# Analysis and performance benchmark of modern adaptive sorting algorithms

Daniel Timko

June 2024

## 1 Motivation

Sorting data is a very common algorithmic problem that is theoretically very well researched and it might seem that there is not much space for improvement left in this area.

However, what about practical considerations? We want our sorting algorithms to work well with real-world data. For example, in practice there is oftentimes already some pre-existing partial order in the input data. Is it possible to exploit that to save some time?

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

## 2 Natural Merge Sort

Let's consider the scenario where we want to use merge sort to sort an input array of $N$ elements (let's consider $N$ to be a power of 2 for simplicity) with its right half being completely sorted already. If we knew that a priori, we could save sorting one half of the array, which would save a significant amount of time. Natural merge algorithms are all about detecting this pre-existing order and exploiting it.

The concept of natural merging was first rigorously formalised and described in 1973 by D. E. Knuth in [1], although the idea already existed before (e.g., by A. W. Case in [2]).

The general high-level idea of Natural Merge Sort (NMS) algorithm can be captured by the pseudocode in Listing 2. Implementation details, of course, can be found in the code appendix. First, let's define a run as a sub-array that is already sorted. Each array can be decomposed to several concatenated runs. Of course, ideally the runs would be as long as possible already in the input array.

```
1  runs := find_runs(arr)
2  while runs.size() > 1 do
3      for run1, run2 in runs do
4          merge(run1, run2)
```

Listing 1: High-level idea of Natural Merge Sort

On line 1 of the algorithm, `find_runs` function iterates through the input array and finds its initial run decomposition. It compares the neighboring elements and when the invariant breaks, finishes the current run and starts the new one. Besides ascending runs (we assume we want to sort the array in the ascending order), it also detects descending runs and reverses them immediately.

Once we have the run decomposition, we can start merging them, which we do pair-wise. On line 3, we merge distinct pairs of runs, in the order given by the run decomposition, until we iterate through all of them. We repeat that (line 2) 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 that Knuth proposed [1] slightly varies in the merging process because instead of merging 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.

We can observe that the traditional top-down Merge sort is similar to a NMS with initial segment (run) size being equal to 1, each element being a separate run because the pre-existing order is ignored. However, in NMS, we have to do the extra initial work of detecting the runs, reversing the decreasing runs, etc. Secondly, top-down 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 to merge together in NMS can vary significantly, which is suboptimal. The `merge` operation as we know it works best when merging the arrays of the same (or similar) size.

Therefore, for truly random input arrays with no preponderance of existing order, NMS might not be 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 $1/2$ [1], 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.

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 determining from input array if it is worth to continue with NMS or a different, non-adaptive sorting algorithm. However, that has an additional cost as well.

# 3 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 twists and further optimisations) since version 2.3 up to 3.11 when it was replaced by Powersort [7]. It is also used as a default sorting algorithm in Java [4], since version Java SE 7 (2011) up to this day (Java SE 20).

It belongs to the family of natural merge algorithms. Its core idea is essentially the same as the Natural Merge Sort, but it introduces many additional practical enhancements and optimisations, which makes it more powerful and more complicated.

Because it is used in the reference implementations of Python and Java, it is not surprising that there is a lot of different possible versions of Timsort with different number of implemented optimisations aiming to improve the performance constants. I will focus only on the most important ones, those that really make Timsort be what it is.

## 3.1 Core algorithm

```
1  runs := find_runs(arr)
2  S := empty stack
3  while runs ≠ ∅ do
4      run = runs.pop()
5      S.push(run)
6      while true do
7          h := S.size()
8          if h >= 3 && s_1 >= s_3 then merge(S_2,S_3)
9          else if h >= 2 && s_1 >= s_2 then merge(S_1,S_2)
10         else if h >= 3 && s_1+s_2 >= s_3 then merge(S_1,S_2)
11         else if h >= 4 && s_2+s_3 >= s_4 then merge(S_1,S_2)
12         else break
13 while S.size() > 1 do
14     merge(S_1,S_2)
```

Listing 2: Core procedure of Timsort algorithm, inspired by [5]

Listing 2 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 3. We maintain a stack, to which we push the runs one by one. After every push, if needed, we merge some of the top-most runs on the stack (lines 6-12) – more on this part in 3.2. After all runs are processed, we finish up with gradually merging two top-most runs on the stack until we have only a single run remaining, and thus the whole array is sorted.

### 3.2 Merge policies

As was already mentioned, merging runs with drastically different sizes leads to the performance degradation of the merge procedure, because a big chunk of the elements from the bigger array is already sorted and would just be copied over. Therefore, we want to merge the arrays of similar size.

That is the goal of the conditions on lines 8-11 in Listing 2 that are not so intuitive at the first glance. They are establishing that the runs on top of the stack (top four runs to be precise) 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 merges are always performed in a favorable order. Additionally, this gradual merging process keeps the stack small (logarithmic, [3, 5]) and hence reduces the number of runs we have to remember at any given point. With random input data, all runs are likely to have length equal to the MIN_RUN constant (more on that in 3.3), achieving a perfectly balanced run profile, with only exception possibly being at the very end. [3]

One important note here 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.

### 3.3 Minimal Run Length

The most common optimisation for Timsort is to fix the minimal size of 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, the insertion sort is an effective algorithm for the small arrays (and also for nearly sorted arrays, which goes in hand with 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 minimal 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]*

The following pseudocode demonstrates how the Insertion sort is utilized in the process of creating the run profile, with MIN_RUN being the constant for minimal size of the run. The code hides the logic of detecting the next ascending/descending sequence, which is trivial but unnecessarily technical.

```
1  function find_runs(arr):
2      runs := []
3      i := 0
4      while i < arr.size() do
5          j := find_next_natural_run(arr, i)
6          size := j-i+1  // size of the next natural run
7          if size < MIN_RUN then  // run is too short
8              j := min(i+MIN_RUN-1, arr.size()-1)  // end index
9              // insert the remaining elements
10             insertion_sort(arr, i, i+size, j)
11         runs.add(i, j)
12         i := j+1
13     return runs
```

Listing 3: Using Insertion sort to enforce minimal run size

The only new addition to the function, when compared to Natural Merge Sort algorithm, 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 the correct order. The *insertion_sort* function is adapted to consider the first $N$ elements of the input sub-array already sorted, but otherwise is the same as the traditional insertion sort.

## 3.4 Binary Insertion sort

Timsort is in fact 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 switches 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 (although not necessarily), improving the constant factor.

The overall procedure as outlined in Listing 3 remains the same, except the slight adaptation of *insertion_sort* function.

## 3.5 More optimisations

There are several more engineering tricks that can be used to try to optimize even more. I did not implement these myself, but I consider them interesting enough for an honorable mention.

### 3.5.1 Galloping mode

Galloping is a heuristical technique used during the merge step to try and skipjump over multiple elements of the run if we "suspect" that there is a long sorted subsequence in it. For example, when merging runs [1,2,3,4,6] and [5,7,8,9,10], it would be very useful instead of simply picking element 1 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, it is not always efficient.

The technique keeps track of the minimum galloping treshold, which is a number of elements that need to be picked from the same run consecutively in order to trigger this galloping mode.

One possible case when this could be beneficial is that with `minrun` size constraint in place, it could happen that we have a really long natural run which is splitted into multiple consecutive artificial runs of size `minrun`, which is not optimal. When merging such runs, galloping would come in very handy.

### 3.5.2 In-place merging

As we know, mergesort is not merging in-place but requires $n + m$ extra space ($n$ and $m$ being sizes of the two arrays). There are also well-known variations that are in-place but with a big time overhead. There is also a version acting as a compromise between the previous two approaches, as it is not completely in-place, but reduces extra space to $min(n, m)$ with a relatively small time overhead. This method is also used in many Timsort implementations.

### 3.5.3 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 improvement:

*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]*

### 3.6 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 [6]. However, this proof is out of the scope of this work.

## 4 Powersort

The main complexity of natural merge algorithms lies in determing what order should the runs be merged in. This is called merge policy. With the optimal merging order, we have a good potential to really leverage the benefits of adaptive sorting. However, with a bad merge policy, we can reach even quadratic slowdown.

Timsort's merge policy is based on the 4 rules maintaining the invariant on the top of the run stack. The problem is that it has a "bad case" (also named "Timsort-drag" in [7]) – 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 [8]. This along with the complexity, lack of transparency, and difficulty of analysis of Timsort's merge policy lead to the search for something better.

Let's now describe Powersort – an algorithm which uses a different merge policy. As was mentioned earlier, Powersort replaced Timsort in CPython implementation of its sorting function since version 3.11.

### 4.1 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.

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. 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 [9]. In short, it recursively keeps looking for run boundaries closest to the middle, which approximates the tree to be nearly perfectly balanced.

## 4.2 Algorithm

Once we have constructed the nearly-optimal BST, the second step is to iterate through the intervals from left to right, pushing them to the stack (same as in Timsort) and applying the merge policy. During this iterative process, we keep computing the so-called power (hence Powersort) of the boundary between the two adjacent intervals (also can be interpreted as the power of their parent node in BST), more on that in 4.3. Powersort merge policy has a single rule – the power on the stack is always increasing (as opposed to Timsort's 4 non-trivial rules). This process essentially reconstructs the merge tree. The whole algorithm is implemented by the following pseudocode.

```
1  function powersort(arr)
2      X := empty stack
3      P := empty stack
4      s1 := 0
5      e1 := find_next_run(arr, s1)
6      while e1 < n-1 do
7          s2 := e1 + 1
8          e2 := find_next_run(arr, s2)
9          p := power(s1, e1, s2, e2, n)
10         while P.top() > p do
11             (s1, e1) := merge(arr, X.pop(), (s1, e1))
12             P.pop()
13         X.push((s1, e1))
14         P.push(p)
15         s1 := s2
16         e1 := e2
17     while X ≠ ∅ do
18         (s1, e1) := merge(arr, X.pop(), (s1, e1))
```

Listing 4: Powersort pseudocode, inspired by [7]

First, we initialize the stack for runs and for powers. The `while` loop on the line 6 implements the process described above. In a single run, it finds and examines pairs of adjacent runs, 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.

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 (to smaller power) in the logical BST hierarchy.

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

Note that MIN_RUN + insertion sort optimisation as described for Timsort can be used for Powersort in the same way.

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 still resides in cache, and its retrieval is thus faster.

### 4.3 Power

So what really is that mysterious power? For any 2 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 Munro&Wild's [7] formal definition and the pseudo-code for its calculation.

**Definition: Node Power** Let $\alpha_0, \ldots, \alpha_m, \sum \alpha_j = 1$ be leaf probabilities. For $1 \leq j \leq m$, let $\circledast j$ be the internal node separating the $(j-1)$st and $j$th leaf. The *power* of (the split at) node $\circledast j$ is

$$P_j = \min \left\{ \ell \in \mathbb{N} : \left\lfloor \frac{a}{2^{-\ell}} \right\rfloor < \left\lfloor \frac{b}{2^{-\ell}} \right\rfloor \right\},$$

where

$$a = \sum_{i=0}^{j-1} \alpha_i - \frac{1}{2}\alpha_{j-1}, \quad b = \sum_{i=0}^{j-1} \alpha_i + \frac{1}{2}\alpha_j.$$

($P_j$ is the index of the first bit where the (binary) fractional parts of $a$ and $b$ differ.)

```
function power(s1, e1, s2, e2, n)
    length1 := e1 - s1 + 1
    length2 := e2 - s2 + 1
    l := 0
    a := (s1 + length1/2 -1)/n
    b := (s2 + length2/2 -1)/n
    while floor(a * 2^l) == floor(b * 2^l) do
        l += 1
    return l
```

Listing 5: Calculating the power of an interval boundary

# 5 Benchmarks

The final part of this work is the benchmark of the described sorting algorithms and the interpretation of its results. We are especially interested in the impact of presortedness on the performance. Two measures of the presortedness I have taken into account in my benchmarks are:

- Number of runs in the data (with random entropy)
- Entropy of the run profile $r_1 ... r_q$ for the fixed number of runs $q$

As for the performance indicators, I have measured:

- Number of key comparisons
- CPU time

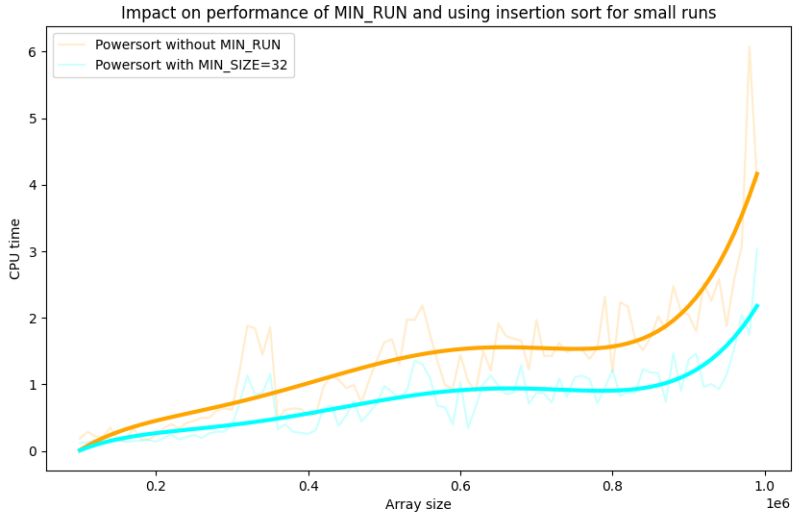
Besides the 3 algorithms that I implemented (NMS, Timsort, Powersort) I have included in my benchmarks:

- Python's `sort()` function – mostly out of curiosity of how much better is the official implementation with all its heroic optimisations
- Traditional top-down Merge sort – this is a bit unfair, because it does not take advantage of presortedness in the data at all. However, I still think it could be interesting as a baseline.

I have ran all these benchmarks with Python version 3.12.3. and used fixed MIN_RUN=32 for Timsort and Powersort.

## 5.1 Impact of MIN_RUN and usage of insertion sort on small runs

First, let's see confirmed experimentally that fixing the minimum run size and using binary insertion sort to sort all runs below that size indeed improves the performance of an algorithm (in this case Powersort) by a constant factor.

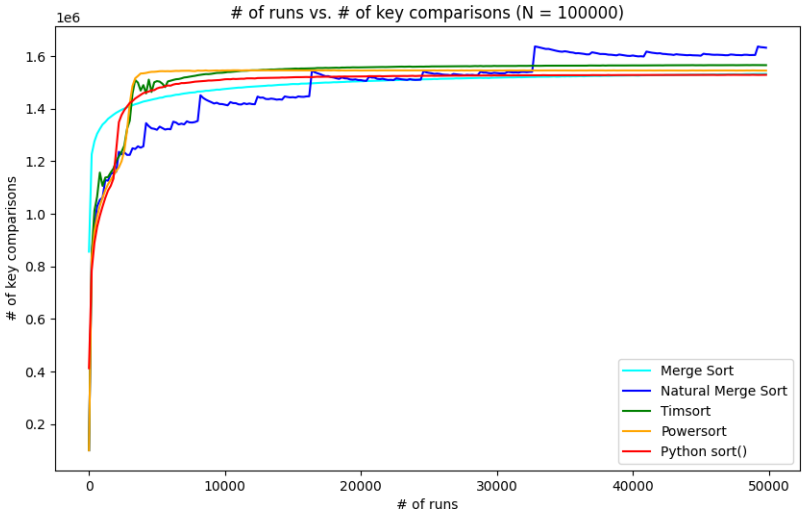



## 5.2 Number of runs vs. Number of key comparisons

Next benchmark shows the impact of number of runs present in the input on the number of key comparisons performed during the sorting. We can see that all algorithms follow a similar trend, perhaps too similar.

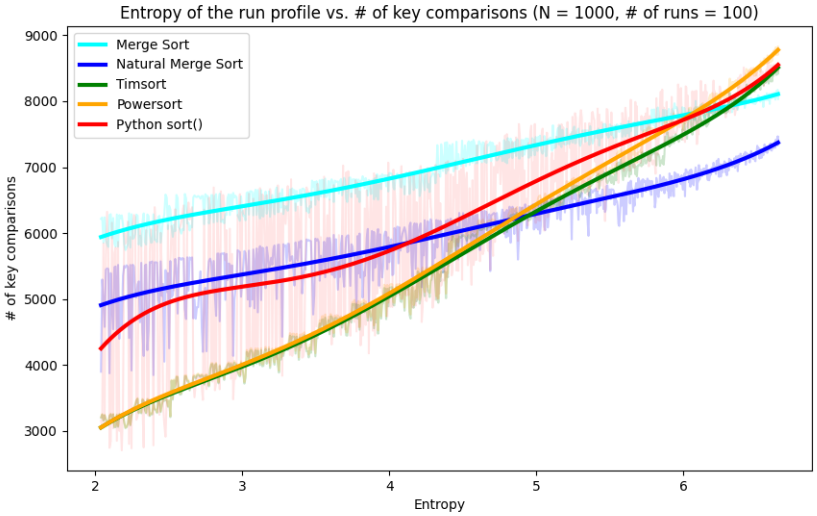
We can still observe something though. First of all, there is a slightly increasing trend for all algorithms, to some degree, which means that they are achieving their goal to exploit presortedness. Secondly, for small enough number of runs ($\sim$5000, so $\sim$20 elements in a run on average), the adaptive algorithms work better than the baseline top-down Merge sort. However, I personally expected this threshold to be much higher.

One might ask how is it possible that the presortedness has an impact on the performance for non-natural merge sort as well. That is because when the input is (almost) sorted, the merge operation performs less comparisons, and more elements are just copied over. So there is indeed an indirect impact.



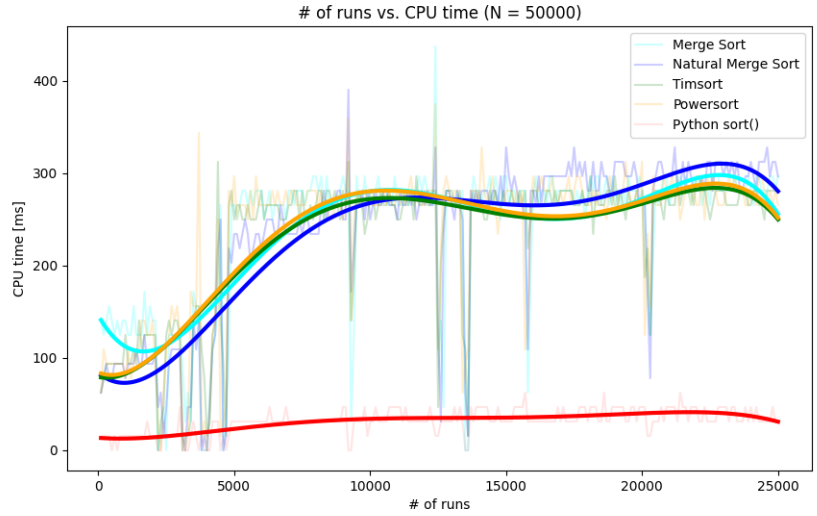

### 5.3 Entropy of the run profile vs. Number of key comparisons

I consider this result to be the most interpretative. As was already mentioned several times, the merging works best overall for the pairs of arrays of similar size. The lower the entropy, the more homogeneous the run profile is, merge operation performs better (less intermediate merges are needed) and less comparisons are performed overall.

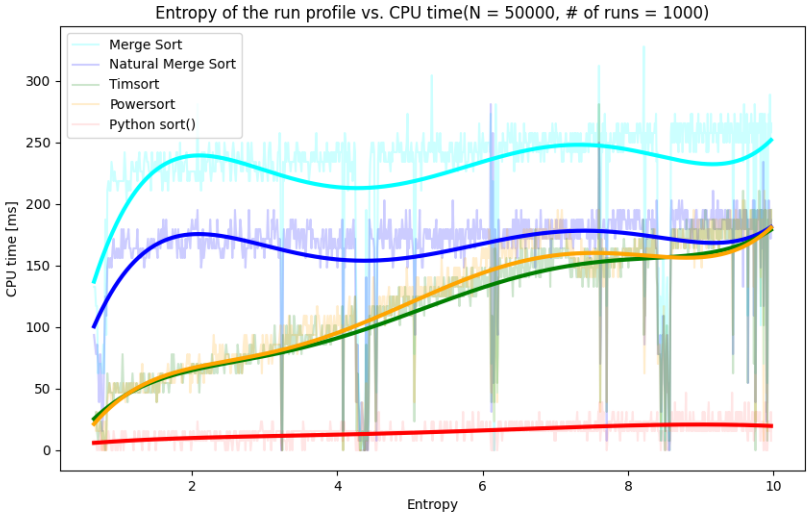



## 5.4 Number of runs vs. CPU time

Finally, in the last 2 benchmarks we can see that, as could be expected, CPU time is also affected in a negative way with the increasing number of runs (less presortedness) and also with increasing entropy (same argument as for 5.3):



### 5.5 Entropy of the run profile vs. CPU time




## 6 Possible future research and improvements

- Implement more engineering techniques for Timsort and Powersort, such as galloping and in-place merging to try to obtain more substantial results.
- Including measure of cache misses in the benchmarks could be interesting.
- Compare with other interesting algorithms like Peeksort [7] or Adaptive Shivers sort, presented and explained in a lot of detail in [10]

## A Archive structure of the implementation part

- `benchmark_versions/` – directory containing all the same algorithms but with the added counting of key comparisons
- `graphs/` – directory with all generated figures
- `natural_merge_sort.py`
- `timsort.py`
- `powersort.py`
- `random_input_generators.py` – utility functions for generating random arrays with specific properties

- `benchmarks.py` – script for execution of benchmarks
- `plotting.py` – utility functions for plotting the graphs
- `requirements.txt`

# References


[1] Knuth, Donald E. (1973). The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, Section 5.2.4.2: Natural Merging

[2] Case, Arthur W. (1951). Serial Files with High Activity. Proceedings of the ACM National Meeting.

[3] Python Software Foundation, written by Tim Peters. (n.d.). CPython List Sort Documentation. Retrieved 13 June 2024, from https://svn.python.org/projects/python/trunk/Objects/listsort.txt .

[4] OpenJDK Project. (n.d.). TimSort implementation in Java. GitHub. Retrieved June 13, 2024, from https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/TimSort.java

[5] Auger, Nicolas & Nicaud, Cyril & Pivoteau, Carine. (2015). Merge Strategies: from Merge Sort to TimSort.

[6] Auger, Nicolas & Jugé, Vincent & Nicaud, Cyril & Pivoteau Carine. On the Worst-Case Complexity of TimSort. 26th Annual European Symposium on Algorithms (ESA 2018), Aug 2018, Helsinki, Finland.

[7] J. I. Munro & S. Wild. (2018) Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs.

[8] Sam Buss and Alexander Knop. Strategies for stable merge sorting, January 2018. arXiv:1801.04641.

[9] Mehlhorn, Kurt. (1977). A Best Possible Bound for The Weighted Path Length of Binary Search Trees. SIAM Journal on Computing, v.6, 235-239 (1977). 6. 10.1137/0206017.

[10] Jugé, Vincent. (2023). Adaptive Shivers Sort: An Alternative Sorting Algorithm. Proceedings of the ACM Symposium on Algorithmic Principles of Programming Languages (APPL).