Investigating the efficiency behind Python's Sorting through the underlying C code

Randy Panopio rpanopio@sfu.ca, 301294517

Introduction

Investigating the performance and implications of certain algorithms within the context of this class, I was drawn into investigating Python's sorting algorithm. Despite Python being an interpreted language, many of its functionality and libraries use C or C++ with a wrapper written around them to interoperate with the rest of Python. According to Python (3.10) documentation, the `list.sort()` and `sorted()` functions use the Timsort algorithm. The underlying C code, I looked through the official CPython project, where the optimized C Timsort is available for inspection. With this in mind, I began to investigate and conduct timing tests against Timsort which powers Python 3.10's sorting functionality.

Pre-Testing Analysis

The way the Timsort algorithm functions, as described by the original contributor Tim Peters, as *"[Timsort] describes an adaptive, stable, natural merge sort. (…) It has supernatural performance on many kinds of partially ordered arrays (less than lg(N!) comparisons needed, and as few as N-1)".* Timsort is derived from merge sort and insertion sort, tuned to work well with real-world data. Its time complexity is defined for the worst case as O(n log n), an average case of O(n log n), and best case of O(n). A fairly efficient algorithm that attempts to reduce the pitfalls of the original merge and insertion sort algorithms. By nature of this algorithm, the worst-case space complexity of O(n), and can require an auxiliary array/pointers. Which could imply that it may have issues with memory locality. The actual CPython Timsort that Python uses is far more complex and optimized specifically to work with the various structs to interop with Python. Rather than attempting to rewrite the algorithm from scratch based on the CPython implementation in C myself, I opted to utilize an open source C implementation of Timsort (initially as a port of Java's Timsort over to C) for my analysis. Although I cannot definitely state I am measuring the C code that Python uses, it is a close approximation. I also chose to also conduct the same testing against a merge sort and insertion sort respectively to have a better understanding as a baseline against Timsort.

Testing Setup

For measuring the performance of Timsort against Insertion Sort and Merge Sort, I chose to generate sorted, reverse sorted, and randomized arrays of length 65536 (2^16) of type int64_t and conducted 32 runs, averaging them for my comparisons. Since Insertion sort inherently has O(n^2) time complexity, I decided to not test the algorithm against the larger data sets as it would have taken far too long to measure (I had one run saved against size: 1048576 random insertion sort took 385632.7 ms!). I also generated the same datasets and attempted to cast it into 64 Bit Integers using numpy's `np.int64` data type to test against Python `list.sort()`. This is to ensure that I can normalize my findings, and have a somewhat reasonable comparison against C.

I measured timing performance on my WSL machine, running on an AMD Ryzen 9 7950X3D 16-Core Processor. This is so I can test my Python and C code easier. I used `clock_gettime` to measure the expected run time of each test for the Timsort, Merge, and Insertion C code, and `time.time()` for the Python `list.sort()` code. For testing branching and cache misses, I used the CSIL machine to conduct my analysis, running on an Intel(R) Core(TM) i7-9700K CPU @

3.60GHz Processor which reported to have 256KiB L1, 1.5MiB L2, and 12MiB L3 caches respectively. I used Perf to record the behavior of the algorithms regarding branch prediction performance.

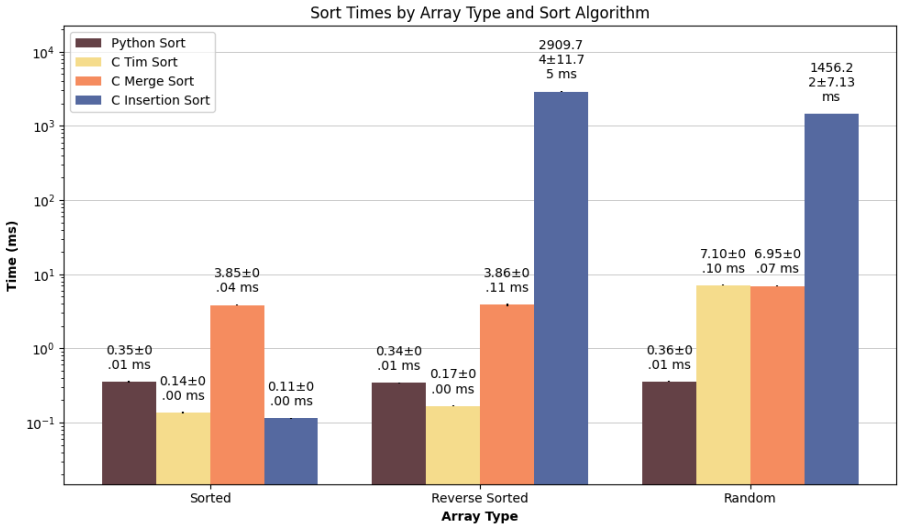

Figure 1: Comparisons of Python .sort, C Tim, C Merge, and C Insertion sorting algorithms

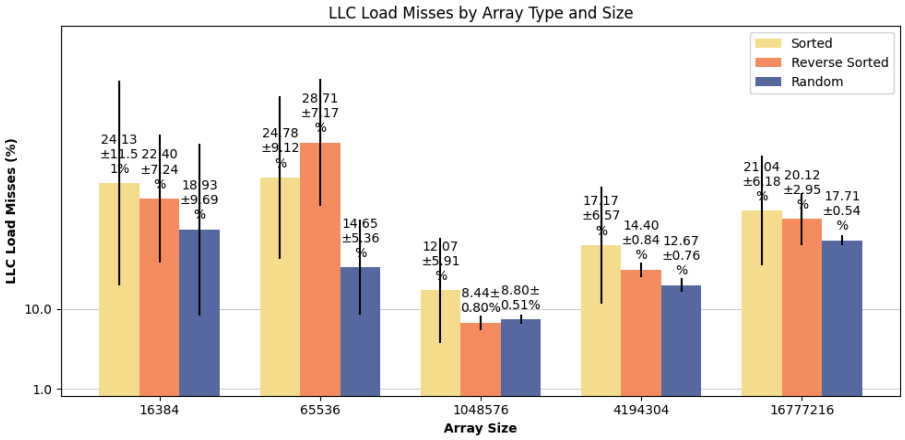

Figure 2: Perf llc-load-misses percentages of the C Timsort under Sorted, Reverse Sort, and Random arrays.

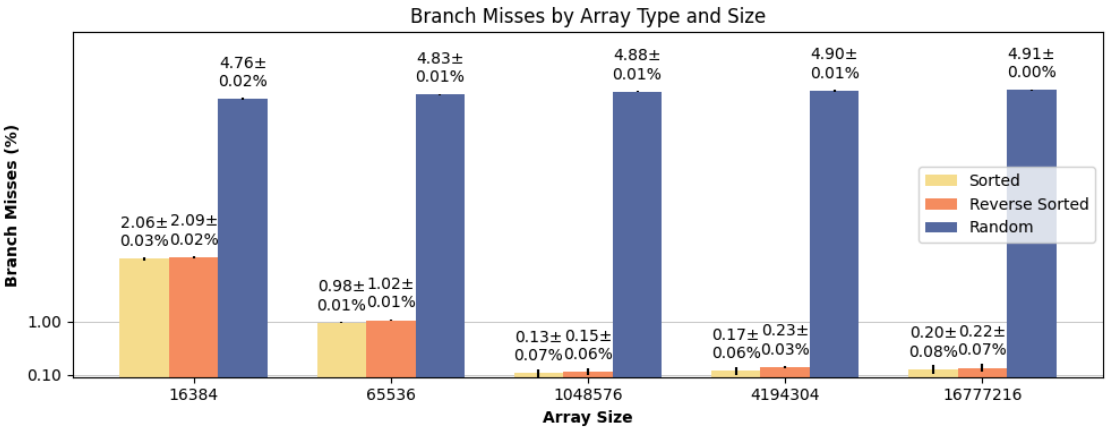

Figure 3: Perf branch_misses percentages of the C Timsort under Sorted, Reverse Sort, and Random arrays.

For measuring memory locality of Timsort, I generated randomized arrays of type `int64_t` and ran 10 runs for each case and averaged them. I've created arrays of length (powers of 2) 16384, 65536, 1048576, 4194304, 16777216, with cases 1-3 intended to fit within caches, while cases 4 and 5 do not. I used the flag `LLC-load-misses` with Perf to record my analysis.

Result Analysis

With the C timings, there are some expected behaviors immediately apparent. The run times of Merge sort are consistent regardless of input, while Insertion degenerates in reverse sort, and quite close to randomized arrays. My results show that while Timsort gets slightly outperformed by Insertion sort for sorted arrays, it performs much better than Merge sort. We can clearly see the advantage of Timsort comparing it to its inspired algorithms. To note, the Python timings seem to perform even better on randomized arrays in Python itself, but seeing that the resulting timings across all data types is so close, there may be other shenanigans going under the hood, if anything the way the timing functions used to aggregate these data in itself is also a problem, however it is interesting to see how close the timings are to the C code, which we can see the performance of the underlying C code in action.

When measuring the cache misses on Perf, I noticed that there was a lot of variance between the percentages of the run. This also happens on sorted arrays, which was interesting, even in cases where the whole array should fit inside the cache, `LLC-load-misses` were still reported. This may be due to the fact that Timsort uses auxiliary data as part of its algorithm. What is interesting is that in cases where the data supposedly fit within L1 cache, the cache misses increases, then suddenly goes down in Case 3, steadily increasing again. This variance could still be due not having enough runs to have a conclusive result, but this pattern could also be part of how the Timsort algorithm chooses to run a merge approach, vs the insertion approach of the algorithm.

Analyzing the branch misses of Timsort, there is a surprising pattern where the larger the dataset, the more predictable it becomes. This could be that the processor is able to predict the behavior of Timsort as it continues to execute over time. What is interesting is that both reverse sort and sorted behave identical, and this could be due to the processor being able to predict its behavior, but I believe it is primarily how Timsort decides to sorts reverse sorted array efficiently, with this finding somewhat backed up by the fact that reverse sorted arrays timing performance behaves similarly to sorted arrays.

Conclusion

After my analysis, it showed the real impact of algorithms and the tradeoffs designers choose to implement as a standard. It's clear that Timsort was chosen to manage the tradeoffs of O(n) efficiency of sorted/reverse sort arrays, while attempting to maintain O(n log n) for the vast majority of use cases. Another decision evident is the choice of trading better performance in branch prediction than it does with memory locality. Perhaps this was chosen as code execution in modern hardware performs better with more predictable code over memory locality.

Timsort is a very efficient sorting algorithm and Python utilizes the efficiency of running native C code to speed up common yet critical tasks like sorting "under the hood". The same sorting algorithm even inspired the Rust language's default sorting algorithm Despite that, I have also learned that the field of sorting is constantly evolving, and that Python 3.11 has opted to use an even more efficient sorting algorithm powersort.