Project 1

Integration of Mergesort & Insertionsort

Mabel, Ming Kai, Aditi

01 - Algorithm Implementation

```python
def hybrid_sort(arr, l, r, S):
    key_comparison = 0

    if r - l + 1 <= S: # If subarray is lesser than subarray
        subarray = arr[l:r + 1]  # Extract the subarray
        key_comparison += insertion_Sort(subarray)  # Sort the subarray with Insertion Sort
        arr[l:r + 1] = subarray  # Insert sorted subarray back into array
    else:
        if l < r:
            # Use standard merge for larger subarrays
            m = l + (r - l) // 2

            # Sort 1st and 2nd halves
            key_comparison += hybrid_sort(arr, l, m, S)
            key_comparison += hybrid_sort(arr, m + 1, r, S)

            # Merge both the sorted halves
            key_comparison += merge(arr, l, m, r)
    return key_comparison
```

Tips

```python
def insertion_Sort(arr):  # Returns number of key comparisons.
    key_comparison = 0
    for i in range(1, len(arr)):
        temp = arr[i]
        j = i - 1
        # Move elements of arr[0..i-1], that are
        # greater than temp, to one position ahead
        # of their current position
        while j >= 0 and temp < arr[j]:
            key_comparison += 1
            arr[j + 1] = arr[j]
            j -= 1
        # One more comparison happens after the while loop, if j >= 0
        if j >= 0:
            key_comparison += 1
        arr[j + 1] = temp
    return key_comparison
```

Hybird_sort function enhances sorting efficiency by dynamically switching between insertion sort for smaller subarray and merge for large ones

01 - Algorithm Implementation

Algorithm

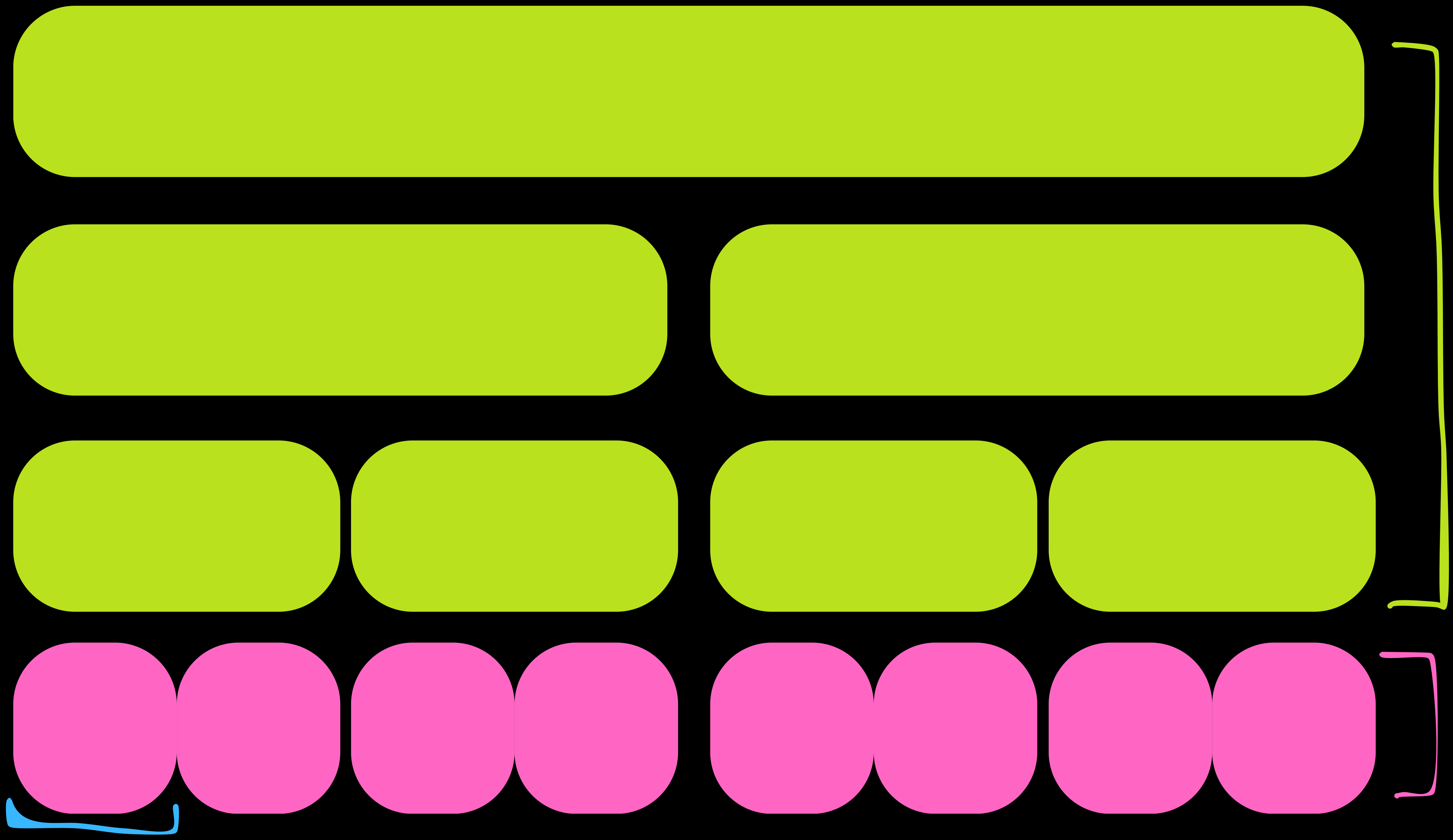
```python
def merge(arr, l, m, r): # Not sorted inplace.
    key_comparison = 0
    # Find sizes of two subarrays to be merged
    n1 = m - l + 1
    n2 = r - m

    # Create temp arrays
    L = [0] * n1
    R = [0] * n2

    # Copy data to temp arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0  # Initial index of first subarray
    j = 0  # Initial index of second subarray
    k = l  # Initial index of merged subarray

    while i < n1 and j < n2:
        key_comparison += 1;
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[], if any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[], if any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

    return key_comparison
```

Flow of Hybrid Sort:

1. Check if Array Size is lesser than threshold
    a. Extract the sub array from current array
    b. Sort subarray with insertion sort
    c. insert the subarray back to current array
2. If array size is bigger than threshold
    a. If left pointer is smaller than right pointer
        i. Call hybrid sort with 1st half of array
        ii. Call hybrid sort with 2nd half of array
        iii. Merge and sort both halfs of the array

02 - Time Complexity Analysis

Theoretical Analysis for Hybrid Sort



kth layer:

$$S = \frac{n}{2^k}, \quad k = \log_2 \frac{n}{S}$$

$$\log_2 \frac{n}{S} \cdot n$$

$$\frac{1}{4} S^2 \cdot \frac{n}{S} \quad -> \quad \frac{1}{4} S \cdot n$$

$$\frac{1}{2} \cdot \frac{(n-1)(n+2)}{2} \quad -> \frac{1}{4} S^2$$

02 - Time Complexity Analysis

Theoretical Analysis for Hybrid Sort Continued... ↴

Putting it together....

$$n \cdot \log_2 \frac{n}{S} + \frac{1}{4} S \cdot n \quad = O\left(n \cdot \log_2 \frac{n}{S} + S \cdot n\right)$$

Key Comparison (Fixed Threshold S, Varying Input N)

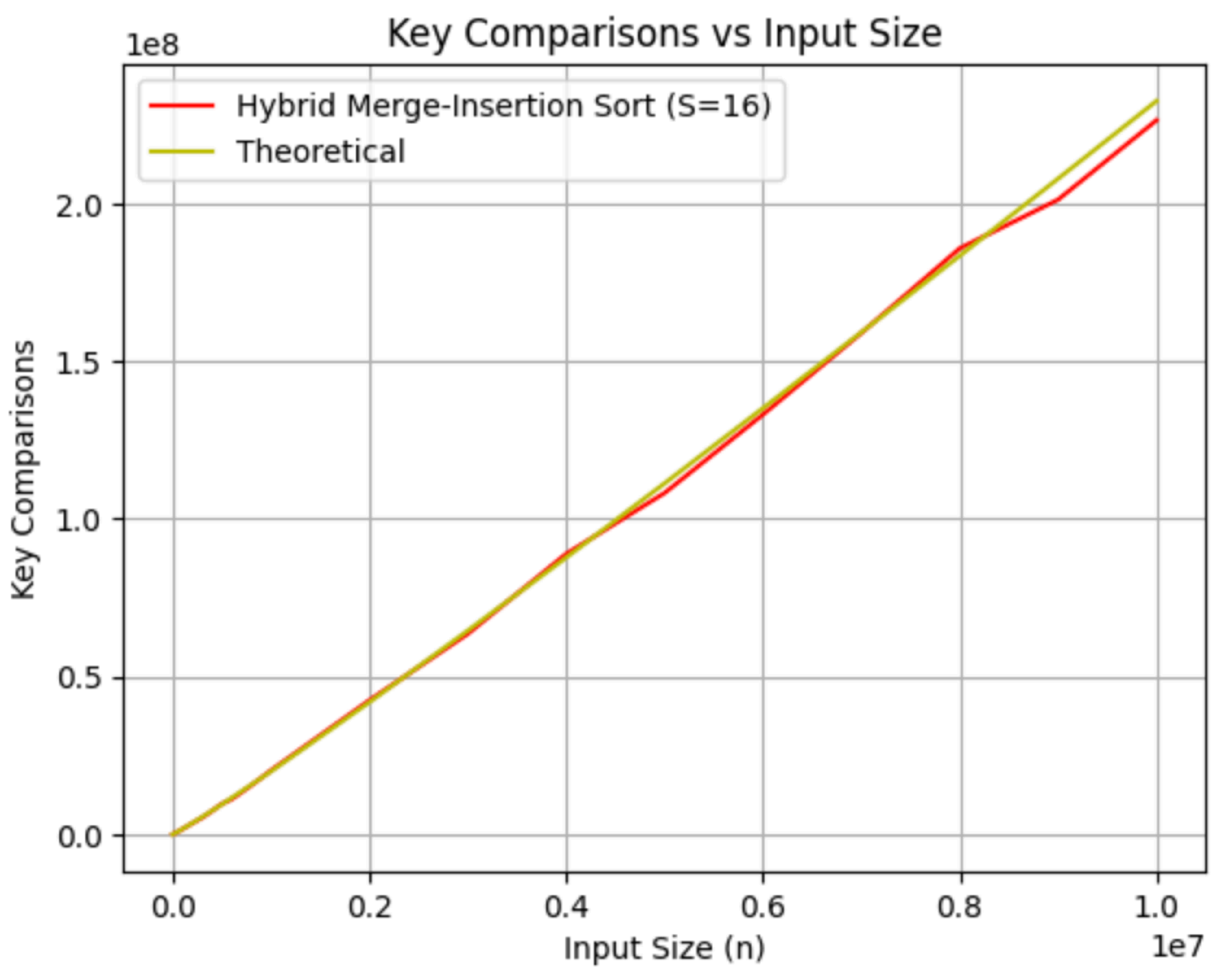
Method

1. Fixed S and varying n -> S threshold fixed at 16
2. Loop over different n sizes
3. Record Key Comparisons and Time for each
4. Plot graph to compare

Key Comparison (Fixed Threshold S, Varying Input N)

√ both theoretical & empirical models match closely

$$n \cdot \log_2\left(\frac{n}{16}\right) + \frac{16}{4} \cdot n$$

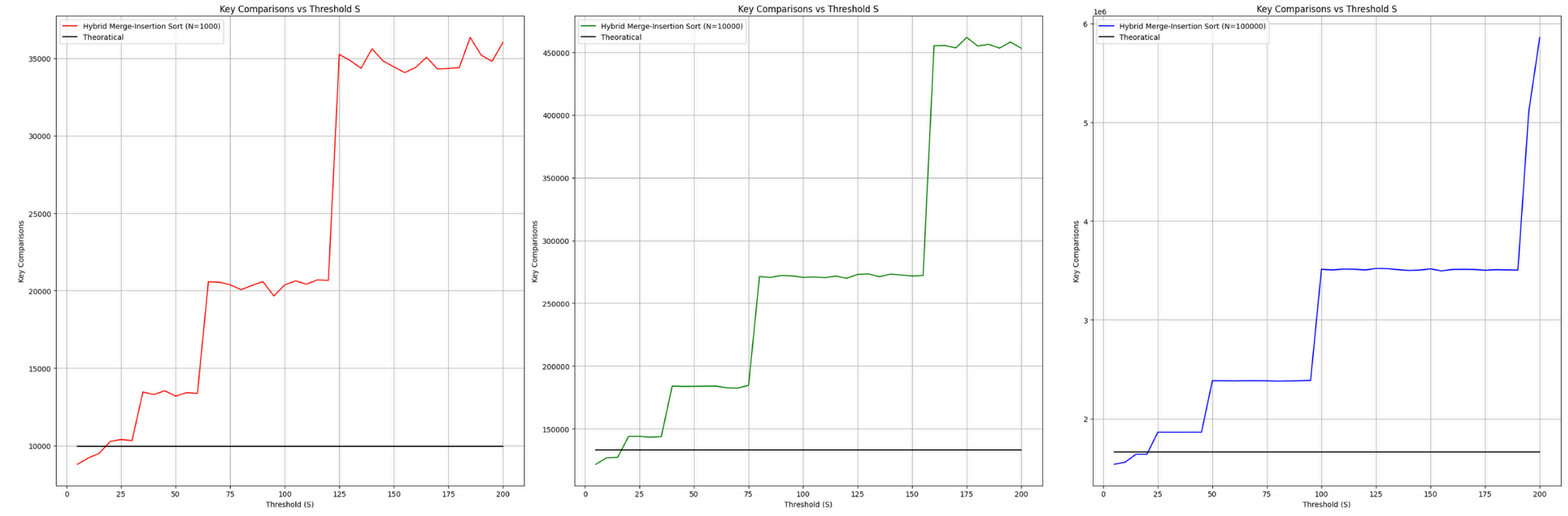


Key Comparison (Varying Threshold S, Fixed Input N)

Method

1. Fixed N size at 100,000
2. Loop and run over hybrid sort for each S
3. Record Key Comparisons and Time
4. Plot Graph

Key Comparison (Varying Threshold S, Fixed Input N)



- Both theoretical & empirical models match closely are quite different
- As n is fixed, input size doesn’t increase, thus theoretical model remains a straight line

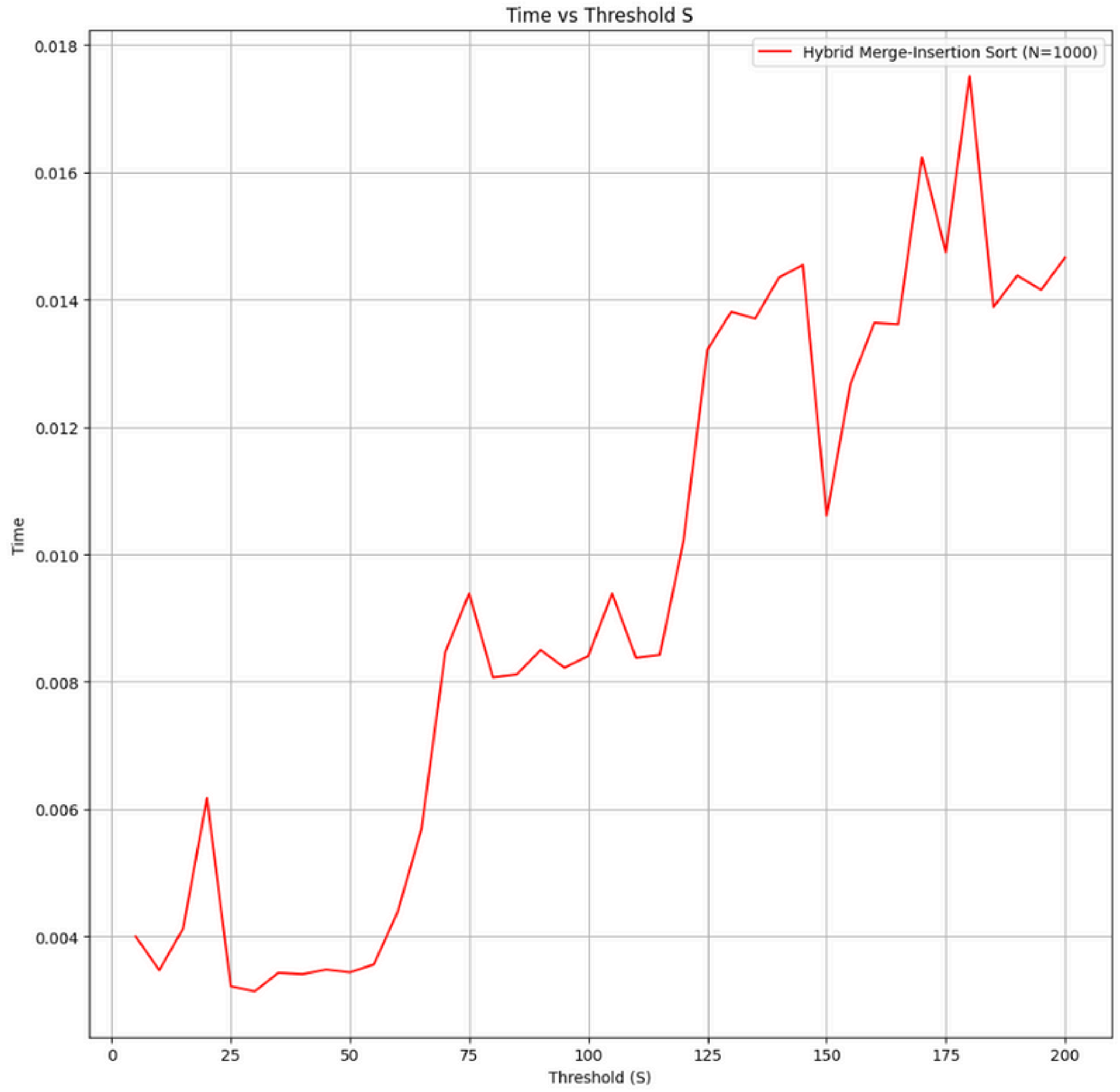
$$n \cdot \log_2\left(\frac{n}{16}\right) + \frac{16}{4} \cdot n$$

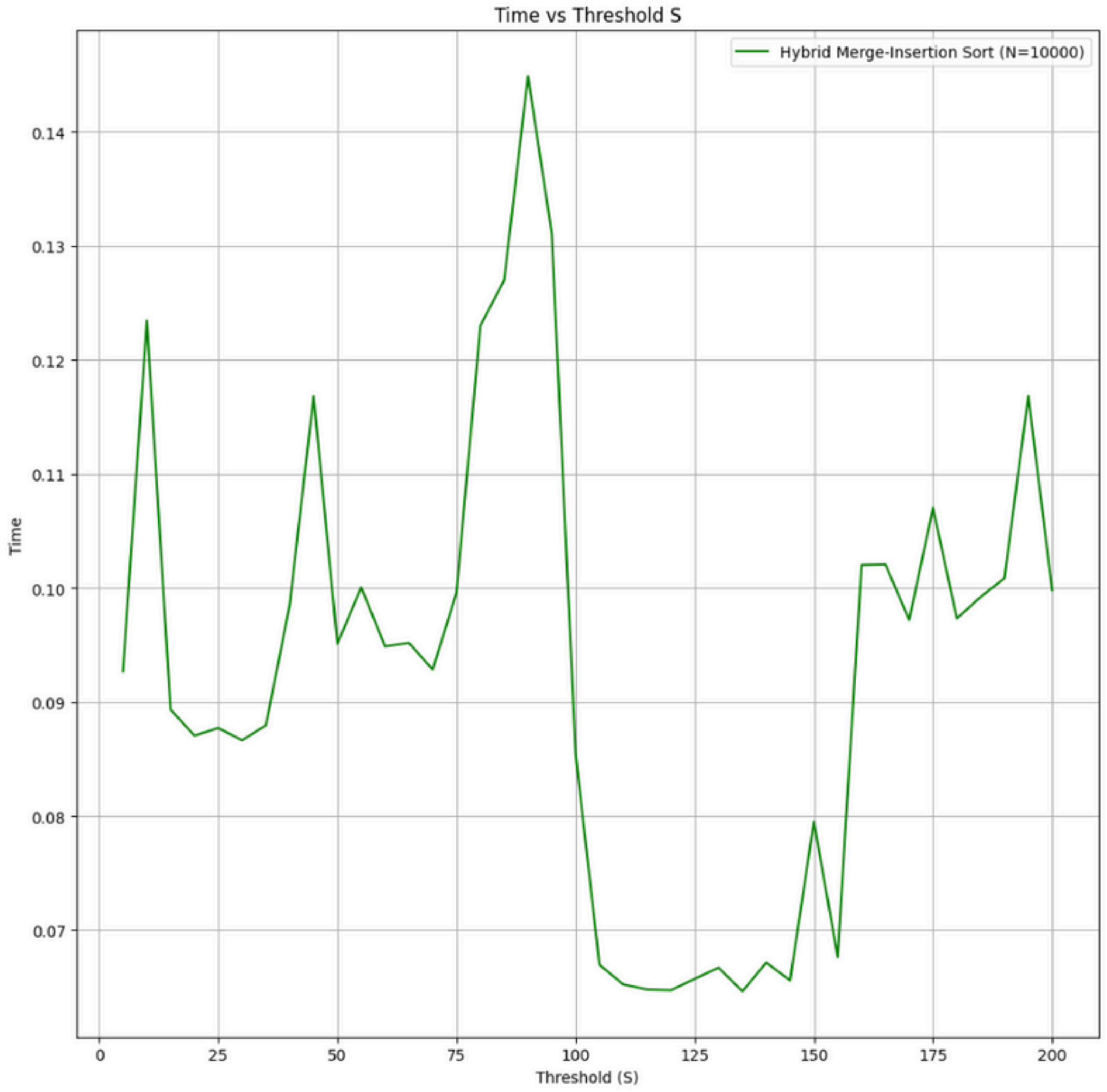
Optimal Value of S

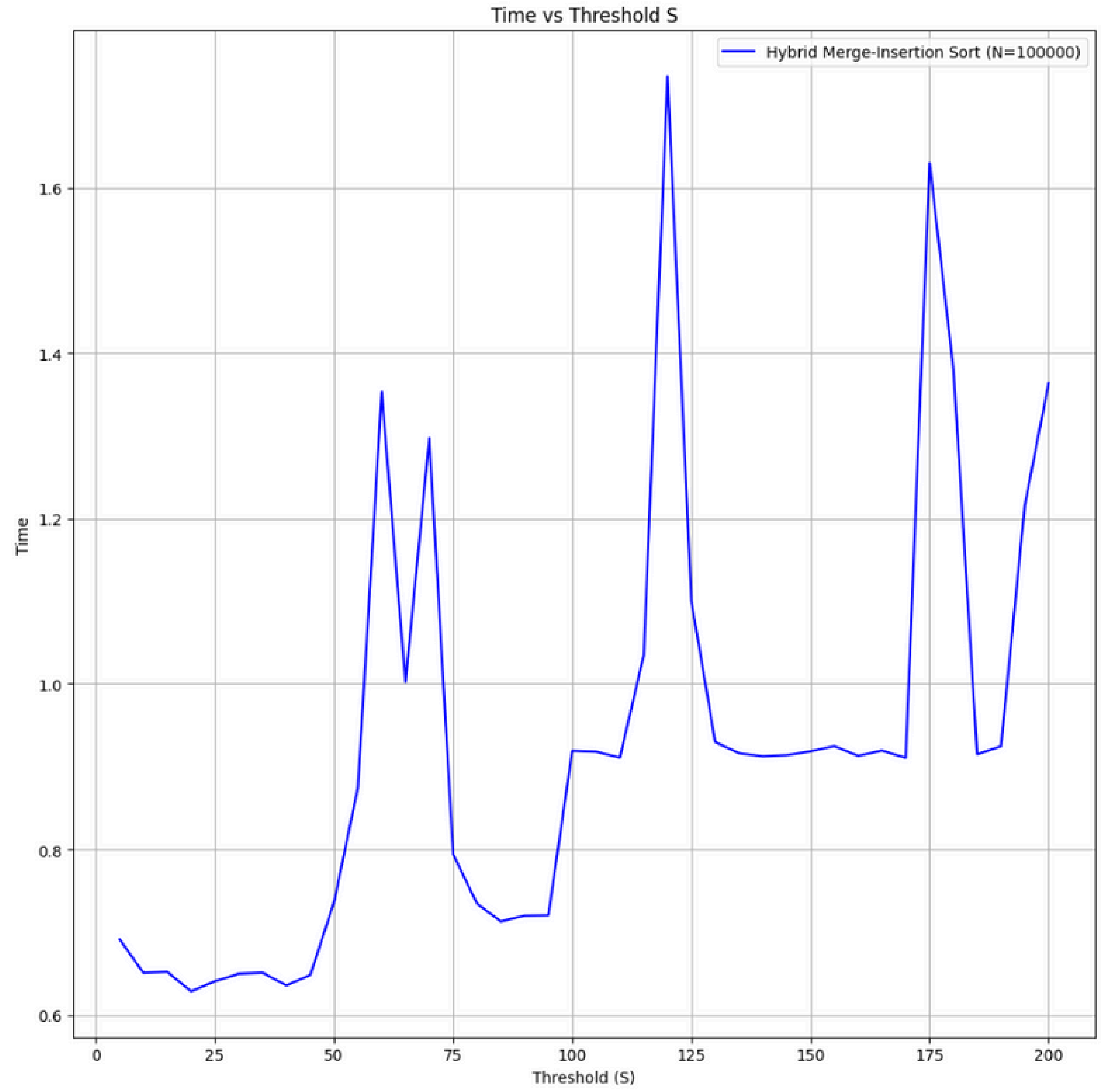
Method

1. Run for n = 1 000, 10 000, 100 000
2. Plot against Threshold and Time
3. Derive Optimal S from graph

Optimal Value of S

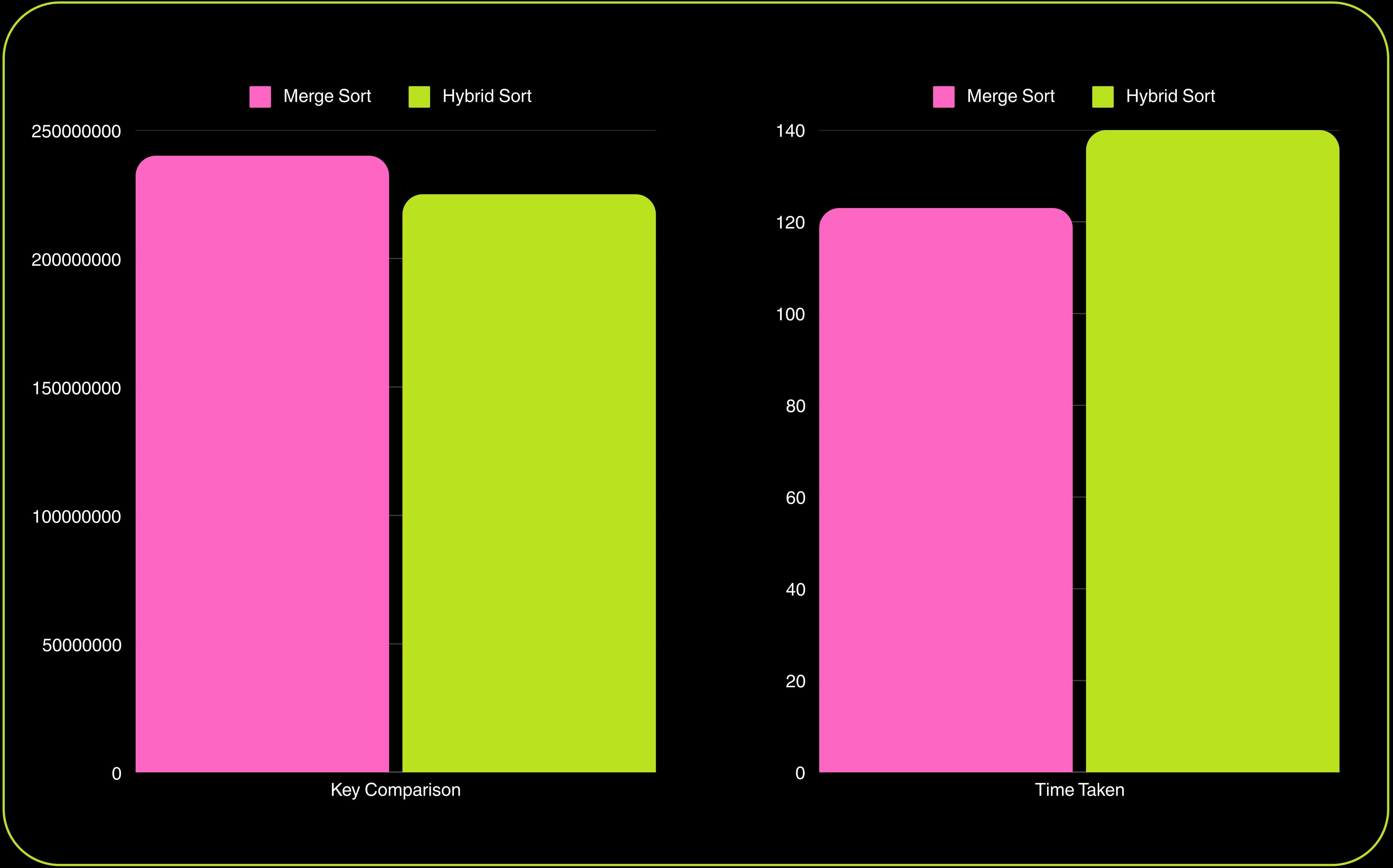






- The optimal S is the value where the time is the lowest.
- In the graph, there is a low point around $S \approx 30$–35, where the time is minimal.
- Optimal S seems to be 31

03 - MergeSort Comparison



- Optimal S: 31
- Dataset: 10 Million Integers

- Merge Sort
  - Key Comparisons: ≈230 000 000
  - CPU Time: ≈140s

- Hybrid Sort
  - Key Comparisons: ≈240 000 000
  - CPU Time: ≈130s

04 - Conclusions

- Hybrid Sort is better in terms of merging:
  - More efficient; lower CPU time usage
- Hybrid Sort had best CPU time performance with varying amounts of input data:
  - Merging small sorted subarrays to minimise comparisons and swaps

Thanks

Q&A