TimSort Executive Summary

Aidan Tan, Sarah Warren, Philip Yao

Introduction/ Problem Statement:

Efficient sorting has a wide range of applications in computer science. Traditional sorting algorithms like Merge Sort or Quick Sort assume that the input is completely unordered. Timsort is a hybrid sorting algorithm designed to solve the general problem of sorting data efficiently, but is optimized for patterns commonly found in real-world data. Naturally occurring data often contains partially sorted segments, or "runs," and strictly decreasing segments. This leads to them doing unnecessary work, resulting in suboptimal performance. Timsort has a runtime of O(nlogn) in the average and worst case, and an optimal O(n) time complexity when the input is already sorted or nearly sorted. Timsort is also a stable sorting algorithm, meaning that elements with equal value remain in the same order as their input. TimSort was introduced by Tim Peters in 2002 and is now the default sorting algorithm in Python and Java.

Overall Intuitive Approach / Solution:

Timsort combines Insertion Sort, which is fast on small and partially ordered inputs, and uses the strategy used in Merge Sort to combine those runs. It also looks for strictly decreasing segments of data and reverses them. When merging runs together, the algorithm uses a stack to ensure that merging is balanced. Galloping mode is another optimization technique that is triggered when one run keeps winning merge comparisons. In this case there is an exponential or binary search to find the next element, and the algorithm will copy a section of the run into the merged list to avoid one-by-one comparisons and merges when unnecessary. A dataset of daily stock prices is a good example of data that Timsort is designed to be very fast at handling. The prices change gradually and follow patterns, so the list will already be partially sorted and have decreasing runs that the algorithm will detect and reverse.

Implementation:

Timsort works in three stages. It first determines the minimum size of the run, separates the data into runs, and merges them. As an adaptive sorting algorithm, Timsort determines some variables based on its input. MinRun, which is the size of the runs that the algorithm initially splits into, is determined by the size of the input. The size divided by the minimum run should be equal to or slightly less than a power of two so that the merging is as stable as possible. The minimum run is between 32 and 64, inclusive.

```
def calculate_min_run(n):
    r = 0
    while n >= 64:
        r = r | (n & 1)
        n = n >> 1
    return n + r
```

The function for MinRun takes the size of the input, n, and instantiates an offset variable, r = 0. While n is greater than or equal to 64, which is the maximum run size, r = r | (n & 1). The next operation is n >> 1, which effectively divides n by two and rounds down to the nearest even number. The final value of minRun is the sum of n and r. This method ensures that the minimum size of the runs is as close to or equal to a power of 2. This makes merging efficient because each merge reduces the problem by 2.

```
def timsort(arr):
    n = len(arr)
    min_run = calculate_min_run(n)
    run_stack = []

    i = 0
    while i < n:
        run_end = min(i + min_run - 1, n - 1)
        insertion_sort(arr, i, run_end)
        run_stack.append((i, run_end))
        i = run_end + 1

        merge_collapse(arr, run_stack)

    while len(run_stack) > 1:
        merge_at(arr, run_stack, len(run_stack) - 2)
```

Once MinRun is determined, the algorithm passes through the data to split it into $\frac{size(data)}{MinRun}$ number of runs. The data is split with one pass through the list. If the run is slightly increasing and the next element is decreasing, Insertion sort is used to sort this element in its place within the run. If the current run is strictly decreasing, and the next element increases, the algorithm reverses the current run so that it is now increasing, and then uses Insertion sort to insert the next element. Insertion sort works best on smaller and partially ordered inputs. If any runs are strictly decreasing, it reverses these segments. At the end of the pass through the list, the input has separated into sorted chunks.

```
def insertion_sort(arr, left, right):
    if left < right and arr[left] > arr[left+1]:
        is_desc = True
        for k in range(left, right):
            if arr[k] < arr[k+1]:
                is_desc = False
                break
        if is_desc:
            i, j = left, right
            while i < j:
                arr[i], arr[j] = arr[j], arr[i]
                i += 1
                j -= 1
            return
    for i in range(left + 1, right + 1):
        temp = arr[i]
        j = i - 1
        while j >= left and arr[j] > temp:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = temp
```

The next step is to use balanced merging to combine the runs efficiently. The algorithm stores the runs on a stack, and the merge policy involves runs X, Y, and Z (X at the top). It maintains the rule that |Z| > |Y| + |X|, and |Y| > |X|. This means that the run furthest to the top of the stack, Z, should be greater than its successor, Y, which should be greater than the size of the run on the top of the stack. If |X| + |Y| ≥ |Z| , runs X and Y are popped off the stack, merged using Merge sort, and put back on the stack.

```
def merge_collapse(arr, run_stack):
    while len(run_stack) > 2:
        X = run_stack[-3]
        Y = run_stack[-2]
        Z = run_stack[-1]
        lenX = X[1] - X[0] + 1
        lenY = Y[1] - Y[0] + 1
        lenZ = Z[1] - Z[0] + 1

        if lenX <= lenY + lenZ or lenY <= lenZ:
            if lenX < lenC:
                merge_at(arr, run_stack, len(run_stack) - 3)
            else:
                merge_at(arr, run_stack, len(run_stack) - 2)
        else:
```

```
            break

    if len(run_stack) == 2 and (run_stack[-2][1] - run_stack[-2][0] + 1) <=
(run_stack[-1][1] - run_stack[-1][0] + 1):
        merge_at(arr, run_stack, len(run_stack) - 2)

def merge_at(arr, run_stack, i):
    start1, end1 = run_stack[i]
    start2, end2 = run_stack[i + 1]

    merge(arr, start1, end1, end2)
    run_stack[i] = (start1, end2)
    del run_stack[i + 1]

def merge(arr, left, mid, right):
    left_part = arr[left:mid + 1]
    right_part = arr[mid + 1:right + 1]
    i = j = 0
    k = left
    while i < len(left_part) and j < len(right_part):
        if left_part[i] <= right_part[j]:
            arr[k] = left_part[i]
            i += 1
        else:
            arr[k] = right_part[j]
            j += 1
        k += 1
    while i < len(left_part):
        arr[k] = left_part[i]
        i += 1
        k += 1
    while j < len(right_part):
        arr[k] = right_part[j]
        j += 1
        k += 1
```

“Galloping mode” is an optimization that speeds up merging when two runs have very different values. If the elements compared in run A are greater than those in run B more than *MinGallop* times, this optimization is triggered. It works by binary or exponentially searching run B for the next element that exceeds A’s current winner. When this element is found, the chunk of elements is copied and merged into place. This optimization cuts down on comparisons when one list is much greater than another, which can be a common occurrence in the target datasets.

```
1   def gallop(x, arr, start):
2       hi = 1
3       n = len(arr)
4       while start + hi < n and x > arr[start + hi]:
5           hi *= 2
6       lo = hi // 2
7       hi = min(start + hi, n)
8       while lo < hi:
9           mid = (lo + hi) // 2
10          if x > arr[mid]:
11              lo = mid + 1
12          else:
13              hi = mid
14      return lo
```

Summary of Programming Challenge:

The programming challenge describes a lighthouse that determines the docking order of ships at sea. Its beam rotates counterclockwise, identifies ships based on their position, visibility, and proximity to the lighthouse's center at (0,0). The beam has constraints on the number of ships it can process at one time, simulating Timsort's structure of sorting smaller segments of its input, or runs. Furthermore, the beam has a set radius meaning only certain ships will be visible and able to dock. The aim of this challenge is to provide an environment where Timsort would be most efficient while modifying it with the twist of a graph and coordinate based angle comparator. This programming challenge meets those learning objectives by adding certain constraints to the solution so that it necessitates Timsort - for example, the number of ships able to be processed at one time naturally fits into Timsort's idea of "runs." An autograder configured for this challenge would have strict time constraints to ensure the algorithm includes the optimizations that make Timsort fast.

Key Ideas for Solving Programming Challenge:

The main points to consider when approaching the programming challenge are: Timsort's runs, angle comparison constraints, and stable sorting.

When solving the problem, the first key idea is the predefined run size. The input specifies how many ships the lighthouse can process at a time which can be interpreted as *MinRun* size, intuitively aligning with how Timsort organizes data into manageable chunks. This ensures the solution addresses Timsort's strength in partially ordered segments. If not adhered to, depending on the size of the dataset, it may become inefficient e.g. if MinRun is hardcoded to 64 but the input size is smaller, the algorithm may default to insertion sort which can be less efficient than Timsort. There are a number of test cases with sizes ranging from less than 10 ships to 1,000,000

ships in varying orders which requires the algorithm to use well defined run sizes to efficiently sort the data.

The second key idea is the polar angle comparator. Ships are sorted not by value alone but by their polar angle relative to the lighthouse's center at (0, 0), requiring careful handling of edge cases like overlapping angles. This geometric ordering makes the sorting nontraditional, adding a layer of complexity which must be addressed. There are a number of edge cases in the tests which require a robust angle comparator such as case 2 which addresses identical angles in which each ship is along the line x=y and case 6 and 7 in which ships are sorted in forward and reverse polar order respectively.

The last key idea is stable sorting. In cases where multiple ships occupy the same coordinates or angular direction, their original input order must be preserved as specified by the programming problem. This maps directly to Timsort's design as a stable sorting algorithm. This is addressed in test cases 8 and 9 which place ships along x=0 and y=0 respectively meaning that all ships are on the same angle and order must be preserved.

An additional challenge is optimizing the code to pass with certain time constraints in the autograder. The tests are set up with many worst case scenarios and to pass all tests, the implementation must correctly and efficiently trigger run reversal, galloping mode, and other key Timsort optimizations. These challenges all provide interesting twists, which when initially presented with may be confusing, but come with sleek and intuitive solutions once discovered. They are possible to self derive and the programming problem itself gives a number of pointers in terms of direction. The expected complexity of a correct implementation is consistent with Timsort's complexity of O(n log n) in the general case, and O(n) in best-case scenarios where ships are pre-sorted.

1	*simple test case - programming challenge example*
2	*edge case - every ship along y = x*
3	*edge case - every ship along fixed radius*
4	*edge case - duplicate ship coordinates*
5	*edge case - no ships in query*
6	*edge case - 20 ships provided in sorted polar order*
7	*edge case - 20 ships provided in reverse polar order*
8	*edge case - 20 ships along x = 0*
9	*edge case - 20 ships along y = 0*
10	*runtime: 1,000 random ships and 10 queries*
11	*runtime: 5,000 random ships and 10 queries*
12	*runtime: 10,000 random ships and 10 queries*
13	*runtime: 50,000 random ships and 10 queries*
14	*runtime: 150,000 sorted ships and 10 queries*
15	*runtime: 150,000 reverse sorted ships and 10 queries*

```
16  runtime: 150,000 random ships and 10 queries
17  runtime: 150,000 sorted ships with 150,000 reverse sorted ships and 10 queries
18  runtime: 300,000 sorted ships and 10 queries
19  runtime: 300,000 reverse sorted ships and 10 queries
20  runtime: 300,000 random ships and 10 queries
```

Conclusion:

Timsort uses a variety of strategies from other sorting algorithms, mainly Insertion and Merge sorts, to make it adaptable to its input and efficient on real world data. Variables like MinRun and MinGallop that are determined based on each unique dataset make this algorithm adaptable and able to be very efficient no matter its input. Timsort's adaptability and stability make it ideal for use in many programming languages' sorting libraries, like Java and Python.

References:

https://www.kirupa.com/sorts/timsort.htm
https://www.youtube.com/watch?v=0Dg41UEK3Io
https://github.com/kirupa/kirupa/blob/master/data_structures_algorithms/timsort.htm
https://www.chrislaux.com/timsort