JESSESORT

Jesse Lew
NYU
jl8429@nyu.edu

ABSTRACT

This paper introduces a novel sorting algorithm, JesseSort, and a new data structure, called a Rainbow. By utilizing this new data structure, our sorting algorithm achieves a best case runtime of $O(n)$ and worst case $O(n \log_2 n)$. Of note, JesseSort is capable of handling natural runs in $O(n)$ time and excels in low-diversity situations with repeated values. Our unoptimized algorithm achieves a faster runtime than Python's default sorting algorithm, offering a competitive alternative. Several areas for potential optimization of this new algorithm are discussed. The source code used in this paper is publicly available[1].

***K*eywords** sorting · algorithm

1 Introduction

Sorting is a fundamental operation in computer science, supporting a wide range of applications in data processing and search optimization. While classical sorting algorithms such as QuickSort [1], MergeSort [2], and HeapSort [3] operate with an average-case time complexity of $O(n \log_2 n)$, new research often pushes the theoretical and practical limits of sorting efficiency. The development of faster algorithms can impact performance significantly in domains where large-scale data ordering is critical, such as internet search, database management, real-time analytics, and machine learning preprocessing.

In this paper, we introduce a novel sorting algorithm, JesseSort, which also achieves a time complexity of $O(n \log_2 n)$, however it approaches $O(n)$ runtime in certain situations. To do this, our approach leverages a new data structure that exploits the harmonic series, called a Rainbow. This data structure can quickly absorb natural runs that may be present in the input data. By keeping search space small, JesseSort outperforms classical sorting techniques dramatically.

We provide a formal analysis of the algorithm's mathematical complexity that demonstrates its theoretical advantage over conventional methods. We share runtime comparisons against Python's default sorting algorithm [4]. Additionally, we identify situations where JesseSort either excels or struggles. Our results highlight the algorithm's efficiency and scalability, showcasing its potential as a practical state-of-the-art alternative to traditional algorithms.

2 Data Structure

In this section, we introduce a new data structure called a Rainbow. We also introduce a variation, called a Split Rainbow, that is useful for our sorting algorithm.

2.1 Rainbows

A Rainbow is an array of array-like structures with a special feature: the first and last values of each subsequent subarray are guaranteed to be in sorted order. These first and last values of each subarray or "band" form a "base array", which serves as the search space of our algorithm. When adding a new value to the Rainbow, one need only consider these front and end band values and append the new value as the new front or end of that band. The first band of the Rainbow

[1] Source code: https://github.com/lewj85/jessesort

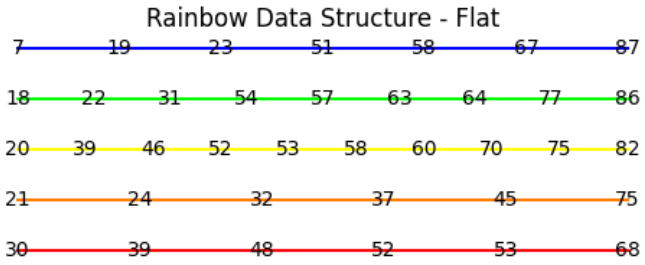

Figure 1: A Rainbow is an array of arrays. Note the sorted order of the first and last values of each adjacent subarray.

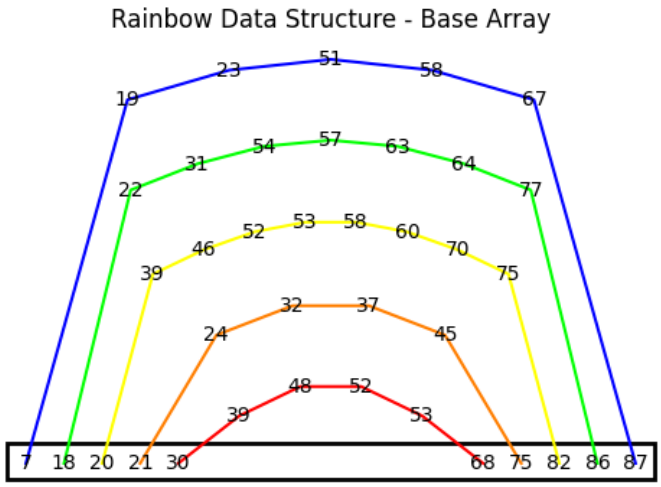

Figure 2: The same Rainbow as in Figure 1, but bent to reveal the base array, highlighted in a black rectangle.

will include the lowest and highest values in our search space, the second band will include the second lowest and second highest, and so forth.

Figure 1 shows a flat rainbow. Figure 2 shows one bent into shape, with the base array highlighted. Note that only the base array values are needed or considered when adding a new value to the Rainbow. As a band could therefore have near-infinite values in it, but the search space would only include its 2 ends, one can easily see how a Rainbow data structure provides value at large scale.

2.2 Split Rainbows

A Split Rainbow is a variation that divides the bands of a standard Rainbow into 2 parts. The usefulness of this becomes evident as one begins to build a Rainbow. When inserting a new low value to a band, one would need to push all values in the band over if using a traditional array or list data structure. We could use a linked list instead to allow for faster front-end insertion, but we discuss later why this is suboptimal.

We propose splitting these bands into 2 parts, allowing us to once again use arrays or lists. The only modification one needs here to avoid front-end insertion is to reverse the order of the lists in the bottom half of the Split Rainbow. By doing so, one can append a new low value to the end of a band in this lower half without dealing with front-end insertion issues or linked lists.

While the length of a Rainbow's bands can technically be arbitrary, a natural pattern emerges when sorting purely random values. Figures 3 and 4 show this pattern for Split Rainbows. We note the difference in slope steepness between the outer edges and the middle of the arches.

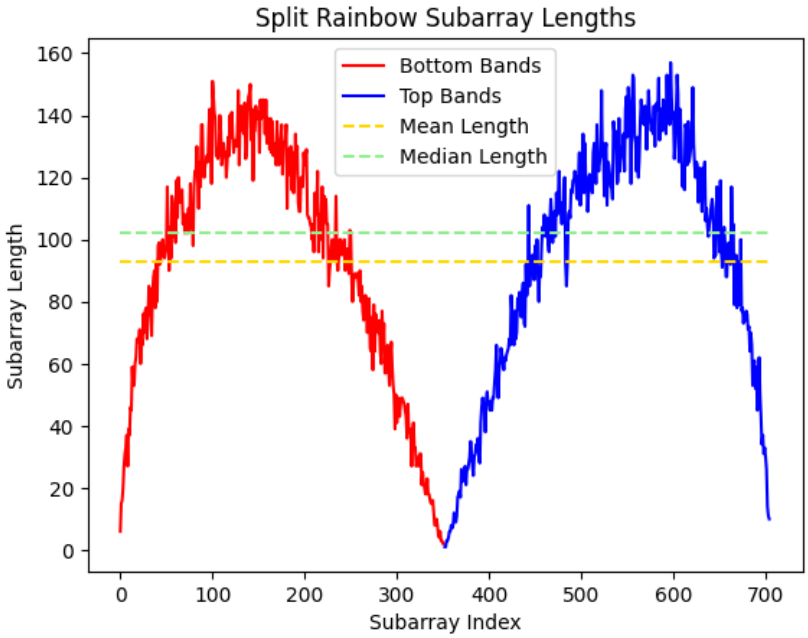

Figure 3: The lengths of each subarray in a Split Rainbow.

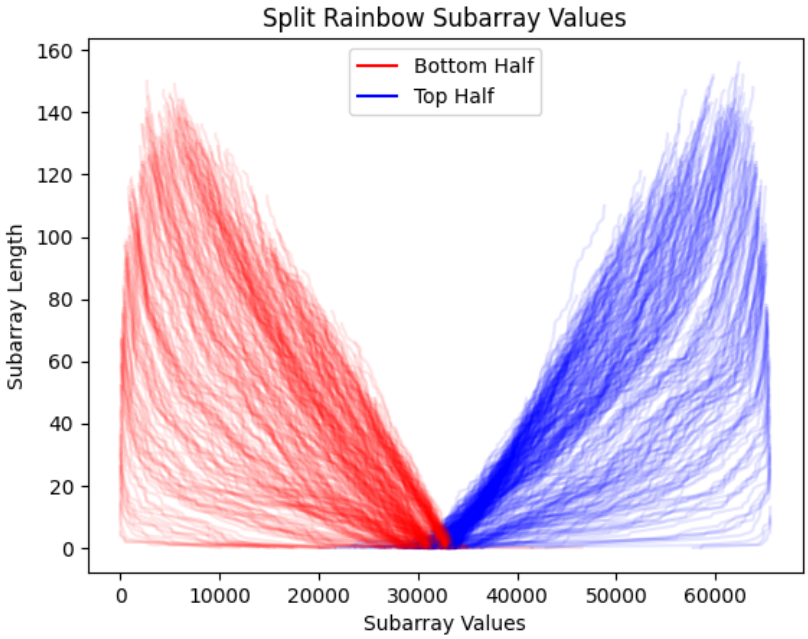

Figure 4: The values of each subarray mapped in a Split Rainbow.

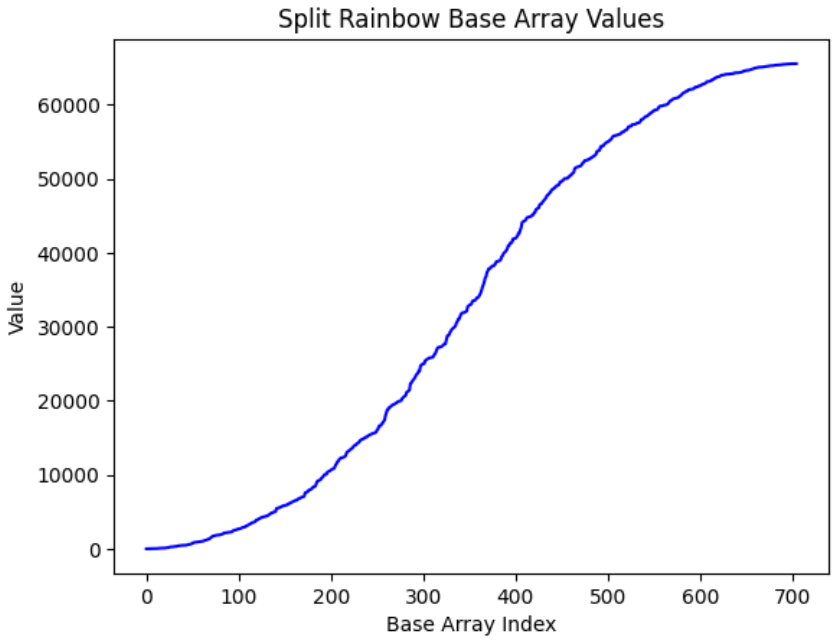

Figure 5: The values in the base array form a sigmoid pattern.

2.3 Base Array

As values are inserted into to the Rainbow or Split Rainbow, new bands may need to be added for middle values that are not lower than the first value or higher than the last value of the last band. Adding a band causes the base array to expand from the middle out. New values push the bottom values of the base array lower and the top values higher. As seen in Figure 5, this causes a sigmoid pattern to emerge for the values in the base array.

We make a copy of the base array values specifically for faster search purposes. Using a copy allows us to keep these values adjacent to each other in memory. Specifically, we aim to keep the base array small enough to fit into L1-L3 cache, which speeds up the insertion process. Whenever we refer to the "base array", it is this separate array of copied values.

As we have mentioned, the need to add a new band only arises if the new value falls between the first and last values of the last band in the Rainbow. Otherwise, if new values are smaller than any of the values in the bottom half of the base array or larger than any of the values in the top half, inserting these values will not extend the length of the base array. The math and complexity analysis of band insertion is provided in section 5.

3 Algorithm

In this section, we introduce JesseSort's pseudocode and a variation that improves its speed on traditional hardware.

3.1 JesseSort

At a high level, there are 2 phases to JesseSort: generating a Rainbow and merging its bands. Pseudocode for this is in Algorithm 1.

To generate the Rainbow, we loop through each value in the unsorted input and search for an appropriate band to add it to. We only need to search through the values in the base array and can use a binary search strategy to find the proper index in $O(\log_2 k)$ time, where k is the length of this base array.

If the new value is greater than the first element of the last band and less then the last element of the last band, then we need to create a new band with this value. The value will be added to a new subarray and also inserted into the middle index of the base array.

Algorithm 1 JesseSort

```
Input: Unsorted array
Output: Sorted array
Create Rainbow with first 2 values
Sort these 2 manually
Create base array with first 2 sorted values
index ← 0
// Phase 1:  Fill the Rainbow
for each value in unsorted array after first 2 do
    while True do
        if index is at the middle of the base array then
            if base array length is even then
                if value is between (> and <) the 2 middle elements then
                    Append new band with value to end of Rainbow
                    Insert value at middle of base array
                    break
                end if
            else
                if value is between (> and <) the 2 middle-left elements then
                    Add value to the front of last band
                    Insert value before middle of base array
                    break
                else if value is between (> and <) the 2 middle-right elements then
                    Add value to the back of last band
                    Insert value after middle of base array
                    break
                end if
            end if
        else
            if index in bottom half and value ≤ base array[index] and value > base array[index-1] then
                Insert value at the front of Rainbow[index] band
                base array[index] ← value
                break
            else if index in top half and value ≥ base array[index] and value < base array[index+1] then
                Insert value at the end of Rainbow[index] band
                base array[index] ← value
                break
            end if
        end if
        index ← BINARYSEARCH(base array, index, value)
    end while
end for
// Phase 2:  Merge the bands
while number of bands > 1 do
    for band_i, band_j in MERGEPOLICY(Rainbow) do
        MERGE(band_i, band_j)
    end for
end while
Sorted array = Rainbow[0]
return Sorted array
```

If we are in the bottom half the base array, we check to see if the new value is less than or equal to the base array value at the current index but still greater than the value at index-1. If we are in the top half the base array, we check to see if the new value is greater than or equal to the base array value at the current index but still less than the value at index+1. If the current index location is not suitable for placement, we update the current index using the binary search strategy mentioned previously and loop again. Once we isolate a location in the base array for this new value, we place this value into the band, update the copy of the base array, and start on the next unsorted value.

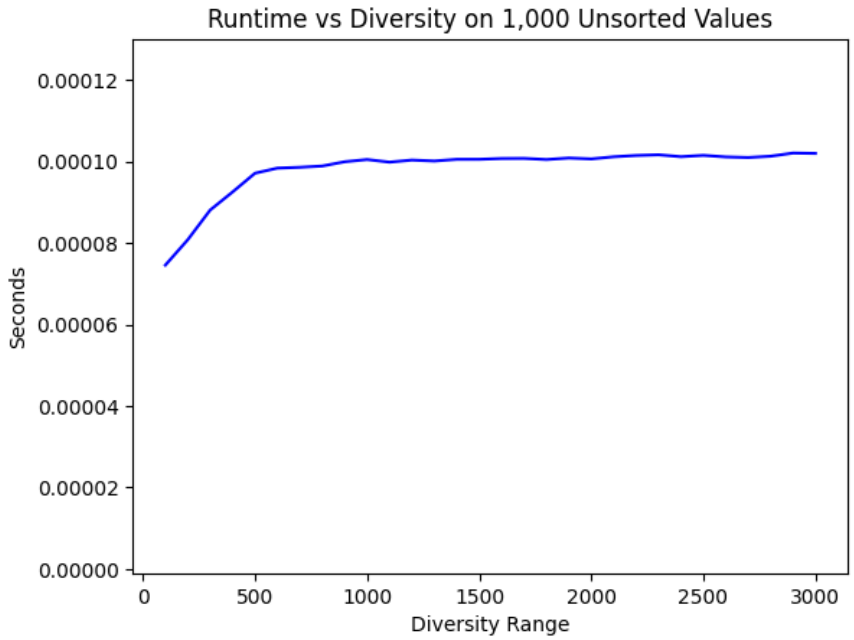

Figure 6: Comparison of JesseSort speed on 1,000 unsorted values using different diversity ranges.

Note that we maintain our current index marker rather than reset it after every placement, so the next value in a natural run will potentially begin its search while already in the proper base array index location. This pushes us towards $O(n)$ runtime for these sections of natural runs.

3.2 Arrays vs Linked Lists

The bands of a Rainbow can be any array-like structure that can maintain the sorted order of its elements. Standard arrays and lists are well-suited for this, however linked lists can also be used. As JesseSort requires front-end insertion approximately half the time, using arrays or lists would add significant additional overhead. To avoid this, we could use linked lists for these bands instead, as adding new nodes to the front of a linked list requires virtually no overhead.

Our initial speed tests showed that using arrays (Python lists) and a Split Rainbow was significantly faster than using linked lists with a standard Rainbow. We also tested using a value-only copy of the base array during the insertion phase to limit having to access nodes prior to the final merge. This only sped up runtime slightly, making both linked list options uncompetitive with the array method that uses Split Rainbows.

3.3 Diversity

We find that JesseSort excels when the diversity or uniqueness of the unsorted values is low. As seen on the left side of Figure 6, JesseSort performs up to 30% faster on 1,000 unsorted values as diversity decreases and values are repeated. The "diversity range" in the x-dimension of the figure represents the range $[1, x]$ used when filling our unsorted array with random values. We discuss ways to exploit this in Section 7.

4 Performance

We perform speed tests of JesseSort, comparing it against Python's default sorting algorithm. We used Python 3.13 for our experiments, which utilizes Timsort [5] with an optimized merge tree, Powersort [6]. Speed tests were performed by using the built-in Python function "sorted( )". All tests were done using purely random values with a diversity range equal to the number of unsorted values, $[1, x]$.

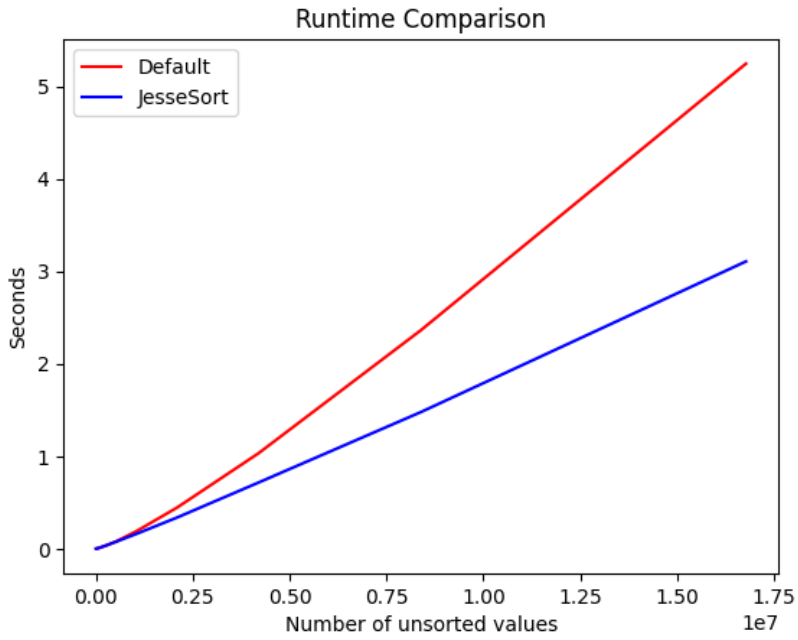

Figure 7: Runtime of Python's default sorted() function vs JesseSort with different size n.

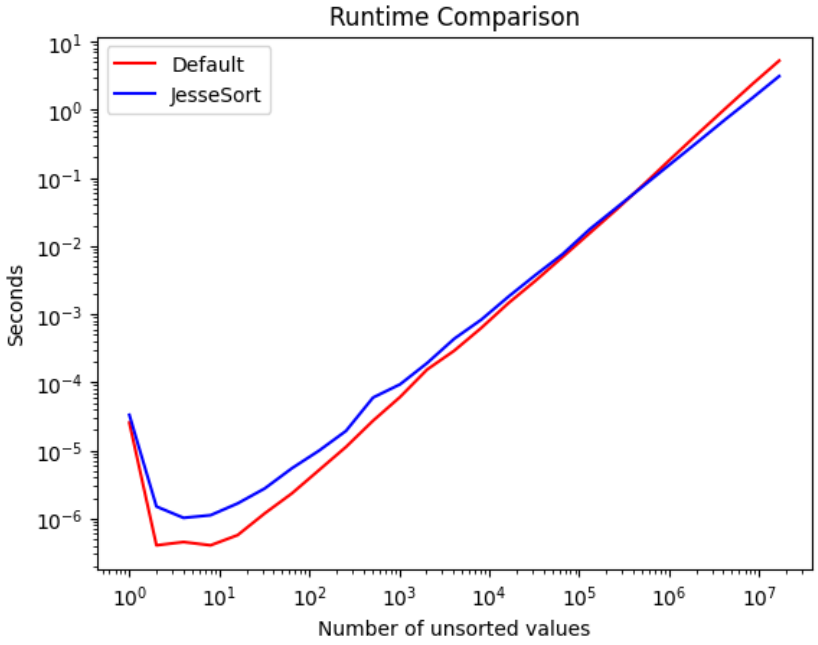

Figure 8: The same information as Figure 7, but with log-scale axes.

Figure 7 shows that JesseSort is significantly faster as the number of unsorted values n increases. For big data use cases, JesseSort is the clear winner. However, Figure 8 shows a different perspective of this same data. We find that for smaller $n \lesssim 262144$, JesseSort is slower than Python's default sort. We note that this version of JesseSort is unoptimized, so speed may improve as variations and optimizations are implemented.

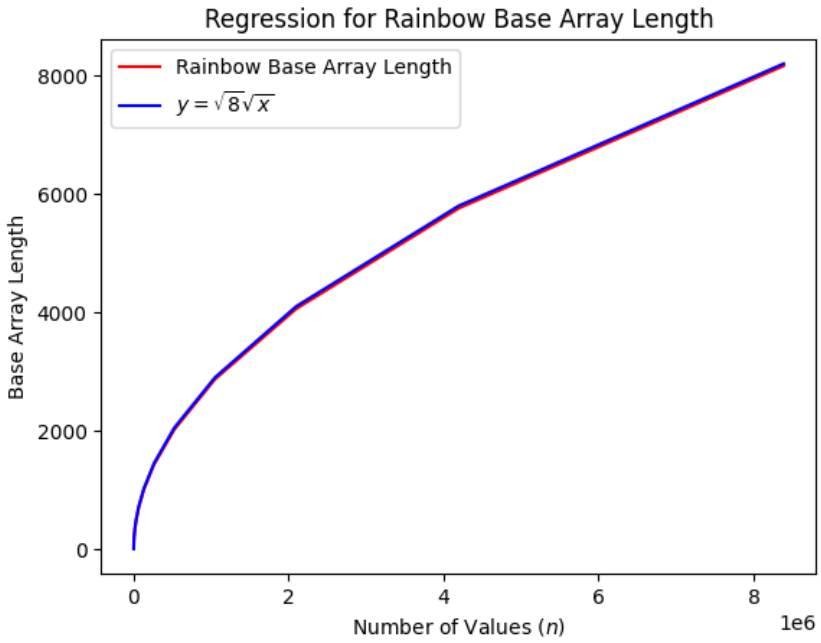

Figure 9: The length of the base array of a Rainbow is mapped with a regression line. The results from repeated trials and the regression line overlap tightly.

5 Analysis

To analyze the time complexity of JesseSort, we break down its two main components: the insertion phase and the final merging phase.

5.1 Insertion Phase Complexity

The key operation in the insertion phase is determining the correct position of a new element in the Rainbow's base array. This is achieved using a binary search, which takes $O(\log_2 k)$ time, where k is the length of this base array. Therefore, to calculate the overall runtime, we must first decipher the value of k.

We observe that the growth of k is governed by the probability of a new element falling within the last band's bounds in the middle of the base array (the "expansion zone"). This triggers an expansion either from: (1) the creation of a new band or (2) the extension of a band with only 1 value. This probability of expansion p follows a harmonic sum pattern with an oscillating numerator:

$$\begin{aligned} p &= \frac{1}{1} + \frac{2}{2} + \frac{1}{3} + \frac{2}{4} \cdots \\ &= 1.5 \sum_{m=1}^{n} \frac{1}{m} \\ &= 1.5H(n) \end{aligned}$$

where the chance of extension at each step is proportional to the inverse of the current number of elements. The harmonic sum $H(n) \approx \ln(n) + \gamma$, where $\gamma \approx 0.577$ (Euler-Mascheroni constant [7]), suggests that k initially follows $O(\ln n)$ growth. This would make the runtime complexity of this phase of JesseSort $O(n \log_2 \ln n)$.

However, in practice, the expansion zone dynamically widens over time as more elements are added to the band in the middle of the base array, which in turn increases the likelihood of further extensions. This resulted in additional testing where we modeled this growth factor and found it to be greater than $O(\ln n)$ while still being sublinear. We determined empirically that the length of the base array for any given n is related to $\sqrt{n}$. This can be seen in Figure 9, where we fit a regression line to the mean of multiple JesseSort trials of various input sizes.

The determined regression formula $y = \sqrt{8}\sqrt{x}$ is a near-perfect match. We point out that the coefficient $\sqrt{8}$ appears non-arbitrary. Given $\log_2 \sqrt{8} = 1.5$, we see the return of the mean oscillating numerator from our harmonic sum above. We continue our complexity analysis:

$$O(n \log_2(\sqrt{8}\sqrt{n})) = O(n(\log_2 \sqrt{8} + \log_2 \sqrt{n}))$$

$$= O(n(1.5 + \frac{1}{2} \log_2 n))$$

$$= O(1.5n + \frac{n}{2} \log_2 n))$$

We note that this could be further simplified to $O(n \log_2 n)$, however this would not reflect the sublinear nature of base array length k. Likewise, while $O(n \log_2 k)$ is more appropriate, this does not immediately reveal k's relationship to n. We reserve simplifications until the overall complexity analysis later in this paper.

5.2 Merging Phase Complexity

Once all elements are inserted, JesseSort merges the bands two at a time in a recursive manner. There are various merge policies that can employed to select intelligent pairs and reduce the overall complexity. These are discussed in the 7 section.

The number of merges is dependent on the number of bands, and the number of bands is related to the length of the base array k. We could therefore summarize this phase as having an approximate runtime of $O(n \log_2 k)$, but we found a deeper analysis to be helpful.

When k is even, the number of bands is $\frac{k}{2}$. When k is odd, the number of bands is $\frac{k+1}{2}$. These average out to $\frac{k+0.5}{2} = \frac{k}{2} + 0.25$. Note that in a Split Rainbow these bands are cut in two, doubling the number of subarrays for merging, resulting in $2(\frac{k}{2} + 0.25) = k + 0.5$.

Each round of merging processes O(n) elements. The number of merging rounds required is logarithmic, but based on band counts rather than on n. For standard Rainbows, this results in a merging complexity of:

$$O(n \log_2(\frac{k}{2} + 0.25)) = O(n \log_2(\frac{\sqrt{8}\sqrt{n}}{2} + 0.25))$$

$$= O(n \log_2(\sqrt{2n} + 0.25))$$

$$\approx O(n \log_2(\sqrt{2n}))$$

$$= O(n\frac{1}{2} \log_2(2n))$$

$$= O(\frac{n}{2}(\log_2 2 + \log_2 n))$$

$$= O(\frac{n}{2}(1 + \log_2 n))$$

$$= O(0.5n + \frac{n}{2} \log_2 n))$$

While we would prefer to keep constants until the final complexity analysis, the 0.25 here was negligible and prevented further analysis. For Split Rainbows, we would also throw away the negligible 0.5 constant, which would yield an equivalent complexity to the insertion phase:

$$O(n \log_2(k + 0.5)) \approx O(n \log_2(\sqrt{8}\sqrt{n}))$$

$$= O(1.5n + \frac{n}{2} \log_2 n))$$

5.3 Overall Complexity

Here we sum the complexities from both the insertion and merging phases of JesseSort:

$$\begin{aligned} &O(1.5n + \frac{n}{2}\log_2 n)) + O(0.5n + \frac{n}{2}\log_2 n)) \\ &= O(1.5n + \frac{n}{2}\log_2 n + 0.5n + \frac{n}{2}\log_2 n)) \\ &= O(2n + n\log_2 n) \end{aligned}$$

When using Split Rainbows, we need only add $+n$ to account for the additional merge loop:

$$\begin{aligned} &O(1.5n + \frac{n}{2}\log_2 n)) + O(1.5n + \frac{n}{2}\log_2 n)) \\ &= O(3n + n\log_2 n) \end{aligned}$$

In either case, this simplifies to:

$$\approx O(n\log_2 n)$$

This complexity accounts for the full interplay between the harmonic sum, widening expansion zones, and recursive merging. While each of the two steps of JesseSort was able to capitalize on sublinear search space and complete in approximately $O(n\log_2 k)$, their summed efforts unfortunately exceeded $O(n\log_2 n)$.

Empirically, the average complexity is somewhat better than $O(n\log_2 n)$, as JesseSort can capitalize on natural runs that can slow the growth of the expansion zone significantly. To make a case for JesseSort's adoption, we point once again to its performance speed in practice. We also investigate several variations and optimizations beyond the use of Split Rainbows in Section 7.

6 Challenges

Like Timsort, JesseSort can gracefully handle natural runs in $O(n)$ time, but only under certain conditions. These runs need not be of any minimum size, which offers an advantage over Timsort. However there are 2 situations where JesseSort is at a disadvantage: (1) when an ascending natural run occurs in the bottom half of the base array, and (2) when a descending natural run occurs in the top half of the base array. This accounts for approximately 50% of natural runs.

In these situations, the binary search for an adjacent band is triggered after each value in the run, pushing the runtime to the insertion phase's worst case. This can potentially make many bands jump farther in value than they would from purely random input values, which results in generally shorter bands. This can also result in potentially sequential values in adjacent base array indices that can no longer be extended until their adjacent bands do. Shorter bands would result in a higher band count, which would slow both phases of JesseSort. We discuss ways to either mitigate these problems or simply avoid this situation altogether in Section 7.

When natural runs fall into the ideal half of the base array and stay within the bounds of adjacent bands, insertion can be processed in $O(n)$ time. More often, these natural runs will have to be broken into chunks that lay across multiple adjacent bands in order to enforce the sorted base array pillar of Rainbows. While this remains near the best case runtime, breaking these natural runs into chunks is still technically suboptimal as these will need to be merged together later anyway.

Finally, we note that when natural runs cross the center of the base array, they switch from best case to worst case or vice versa, again balancing out approximately 50% of the time. This reinforces JesseSort's average runtime complexity as somewhere between $O(n)$ and $O(n\log_2 n)$.

7 Variations and Optimizations

In this section, we discuss potential optimizations of JesseSort that may have an improved overall runtime. While some strategies alter and improve upon the original algorithm significantly, these still include an insertion phase that utilizes Rainbows followed by a merge phase. Thus, we consider these variations rather than new completely new algorithms.

7.1 Merge Optimization

Selecting an optimal merge policy for JesseSort's merge phase can improve overall speed. Naively pairing adjacent or mirrored subarrays will usually be suboptimal, as they may have significant length differences. We could use a greedy pairing policy [8], always taking the smallest subarrays, however this is suboptimal as well. Powersort has been shown to find an optimal merge tree structure, improving on Timsort's naive merge policy. We could use Powersort's optimal merge policy to improve JesseSort's merge phase runtime as well.

7.2 Split Rainbow First Merge

As shown in Figure 4, we find that there is relatively little overlap of lower half and upper half bands in a Split Rainbow. We could modify the first merge iteration of the merge phase to exploit this. For each mirrored pair of subarrays from the lower and upper halves, we could use binary search to find the start index of their overlap to avoid unnecessary comparisons in this initial merge iteration. This would cost additional overhead but could reduce the number of comparisons needed during the merge phase by almost $O(n)$.

7.3 Pruning and Maximum Base Array Size

As shown in Figure 5, the values at the ends of the base array get pushed towards the extremes (low values get lower, high values get higher), making it less likely for new values to need to access these low-index bands of the rainbow. Given this, we could dynamically prune the rainbow of these early bands, sending them to a CPU for parallel processing and merging while the rest of the rainbow continues to be built. The ideal location for this pruning logic would be when a new band must be added (after lines 12, 18, and 22 of Algorithm 1), because shifting/extending portions of the base array must occur at these times anyway.

To trigger the pruning, we could add a maximum size parameter for the base array and prune the lowest-index band (or multiple) as this maximum size limit is reached. Using a predetermined length base array would also allow for an easier implementation of C arrays instead of dynamically-sized Python lists, which would be faster. A "merge queue" would add additional overhead to process these pruned bands, but would allow for parallel processing across multiple CPUs.

Pruning bands keeps the base array small, which keeps the search space for insertion small, which speeds up the algorithm. Additionally, this can make it more likely for the base array to fit in its entirety in L1-L3 cache, offering more speed to the insertion phase. This would potentially add many additional bands to merge in the second phase however, which may not lead to any overall speed increase. Therefore, pruning only makes sense if the pruned bands could be processed in parallel during the insertion phase.

We note that both aggressive pruning and/or setting a maximum base array size too low could lead to the last few bands at the middle of the base array growing too quickly, which inversely affects the speed, as the expansion zone growth factor would increase. More research is required to find the optimal pruning policy and an optimal maximum base array size in relation to n and/or available hardware.

7.4 Parallelization and Double Rainbows

In the insertion phase of JesseSort, we could divide the unsorted array into two parts and create separate Rainbows for each section in parallel. This "Double Rainbow" strategy could easily be extended to any number of parallel Rainbows that would maximize the use of available hardware. We could again utilize a merge queue to process 2 fully sorted sections at a time, as the parallel JesseSorts individually finish.

In the merge phase of JesseSort, we could make use of Cython's parallel functions like "prange" [9]. Merging pairs of bands in parallel would greatly speed up this phase of the algorithm.

7.5 Natural Runs

As mentioned in Section 6, natural runs may be detrimental if found in a suboptimal half of the base array. To avoid this, we could add checks for natural runs, recycling one of the best features of Timsort at the cost of more overhead. Runs (perhaps of some minimum length) that begin in the suboptimal half of the base array could be handled differently. They could be processed in reverse order or, more optimally, placed directly into a merge queue for parallel processing. This would synergize well with a pruning strategy that also makes use of a merge queue during the insertion phase.

This variation could potentially outperform Timsort, as all non-natural runs would be processed using Rainbows in roughly $O(n \log_2 k)$ time rather than relying on Insertion Sort's $O(n^2)$ for 32-value chunks at a time. While Insertion Sort would require $\frac{32*31}{2} = 496$ average comparisons for every 32 unsorted values, using JesseSort would only cost

$3 * 32 + 32 \log_2 32 = 96 + 32 * 5 = 256$ average comparisons to yield the same 32-length sorted subarray. We could try utilizing miniature dedicated Split Rainbows (per the Double Rainbow strategy mentioned previously), though the overhead may prove too costly.

7.6 Early Freezing

We have already shown how the insertion phase would have a complexity of $O(n \log_2 \ln n)$ if not for an ever-expanding middle gap. We could freeze the base array early after some number of steps m, and use this frozen base array as the search space for future unsorted values. This would prevent the gap from continuing to increase, but would also mean we could not further build this initial Rainbow.

Instead, the frozen base array could be used as a filter. First, we would make a new array that would hold sub-structures, such as subarrays or binary heaps or Rainbows. When the correct frozen base array index is isolated, the value could be sent to the sub-structure at that same index in the new array for further sorting. The sorting space of this sub-structure would be much smaller, specifically the average distance between 2 values in the frozen base array. As the frozen base array size is dependent on m steps, rather than n, this gives it a size of $\sqrt{8}\sqrt{m}$, meaning each sub-structure would have to sort only $\frac{n-m}{\sqrt{8}\sqrt{m}}$ values.

The number of values each sub-structure has to sort is not the only consideration, but also the range or diversity of these values. Depending on how big we wish to set m, the range of values in these sub-structures can potentially be very small. We propose using a sorting algorithm here that excels at low-diversity situations with repeated values, such as JesseSort.

We could freeze the sub-Rainbows and sub-sub-Rainbows after some number of steps, using recursive logic to prevent these gaps from widening as well. We note that this is reminiscent of Mergesort's method of recursive division. More research is required to find an optimal depth for nested frozen Rainbows.

Using just one level of nested JesseSorts and sub-Rainbows would lead to an overall runtime cost of:

$$O(1.5m + \frac{m}{2} \log_2 m)$$

prior to the freeze, plus:

$$O((n-m)(\log_2 \ln m))$$

to filter new values, plus:

$$O(\sqrt{8}\sqrt{m}(3\frac{n-m}{\sqrt{8}\sqrt{m}} + \frac{n-m}{\sqrt{8}\sqrt{m}} \log_2(\sqrt{8}\sqrt{\frac{n-m}{\sqrt{8}\sqrt{m}}})))$$

$$= O(3(n-m) + (n-m)\log_2(\sqrt{8}) + \frac{(n-m)}{2} \log_2(\frac{n-m}{\sqrt{8}\sqrt{m}}))$$

$$= O(4.5(n-m) + \frac{(n-m)}{2} \log_2(\frac{n-m}{\sqrt{8}\sqrt{m}}))$$

to sort the values in each of the $\sqrt{8}\sqrt{m}$ sub-Rainbows, plus:

$$O(1)$$

to stack the results, as no comparisons are needed due to each sub-Rainbow being between set bounds, plus:

$$O(1.5m + \frac{m}{2} \log_2 m)$$

to merge in the bands from the original frozen Rainbow. If we select $m \ll n$, we can simplify this overall complexity by dropping m values:

$$O(1.5m + \frac{m}{2}\log_2 m + (n-m)(\log_2 \ln m) + 4.5(n-m) + \frac{(n-m)}{2}\log_2(\frac{n-m}{\sqrt{8}\sqrt{m}}) + 1 + 1.5m + \frac{m}{2}\log_2 m)$$

$$\approx O(5.5n + \frac{n}{2}\log_2 n)$$

which further simplifies to $O(n \log_2 n)$, but with every n functionally smaller due to the subtraction and division of other elements. We can see how repeating this pattern at greater depths may yield further functional reductions to complexity. Further research is required to select an optimal value m at each level of this JesseSort variation.

7.7 Hybrid Sorting

JesseSort could be adapted to use a hybrid sorting method. When the distance between the first and last values of the last band (in the middle of the base array) is between some thresholds $[t_{min}, t_{max}]$, we could process new values that fall into the expansion zone through a secondary sorting algorithm. This secondary algorithm could be another JesseSort, again adapting a recursive strategy, or it could be some other sorting algorithm. For example, when dealing with integers, we could utilize a Counting Sort [10] to process these values in $O(n)$ time.

Adopting this hybrid method would speed up the insertion phase of the algorithm significantly, as it would greatly reduce the need to add new bands to the Rainbow. This would push the insertion phase's complexity closer to its harmonic sum runtime of $O(n \log_2 \ln n)$, but the runtime would be influenced by choice of secondary sorting algorithm.

7.8 n-dim Rainbows

While we have not yet tested the use of higher dimensional Rainbows or base arrays, there may be opportunities for optimization that utilize k-way comparisons in JesseSort's insertion phase or k-way merges in its merge phase. These present interesting areas for future research.

8 Conclusion

In this paper, we have introduced a new $O(n \log_2 n)$ sorting algorithm, JesseSort. We have also introduced a new data structure, Rainbows, and a Split Rainbow variation to avoid the use of potentially slow linked lists. We have shown the value that Rainbows can provide by limiting the search space for new input values to its relatively small base array. We have demonstrated that JesseSort's performance is superior to Python's default Powersort for reasonably large n. We have shown cases where JesseSort does well and where it faces challenges. Finally, we have offered several variations for optimization that can improve its overall runtime. We hope this inspires new research into sorting algorithms and creative uses of Rainbows.

References

[1] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[2] John von Neumann. First draft of a report on the edvac. 1945. Mergesort was introduced in this seminal report.

[3] J. W. J. Williams. Algorithm 232: Heapsort. In *Communications of the ACM*, volume 7, pages 347–348. Association for Computing Machinery, 1964.

[4] Python Software Foundation. Python language reference, version 3.x, 2023. Available at `https://www.python.org`.

[5] Tim Peters. Timsort: A fast, stable sorting algorithm. Python Enhancement Proposal (PEP) 450, 2002. `https://github.com/python/cpython/blob/main/Objects/listsort.txt`.

[6] J Ian Munro and Sebastian Wild. Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. *arXiv preprint arXiv:1805.04154*, 2018.

[7] Leonhard Euler and Lorenzo Mascheroni. The euler-mascheroni constant, γ, 1734. First introduced by Euler and later studied extensively by Mascheroni. For more, see `https://mathworld.wolfram.com/Euler-MascheroniConstant.html`.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998. Discusses merging strategies, including greedy approaches.

[9] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 13th Python in Science Conference (SciPy)*, pages 29–36, 2011.

[10] Harold H. Seward. Automatic data processing. *Massachusetts Institute of Technology (MIT) Computation Laboratory*, 1954. Counting Sort was introduced as part of a system described in this work.