Weekly Challenge 03
Timsort: Merge Algorithm Analysis
CS/CE 412/471 Algorithms: Design & Analysis (2025)

Objective

To understand how the nature of the input influences the number of operations during the merge phase of Timsort.

Description

Timsort is similar to merge sort but starts by identifying already sorted segments, called runs, in the array. Runs contain elements in either ascending or descending order (see Figure 1). While random data has few runs, real-world data often has many, making Timsort faster than plain merge sort and achieving linear time in the best case.

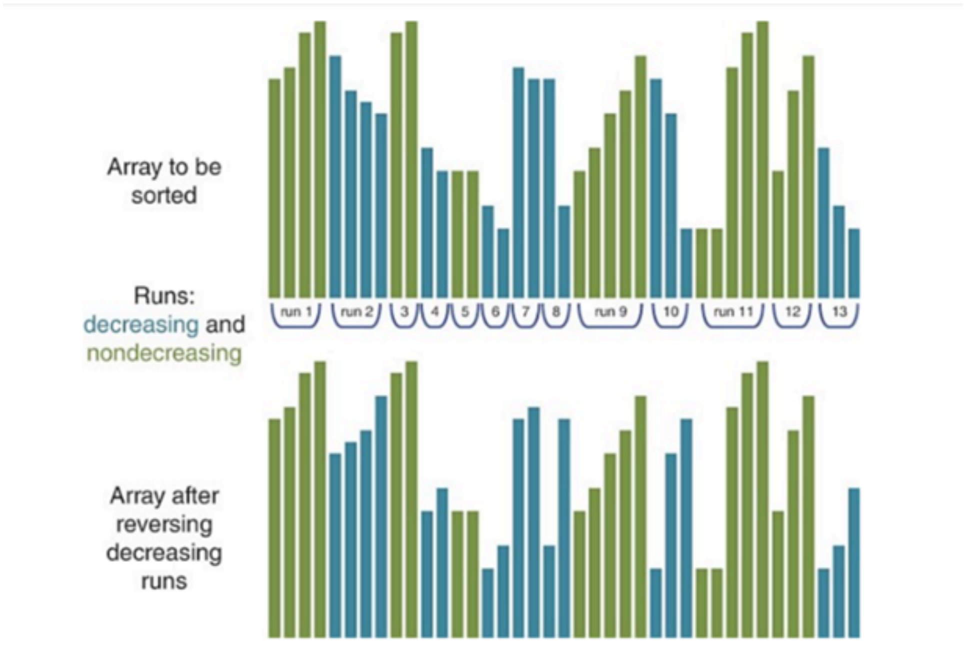


Figure 1: Illustration of how Timsort identifies runs. Taken from *Data Structures and Algorithms in Python* by Robert Lafore, Alan Broder, and John Canning.

Timsort processes the array by finding runs and storing their bounds on a stack. It uses rules to decide when to merge runs and ensures certain conditions are met for the top three runs on the stack, labeled A, B, and C (with A at the top):

- $\text{len}(A) > \text{len}(B) + \text{len}(C)$
- $\text{len}(B) > \text{len}(C)$

If the first condition is violated, the smaller of A and C is merged with B, and the result replaces the merged runs in the stack. Merging continues until both conditions are satisfied.

Algorithms

Timsort_Algorithm

1. Set `low` to zero.
2. While `low` is less than the length of the array:
   (a) Find the length of the run starting at `low`.
   (b) If this is a decreasing run, reverse it.
   (c) Add this run to the stack.
   (d) Set `low` to `low + length of the run`.
   (e) Collapse-merge the stack to satisfy the invariants.
3. Force-merge the stack until there is only one run on it.

Collapse-Merge Procedure

1. While the stack size is > 1:
   (a) Let `top` be the index of the top of the stack.
   (b) If *top* > 1 and If `stack[top-2].length` $\leq$ `stack[top-1].length + stack[top].length`:
      i. If `stack[top-2].length` $<$ `stack[top].length`, merge `stack[top-2]` and `stack[top-1]`.
      ii. Else, merge `stack[top-1]` and `stack[top]`.
   (c) Else if `stack[top-1].length` $\leq$ `stack[top].length`, merge `stack[top-1]` and `stack[top]`.
   (d) Else, exit the loop.

Force-Merge Procedure

1. While the stack size is > 1, merge `stack[top-1]` and `stack[top]`.

Merge_Algorithm

```
1. C := new empty list
2. while A is not empty and B is not empty do
    (a) if head(A) $\leq$ head(B) then:
        i. append head(A) to C
        ii. drop the head of A (move to next element of A)
    (b) else:
        i. append head(B) to C
        ii. drop the head of B (move to next element of B)
3. while A is not empty do:
    (a) append head(A) to C
    (b) drop the head of A
4. while B is not empty do:
    (a) append head(B) to C
    (b) drop the head of B
```

Task

For arrays of sizes 20, 144, and 240 (or nearest for case 2):

1. Calculate the maximum number of comparisons at line 2(a) of the `Merge_Algorithm()` required to sort an array.
2. Plot the number of comparisons against the array size to identify the order of growth.
3. In 4–5 lines, explain the observed trend.

Cases

Case 1: Alternating high and low values. Each run is of size 2. For 20 elements, runs are of the form: [2][2][2][2][2][2][2][2][2][2] (10 runs of size 2).
Case 2: Follows the Golden Ratio. For 20 elements, runs are of the form [1][1][2][3][5][8] (6 runs).
Case 3: Equal-size runs. For 20 elements, runs are of the form: [4][4][4][4][4] (5 runs). Extend similarly for 144 and 240 elements.

References

1. *Data Structures and Algorithms in Python*, by Robert Lafore, Alan Broder, John Canning.

2. *Python Algorithms: Mastering Basic Algorithms in the Python Language*, by Magnus Lie Hetland.

3. *Data structures abstraction and design using Java*, by Elliot B. Koffman, Paul A. T. Wolfgang.

Rubric

Criteria	**Excellent (4)**	**Good (3)**	**Fair (2)**	**Poor (1)**
Calculation of Comparisons	Correctly calculates the maximum number of comparisons (worst case) for all given array sizes (20, 144, 240).	Calculates the maximum comparisons for all cases with minor inaccuracies.	Partial or incorrect calculations for one or more array sizes.	Fails to calculate comparisons for most or all cases.
Plot Creation	Generates a clear and accurate plot of comparisons against array sizes, with proper labels, titles, and legends to illustrate the order of growth.	Creates a plot showing comparisons and growth trends but lacks proper labeling or minor errors in presentation.	Creates a plot, but it is unclear, mislabeled, or does not represent the growth trend accurately.	No plot provided or plot is irrelevant/ inaccurate.
Analysis of Observed Trend	Provides a concise and insightful explanation of the trend (e.g., O(n log n) complexity), connecting it to algorithm behavior and implications for scalability.	Provides a good explanation of the trend, but lacks depth or connections to algorithm behavior and implications.	Provides a basic explanation of the trend but misses key insights, such as order of growth or implications.	Explanation is missing, unclear, or incorrect.

1. Excellent: 18-20 points

2. Good: 14-17 points

3. Fair: 10-13 points

4. Poor: $\leq$10 points

Weekly Challenge 03 Report: Timsort Merge Analysis

bq08283

CS/CE 412/471 Algorithms: Design Analysis
Habib University
Spring 2025

Introduction

This report examines the number of comparisons required during the merge phase of the Timsort algorithm for arrays of sizes 20, 144, and 240. The results for three distinct cases are analyzed and compared.

Maximum Number of Comparisons

- **Case 1: Alternating high and low values**
  - Size 20: 63
    For an array of size 20, the subarrays are divided as follows:

Subarray Number	Subarray Size
1	2
2	2
3	2
⋮	⋮
10	2

Table 1: Initial Subarray Sizes

The formula for total comparisons during the merge is:

$$m + n - 1$$

where:

  * m is size of the first subarray
  * n is size of the second subarray

The calculations for merge comparisons are as follows:

$$\begin{aligned}
\text{Step 1: } & [(2+2)-1] \times 5 = 15 \\
\text{Step 2: } & [(4+4)-1] \times 2 = 14 \\
\text{Step 3: } & [(8+8)-1] \times 1 = 15 \\
\text{Step 4: } & [(16+4)-1] \times 1 = 19
\end{aligned}$$

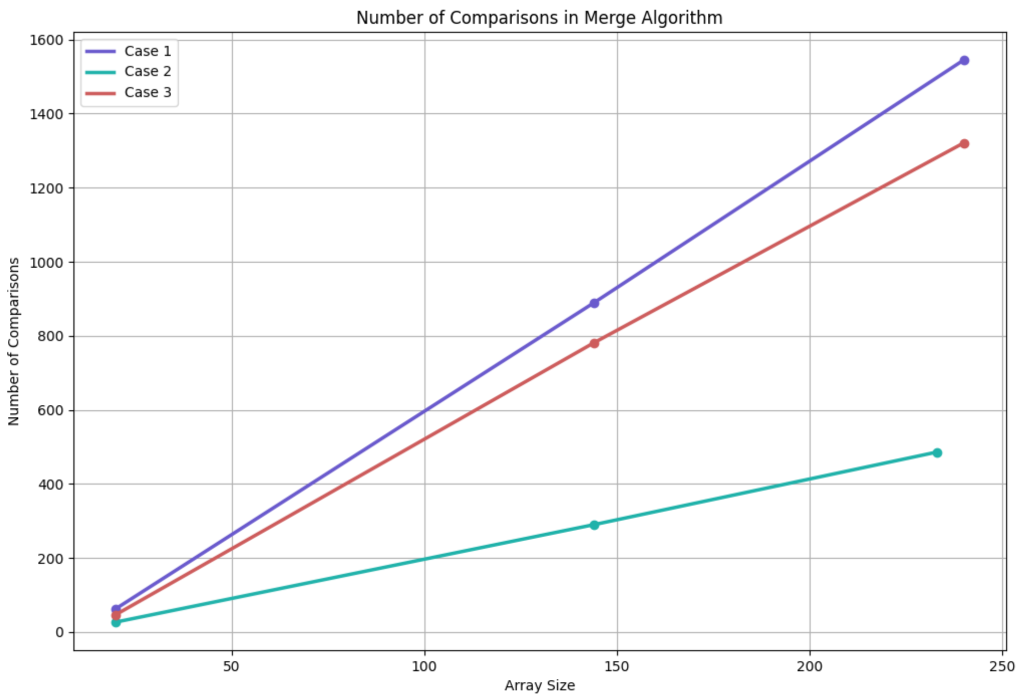
Summing the values from each step:

$$15 + 14 + 15 + 19 = 63$$

- Size 144: 889
- Size 240: 1545

- **Case 2: Follows the Golden Ratio**
  - Size 20: 27
  - Size 144: 290
  - Size 240: 486
- **Case 3: Equal-size runs**
  - Size 20: 48
  - Size 144: 781
  - Size 240: 1365

Visualization



Trend Analysis

Though their gradients vary because of the initial run sizes, the trend indicates that all three cases follow the $O(n)$ growth.

- Case 1 has the steepest gradient, because smaller runs result in more frequent merging and higher comparisons.
- Case 2 has a least steep gradient, reflecting the smallest number of comparisons because Fibonacci-like run sizes lead to fewer initial merges, which reduces the overall number of comparisons.
- Case 3 has a moderate gradient, falling between Cases 1 and 2. Although there are fewer merges overall than in Case 1, the comparisons are higher than in Case 2 because of the uniform run distribution.

Though other cases match more with the worst-case complexity, Case 2 is the most efficient, showing a performance closer to the best-case complexity.