only work with pointers to these complex data types, instead of actually copying the full data from one memory location to another.

A pleasant property of this metric is its determinism. Assuming that the sorting algorithms themselves are also deterministic, then with the same input, the number of comparisons is always the same, even using different machines to run the benchmarks.

2 Algorithms

2.1 Natural Merge Sort

As was explained before, the adaptive sorting algorithms are about detecting the pre-existing order in the provided input and taking advantage of it. In the case of natural merge algorithms, this pre-existing order is represented by already sorted runs, detecting it means constructing the run profile, and exploiting it means merging the runs in some systematic order.

The concept of natural merging was first rigorously formalised and described in 1973 by D. E. Knuth in [2], although the idea already existed before.

The general high-level idea of Natural Merge Sort (NMS) algorithm can be captured by the pseudocode in Algorithm 2. Implementation details, of course, can be found in the attached code archive.

Algorithm 2 High-level idea of Natural Merge Sort

```
1:  function NMS(arr)
2:      runs ← FindRuns(arr)      ▷ Whole runs, not just their indices
3:      while runs.length() > 1 do
4:          new_runs ← []
5:          for run_1, run_2 in runs do          ▷ Iterate runs pair-wise
6:              merged_run ← merge(run_1, run_2)
7:              new_runs.append(merged_run)
8:          end for
9:          runs ← new_runs
10:     end while
11:     return runs[0]
12: end function
```

On line 2 of the algorithm, `FindRuns` function iterates through the input array and finds its initial run decomposition. This is done using the algorithm already outlined in Section 1.2 with one addition. After finding a whole decreasing run, it is immediately reversed in-place in the original array. This is done out of convenience because at the end of the algorithm, we want to end up with a single run that is increasing.

An alternative approach would be to remember for all runs whether they are increasing or decreasing, and adjust the merging procedure to take this information into account. However, simple reversing is easier to implement. That way all the runs are increasing after line 1. A slightly more optimized implementation of `FindRuns` will be presented later in Algorithm 4. Note that the function returns whole runs rather than just their indices.

Having the run decomposition, the runs start to get merged, which is done pair-wise from left to right. Line 5 iterates through the distinct neighboring pairs of runs, which are merged on line 6. We repeat this process (line 3) until we have only one run left, at which point the whole array is sorted. Therefore, the number of runs halves in each iteration.

I should mention that the version of Natural Merge Sort proposed by Knuth [2] slightly varies in the merging process because instead of merging pairs of adjacent runs, it "burns the candle at both ends" by always merging a run from the left side of the array with the run from the right side. Effectively, this makes no difference.

Notice that if we decided to ignore the pre-existing order in this algorithm by considering all initial runs to be of size 1, each element being a separate run, we would end up exactly with the bottom-up variation of Merge Sort algorithm, which is obviously not adaptive. In the actual NMS, we have to do the extra initial work of finding the run decomposition and reversing the decreasing runs. Secondly, non-adaptive Merge sort generally merges the subarrays of matching size (except the right-most segment if the size of input is not a power of 2), while the length of two runs being merged in NMS can vary significantly, which is suboptimal.

The `merge` operation as we know it works best when merging the arrays of the same size, otherwise it may end up doing a lot of comparisons for little gain. For example, consider the following two calls of the `merge` procedure. The first call merges runs `[1,2,3,4,5,6,7,8,9]` and `[10]`, and the second one merges runs `[1,3,5,7,9]` and `[2,4,6,8,10]`. Both calls perform the same number of comparisons but the first call extends its longer run by only 1 element, while the second one doubles its size. Clearly, merging runs of different sizes can be ineffective, especially with an unfortunate distribution of elements. The more different the run sizes, the greater deficiency it can lead to.

In conclusion, for truly random input arrays with no preponderance of existing order, NMS is not the most efficient algorithm. Under random conditions, the number of initial runs will be $N/2$, with N being the array length, because we have $K_i > K_{i+1}$ with probability $1/2$ [2], and thus the average initial run length is 2, which is not good when we take into account the previously mentioned additional costs of NMS. In other words, we end up doing a lot of extra work, getting almost no compensation in return.

The worst-case time complexity, of course, stays the same but a lot of practical effort lies in optimizing the constants. One logical idea is to try to implement some kind of heuristic that would determine from the input array if it is worth it to continue with NMS or a different, non-adaptive sorting algorithm. However, that has an additional cost as well.

2.2 Timsort

The algorithm described in the previous chapter is not really being used in practice, but its idea was crucial for creation of other modern adaptive sorting algorithms that are being actually practically used for their good performance and properties. One of them is Timsort, named after its creator Tim Peters who introduced it in 2002. He implemented it to be used as a default sorting algorithm in the CPython implementation of the Python programming language [3] and it was being used (after some tuning and further optimizations) since version 2.3 up to 3.11 when it was replaced by Powersort [4]. It is also used as a default sorting algorithm in Java [5], since version Java SE 7 (2011) up to this day (Java SE 20 at the time of writing this thesis).

It also belongs to the family of natural merge algorithms. Its core idea is very similar to the one of Natural Merge Sort, but it introduces many additional practical enhancements and optimizations, which makes it more powerful and also more complicated.

Algorithm 3 presents the high-level implementation of basic Timsort. Similarly to Natural Merge Sort, it starts with the initial run decomposition and ends with some kind of pair-wise merging of the runs. The main complexity lies in the `while` loop starting on line 4. The algorithm maintain a stack S, to which it pushes the runs one

Algorithm 3 Core part of Timsort, inspired by [6, Algorithm 5]

```
1:  function Timsort(arr, MIN_RUN)
2:      runs ← FindRuns(arr, MIN_RUN)                ▷ Whole runs
3:      S ← empty stack
4:      while runs ≠ ∅ do                            ▷ Process all runs
5:          S.push(runs.pop())
6:          while true do                ▷ Maintaining the stack invariants
7:              h ← S.size()
8:              if h ≥ 3 ∧ s1 ≥ s3 then
9:                  Merge(S2, S3)   ▷ Merges runs 2 and 3 together in S
10:             else if h ≥ 2 ∧ s1 ≥ s2 then
11:                 Merge(S1, S2)
12:             else if h ≥ 3 ∧ s1 + s2 ≥ s3 then
13:                 Merge(S1, S2)
14:             else if h ≥ 4 ∧ s2 + s3 ≥ s4 then
15:                 Merge(S1, S2)
16:             else
17:                 break
18:             end if
19:         end while
20:     end while
21:     while S.size() > 1 do                        ▷ Final pair-wise merging
22:         merge(S1, S2)
23:     end while
24:     return S1
25: end function
```

by one. For convenience, the pseudocode uses the following notation: S_i represents i-th topmost run on the stack and s_i represents its length. For example, operation $\texttt{MERGE}(S_1, S_2)$ is just a sugar for $run_1 = S.pop();\ run_2 = S.pop();\ S.push(\texttt{MERGE}(run_1, run_2));$.

After every push, if needed, the algorithm merges some of the top-most runs on the stack (lines 6-19) – more on this part later. After all the runs are processed, it finishes up with gradually merging two top-most runs on the stack until only a single run remains (lines 21-22), and thus the whole array is sorted.

Merge policy

As was already explained in Section 2.1, merging runs with drastically different sizes leads to the performance degradation of the merge procedure. Therefore, the aim is to merge the arrays of similar size in order to achieve maximum efficiency.

That is the goal of the four conditions on lines 8-15 in Algorithm 3 that are not so intuitive at the first glance. They are establishing that the four runs on top of the stack always hold certain invariants that cause the run lengths to be balanced as closely as possible. If any of those invariants is broken, the corresponding rule triggers merging on-the-fly to restore them. Hence, the final merging is always performed in a favorable order. Additionally, this gradual merging process keeps the stack small (in fact, logarithmic [3, 6]), and hence reduces the number of runs the algorithm has to remember at any given point.

With random input data, all runs are likely to have their length equal to the `MIN_RUN` constant (more on that in 2.2.1), achieving a perfectly balanced run profile, with the only exception possibly being at the very end. [3]

Stability

One observation worth mentioning is that Timsort is a stable sorting algorithm, which means that it preserves the original order of the elements with equal value. That can be observed from the fact that we always merge carefully only the adjacent pairs of runs, and the merging procedure itself is stable by default. An interesting fact is that CPython's sorting algorithms before Timsort's introduction in

2002 (before version 2.3) were variations of Quicksort that were all famously not stable.

Time complexity

When Timsort was first introduced in 2002, Tim Peters stated its worst-case time complexity to be $\mathcal{O}(n * logn)$ without a proof. The analysis of the merging process is not obvious and was properly proved only in 2015 in [7]. However, this proof is out of the scope of this work.

2.2.1 Optimizations

Because Timsort is used in the reference implementations of Python and Java, it is not surprising that it was an object of extensive optimization studies over the years, using various engineering techniques aiming to improve its performance constants in practice. As a result, numerous variations of the algorithm exist, differing in the number of implemented optimizations. In this section, I will focus only on the most significant of them, those that define Timsort the most.

In the implementation part of this thesis, I have implemented the following ones: minimal run length, binary insertion sort, and galloping mode.

Minimal Run Length

The most common optimization for Timsort is to set the minimal size for runs and use Insertion sort to keep them ordered.

There are several reasons to do this. The first goal is to achieve a more balanced profile of runs in terms of their size, for reasons discussed previously. Secondly, Insertion Sort is an effective algorithm for small arrays, and also for nearly sorted arrays, which goes in hand with the whole idea of exploiting the natural order of input. On the other hand, Merge Sort is not the most effective for small arrays, because of its overhead in the merging process and recursion. This concept of combining multiple sorting techniques is called hybrid sorting.

The minimal run size is often fixed as 32, which has empirically proven to be a good threshold for using Insertion Sort. Because it is a power of 2, it also aligns well with the implementations of CPU and

memory architectures, namely for caching purposes, likely reducing the low-level overhead. As written by Tim Peters himself,

> *When N is a power of 2, testing on random data showed that minrun values of 16, 32, 64 and 128 worked about equally well. At 256 the data-movement cost in binary insertion sort clearly hurt, and at 8 the increase in the number of function calls clearly hurt. Picking *some* power of 2 is important here, so that the merges end up perfectly balanced* [...]. *We pick 32 as a good value in the sweet range; picking a value at the low end allows the adaptive gimmicks more opportunity to exploit shorter natural runs.* [3]

Algorithm 4 demonstrates how the Insertion sort is utilized in the process of constructing the run profile, with `MIN_RUN` being the input constant for the minimal size of runs. The function `FindNextRun` abstracts away the logic of detecting the next ascending/descending sequence, which is trivial but too technical, returning the ending index of the next natural run.

Algorithm 4 Using Insertion sort to enforce minimal run size

```
1:  function FindRuns(arr, MIN_RUN)
2:      runs ← []
3:      i ← 0
4:      while i < arr.length() do
5:          j ← FindNextRun(arr, i)                    ▷ end index
6:          size ← j − i + 1                ▷ size of the next natural run
7:          if size < MIN_RUN then
8:              j ← min(i + MIN_RUN − 1, arr.length() − 1)
9:              ▷ Consider indices [i : i + size] to be already sorted
10:             InsertionSort(arr, i, i + size, j)  ▷ extend to MIN_RUN
11:         end if
12:         runs.append(arr[i : j])
13:         i ← j + 1
14:     end while
15:     return runs
16: end function
```

The only new addition to the function, when compared to NMS, is the `if` block on line 7 that extends the run if it is too short. More precisely, if the size of the natural run is N, the algorithm uses Insertion sort to insert the remaining (`MIN_RUN` - N) elements (paying attention to the array bounds) into their correct position in the run. The `InsertionSort` function called on line 10 is adapted to consider the first `size` elements of the input subarray (from index `i` to `i+size`) already sorted, but otherwise is the same as the traditional Insertion Sort.

Binary Insertion sort

Timsort is a hybrid sorting algorithm because it combines the ideas of Merge sort and Insertion sort to make the best out of both of them.

An additional enhancement to this is to use the Binary Insertion sort, which is an alternation that first finds the insertion point for the element using binary search and then performs the necessary swaps but without the need for further comparisons. Of course, its time complexity remains $\mathcal{O}(n^2)$ but it is likely to decrease the number of needed comparisons, improving the constant factor. In summary, it does not improve the number of swaps, which is still $\mathcal{O}(n^2)$ in the worst case, but it improves the number of comparisons from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$.

The overall procedure outlined in Algorithm 4 remains the same, except the adaptation of `insertion_sort` function to incorporate the binary searching.

Galloping mode

Galloping is a heuristical technique used during the merge step to try and skipjump over multiple elements of one subarray under "suspicion" that there follows a long subsequence in it with all of its elements smaller than the next element from the other subarray. For example, when merging runs [1,2,3,4,6] and [5,7,8,9,10], it would be very useful instead of simply comparing and picking elements one at a time as we are used to, to be able to jump over 4 elements([1,2,3,4] at once (as they are all lower than 5), saving 3 comparisons.

This feels a bit like cheating and, of course, not always works well. Because this approach is heuristical, it can happen that it makes a wrong decision, resulting in extra comparisons. Therefore, galloping is not guaranteed to be efficient.

The heuristic keeps track of the minimum galloping threshold, which is a number of elements that must be picked from the same run consecutively in order to trigger the galloping mode. Successful galloping decreases the galloping threshold, whereas committing a mistake increases it. Tim Peters himself conducted some benchmarks in order to justify the use of galloping and determined that the best initial value for the galloping threshold is 7 [3].

After entering the galloping mode, the algorithm searches for the correct index in the run that triggered galloping, to which the next element from the other run should be correctly inserted. In other words, it determines exactly how many elements can be skipped. This is done in two steps. First, exponential search is used to find an interval with size 2^k, in which the target index resides. Then, the exact index is found using binary search in this interval. The natural question to arise is why not use the binary search straight away. This two-step search is a technique that is known to be more efficient if the searched element is expected to be reasonably close to the beginning of the array, but not too close for the linear search to be more effective. This is likely to be the case also here. When galloping mode is triggered, it means that the algorithm expects to be able to skipjump over some elements, but probably not *too* many because in the real world, such data would be probably very unlikely.

Galloping can be particularly effective if the runs exhibit some degree of reasonably long interval patterns. It is debatable how specific this use case is, but it is safe to say that real-world data contains many more patterns and regularities than random data.

For the sake of simplicity, the galloping logic was omitted from the pseudocodes but is fully implemented in the attached source codes.

In-place merging

The traditional Merge Sort is not merging in-place but requires $\mathcal{O}(n)$ extra auxiliary space to store the intermediate arrays. There are also known variations that *are* in-situ but with some time overhead, for

example the algorithm by Huang and Langston [8] with modest $\mathcal{O}(1)$ extra time. This can be useful in some special environments with limited space, but can prove useless otherwise due to its high constant time factors. That is why it is considered impractical for most real-world cases.

For Timsort, Tim Peters proposed and decided to use a merging algorithm [3] acting as a compromise, as it is not completely in-place, but for a constant time overhead reduces the auxiliary space needed by `merge` procedure to $\mathcal{O}(min(n, m))$, with n and m being the sizes of the subarrays to merge. It is supposed to strike a balance between performance and memory efficiency.

Minrun calculation

Earlier I have stated that the `MIN_RUN` size is often fixed as 32, which is not completely true. It is a good choice, but it was discovered that in some cases, it might be suboptimal. Tim Peters described the intuition behind this, leading to the following refinement:

> *Instead we pick a minrun in range*$(32, 65)$ *such that* $N/minrun$ *is exactly a power of 2, or if that isn't possible, is close to, but strictly less than, a power of 2.* [3]

2.3 Powersort

The main complexity of natural merge algorithms lies in determing what order should the runs be merged in. This is called **merge policy**. The optimal merging order would leverage the benefits of adaptive sorting to the maximum. However, a bad merge policy can cause even quadratic slowdown.

Timsort's merge policy is based on the 4 rules maintaining the defined invariants on the top of the run stack. The problem is that it has a "bad case" (also named "Timsort-drag" in [4]) – a specific input run pattern, with which these merge rules are not optimal and they perform some bad merges, resulting in as much as 50% higher merge cost when compared to the optimal standard Merge Sort [9]. This, along with the complexity, lack of transparency, and difficulty

of analysis of Timsort's merge policy drove algorithm developers to search for something better.

This lead to the dawn of Powersort – yet another natural merge algorithm, which uses a different merge policy. As was mentioned earlier, Powersort replaced Timsort in CPython since the version 3.11.

2.3.1 Merge trees

The order in which the runs are merged for a specific input can be visualized in the form of a merge tree, in which the leafs correspond to the initial runs, the internal nodes to the intermediate merged runs, and the root itself to the final sorted array. Let's define the cost of merging as the sum of lengths from leaves to root, i.e., the external path length. Our goal is to minimize this cost. In this section, I decided not to go into too much theoretical and technical detail for the sake of keeping the scope of this work reasonable. Instead, I review just the basic ideas and the algorithm, and provide its implementation.

It appears that the problem of finding the merge tree with the minimal cost is equal to the problem of finding the optimal binary search tree [4]). There is a well-known dynamic programming solution for this problem running in quadratic time. Unfortunately, that is too slow for a sorting algorithm.

However, there are also linear-time algorithms to compute "nearly-optimal" BST – one with a cost very close to the minimal cost. Powersort uses exactly this concept and it performs very well in practice. More specifically, to calculate such nearly-optimal BST, it uses bisection method, which is quite technical and was described in more detail by Mehlhorn [10]. In short, it recursively keeps looking for run boundaries closest to the middle, which approximates the tree to be nearly perfectly balanced.

2.3.2 Algorithm

Similarly as in Timsort, Powersort (Algorithm 5) iterates through the intervals from left to right, pushing them to the stack (same as in Timsort), and applying the merge policy. The important thing is that the merging tree constructed by the merging policy of Powersort is nearly-optimal w.r.t the given input. During this iterative merging

process, it keeps computing so-called **power** (hence Powersort) of the boundary between adjacent pairs of intervals (also can be interpreted as the power of their parent node in BST), more on that in 2.3.3. Powersort merge policy has a single rule (as opposed to Timsort's 4 non-trivial rules) – the power on the stack is always increasing.

This process essentially simulates the construction of a nearly-optimal BST, thus achieving a nearly-optimal merging order. Importantly, it does not store the entire BST but merely traverses its structure. Also, in this case, the runs are found on-the-fly rather than found and stored beforehand, as in Timsort. This reduces the space overhead of the algorithm as much as possible.

Algorithm 5 Powersort pseudocode, inspired by [4, Algorithm 2]

```
1:  function PowerSort(arr, MIN_RUN)
2:      X ← empty stack                          ▷ stack for runs
3:      P ← empty stack                          ▷ stack for powers
4:      s_1 ← 0                                  ▷ run 1 start
5:      e_1 ← FindNextRun(arr, s_1, MIN_RUN)     ▷ run 1 end
6:      while e_1 < arr.length() − 1 do
7:          s_2 ← e_1 + 1                        ▷ run 2 start
8:          e_2 ← FindNextRun(arr, s_2, MIN_RUN) ▷ run 2 end
9:          p ← Power(s_1, e_1, s_2, e_2, arr.length())   ▷ p is integer
10:         while P.top() ≥ p do    ▷ powers in P must be increasing
11:             (s_1, e_1) ← Merge(arr, X.pop(), (s_1, e_1))
12:             P.pop()
13:         end while
14:         X.push((s_1, e_1))
15:         P.push(p)
16:         s_1 ← s_2
17:         e_1 ← e_2
18:     end while
19:     while X ≠ ∅ do
20:         (s_1, e_1) ← Merge(arr, X.pop(), (s_1, e_1))
21:     end while
22:     return arr
23: end function
```

The pseudocode of Algorithm 5 starts by initializing the stacks for runs (X) and for powers (P). The `while` loop on the line 6 implements the process described above. In a single run, it finds and examines pairs of adjacent runs ((s_1, e_1) and (s_2, e_2)), calculates the power of their boundary, and maintains the invariant of the merge policy, keeping the powers on the stack in a strictly increasing order. That is done on line 11 by merging the top-most run of X with the next found run (s_1, e_1), until the resulting run has a higher power than the new top-most run, and can be pushed back to the stack.

This works because merging two intervals is guaranteed to create a bigger run with power equal to the smaller of the two powers, because we are moving bottom-up (leaf-to-root, to a smaller power) in the logical BST hierarchy.

Note that in this pseudocode, for convenience, the `MERGE` function is slightly adapted from Algorithm 1. Instead of taking whole runs as arguments and simply returning the merged run, it takes the indices of the runs as arguments, along with a reference to the original array, merges these runs by directly modifying the array, and returns the indices of the merged run.

The final `while` loop on line 19 merges the remaining pairs of runs on the stack until finally the entire array is sorted.

Note that all the optimizations described for Timsort in Section 2.2, including `MIN_RUN` and the insertion sort, can be used for Powersort in the same way. On the high level, the two algorithms differ only in their merge policy.

One interesting property of Powersort (and also Timsort) is that it has a good potential to work well with CPU caches. Because it scans the input in a way such that it only works with the top-most run of the stack and the most recent detected run, it has a good chance that the latter (or even both) still resides in cache, and thus its retrieval is faster.

2.3.3 Power

For any two runs, the power of their boundary essentially represents the depth that their merge has in the BST. The smaller the power, the higher in the tree the merged run would reside. From this, combined with the fact that the powers are strictly increasing in the stack, we can

conclude that the size of the stack is logarithmic with respect to the number of runs, because $\lfloor log(n) \rfloor + 1$ is the height of the optimally balanced tree.

Without continuing too deeply, I only attach pseudocode (Algorithm 6) for the calculation of power from Munro&Wild's work [4] that also contains its formal definition, which is out of scope of this thesis.

Algorithm 6 Calculating the power of an interval boundary

```
1:  function Power(s_1, e_1, s_2, e_2, n)
2:      length_1 ← e_1 − s_1 + 1
3:      length_2 ← e_2 − s_2 + 1
4:      l ← 0
5:      a ← (s_1 + length_1/2 − 1) / n
6:      b ← (s_2 + length_2/2 − 1) / n
7:      while ⌊a ∗ 2^l⌋ = ⌊b ∗ 2^l⌋ do
8:          l += 1
9:      end while
10:     return l
11: end function
```

3 Benchmarks

The experimental part of this work consists of defining and implementing a set of performance benchmarks for the selected algorithms and interpreting the results. Based on the information presented in the previous chapters, the experimental results are expected to confirm that the algorithms perform better on inputs that with pre-existing order.

As was explained in Section 1.2, there are many different ways to measure pre-sortedness of an array. For these benchmarks, we focus on two measures in particular: number of runs and entropy of the run profile.

For the implementation of the benchmark scripts, I used Python with packages like numpy and matplotlib to generate graphs from the obtained results. The scripts also save the same results as raw data to `csv` files.

The algorithms had to be slightly modified to enable their benchmarking, particularly to count the comparisons. Each algorithm keeps a comparisons counter, increasing it every time two array elements are compared during their runtime, whether it's in the `merge` procedure or during the discovery of runs. All algorithms have their separate benchmark version in the source code archive.

The code also implements support for measuring the execution time of the algorithms, but this performance metric was not included in the benchmarks.

3.1 Random Input Generators

The input arrays for benchmarking purposes are generated randomly. However, of course, generating completely random arrays wouldn't prove much. The implemented random input generators are able to generate random inputs while being able to control some of its properties.

Input length

Naturally, the first of these properties is the length of the array (N), because we want to collect results and prove the point for inputs of arbitrary lengths.

Number of runs

Next, because the experiments focus on presorted inputs w.r.t. the number of runs, it was necessary to implement a way to generate arrays with a specific number of runs, provided that this number is valid, i.e., from range $[1, \lfloor \frac{N}{2} \rfloor]$ as the minimal run length is 2. This is done in Algorithm 7.

The function is parametrized by N and K, along with *min* and *max*, which determine the bounds for values in the array. First, it generates a random run profile for the array. Based on this profile, it creates the array by concatenating K runs one by one, using function `generate_random_run` to generate them. It first randomly decides whether the run will be increasing or decreasing. Then it generates the elements from the provided range in an increasing order, while taking into account the special bounds for the first element. That is because finishing one run imposes a special condition on the first element of the next run. Specifically, it has to be either lower than the last element of the previous run if the previous run was increasing, or higher if it was decreasing. The pseudocode abstracts this functionality away for the sake of conciseness. For a working implementation, there are a few other caveats and edge cases to consider. For example, it should be handled that not all elements in the newly generated run are the same, because then the next run would inevitably extend the current one, changing the run profile.

Entropy of the run profile

The same goes for the entropy of the array's run profile (hereafter also just "entropy"). Unlike for runs, we generally don't want to be able to generate arrays with an exact entropy value. That is because, first of all, a run profile with such exact entropy value might not even exist because for any fixed N there is a finite number of possible profiles, but entropy by definition is a real number and comes from an infinite

Algorithm 7 Generating random array with the provided number of runs

```
1:  function GenerateRandomArray(N, K, min, max)
2:      profile ← random array with length K and sum N
3:      array ← []
4:      last_run_increasing ← false
5:      first_min ← min ▷ Lowest possible value for the 1st element
6:      first_max ← max ▷ Highest possible value for the 1st element
7:      for i in [0..K] do
8:          if i ≠ 0 then        ▷ No 1st element bounds for the 1st run
9:              if last_run_increasing then
10:                 first_min ← min
11:                 first_max ← array.last() − 1
12:             else
13:                 first_min ← array.last() + 1
14:                 first_max ← max
15:             end if
16:         end if
17:         run, last_run_increasing ← GenerateRandomRun(
                profile[i], min, max, first_min, first_max
            )     ▷ Generate a new run with the specified parameters
18:         array += run          ▷ Append the generated run to the end
19:     end for
20:     return array
21: end function
```

interval. But even if we assume that the provided entropy is "valid", this value doesn't mean anything without a context, which is the length of the array (N) and the number of runs (K). For different arrays, the same value of entropy can mean different things. For $K = 100000$, $H = 4$ would mean that the run profile is very skewed, while for $K = 16$ it means a completely uniform run profile.

Instead of classifying arrays based on the exact value of entropy $H(P)$ of their run profile (P), they can be classified more generally, based on the normalized "distance" $(h(P))$ of this value from the minimal and the maximal possible entropy out of all run profiles with the same N and K. Let's call this distance normalized entropy.

The maximal possible entropy occurs when the run profile is most uniform: $P_{max} = (r_1, ..., r_K) \mid \forall i, j \in [1, K] : |r_i - r_j| \leq 1$. On the other hand, the minimal entropy is when the run profile is most skewed, i.e., $P_{min} = (r_1, ..., r_K) \mid r_1 = N - 2 * (K - 1) \wedge \forall i \in [2, K] : r_i = 2$. Having this, we can define $h(P) = \frac{H(P) - H(P_{min})}{H(P_{max}) - H(P_{min})}$, where it can be easily seen that $h(P) \in [0, 1]$.

This deserves to be shown on an example. For input `[1,2,3,4,1,2,3,4,5,6,7,1,2,3,1,2]` we have $N = 16, K = 4, P = (4, 7, 3, 2), P_{min} = (10, 2, 2, 2), P_{max} = (4, 4, 4, 4), H(P_{min}) \approx 1.55, H(P_{max}) = 2, H(P) \approx 1.85$, and finally, $h(P) \approx 0.66$.

Accepting (N, h_{min}, h_{max}) as arguments, the implemented input generators can generate random arrays with $h(P) \in [h_{min}, h_{max}]$ using the same algorithm as was described in Section 3.1 with the only difference being in the generation of the run profile. Instead of generating a completely random profile, Algorithm 8 is used.

It starts with selecting a random number of runs (K) for the profile. One might consider this odd because we want to show how the sorting algorithms perform with different levels of presortedness, while with this line of code, the average K is $N/4$, thus an average run length is always 4. However, with this generator we will use entropy as the measure of presortedness and the number of runs is irrelevant. The next step is calculating H_{min} and H_{max} for run profiles with the given N and K. Then, we start generating run profiles with the given N and K in such a manner that their entropy is increasing. For each one, calculate absolute (H) and normalized (h) entropy. If h is from the desired range, the run profile is returned.

Algorithm 8 Generating random run profile with normalized entropy from the provided range

function GenerateRunProfile(N, h_{min}, h_{max})
    $K \leftarrow$ `random integer from [2,` $\lfloor \frac{N}{2} \rfloor$`]`
    $H_{min} \leftarrow$ CalculateMinEntropy(N, K)
    $H_{max} \leftarrow$ CalculateMaxEntropy(N)
    for $profile$ in GenProfilesWithIncreasingH(N, K) **do**
        $H \leftarrow$ CalculateEntropy($profile$)
        $h \leftarrow (H - H_{min})/(H_{max} - H_{min})$
        if $h_{min} \leq h \leq h_{max}$ **then**
            return $profile$ ▷ The first profile from desired range
        end if
    end for
end function

Algorithm 9 Calculating entropy of a run profile and entropy bounds

function CalculateEntropy(P)
    $N \leftarrow P.sum()$
    return $-\sum_{r \in P} \frac{r}{N} \, log_2 \frac{r}{N}$
end function

function CalculateMaxEntropy(K)
    return $\log_2 K$
end function

function CalculateMinEntropy(N, K)
    ▷ Min entropy is when the profile is most skewed: [X,2,2,...,2,2]
    $x \leftarrow N - 2 * (K - 1)$ ▷ Value of X (remainder)
    $n2 \leftarrow 2/N$ ▷ Normalized value of 2
    $nx \leftarrow x/N$ ▷ Normalized value of X
    $e2 \leftarrow -n2 * \log_2(n2)$ ▷ Entropy "contribution" of a single 2
    $ex \leftarrow -nx * \log_2(nx)$ ▷ Entropy "contribution" of X
    return $ex + e2 * (K - 1)$ ▷ Total entropy
end function

Algorithm 10 Producing run profiles with increasing entropy

```
1:  function GenProfilesWithIncreasingH(N, K)
2:      profile ← [2] * K
3:      profile[0] ← N − 2 * (K − 1)
4:      yield profile
5:      i ← 1
6:      while profile[0] > profile[i] do
7:          profile[i] += 1
8:          profile[0] -= 1
9:          yield profile
10:         i ← i % (K − 1) + 1
11:     end while
12: end function
```

Algorithm 9 implements the calculations related to entropy. To calculate the entropy of a run profile, it uses the formula from Chapter 1. As discussed before, the highest possible entropy for a run profile with K runs occurs when the profile is completely uniform – all numbers having the same value – and is $log_2(K)$ by definition. The lowest possible entropy occurs when the profile is as far from uniform as possible: $[X, 2, 2, ..., 2, 2]$. Calculating the entropy of such profile follows the same idea as before – combining the contribution of every individual element towards the total entropy. The `CalculateMinEntropy` function leverages the fact that the element 2 is present $K - 1$ times in the profile. Alternatively, we could replace lines 10-14 with simply constructing the described profile and passing it to the `CalculateEntropy` function defined above. However, this would increase the time complexity of the function to linear.

Algorithm 10 describes how the profiles are generated so that their entropy increases. It's important to note that this process is deterministic and does not generate **all** possible profiles for given N and K. It generates a subset of them and it guarantees that they are returned sorted by their entropy from the lowest to the highest. The function operates as a generator in a Python-like manner, producing profiles one at a time using the yielding mechanism. First step (Lines 2-3) is to construct the most skewed profile possible, i.e., with the lowest entropy. Such profile consists of all runs containing the lowest possi-

ble number of elements (2) except one run that contains all the rest. Each iteration of the loop (Lines 5-10) creates the next run profile by moving one element from the biggest run to the smallest one, which makes the next profile less skewed and more uniform.

Although not being an explicit requirement, a possible improvement to make this process more random would be to not select the first satisfying run profile, but to collect all of them and select a random one. Otherwise, the h of the profile returned in Algorithm 8 will always be only very closely above h_{min}. This improvement has been implemented in the benchmark source codes but I left it out from the pseudocode for the sake of simplicity.

3.2 Scope Definition

After implementing the random input generators, the next step is to define the benchmarks themselves. The benchmarks measure the performance of the following algorithms:

1. Natural Merge Sort (Section 2.1)
2. Timsort (Section 2.2), including the following optimizations, as described in Section 2.2.1: minimal run length, binary insertion sort, and galloping mode
3. Powersort (Section 2.3), including the same optimizations as for Timsort
4. Python reference sorting algorithm (invoked by calling `sorted()` or `.sort()`), which is an implementation of Powersort with many additional low-level optimizations
5. Merge Sort, which will serve as the baseline to compare other algorithms with

Using traditional top-down Merge Sort as the baseline makes sense because it is a non-adaptive algorithm that does not consider presortedness. That comparison will clearly highlight the performance differences between the adaptive algorithms and a non-adaptive one.

Of course, it is crucial to ensure that all these algorithms are tested with the same inputs throughout the benchmark. Each time an input

array is generated, it is used for **all** the algorithms, and the results are collected and grouped together for further analysis.

Counting comparisons of the Python reference sorting algorithm

Listing 1 shows a Python-native way of counting the comparisons made by the Python reference sorting function without needing to access or modify its implementation. On line 34, each element of the input array is wrapped in a `Comparable` class that overrides all of its comparison-related magic methods (`__gt__`, `__le__`, etc.) to additionally increment a static counter. Therefore, after the sorting is finished, this counter reflects the number of times these magic methods were called across all elements.

```python
class Comparable:
    # Static variable to track the number of comparisons
    comparison_count = 0

    def __init__(self, value):
        self.value = value

    def __lt__(self, other):
        Comparable.comparison_count += 1
        return self.value < other.value

    def __le__(self, other):
        Comparable.comparison_count += 1
        return self.value <= other.value

    def __gt__(self, other):
        Comparable.comparison_count += 1
        return self.value > other.value

    def __ge__(self, other):
        Comparable.comparison_count += 1
        return self.value >= other.value

    def __eq__(self, other):
        Comparable.comparison_count += 1
        return self.value == other.value

    def __ne__(self, other):
        Comparable.comparison_count += 1
        return self.value != other.value

def run_python_sort_for_comparisons(arr):
    Comparable.comparison_count = 0
    wrapped_arr = [Comparable(x) for x in arr]
    sorted(wrapped_arr)
    return Comparable.comparison_count
```

Listing 1: Counting comparisons of the Python reference sorting function

4 Experimental evaluation

The following sections present three different sets of experiments, each focusing on a different category of inputs. All these categories cover arrays of varying lengths, generally ranging from 100 to 1000000, in order to assess whether the results are affected by input size, which serves as the variable plotted on the X-axis. Generating inputs for every possible input size within this range would be computationally very expensive. Therefore, only every x-th value is selected, with x progressively increasing, starting with "jumps" of 10 for sizes 10-100, then 100 for sizes between 100-1000, followed by 1000 for size 1000-10000, and so on. To conveniently visualize this, the X-axis is displayed on a logarithmic scale (log_{10}), as the data points for small sizes would otherwise be too close to each other and difficult to distinguish. Additionally, rather than generating a single array for each selected size, multiple samples (10 by default) are generated to reduce dispersion and eliminate outliers.

The Y-axis displays the number of element comparisons performed during the sorting process, which is the key metric that we are measuring. The reasoning behind choosing this performance measure was explained in Section 1.3. The absolute number of comparisons can be a bit difficult to interpret visually. That is why instead, as was already hinted earlier, we calculate the difference in the number of comparisons from the baseline algorithm (Merge Sort). To make the results more digestible, we express it as a percentage instead of the absolute difference. A positive value indicates an increase in performance, while a negative value means a decline.

To further minimize variability and provide a clearer insight, a second layer is added with smoothed out curves. This smoothing was achieved using the polynomial fitting method provided by the NumPy package, which fits a polynomial of a certain degree to the data points using the least squares fitting, minimizing the mean square deviation. This layer is displayed in opaque colors, while the raw results are more transparent.

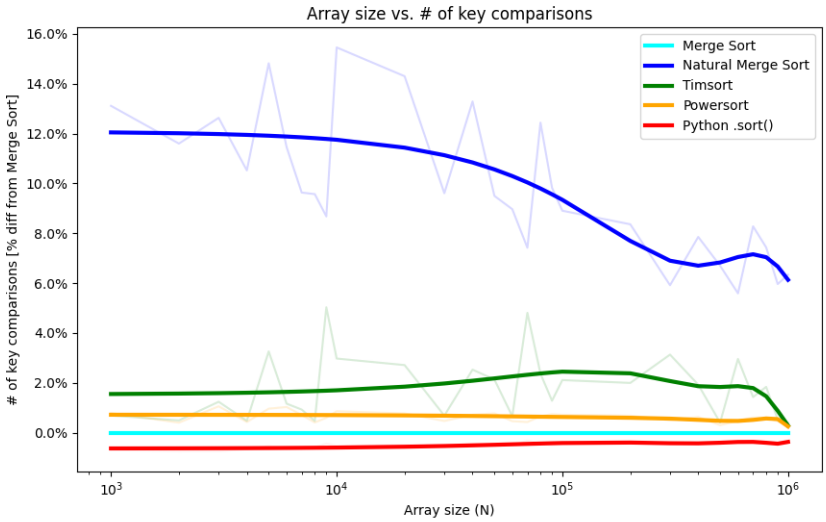

Figure 4.1: Benchmark on arrays without any presortedness

4.1 Results on random arrays

The first benchmark works on completely random inputs, without considering any presortedness. This is done to demonstrate that in such case, the adaptive algorithms generally perform **worse** than the non-adaptive baseline. That is completely logical because they perform extra operations that aim to exploit the pre-existing order. If there is not any, then these operations are just an overhead.

This can be clearly seen on Figure 4.1. The performance overhead in terms of extra comparisons is not big but it's clearly there. The only algorithm with a slightly favorable result is the native Python sort, likely due to its extensive optimizations. Natural Merge Sort suffers from the biggest computational overhead.

4.2 Results based on the number of runs

The next benchmark evaluates the performance of the algorithms on inputs with varying levels of presortedness, using the number of runs (K) as its measure. The inputs are grouped in 3 categories:

1. random: $K = N/2 \implies$ the average run length is 2
2. presorted: $K = N/50 \implies$ the average run length is 50
3. heavily presorted: $K = N/500 \implies$ the average run length is 500

Figure 4.2 shows the results for the first category. This experiment, where K equals to **exactly** $N/2$, is very similar to the one on Figure 4.1, where $K \approx N/2$ on average (completely random array has an average run length of 2). As expected, we can see that the graph looks almost identical.

The experiment with $K = N/50$ incorporates more presortedness into the inputs, and Figure 4.3 clearly shows that it affects the performance of all the adaptive algorithms significantly, by tens of percents. It can also be seen that the results for smaller arrays are more volatile and they stabilize with the increasing input size. One reason for that is the `MIN_RUN=32` constant that can have a big impact especially when the size of the input is relatively small.

This is taken one step further using the third group of inputs with $K = N/500$, where the performance boost is even bigger, ranging from around 30% to 50%, as can be seen on Figure 4.4.

Comparing the adaptive algorithms between themselves, they all perform quite similarly on presorted data, with a slight edge for the Python `.sort()` and Powersort.

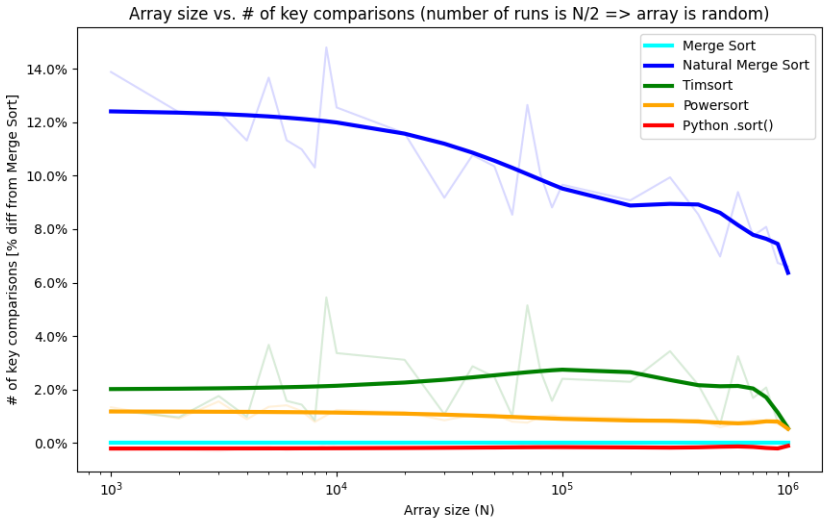

Figure 4.2: Benchmark on arrays with K=N/2 (least presorted)

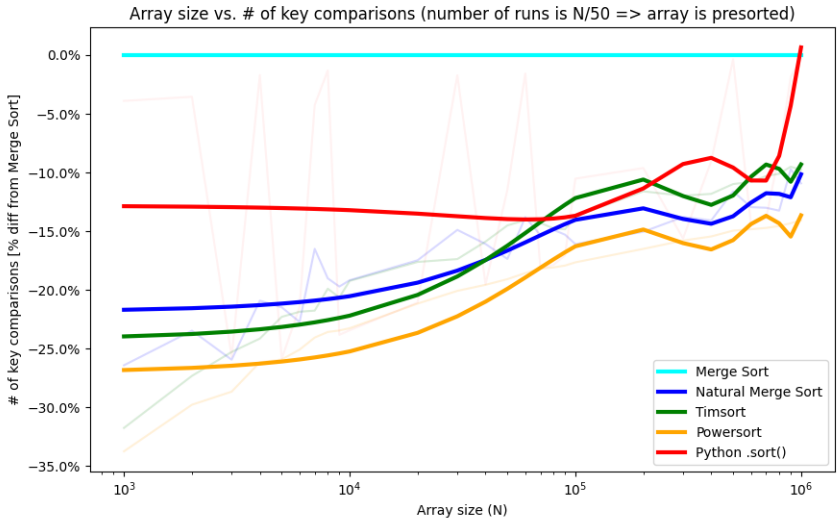

Figure 4.3: Benchmark on arrays with K=N/50 (presorted)

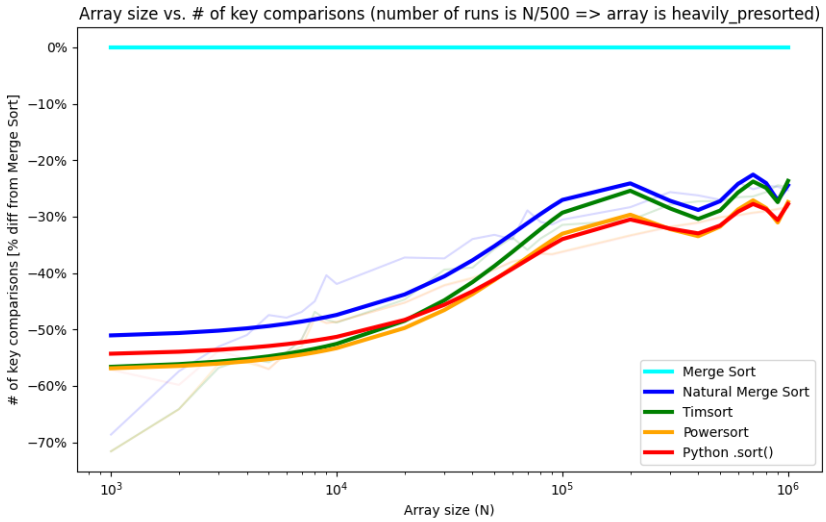

Figure 4.4: Benchmark on arrays with K=N/500 (heavily presorted)

4.3 Results based on the entropy of the run profile

The same approach is followed for the benchmark with the entropy being the measure of presortedness. It is important to note that the number of runs is random in these experiments. We are only interested in the performance impact of the entropy. We again define 3 input classes based on the entropy characteristics of their run profile:

1. heavily uniform: $h \in [0.9, 1.0]$
2. non-uniform: $h \in [0.4, 0.6]$
3. very skewed: $h \in [0.1, 0.2]$

Figure 4.5 shows that when the run profile is very close to uniformity, the performance change is almost negligible.

The second category tests more presorted inputs, which means lower entropy and a more skewed run profile. The results in Figure 4.6 indicate a performance increase of around 5% to 15%.

This is taken further with the third category, represented by very skewed run profiles. Here, the performance increase is even higher, all the way to 30% in some cases. Unlike in Section 4.2, the differences between individual algorithms can be seen more clearly here. Timsort and Powersort perform best, with NMS stagnating behind.

The reason for that is that in skewed run profiles, the differences in size between individual runs are generally higher. Timsort and Powersort take that into account and have their merge policies designed to merge the runs of similar size. On the other hand, the merge policy of NMS is primitive and always merges left-to-right, not considering the size of runs. This can cause short runs to merge with much longer ones, adding many extra comparisons.

Then one could ask how it is possible that NMS still has a stable $\approx$ 5% performance increase in all categories. This is caused by the nature of the experiment – as was mentioned earlier, average run size is 4 instead of 2, which means the input is always slightly presorted w.r.t the number of runs, which increases performance, as was demonstrated in Section 4.2. This is not a flaw but a desired behavior because if the average run size was close to 2, then it would be impossible to generate arrays with lower levels of entropy.

As it turns out from experiments, lower entropy has a positive impact on the performance of natural merge algorithms. I consider it useful to also explain the intuition behind this, as it may seem somewhat counterintuitive. It was mentioned repeatedly that it's better to merge runs with similar size. However, lower entropy generally means that the sizes of runs are *less* similar. It is important to realize here that a skewed run profile does not imply that the bigger runs will be getting merged with smaller ones more often. Smaller runs can be merged together with other smaller runs, then merged with bigger runs, and so on. On the other hand, compared to a uniform run profile, in a skewed one there are generally longer chunks of array that are already sorted, thus saving on initial comparisons, and those smaller chunks have a higher probability of falling under the `MIN_RUN` constant, thus being sorted more efficiently by Insertion Sort. Therefore, we can conclude that decreasing the entropy is indeed likely to decrease the number of performed comparisons and that the experimental results make sense.

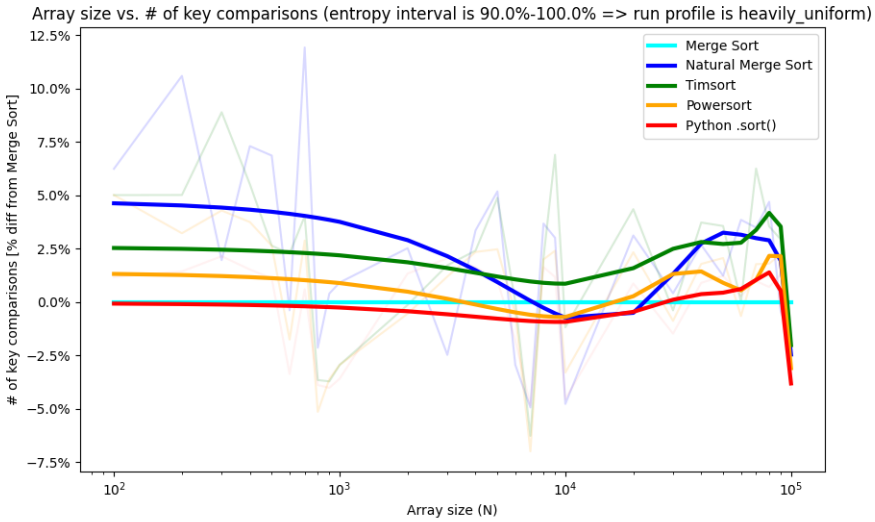

Figure 4.5: Benchmark on arrays with entropy 90%-100% (heavily uniform runs)

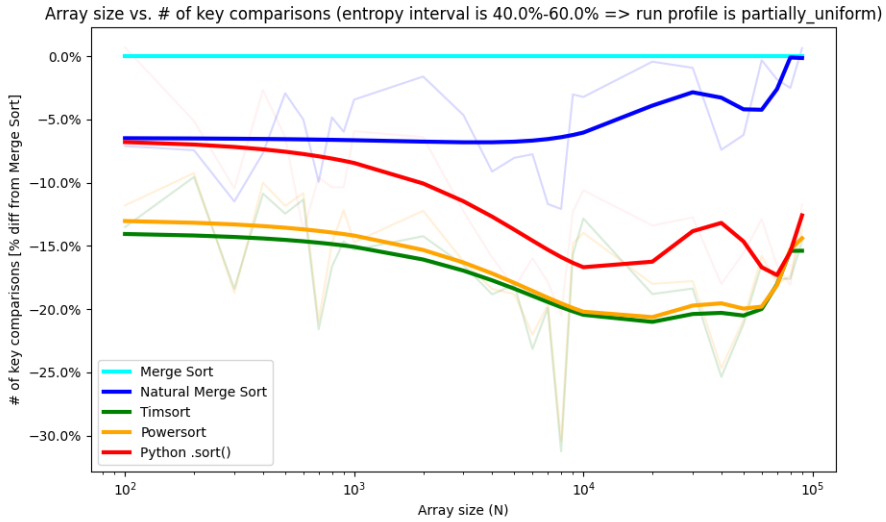

Figure 4.6: Benchmark on arrays with entropy 40%-60% (non-uniform runs)

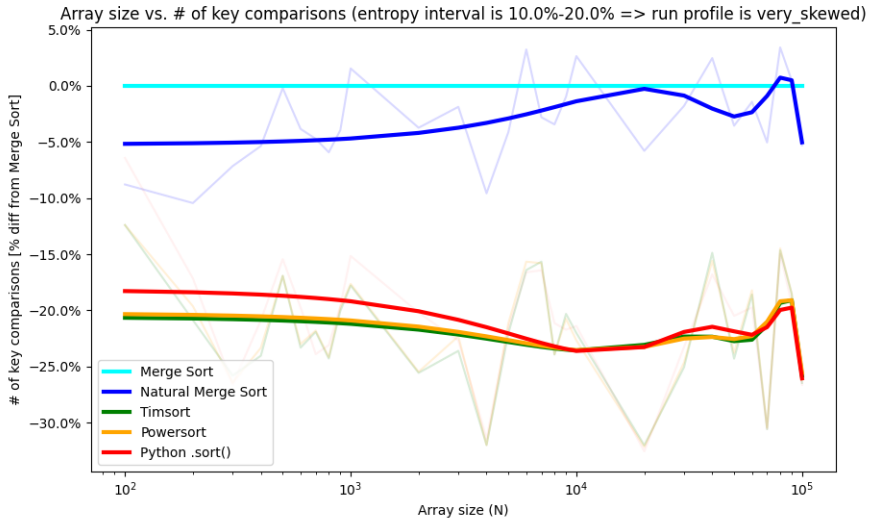

Figure 4.7: Benchmark on arrays with entropy 10%-20% (very skewed)

4.4 Performance impact of the optimizations

This section contains two additional demonstrative measurements that show the impact of the optimizations discussed in Section 2.2.1. The optimizations are used both in Timsort and Powersort, and the experiments in this section examine their impact in Powersort.

Figure 4.8 shows that the `MIN_RUN` optimization does indeed improve the performance. The experiment examines two variants of Powersort: one that uses Insertion Sort for short runs to enforce the minimum run size of 32 and another that does not implement this optimization. The results show that on random arrays, the optimization boosts the performance by approximately $5 - 10\%$, which is a notable improvement.

Figure 4.9 does the same for the galloping optimization. It considers three Powersort variants: one that does not use galloping mode, one that uses it with a static galloping threshold of 7, and one that also dynamically adjusts the threshold. With this dynamic tuning refinement, the threshold is decreased by 1 with every successful gallop, and increased by 1 with every unsuccessful one. A gallop is considered successful if it performed less comparisons than the number of elements (comparisons) that it skipped. It is clear from the results that the galloping mode also introduces a non-trivial performance improvement, especially with dynamic tuning of the threshold ($\approx 5\%$).

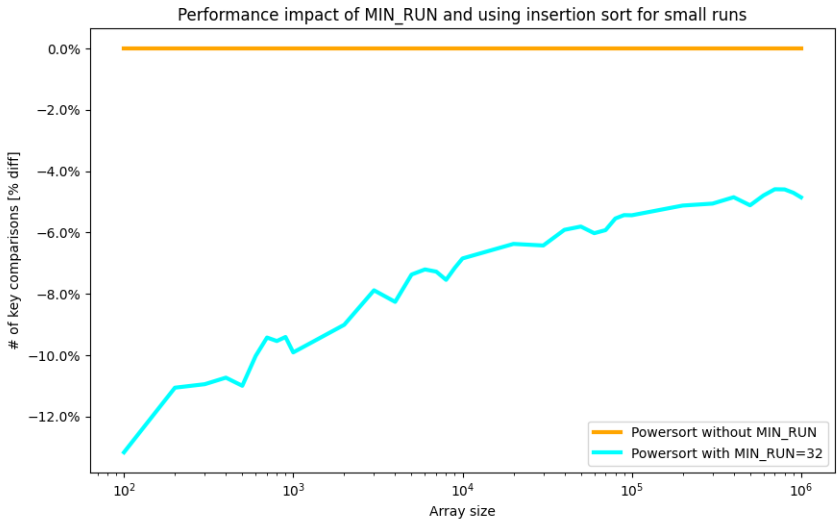

Figure 4.8: Performance impact of `MIN_RUN`

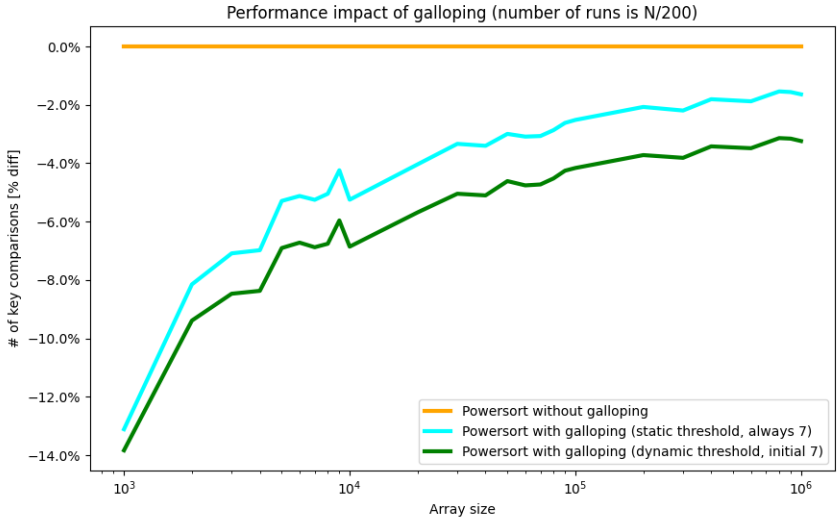

Figure 4.9: Performance impact of using galloping

5 Conclusion

The aim of this work was to survey, compare, and implement various adaptive sorting algorithms, to measure their performance on inputs with different levels of presortedness, and to interpret the results.

The thesis introduced the concept of adaptive sorting and established the necessary theoretical foundations, which were then used to present three of the studied algorithms (NMS, Timsort, and Powersort), showcasing their similarities and differences. While the core idea of the algorithms is similar, they differ primarily in their merging policies.

The experimental benchmark represents the main original contribution of the thesis. We carefully defined the scope of the benchmark and described in detail how the random inputs with the desired properties were generated.

The benchmark results clearly confirmed that the adaptive algorithms generally perform better than the non-adaptive ones on presorted inputs, and worse on inputs that are not presorted. Specifically, they indicated that the performance is positively impacted by a low number of runs and by a non-uniform run profile, implying low entropy. Therefore, we have shown that the algorithms generally perform best on inputs consisting of a small number of unevenly distributed sorted subsequences. Furthermore, Timsort and Powersort demonstrate better performance than Natural Merge Sort on inputs with high entropy. When entropy is random, the differences are not substantial. The results also confirm that the selected Timsort/Powersort optimizations indeed decrease the number of comparisons.

The implementation part of the thesis contains the Python implementations of the three studied algorithms, along with the traditional Merge Sort used as a baseline. Additionally, it includes the code used to generate the inputs, perform the benchmarks, and generate the results and visualizations.

In the future, it would be interesting to extend the scope with other relevant algorithms, such as Peeksort [4] or Adaptive Shivers sort, presented and explained in great detail in [11]. It would also be interesting to include a measure of cache hits/misses, as some algorithms are inherently designed to work with the cache more effectively.

A Source Code Archive Structure

All source code is included in the archive attached to this thesis. The main files and directories of the project include:

- `algorithms/*`
  Implementations of the selected adaptive sorting algorithms
- `benchmark_versions/*`
  Same algorithms as in `algorithms/*`, slightly altered to also measure performance without affecting functionality
- `output/graphs/*`
  Plots (`.png`) generated by the benchmarks
- `output/raw_data/*`
  Raw data (`.csv`) generated by the benchmarks
- `config.py`
  Configurations for the input data to run the benchmarks on
- `random_input_generators.py`
  Functions generating random inputs with desired properties (size, level of presortedness, etc.)
- `output_generation.py`
  Functions generating the benchmark outputs and visualizations
- `benchmarks.py`
  The main code defining and executing the benchmarks
- `requirements.txt`
  Python packages required to run the code
- `README.md`
  Information about the repository and instructions on how to run the code

Bibliography

1. SHANNON, C. E. A mathematical theory of communication. *The Bell System Technical Journal*. 1948, vol. 27, no. 3, pp. 379–423.
2. KNUTH, Donald E. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973. Section 5.2.4.2: Natural Merging.
3. PETERS, Tim. *CPython List Sort Documentation*. Python Software Foundation. Available also from: `https://svn.python.org/projects/python/trunk/Objects/listsort.txt`. Retrieved 13 June 2024.
4. MUNRO, J. Ian; WILD, Sebastian. *Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs*. In: *26th Annual European Symposium on Algorithms (ESA 2018)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, vol. 112, 63:1–63:16.
5. BLOCH, Josh. *Timsort implementation in Java*. Available also from: `https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/TimSort.java`. Retrieved 13 June 2024.
6. AUGER, Nicolas; NICAUD, Cyril; PIVOTEAU, Carine. *Merge Strategies: from Merge Sort to TimSort*. 2015. Available also from: `https://hal.science/hal-01212839`. unpublished research report.
7. AUGER, Nicolas; JUGÉ, Vincent; NICAUD, Cyril; PIVOTEAU, Carine. *On the Worst-Case Complexity of TimSort*. In: *26th Annual European Symposium on Algorithms (ESA 2018)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, vol. 112, 4:1–4:13.
8. HUANG, B-C.; LANGSTON, M. A. *Fast Stable Merging and Sorting in Constant Extra Space*. *The Computer Journal*. 1992, vol. 35, no. 6, pp. 643–650.
9. BUSS, Sam; KNOP, Alexander. *Strategies for Stable Merge Sorting*. In: *Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms*. 2019, pp. 1272–1290.

10. MEHLHORN, Kurt. *A Best Possible Bound for The Weighted Path Length of Binary Search Trees*. *SIAM Journal on Computing*. 1977, vol. 6, no. 2, pp. 235–239.

11. JUGÉ, Vincent. *Adaptive Shivers Sort: An Alternative Sorting Algorithm*. In: *Proceedings of the 31th Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2020, vol. 20.