Project 1

Integration of Merge Sort & Insertion Sort

Presented by:

BRANDON JANG JIN TIAN

KEITH LIM EN KAI

GOH ANNA, NINETTE

TEE QIN TONG BETTINA

Table Of Contents

1 Algorithm Implementation

Hybrid Merge Sort: **Merge Sort + Insertion Sort**.

Merge Sort: (Divide and Conquer)

\+ **Stable** and **Predictable** Sorting Performance

\- **Inefficient** for Small Lists (**$O(n \lg n)$** for Best/Average Case)

Insertion Sort: (Incremental Approach)

\+ **Efficient** for Small Lists (**$O(n)$** for Best Case)

\- **Inefficient** for Large Lists (**$O(n^2)$** for Worst Case)

1 Algorithm Implementation

Threshold **S**: Size of Subarray

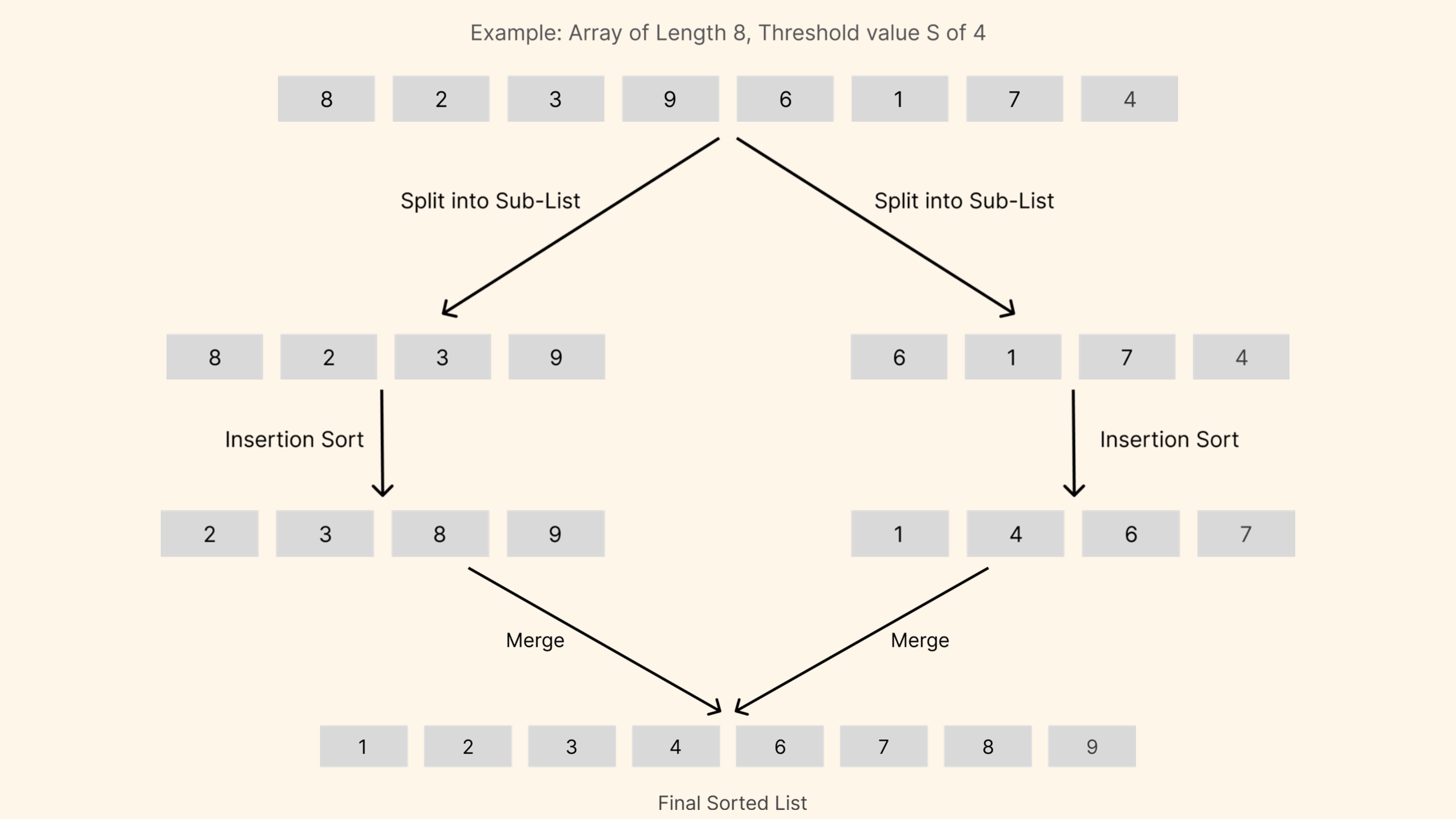
- At which point, algorithm switches from **Merge Sort** to **Insertion Sort**.

Code Snippet: (**Insertion Sort**)

```
def insertion_sort(arr):
    comparisons = 0
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0:
            comparisons += 1  # Increment comparisons for each comparison made
            if key < arr[j]:
                arr[j + 1] = arr[j]
                j -= 1
            else:
                break
        arr[j + 1] = key
    return comparisons
```

Code Snippet: (**Hybrid Merge Sort**)

```
def hybrid_merge_sort(arr, S):
    comparisons = 0
    if len(arr) <= S:
        comparisons += insertion_sort(arr)  # Count comparisons in insertion sort
    else:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        comparisons += hybrid_merge_sort(left_half, S)
        comparisons += hybrid_merge_sort(right_half, S)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1
            comparisons += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

    return comparisons
```


Example: Array of Length 8, Threshold value S of 4
8 2 3 9 6 1 7 4
Split into Sub-List
Split into Sub-List
8 2 3 9
6 1 7 4
Insertion Sort
Insertion Sort
2 3 8 9
1 4 6 7
Merge
Merge
1 2 3 4 6 7 8 9
Final Sorted List

1 Algorithm Implementation

What value of **S**?

- Find out through: **experimentation** and **analysis**.

2 Generating Input data

Code Snippet: (**Generating Input Data**)

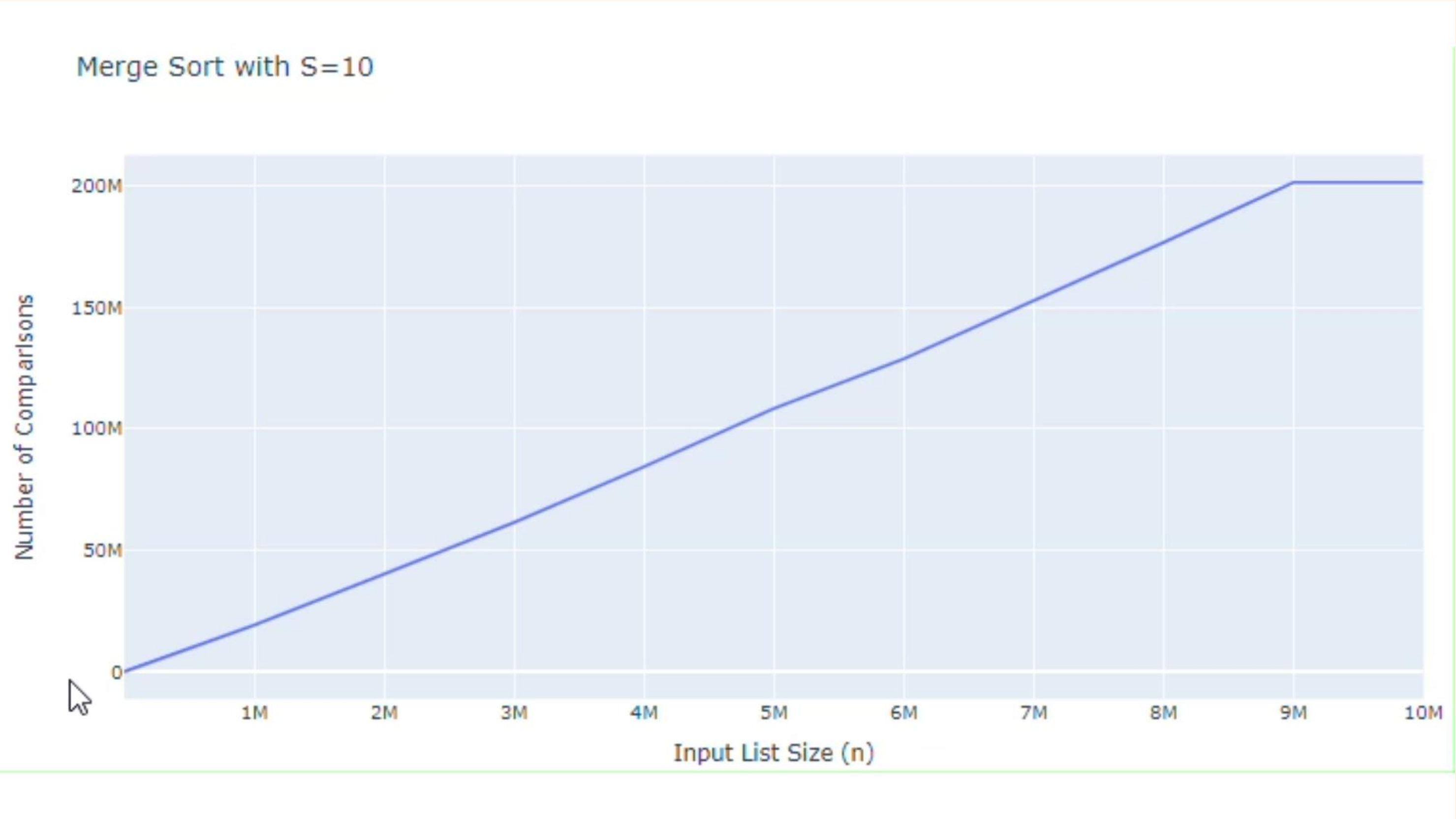
```
import random

datasets = {}
x = 10000000  # largest number you allow for your datasets

# size is the size of the array
# generating a dictionary (datasets) of arrays of increasing sizes, in a range from 1,000 to 10 million
# here we have array size increment of 100,000
# no duplicate elements in each array
for size in range(1000, 10000000, 1000000):
    dataset = random.sample(range(1, x + 1), size)
    datasets[size] = dataset

# including array with size of 10 million
dataset = random.sample(range(1, x + 1), size)
datasets[10000000] = dataset
```

3 Time Complexity Analysis

1. Number of Key Comparisons VS Size of Input List n
   [Fixed S value]
2. Number of Key Comparisons VS S Value
   [Fixed Size of Input List n]
3. Theoretical Analysis of the Hybrid Merge Sort's Time Complexity
4. Comparison between Empirical Results and Theoretical Analysis
5. Determine Optimal S Value

1

Number of Key Comparisons VS Size of Input List n [Fixed S value]

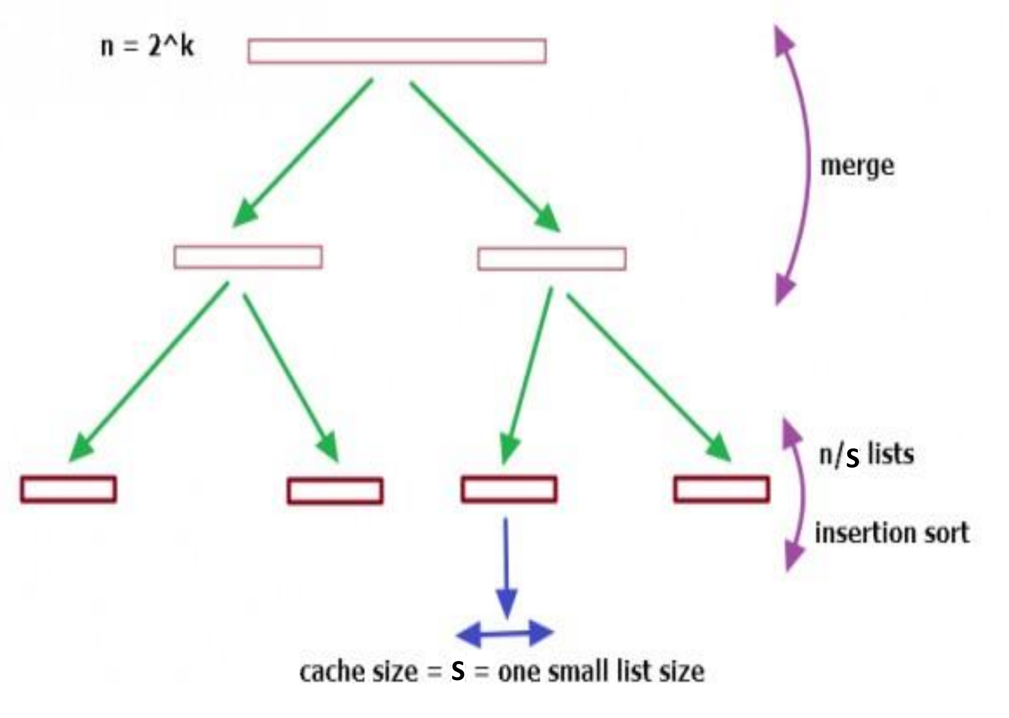


2

Number of Key Comparisons VS S Value [Fixed Size of Input List n]

3 Theoretical Analysis of the Hybrid Merge Sort Time Complexity

Total Complexity = Complexity of Merge Sort + Insertion Sort



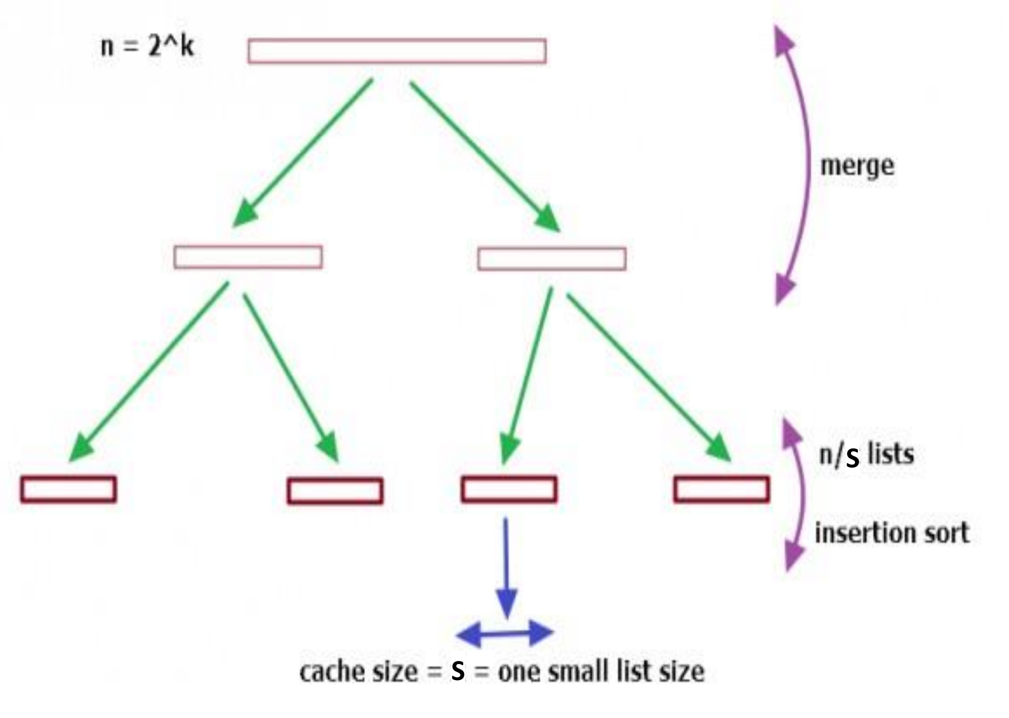
Complexity of Merge Sort

- n levels of splittings from top to bottom - O(n)
- Number of levels = number of sublists - O(log(n/S))
- Complexity - O(nlog(n/S))

3

Theoretical Analysis of the Hybrid Merge Sort Time Complexity

Total Complexity = Complexity of Merge Sort + Insertion Sort



Complexity of Insertion Sort

- Sorting n/S lists each of S elements
- Complexity - O((n/S) INSERTION(S))
  i.e Best Case: O(n)
  i.e Worst Case: $O(n^2)$
  I.e Average Case: $O(n^2)$

3 Theoretical Analysis of the Hybrid Merge Sort Time Complexity

Complexity of Insertion Sort

- Sorting n/S lists each of S elements
- Complexity - O((n/S) INSERTION(S))

  i.e Best Case: $O(n)$
  i.e Worst Case: $O(n^2)$
  I.e Average Case: $O(n^2)$

Complexity of Merge Sort

- n levels of splittings from top to bottom - O(n)
- Number of levels = number of sublists - O(log(n/S))
- Complexity - O(nlog(n/S))

Total Complexity = O((n/S) INSERTION(S) + nlog(n/S))

3

Theoretical Analysis of the Hybrid Merge Sort Time Complexity

Complexity of Insertion Sort

- Sorting n/S lists each of S elements
- Complexity - O((n/S) INSERTION(S))

  i.e Best Case: $O(n)$
  i.e Worst Case: $O(n^2)$
  I.e Average Case: $O(n^2)$

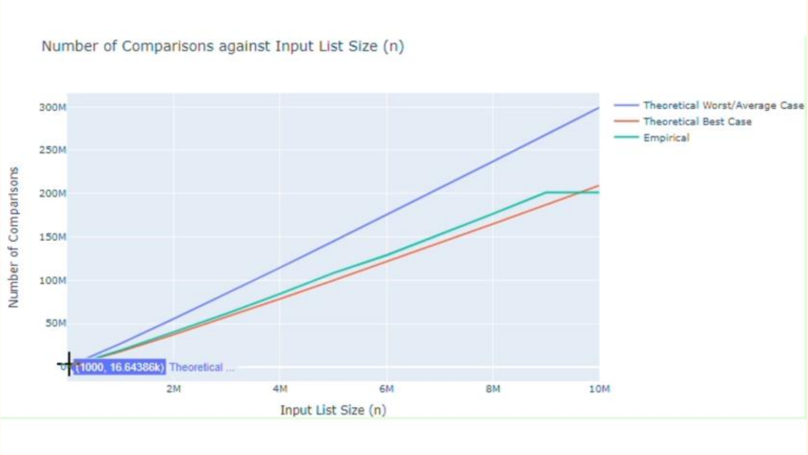
Complexity of Merge Sort

- n levels of splittings from top to bottom - O(n)
- Number of levels = number of sublists - O(log(n/S))
- Complexity - O(nlog(n/S))

Best Case: O(n + nlog(n/S))
Worst Case: O(nS + nlog(n/S))
Average Case: O(nS + nlog(n/S))

4 Comparison between Empirical Results and Theoretical Analysis

No. of Comparisons VS List Size (n)



- Graphs Show Similar Trend
- Empirical Graph (Green) Displays Near Constant Linear Graph

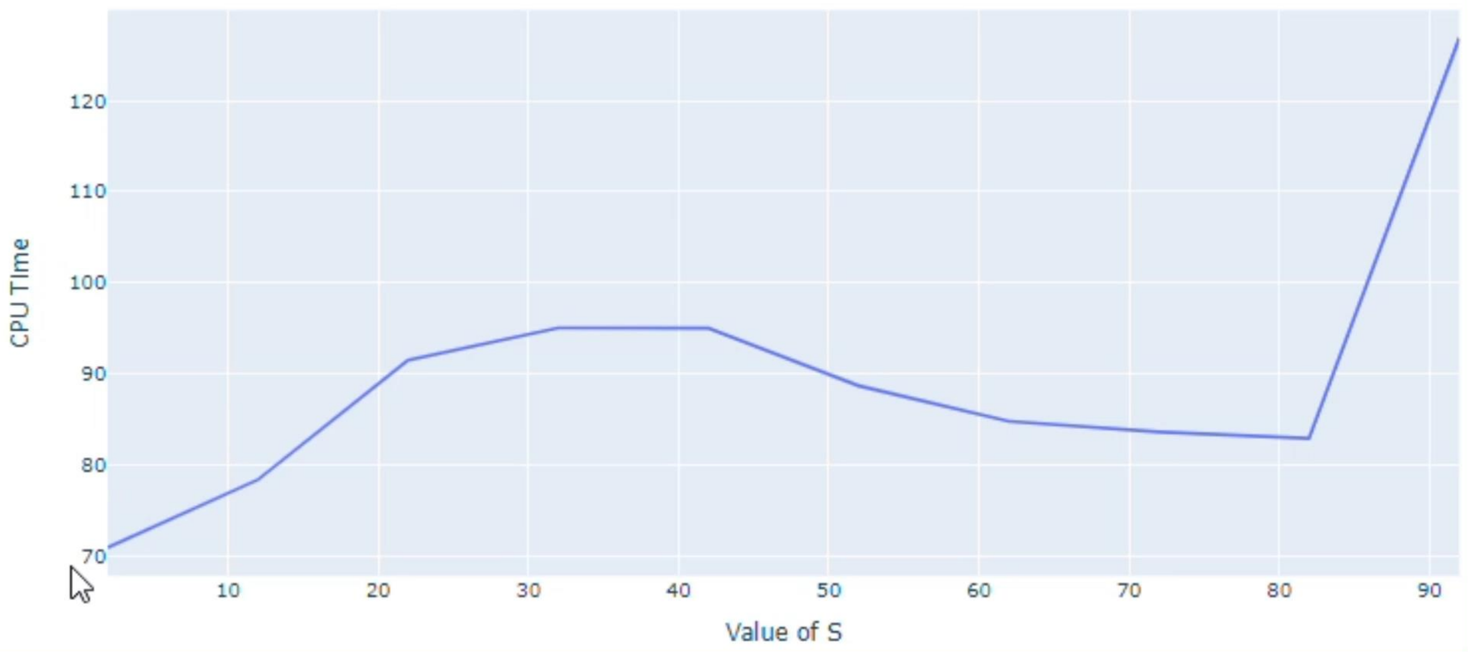
No. of Comparisons VS Value S

- Graphs Show Differing Trend
- Empirical Graph (Green) Displays Overall Increment of Comparisons but periodic sharp increases

5

Determine Optimal S Value

```
for S in range(2, 100, 10):
```



5 Determine Optimal S Value

```
for S in range(2, 23, 1):
```

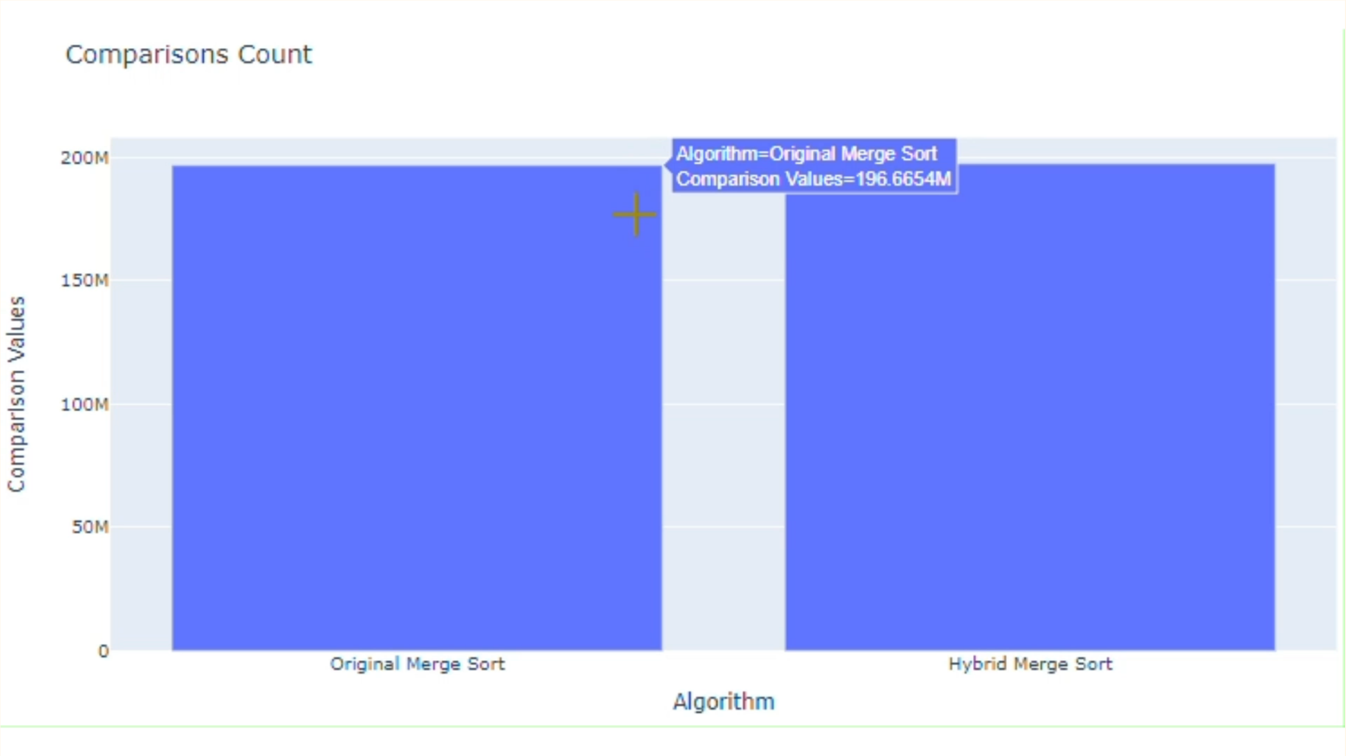
```python
# record start time for cpu_time of original_merge_sort
start_time = time.process_time()

# call original_merge_sort for dataset_10_million
sortedarray, comparisons_count_original = original_merge_sort(dataset_10_million.copy())

# record end time for cpu_time of original_merge_sort
end_time = time.process_time()

# calculate cpu_time of original_merge_sort
cpu_time_orignal = end_time - start_time

print("Number of Key Comparisons is  ", comparisons_count_original)
print("CPU time is ", cpu_time_orignal)
```

```
Number of Key Comparisons is   196665424
CPU time is  85.03125
```

```python
# record start time for cpu_time of hybrid_merge_sort
start_time = time.process_time()

# call hybrid_merge_sort for dataset_10_million
comparisons_count_hybrid = hybrid_merge_sort(dataset_10_million.copy(), optimal_S)

# record end time for cpu_time of hybrid_merge_sort
end_time = time.process_time()

# calculate cpu_time of hybrid_merge_sort
cpu_time_hybrid = end_time - start_time

print("Number of Key Comparisons is  ", comparisons_count_hybrid)
print("CPU time is ", cpu_time_hybrid)
```

```
Number of Key Comparisons is   197371932
CPU time is  67.828125
```

4 Compare with Original Mergesort



4 Compare with Original Mergesort