Andy Cherney

Professor Mark Brody

CS 502-001

22 March 2024

TimSort: Where Inefficient + Efficient = More Efficient?

Introduction

When you consider all the sorting algorithms that we went over this quarter during CS 502, what comes to mind? You would probably think of bubble or insertion sort. All these algorithms are well-known for their slow performance but simple implementation. For larger datasets, one would most likely turn to quicksort or merge sort for their speed. However, what do you think happens when components from both groups are combined to form a hybrid solution? Well, it turns out that in doing so, the new result could outperform even the more efficient algorithms. You would wonder how such a thing is possible. However, sometimes a hybrid solution can be more ideal depending on the input data or at different points of execution (Mervin). One of these algorithms also happens to be the default one implemented in Python, Java, and Javascript: Timsort (Abdalla).

In essence, Timsort is a hybrid sorting algorithm that takes operations from insertion sort and combines them with the ones found in merge sort. By design, it can be used on any real world datasets to give an optimal result (*Tim Sort*). This report will first highlight the overall background of the algorithm as well as how it works. Next, the implementation and tests of this algorithm will be carried out in 2 languages: C and Python. Finally, to evaluate its true efficiency on sorting arrays of integers, this report will also compare the algorithm's performance with both the Gnomesort algorithm's results from Paper 1 and across implementations.

History

Like Gnomesort from paper 1, there is not a whole lot of history available for Timsort. The algorithm was first proposed in 2002 and named after software engineer, Tim Peters, for use in the Python language. Through a mutual colleague, he was in collaboration with Python's founder, Guido van Rossum, about proper design patterns, as the language was still in its early stages at the time (*Interview with Tim Peters*). The algorithm later was proven to be so efficient that other programming languages like Java caught on as well for their default sort implementations (Timur).

Peters' intention was to create an algorithm designed to be optimal for use on real-world datasets by taking advantage of the fact that they contain small subsequences that are already sorted either in ascending or descending order. This is a feature that other sorting algorithms like merge sort and quicksort do not account for, which is why they tend to behave slower on smaller datasets. Therefore, the approach was to first separate out those chunks into blocks called runs, sort them further if necessary, and then merge the new sorted pieces back together (Mervin). This is why 2 components of algorithms are needed to implement this algorithm. Insertion sort is useful, since it performs the best on datasets that are very small in size and partially sorted. The merge feature of merge sort essentially does the grunt work and brings each run together to form the new sorted set. Because it chooses when to apply the merge and insertion sort, Timsort is often referred to as an adaptive sorting algorithm (Abdalla).

Pseudocode

The Timsort algorithm is way more complex in nature than Gnomesort, as it is a hybrid. To understand it completely, it helps to look at it in layers. Figure 1.1 is pseudocode for the main Tim sort algorithm assuming all array indexes are 0-based.

```
function TimSort(A[]):
    let n = length(A)
    minRuns = findRunLength(n)

    for i from 0 to n by minRuns do:
        end = min(i + minRuns - 1, n - 1)
        insertion sort A from i to end

    let mergeSize = minRuns
    while mergeSize < n do
        for left from 0 to n by 2 * mergeSize do
            mid = left + mergeSize - 1
            right = min(left + 2 * mergeSize - 1, n - 1)
            if mid < right then
                merge A on (left, mid, right)
                endif
        mergeSize = mergeSize * 2
```

Figure 1.1: Main TimSort Algorithm Pseudocode

To understand the first layer, a “run” in Tim sort is just a small block of data that is presorted in ascending or descending order. For this algorithm to work efficiently, there needs to be a way to ensure that “all our runs are at least a certain length”. To do this, a minimum run size is first calculated (Bauermeister). While it is not guaranteed that any given run will be that size, the algorithm will correct itself by adding in more elements from the array until the minimum is reached. Only then, a new run is created (Bauermeister). There are various ways to compute the minimum run size and how it is done does not matter a whole lot (Cox). The overall objective is just to minimize the number of runs while keeping the size small enough to be efficient for insertion sort. Figure 1.3 shows the pseudocode for the calculation used in this report.

```
let largeSize = 64
let smallSize = 32
function findRunLength(n):
        let remainder = 0
        if n < smallSize then
           return smallSize
        while n >= largeSize do
           remainder = n mod 2
```

```
            n = floor(n / 2)
        return n + remainder
```

Figure 1.3: Pseudocode to calculate the run length in Timsort

It was determined that choosing the min run length such that the array size n divided by the min run length is close to or exactly a power of 2 is the most optimal way, which is why the range is between 32 and 64 inclusive (Bauermeister). For values in between 32 and 64, the array size can be used as the min run size, since the data size is small enough for insertion sort to be efficient and anything smaller will cap the size at 32. In these cases, the size does not really matter as it is small enough for insertion sort to be efficient anyway.

Next, while any sort could theoretically be used to correct each run, which is another advantage of this algorithm, insertion sort is most common, as it is found to be the most efficient on tiny partially sorted datasets (Mervin). In figure 1.1, the algorithm divides the array into these runs finds the end index for each one, and then runs the insertion sort. Figure 1.2 is the pseudocode for the insertion sort algorithm.

```
function insertionSort(A [], start, stop):
    for i from start + 1 to stop do
        j = i
        while j > start and A[j] < A[j - 1]:
            swap A[j] and A[j - 1]
            j -= 1
```

Figure 1.2: Insertion sort Pseudocode

All the insertion sort does is go through increasing subarrays of the array, compares the next element with its previous one, and swaps if the next is less than the previous.

The final layer is the merge whenever any 2 runs have been created. To merge, the algorithm will first create 2 left and right subarrays and copy the data using the start, middle, and end . Afterwards, using

3 iterators for both the left, right, and main array, it will go through the main array and swap values depending on whether the left and right indexes (i and j) are in both the left and right arrays or not. If true, then the values are copied from the corresponding array after comparing their current values. Otherwise, it will only copy data from the array that has more elements left. This is also what happens under the hood of the merge sort algorithm. Figure 1.4 is the pseudocode for the merge operation.

```
function merge(A[], start, middle, stop):
    left = copy A from start to middle
    right = copy A from middle + 1 to stop

    let i = 0, let j = 0       # Indexes for left and right arrays
    for k from start to stop do
        if i >= length(left) then
            A[k] = A[j]
            j = j + 1
        else if j >= length(right) then
            A[k] = A[i]
            i = i + 1
        else if left[i] < right[j] then
            A[k] = A[i]
            i = i + 1
        else
            A[k] = A[j]
            j = j + 1
```

Figure 1.3: Pseudocode to calculate the run length in Timsort

Back in the main Timsort algorithm, it steps through the runs, which are essentially the subarrays with length equal to the calculated run size. It calculates the left most index of each subarray. The midpoint of each run is the last value of that run, and the right value is the end point of the adjacent run. If the midpoint is less than the end, the current run is merged with the adjacent run. The current run size then doubles until the entire array is sorted.

Implementation 1

Figure 2.1 is the C implementation of the Tim sort algorithm.

```c
#define SMALLRUN 32
#define LARGERUN 64

int min(int a, int b){
    if (a < b)
        return a;
    return b;
}

int findRunLength(int n){
    int r = 0;
    if (n < SMALLRUN)
        return SMALLRUN;

    while (n >= LARGERUN){
        r = n % 2;
        n = n / 2;
    }
    return n + r;
}

void insertionSort(int* list, long start, long stop) {
    for (long i = start + 1; i <= stop; i++) {
        long j = i;
        while (j > start && list[j] < list[j - 1]) {
            int temp = list[j];
            list[j] = list[j - 1];
            list[j - 1] = temp;
            j--;
        }
    }
}

void merge(int* list, long start, long middle, long stop){
    long lSize = middle - start + 1;
    int* left = malloc(lSize * sizeof(int));
    long rSize = stop - (middle + 1) + 1;
    int* right = malloc(rSize * sizeof(int));

    for (long i = 0; i < lSize; i++)
        left[i] = list[start + i];

    for (long i = 0; i < rSize; i++)
        right[i] = list[middle + 1 + i];

    long i = 0, j = 0;
    for (long k = start; k <= stop; k++){

        if (i >= lSize) {
            list[k] = right[j];
            j++;
        } else if (j >= rSize){
            list[k] = left[i];
            i++;
        } else if (left[i] < right[j]){
            list[k] = left[i];
            i++;
        } else {
            list[k] = right[j];
            j++;
        }
    }

    free(left);
    free(right);
}
void timSort(int* list, long n) {

    int minRun = findRunLength(n);

    for (long i = 0; i < n; i += minRun) {
        int end = min((i + minRun - 1), (n - 1));
        insertionSort(list, i, end);
    }

    for (long size = minRun; size < n; size = 2 * size) {
        for (long left = 0; left < n; left += 2 * size) {
            long mid = left + size - 1;
            long right = min(left + 2 * size - 1, n - 1);
            if (mid < right)
                merge(list, left, mid, right);
        }
    }
}
```

Figure 2.1: C Code for Tim sort

In C, the entire algorithm may look quite complex to implement. However, once you understand each layer as well as both the insertion and merge sort algorithms, there is no struggle at all. The code tends to match the pseudocode so easily that many of the operations are directly translatable line by line. Really, the only issue was deciding on how to implement the findRunLength function, since as mentioned in the pseudocode section, there are various implementations of this calculation. Some implementations also just keep the size constant at 32 or 64, while others could optimize for a smaller number of runs in the process. In the end, I decided to implement it as shown in figure 1.2 to optimize for both smaller and larger arrays.

One notable thing that is different in this implementation is the extra for loop in the merging section instead of the while loop. This can be done because C supports iterator step updating in the main loop header and we know that the total size will basically be multiplied by 2 each time as the indexes of each run adjust accordingly. Just like in the C Gnomesort implementation, the size of the array is passed into the function, since C does not have a built in way to calculate length when the arrays are dynamically allocated and are represented as pointers to an integer. Additionally, C does not have automatic memory management, so it is important to free both subarrays after each merge procedure. There is also an additional min function, since C does not have it built in.

The insertion sort, min, find run length, and merge helpers were also quite easy to implement. Merge especially was an almost an exact replica of the merge sort pseudocode discussed in class during week 3. Furthermore, assignment 4 asked us to implement merge sort for an array of strings, so all I needed to do was change the comparison in the merge function to work with integers instead. Similarly with insertion sort, all that needed to be changed was some loop variables that allowed the algorithm to work with start and end indexes.

Implementation 2

Figure 3.1 is the Python implementation of the main Tim sort algorithm.

```python
SMALLRUNS, LARGERUNS = 32, 64

def findRunLength(n):
    r = 0
    if n < SMALLRUNS:
        return SMALLRUNS
    while n >= LARGERUNS:
        n, r = divmod(n, 2)
    return n + r

def insertionSort(list, start, stop):
    for i in range(start + 1, stop + 1):
        j = i
        while j > start and list[j] < list[j - 1]:
            list[j], list[j - 1] = list[j - 1], list[j]
            j -= 1

def merge(list, start, middle, stop):
    left = list[start: middle + 1]
    right = list[middle + 1: stop + 1]

    i, j = 0, 0
    for k in range(start, stop + 1):
        if i >= len(left):
            list[k] = right[j]
            j += 1
        elif j >= len(right):
            list[k] = left[i]
            i += 1
        elif left[i] < right[j]:
            list[k] = left[i]
            i += 1
        else:
            list[k] = right[j]
            j += 1

def timsort(list):
    n = len(list)
    min_run = findRunLength(n)

    for i in range(0, n, min_run):
        end = min(i + min_run - 1, n - 1)
        insertionSort(list, i, end)

    size = min_run

    while size < n:
        for left in range(0, n, 2 * size):
            mid = left + size - 1
            right = min(left + 2 * size - 1, n - 1)

            if mid < right:
                merge(list, left, mid, right)
        size *= 2
```

Figure 3.1: Python Code for Tim sort

In Python, the algorithm is even easier to implement, and syntactically, it matches closer to the pseudocode. There is no struggle at all when implementing it assuming you understand the layers. One notable thing that is different though is unlike the C implementation, this one uses the traditional while loop to go over each run. Although Python does have a way to update the iterator in a for loop header using the walrus operator and a secondary variable, this operator was only introduced in Python version 3.8, so using that implementation of the algorithm would not work on older versions of Python.

A lot more operations can be consolidated as well, since Python is more high level than C. We do not have to worry about freeing memory, implementing certain functions like min, or even writing loops to copy the data in the merge function since Python supports array slicing. Variable unpacking also allows us also to swap the variables in place. Additionally, length can easily be looked up without any overhead, as it is a variable that is automatically managed by the array as it grows or shrinks in size. Even the division and modulo operation in the findRunLength function is just on one line, as the divmod function returns both the floor division and remainder in one step.

Comparison

Between Tim sort and the Gnome sort algorithm from paper 1, there is no doubt that Tim sort is the faster of the 2 algorithms. While Gnome sort, has the advantage of using constant space, since it is a comparison sort for one array in place , Tim sort has a faster time complexity since it relies on insertion sort for only small arrays that are guaranteed to be partially sorted and the merge component of merge sort to put everything back together. It shares a lot more similarity with merge sort with the only difference being that merge sort first recursively splits the arrays until they cannot be split anymore. Only then, will it put everything back in the correct way. The time complexity for this algorithm in its worst and average cases is $O(N\ log_2(N)$ compared to Gnome sort's slow $O(N^2)$ time complexity, a decent improvement. In the best cases though, they both operate in $O(N)$ time (Mervin).

Testing

Just like in paper 1, the testing was run on both Tim sort implementations using the same arrays and the logic was implemented in Python. Because the RNG was already seeded prior to testing, this step was easy to do. The subprocess module was used to execute the C program, pass in each test case to stdin, and capture the sorted output. For all tests except the first 2 edge cases, the arrays were randomly generated with the RNG having a lower bound of -50 and an upper bound of 100. This was done to demonstrate the algorithm handling both positive and negative values. Table 4.1 represents the same test set.

#	Size	Test case	C result	Python result	Passed
1	0	[]	[]	[]	True
2	1	[9]	[9]	[9]	True
3	14	[-13, -15, -47, 84, 2, -2, -47, 67, 5, -35, -8, 77, 97, 79]	[-47, -47, -35, -15, -13, -8, -2, 2, 5, 67, 77, 79, 84, 97]	[-47, -47, -35, -15, -13, -8, -2, 2, 5, 67, 77, 79, 84, 97]	True
4	15	[48, -3, 1, 20, 21, 2, -42, 97, 22, -2, -4, 96, 35, 1, 51]	[-42, -4, -3, -2, 1, 1, 2, 20, 21, 22, 35, 48, 51, 96, 97]	[-42, -4, -3, -2, 1, 1, 2, 20, 21, 22, 35, 48, 51, 96, 97]	True
5	13	[40, 26, 88, -25, -2, -30, 54, -10, 34, 87, 81, 29, 90]	[-30, -25, -10, -2, 26, 29, 34, 40, 54, 81, 87, 88, 90]	[-30, -25, -10, -2, 26, 29, 34, 40, 54, 81, 87, 88, 90]	True
6	12	[-24, -50, 62, 25, 51, 69, -45, 81, -6, 21, -14, 27]	[-50, -45, -24, -14, -6, 21, 25, 27, 51, 62, 69, 81]	[-50, -45, -24, -14, -6, 21, 25, 27, 51, 62, 69, 81]	True
7	12	[6, 21, 1, 71, 32, -10, 32, -12, 94, 18, -20, -1]	[-20, -12, -10, -1, 1, 6, 18, 21, 32, 32, 71, 94]	[-20, -12, -10, -1, 1, 6, 18, 21, 32, 32, 71, 94]	True
8	8	[-1, 80, 19, 98, 37, 77, 69, -16]	[-16, -1, 19, 37, 69, 77, 80, 98]	[-16, -1, 19, 37, 69, 77, 80, 98]	True
9	14	[48, 84, 60, 54, -1, 90, 30, -14, 44, 53, 29, 76, 11, 62]	[-14, -1, 11, 29, 30, 44, 48, 53, 54, 60, 62, 76, 84, 90]	[-14, -1, 11, 29, 30, 44, 48, 53, 54, 60, 62, 76, 84, 90]	True
10	15	[-10, 1, -46, 64, -27, 99, 57, 93, 93, 70, -45, 49, 75, -27, -9]	[-46, -45, -27, -27, -10, -9, 1, 49, 57, 64, 70, 75, 93, 93, 99]	[-46, -45, -27, -27, -10, -9, 1, 49, 57, 64, 70, 75, 93, 93, 99]	True
11	9	[-45, 12, 71, 55, 51, 22, 56, 17, 87]	[-45, 12, 17, 22, 51, 55, 56, 71, 87]	[-45, 12, 17, 22, 51, 55, 56, 71, 87]	True
12	13	[22, -46, 21, -23, -7, -40, 85, 62, 21, -49, -29, 44, -40]	[-49, -46, -40, -40, -29, -23, -7, 21, 21, 22, 44, 62, 85]	[-49, -46, -40, -40, -29, -23, -7, 21, 21, 22, 44, 62, 85]	True
13	4	[75, -2, 95, 12]	[-2, 12, 75, 95]	[-2, 12, 75, 95]	True
14	15	[97, -41, -21, 69, 68, -40, 76, -7, -36, -50, -17, 79, -5, -34, 54]	[-50, -41, -40, -36, -34, -21, -17, -7, -5, 54, 68, 69, 76, 79, 97]	[-50, -41, -40, -36, -34, -21, -17, -7, -5, 54, 68, 69, 76, 79, 97]	True
15	8	[-46, -29, -44, -33, -21, -7, 79, 95]	[-46, -44, -33, -29, -21, -7, 79, 95]	[-46, -44, -33, -29, -21, -7, 79, 95]	True
16	14	[83, -38, -44, 56, 38, 36, 41, 86, -28, -19, 31, -42, -36, -16]	[-44, -42, -38, -36, -28, -19, -16, 31, 36, 38, 41, 56, 83, 86]	[-44, -42, -38, -36, -28, -19, -16, 31, 36, 38, 41, 56, 83, 86]	True
17	10	[71, 65, 56, 51, 29, 28, 85, 76, 64, 62]	[28, 29, 51, 56, 62, 64, 65, 71, 76, 85]	[28, 29, 51, 56, 62, 64, 65, 71, 76, 85]	True
18	13	[40, 78, 56, 90, 93, 66, -8, 59, 54, 68, 92, 25, -7]	[-8, -7, 25, 40, 54, 56, 59, 66, 68, 78, 90, 92, 93]	[-8, -7, 25, 40, 54, 56, 59, 66, 68, 78, 90, 92, 93]	True
19	14	[7, 77, 80, -48, -29, 47, -2, 71, 17, 37, 63, 6, 31, -49]	[-49, -48, -29, -2, 6, 7, 17, 31, 37, 47, 63, 71, 77, 80]	[-49, -48, -29, -2, 6, 7, 17, 31, 37, 47, 63, 71, 77, 80]	True
20	15	[79, 40, 2, 48, -35, -26, 93, -17, 75, 68, -13, 76, 18, 4, 31]	[-35, -26, -17, -13, 2, 4, 18, 31, 40, 48, 68, 75, 76, 79, 93]	[-35, -26, -17, -13, 2, 4, 18, 31, 40, 48, 68, 75, 76, 79, 93]	True

Table 4.1: Tim Sort Algorithm Tests

We can see that the algorithm is accurate for both implementations, as all the tests passed. As mentioned before, the tests are all the same as in paper 1. Tests 1 and 2 are edge cases and test for the empty array and an array with only 1 element respectively. In these cases, the algorithm does not change the original array and yields the same result back. The remaining tests all have varying input sizes ranging between 4 and 15 elements to optimize for space. Additionally, since the randomization for values in the arrays was done with replacement, it is possible for some elements to be repeated such as in tests 4 and 10. Tim sort is also guaranteed to be stable like merge sort, so all ties are kept in the same position. To assess the tests, a function was used to first compare the Python result against the default Tim sort implementation and then the string representation of that array was checked against the C result.

Timings

As mentioned in the comparison section, the theoretical time complexity of the gnome sort algorithm is $O(N\ log_2(N))$. Therefore, as the input size increases by 2 times, we expect the runtime to increase by $2 + \frac{2}{\log_2(N)}$ times, for any integer N larger than 1. As N grows towards infinity, this factor will slowly converge to 2, which is a good improvement compared to gnome sort's constant quadrupling of the runtimes. Table 5.1 below captures the timing in seconds across both implementations and algorithms. Just like in Paper 1, the data was collected using a similar procedure as the tests in Table 4.1, except in this case, the input sizes are not randomized.

	Gnome Sort		Tim Sort	
Size	**C time**	**Python time**	**C time**	**Python time**
1	0.0036079883575439	3.814697265625e-06	3e-06	2.002716064453125e-05
2	0.0034897327423095	3.814697265625e-06	3e-06	1.6927719116210938e-05
4	0.0026938915252685	1.0967254638671877e-05	2e-06	2.09808349609375e-05
8	0.0040822029113769	2.193450927734375e-05	4e-06	8.177757263183594e-05
16	0.002831220626831	3.314018249511719e-05	3e-06	3.409385681152344e-05
32	0.0031731128692626	9.584426879882812e-05	5e-06	6.604194641113281e-05
64	0.00260591506958	0.0003721714019775	1.1e-05	0.0002639293670654
128	0.0031549930572509	0.0014398097991943	2e-05	0.0006182193756103
256	0.0030562877655029	0.0066609382629394	5e-05	0.0005402565002441

512	0.0043809413909912	0.0277392864227294	9e-05	0.001260757446289
1024	0.0044066905975341	0.0849688053131103	0.000196	0.0027310848236083
2048	0.0094079971313476	0.3572700023651123	0.000372	0.0068049430847167
4096	0.038686990737915	1.3045740127563477	0.000811	0.0147640705108642
8192	0.1283876895904541	4.858428955078125	0.001591	0.0282061100006103
16384	0.3987851142883301	18.592221975326535	0.003476	0.0573091506958007
32768	1.720034122467041	81.56968927383423	0.00629	0.1478898525238037
65536	7.761588096618652	402.0783739089966	0.012788	0.2414169311523437
131072			0.028925	0.5135059356689453
262144			0.055916	1.1155850887298584
524288			0.111155	2.066522121429444
1048576			0.219976	4.41825795173645
2097152			0.491725	10.389513969421388
4194304			0.954912	19.741071939468384
8388608			2.145837	41.55577111244202
16777216			4.062533	86.10627794265747
33554432			8.169145	179.5933849811554
67108864			16.881771	361.8881530761719

Table 5.1: Combined table of Gnome sort and Tim sort timings by implementation

Due to some slight overhead when reading and returning the sorted array values in paper 1 for the C implementation, the timing logic of the Timsort implementation was changed slightly. Instead of timing both programs from Python, separate timing logic was implemented in C. The user has full control in what they wish to see by simply passing in command line arguments. In this way, the timing would be confined to just the sort algorithm as opposed to the entire program itself. While it was shown in paper 1 that the overhead does not matter as much for larger datasets, I still chose to minimize it for better accuracy even if it meant for some small precision loss due to the differences in how C times work.

There are also more runs for the Tim sort algorithm timings than gnome sort because I wanted to also see just how far I could increase the data sizes until I wound up with a time that is close to gnome sort's ending runtime. The number of tests was determined by simple trial and error, and about 27 tests were needed, leading to a maximum data size of 2^{26} (67,108,864) elements. Comparing this value to gnome sort's highest data size, this means that Tim sort performs roughly 1024 times more efficiently than Gnome sort overall. The table also shows that the Tim sort runtimes look to be doubling as expected for larger array sizes. Figure 5.2 on the next page also represents 2 plots that compare the available data for both algorithms across both implementations.

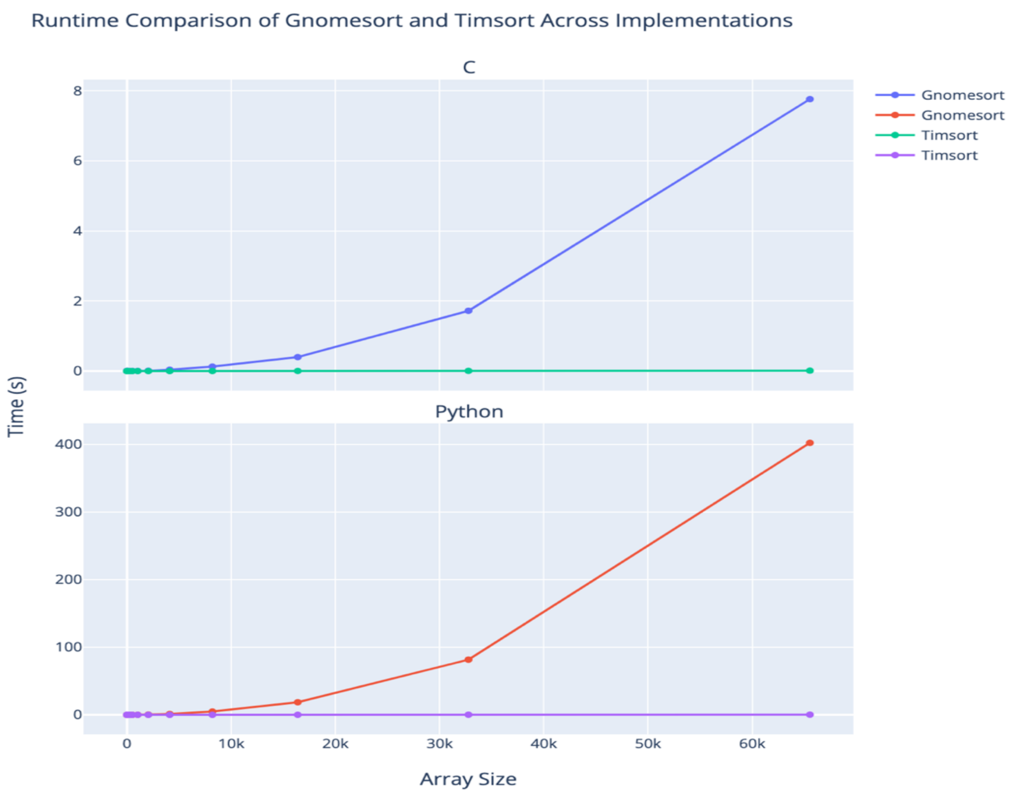

Figure 5.2: Runtimes of algorithms across both implementations

Once again, the plots prove that Tim sort is far better of the 2 algorithms. In both, Gnomesort exhibits the same type of increasing trend in the run times. This trend was shown to be quadratic and corresponds to Gnomesort's $O(N^2)$ time complexity in paper 1. Timsort on the other hand in this graph is relatively flat, which indicates how efficient it truly is. When comparing based on the biggest array size for Gnomesort, the C implementation of the algorithm runs 607.32 times slower than its Timsort implementation. For Python, it runs about 1665.6 times slower. This gap will only increase as more data is tested. While these 2 plots are good at assessing which algorithm is better, to truly capture the scale of performance and assess the empirical time complexity, the data first needs to be normalized through a transformation. Figure 5.3 represents the transformed data using a log transform on both axes.

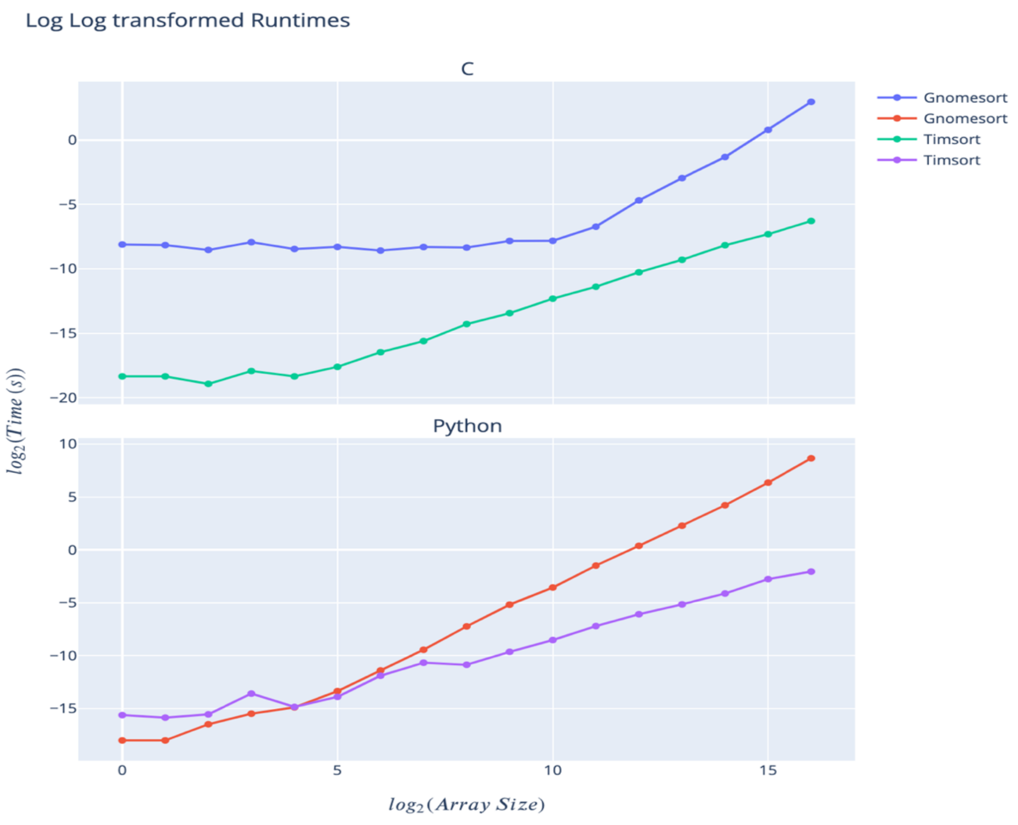

Figure 5.3: Log runtime over log array size

The transformed data also proves that overall, timsort's runtime is far more efficient than gnomesort. There is a small anomaly with Python where the gnomesort implementation is slightly faster for smaller datasets, which could be attributed to the implementation of the run length calculation across languages as well as other factors pertaining to the OS and hardware. However, the times later have a stable trend. We can also see that while the gradient for Tim sort curves is less overall than for Gnomesort, the relationship is somewhat curvilinear. This implies that the Since Tim sort exhibits almost the same exact operations as merge sort, which has $O(N\ log_2(N))$, the empirical time complexity therefore matches the expected time complexity as well, and any differences are just simply variations in the machine itself. Finally, figure 5.4 just compares the 2 implementations of Timsort with each other.

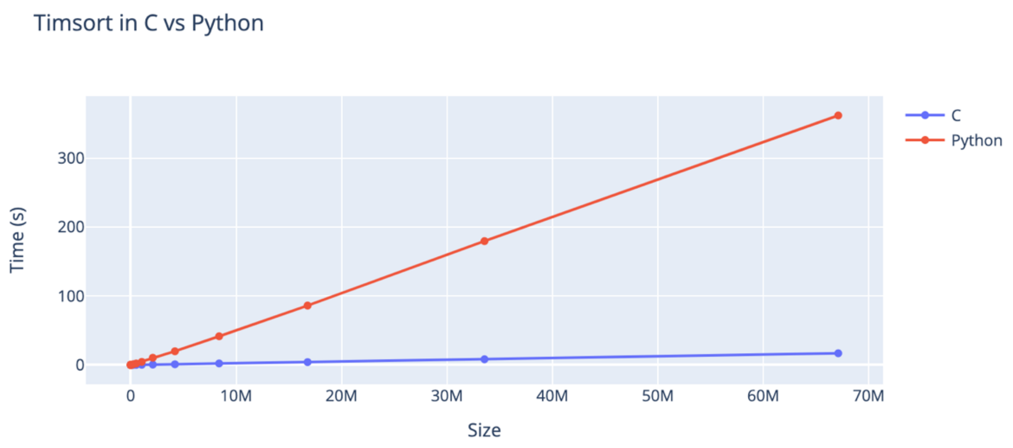

Figure 5.4: Timsort runtime in C vs Python

It is no surprise that just like in paper 1, C is the better language overall when it comes to sorting using Tim sort, as it is more low level and compiled, which leads to better performance. Referencing the table, the highest array size in C took about 16.8 seconds to complete. The same size in Python took 361.9 seconds. Since the growth rate is almost constant between implementations, we can expect that the C implementation would be about 21 times faster than Python. This also makes sense because in paper 1, gnome sort was determined to run 50 times faster in C, and the runtime of Tim sort nearly doubles as the data sizes double instead of quadruples.

Conclusion

Overall, I expected that Tim sort would be better than Gnome sort hands down. You can think of Tim sort as a hybrid between insertion and merge sort, whereas Gnome sort behaves like a hybrid between bubble and insertion sort as mentioned in paper 1. Both algorithms can be considered hybrids. However, while Gnome sort has features from 2 inefficient algorithms, Tim sort leverages insertion sort's inefficiency by only applying it for run correction. Much of the work is just building already sorted runs from parts of the dataset and then merging each part together. This is also the reason why it outperforms efficient algorithms like merge sort. Tim sort is the prime example of an algorithm that when developed and planned correctly, is scalable in solving problems optimally in the real world.

References

Abdalla, Safia. “The Most Important Sorting Algorithm You Need to Know.” *DEV Community*, DEV Community, 10 Feb. 2020, dev.to/captainsafia/the-most-important-sorting-algorithm-you-need-to-know-38e0.

Bauermeister, Rylan. “Understanding Timsort.” *Medium*, Medium, 24 Apr. 2019, medium.com/@rylanbauermeister/understanding-timsort-191c758a42f3.

Cox, Graham. “How Does Timsort Work?” Baeldung on Computer Science, 12 July 2023, www.baeldung.com/cs/timsort.

“Interview with Tim Peters.” *YouTube*, YouTube, 1 July 2012, www.youtube.com/watch?v=1wAOy88WxmY.

Mervin, Eric. “What Is Timsort?” Medium, Medium, 24 Mar. 2021, ericmervin.medium.com/what-is-timsort-76173b49bd16.

Timur, Bekmurzin. “How Javascript Sorts? Timsort Algorithm.” *DEV Community*, DEV Community, 4 May 2023, dev.to/bekmurzintimur/how-arrayprototypesort-works-3kcn.

“Tim Sort.” *Javatpoint*, www.javatpoint.com/tim-sort. Accessed 11 Mar. 2024.