HybridSort Algorithm Analysis

<u>Group 3:</u>
Karthik Raj U2320917J
Ishita Jain U2321668B
Jayakumar Thirunithiyan U2321424H

Overview

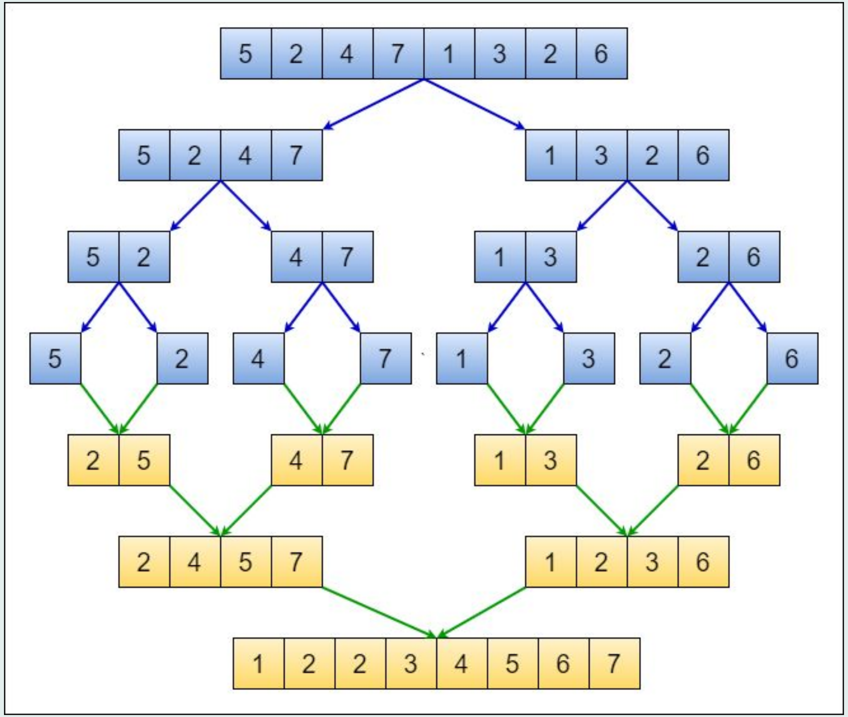
- Implementation of Hybrid Algorithm
- Generate Input Data
- Time Complexity Analysis
- Comparison of Algorithms

MERGE SORT

- Merge Sort is an example of a divide and conquer strategy.

- It divides the array continuously into halves until there are n arrays of size 1 and then merges them in a 'head vs head' manner.

- Height of the MergeSort tree is log(n) i.e array is split log(n) times.



```python
def merge_sort(arr, left, right, S):
    """Performs a hybrid merge sort with insertion sort on small subarrays."""
    if right - left + 1 <= S:
        insertion_sort(arr, left, right)
    else
        if left < right:
            middle = (left + right) // 2
            merge_sort(arr, left, middle, S)
            merge_sort(arr, middle + 1, right, S)
            merge(arr, left, middle, right)
```

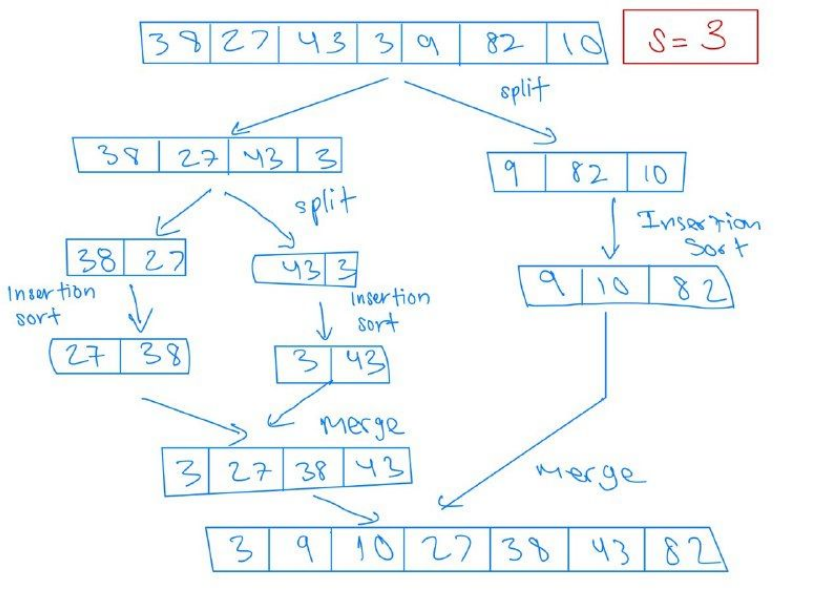
- ***The merge and Merge sort function implementing the algorithm described earlier.***

```python
def merge(arr, left, middle, right):
    """Merges two sorted subarrays into a single sorted subarray."""
    n1 = middle - left + 1
    n2 = right - middle

    # Create temporary arrays
    L = arr[left:middle + 1]
    R = arr[middle + 1:right + 1]

    # Merge the temporary arrays back into arr[left..right]
    i = j = 0
    k = left

    while i < n1 and j < n2:
        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
```

Implementation of the hybrid algorithm

HybridSort Algorithm

- This is also a divide and conquer based algorithm.
- Implements MergeSort until the size of the subarrays is less than the threshold value S.
- Once the threshold value is reached, it uses insertion sort to sort the sub arrays.
- It then merges these sorted subarrays back into the final sorted array.



- ***The array is divided into sub arrays log(n/S) times.***

Implementation of the hybrid algorithm

```
# Hybrid Merge Sort (Merge Sort + Insertion Sort)
def hybrid_sort(arr, left, right, S):
    if right - left + 1 <= S:
        insertion_sort(arr[left:right + 1])
    else:
        if left < right:
            mid = (left + right) // 2
            hybrid_sort(arr, left, mid, S)
            hybrid_sort(arr, mid + 1, right, S)
            merge(arr, left, mid, right)
```

This is the hybrid sort function implementing the algorithm described previously.

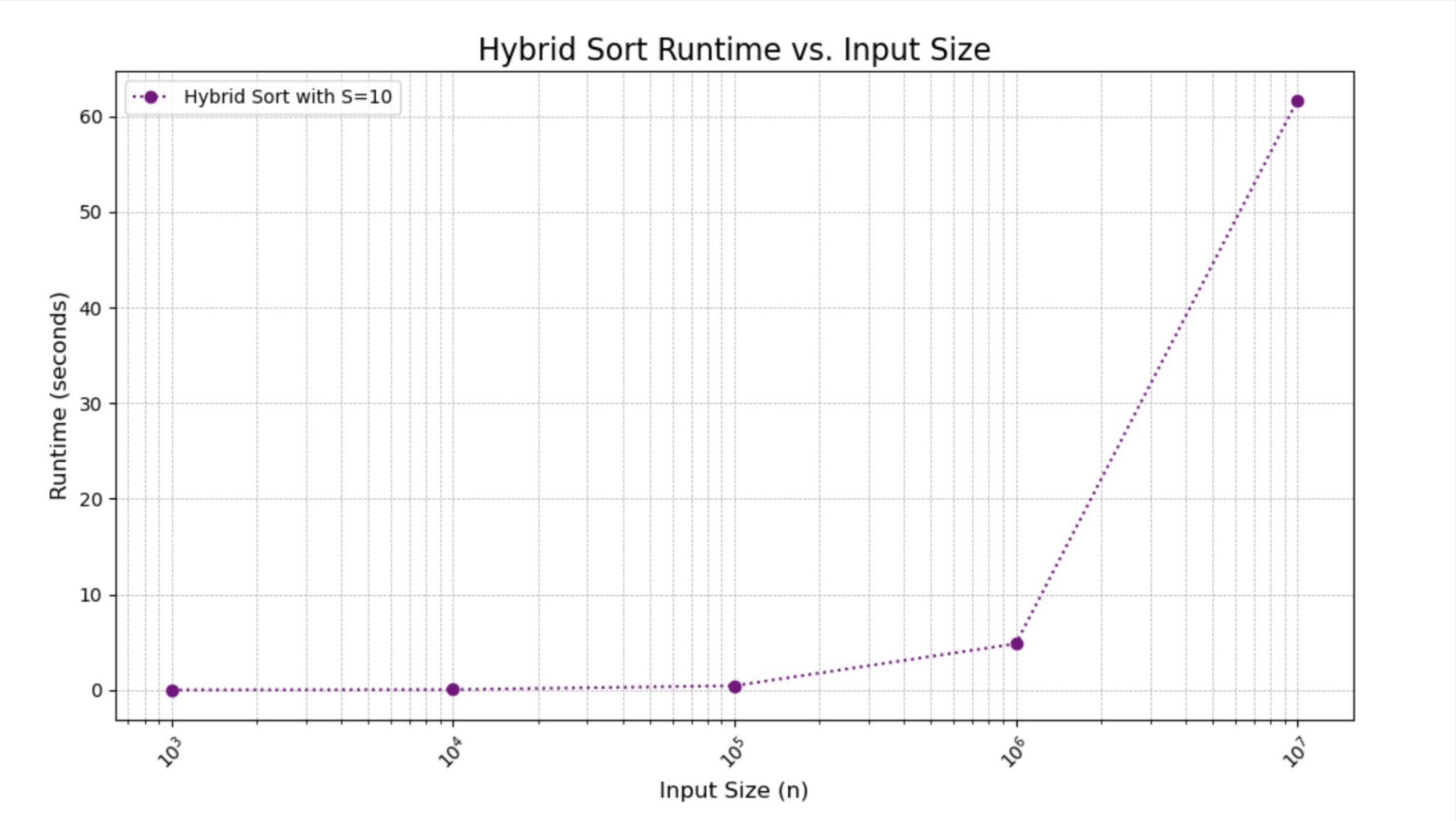
It also uses the merge function previously shown.

```
def insertion_sort(arr, left, right):
    """Performs insertion sort on the subarray arr[left:right+1]."""
    for i in range(left + 1, right + 1):
        key = arr[i]
        j = i - 1
        while j >= left and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Implementation of the hybrid algorithm

```
Sorting completed in 0.004971981 seconds for input 1000

Sorting completed in 0.036343575 seconds for input 10000

Sorting completed in 0.441458464 seconds for input 100000

Sorting completed in 4.831027985 seconds for input 1000000

Sorting completed in 61.609353781 seconds for input 10000000
```

Implementation of the hybrid algorithm



Generate Input Data

Generate Input Data

```python
import random

# Generating 5 different arrays from size 1,000 to 10 million
array_1k = [random.randint(1, 1000000) for _ in range(1000)]
array_10k = [random.randint(1, 1000000) for _ in range(10000)]
array_100k = [random.randint(1, 1000000) for _ in range(100000)]
array_1mil = [random.randint(1, 1000000) for _ in range(1000000)]
array_10mil = [random.randint(1, 10000000) for _ in range(10000000)]

# Storing the arrays in variables for future use in Part C
arrays = {
    "1k": array_1k,
    "10k": array_10k,
    "100k": array_100k,
    "1mil": array_1mil,
    "10mil": array_10mil
}

# Printing only the first 10 elements of each array
print("First 10 elements of array_1k:", array_1k[:10])
print("First 10 elements of array_10k:", array_10k[:10])
print("First 10 elements of array_100k:", array_100k[:10])
print("First 10 elements of array_1mil:", array_1mil[:10])
print("First 10 elements of array_10mil:", array_10mil[:10])
```

Generate Input Data

```
First 10 elements of array_1k: [202179, 652855, 781727, 278717, 853710, 704668, 231470, 550610, 865265, 563278]
First 10 elements of array_10k: [543792, 973116, 786118, 43283, 975621, 622245, 953486, 838332, 780839, 257422]
First 10 elements of array_100k: [748806, 610455, 590206, 544699, 987920, 80355, 576134, 265708, 340396, 439175]
First 10 elements of array_1mil: [811376, 6820, 729318, 851313, 304379, 791334, 622930, 256747, 74248, 930387]
First 10 elements of array_10mil: [5268965, 6011406, 2186839, 1854898, 3330216, 7466911, 6822817, 1832412, 9073715, 4331356]
```

Time Complexity Analysis

Time Complexity Analysis
Part c i

```
S = 10  # Threshold for switching to insertion sort
# Plotting the results
sizes = [1000, 10000, 100000, 1000000,10000000] #
```

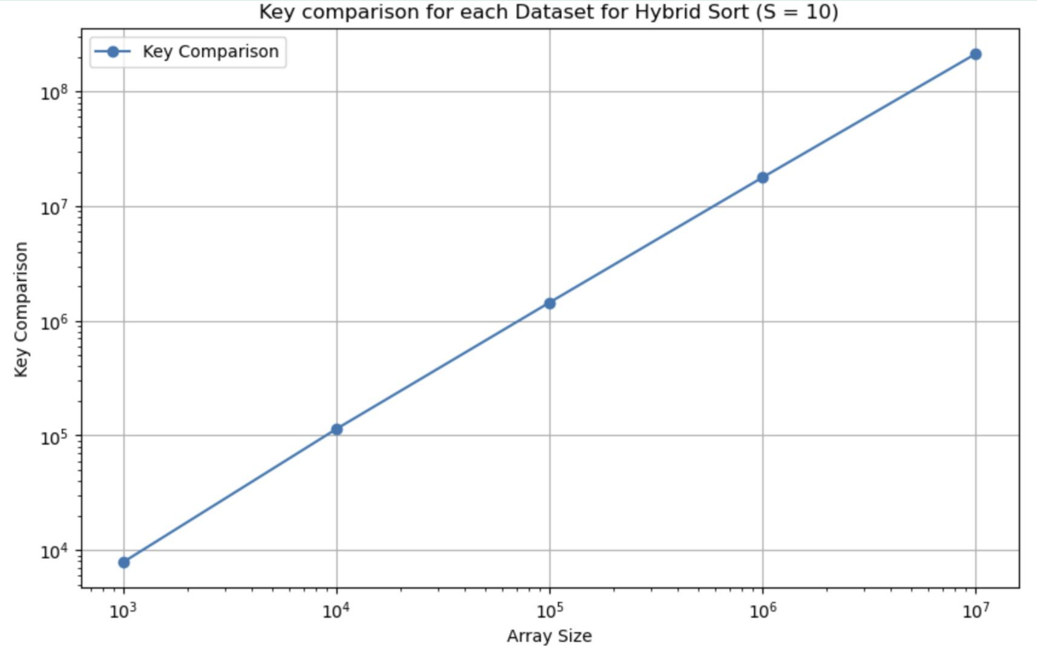
Time Complexity Analysis
Part c i

```python
# Hybrid sort wrapper function
def hybrid_sort_main(arr, S):
    global key_comparisons
    key_comparisons = 0  # Reset comparisons count
    start_t = time.time()
    hybrid_sort(arr, 0, len(arr) - 1, S)
    end_t = time.time()
    runtime = end_t - start_t
    return key_comparisons,runtime
```

Runtime and the key comparisons are calculated at the same time and being returned

Time Complexity Analysis
Part c i

	Array Size	Key Comparisons	Run Time(Seconds)
0	1k	7931	0.001000
1	10k	114198	0.019004
2	100k	1446494	0.250254
3	1mil	17857236	3.145501
4	10mil	213518420	39.456156



Time Complexity Analysis
Part C i

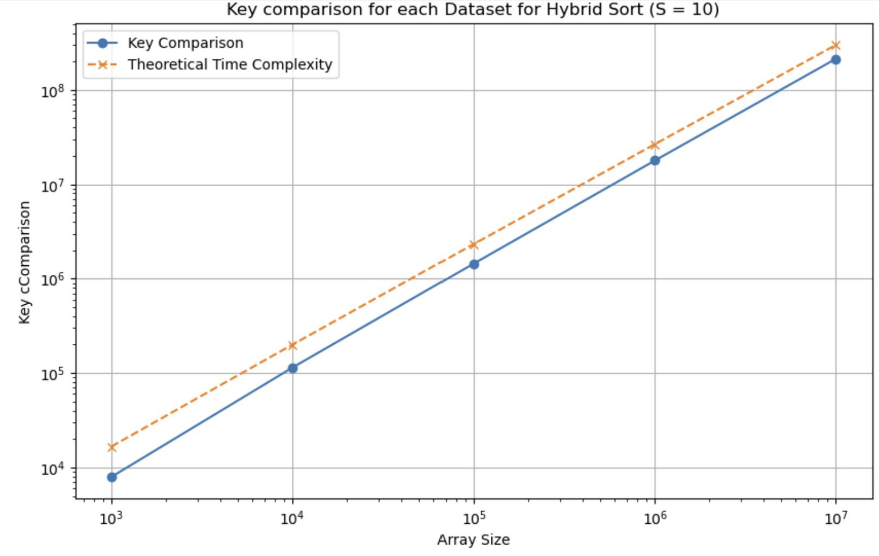
Time complexity(Total) = Time complexity(Insertion Sort) + Time complexity(MergeSort)

```python
# Theoretical time complexity function
def theoretical_complexity_best(n, S):
    return n * math.log2(n/S) + n

# Theoretical time complexity function
def theoretical_complexity_worst(n, S):
    return (n * math.log2(n / S)) + (S * n)


# Part C: Run hybrid sorting on different arrays and collect key comparisons and runtime
S = 10  # Threshold for switching to insertion sort
comparisons_results = {}
runtime_results = {}
theoretical_results_best = {}
theoretical_results_worst = {}
results1 = []
results2 = []
sizes = [1000, 10000, 100000, 1000000, 10000000]
# Use the arrays generated in Part B
for size, array in arrays.items():
    array_copy = array.copy()  # Make sure to sort a copy to avoid sorting the original array
    comparisons, run_time = hybrid_sort_main(array_copy, S)
    comparisons_results[size] = comparisons
    runtime_results[size] = run_time
    theoretical_results_worst[size] = theoretical_complexity_worst(len(array_copy), S)
    theoretical_results_best[size] = theoretical_complexity_best(len(array_copy), S)
    t_best = theoretical_complexity_best(len(array_copy), S)
    t_worst = theoretical_complexity_worst(len(array_copy), S)
    #data_key_comparison = {"Array Size" : size,"Key Comparisons" : comparisons,"Run Time(Seconds)": run_time}
    results1.append({"Array Size" : size,"Key Comparisons" : comparisons,"Run Time(Seconds)": run_time})
    results2.append({"Array Size" : size,"Key Comparisons" : comparisons,"Run Time(Seconds)": run_time,"Theoretical Comparison": t_worst})
```

Best Case	O(n+n*log(n/S))
Worst/Average Case	O(n*S+n*log(n/S))

Time Complexity Analysis
Part c i

	Array Size	Key Comparisons	Run Time(Seconds)	Theoretical Comparison
0	1k	7931	0.001000	1.664386e+04
1	10k	114198	0.019004	1.996578e+05
2	100k	1446494	0.250254	2.328771e+06
3	1mil	17857236	3.145501	2.660964e+07
4	10mil	213518420	39.456156	2.993157e+08



Time Complexity Analysis
Part C ii

Fixed Dataset to '1k'

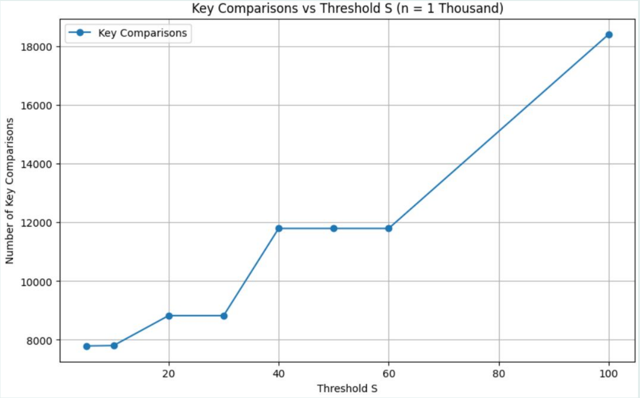
Plotted Key comparison for hybrid sort against S values [5,10,20,30,40,50,60,100]

Plotting Alongside Theoretical Value

```python
# Fix the input size to 1 million elements
fixed_array_size = arrays['1k']

# List of S values to test
S_values = [5, 10, 20, 30, 40, 50, 60, 100]

# Dictionary to store comparisons and runtimes for different values of S
comparisons_for_S = {}
runtime_for_S = {}
theoretical_for_S = {}

# Theoretical time complexity function
def theoretical_complexity(n, S):
    return (n * math.log2(n / S)) + (S * n)

# Run hybrid sorting for different values of S and record key comparisons and runtime
for S in S_values:
    array_copy = fixed_array_size.copy()  # Ensure we sort a copy of the array
    comparisons, run_time = hybrid_sort_main(array_copy, S)  # Sort and track time
    comparisons_for_S[S] = comparisons
    runtime_for_S[S] = run_time
    theoretical_for_S[S] = theoretical_complexity(len(fixed_array_size), S)
    print(f"S = {S}, key comparisons: {comparisons}, run time: {run_time:.4f} seconds")
```

Time Complexity Analysis
Part C ii

```python
# Plotting key comparisons vs S
plt.figure(figsize=(10, 6))
plt.plot(S_values, [comparisons_for_S[S] for S in S_values], marker='o', label="Key Comparisons")
plt.xlabel('Threshold S')
plt.ylabel('Number of Key Comparisons')
plt.title('Key Comparisons vs Threshold S (n = 1 Thousand)')
plt.grid(True)
plt.legend()
plt.show()
```



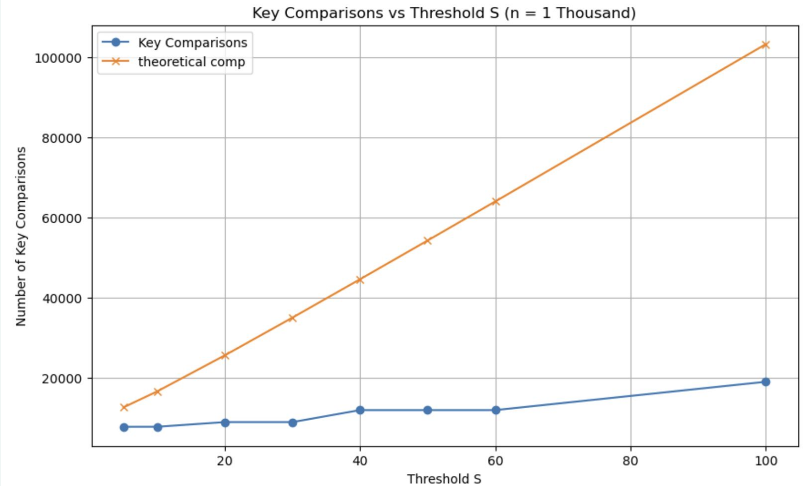
<u>S Value -> Key Comparison</u>

5 -> 7792
10 -> 7805
20 -> 8825
30 -> 8825
40 -> 11796
50 -> 11796
60 -> 11796
100 -> 18415

Time Complexity Analysis
Part C ii

```python
# Plotting key comparisons vs S along with theoretical complexity
plt.figure(figsize=(10, 6))
plt.plot(S_values, [comparisons_for_S[S] for S in S_values], marker='o', label="Key Comparisons")
plt.plot(S_values, [theoretical_for_S[S] for S in S_values], marker='x', linestyle='--', label="Theoretical Key Comparisons")
plt.xlabel('Threshold S')
plt.ylabel('Number of Key Comparisons')
plt.title('Key Comparisons vs Threshold S (n = 1 Thousand)')
plt.grid(True)
plt.legend()
plt.show()
```

```
Theoretical key comparisons:
{5: 12643.856189774724,
 10: 16643.856189774724,
 20: 25643.856189774724,
 30: 35058.89368905357,
 40: 44643.85618977473,
 50: 54321.928094887364,
 60: 64058.89368905357,
 100: 103321.92809488736}
```



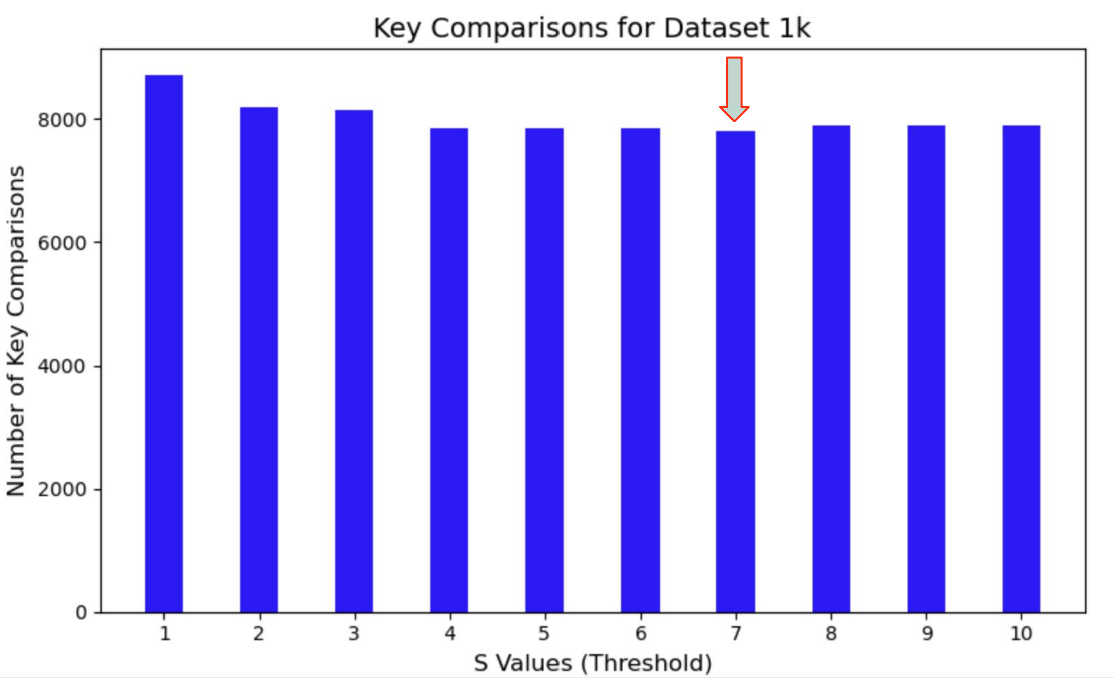
Time Complexity Analysis
Part C iii

S value 5 and 10
Produced smallest key comparisons in Part 2

For a detailed Analysis
We printed key comparisons of S value 1-10 for 1k - 1Mil dataset to find the optimal S value

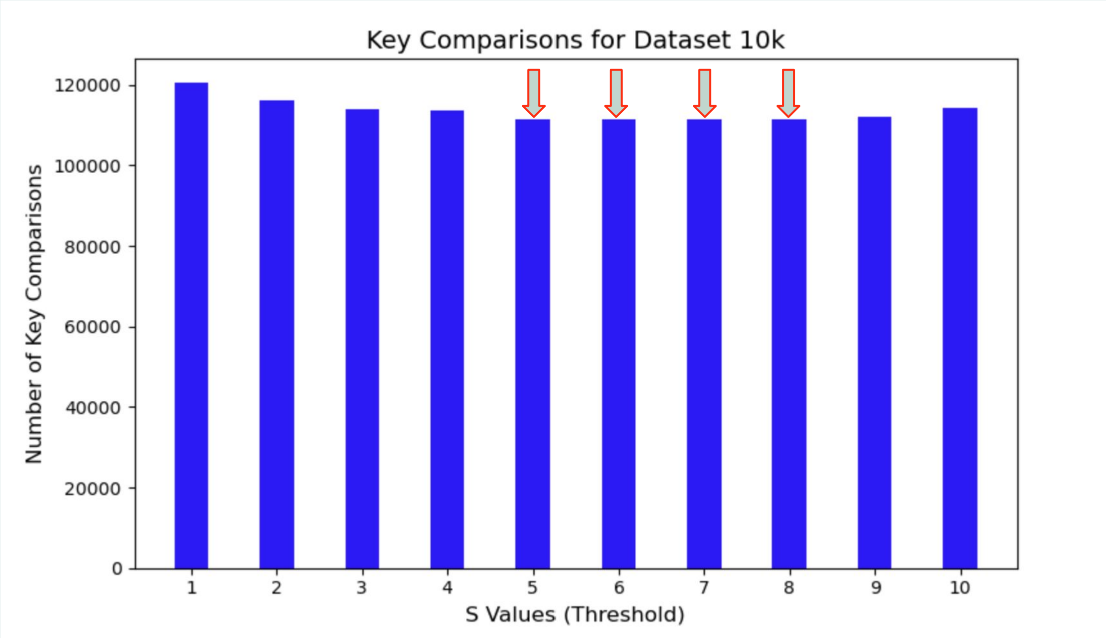
```
# List of S values to test (1 to 10)
S_values = list(range(1, 11))

# Dataset sizes we want to analyze (1k, 10k, 100k)
dataset_sizes = ['1k', '10k', '100k','1mil']

# Dictionary to store the results (comparisons for each dataset size and S value)
comparisons_by_size_and_S = {size: [] for size in dataset_sizes}

# Run hybrid sorting for each dataset size and each value of S
for size in dataset_sizes:
    for S in S_values:
        array_copy = arrays[size].copy()  # Make a copy of the array for sorting
        comparisons = hybrid_sort_main(array_copy, S)
        comparisons_by_size_and_S[size].append(comparisons)
        print(f"Dataset size {size}, S = {S}, key comparisons: {comparisons}")

# Find all S values that result in the minimum number of key comparisons for each dataset size
optimal_S_by_size = {}

for size in dataset_sizes:
    min_comparisons = min(comparisons_by_size_and_S[size])
    optimal_S_list = [S_values[i] for i, comparisons in enumerate(comparisons_by_size_and_S[size]) if comparisons == min_comparisons]
    optimal_S_by_size[size] = optimal_S_list
    print(f"Optimal S values for dataset size {size}: {optimal_S_list}, with {min_comparisons} key comparisons")
```

Time Complexity Analysis
Part C iii



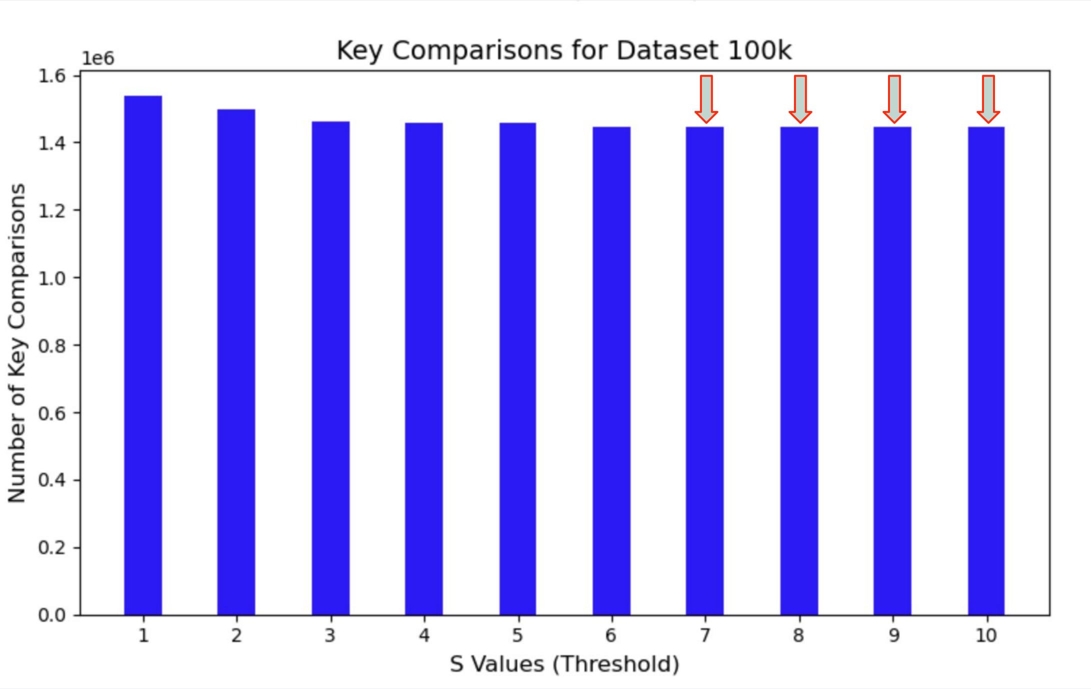

```
Dataset size 1k, S = 1, key comparisons: 8701
Dataset size 1k, S = 2, key comparisons: 8187
Dataset size 1k, S = 3, key comparisons: 8146
Dataset size 1k, S = 4, key comparisons: 7842
Dataset size 1k, S = 5, key comparisons: 7842
Dataset size 1k, S = 6, key comparisons: 7842
Dataset size 1k, S = 7, key comparisons: 7812
Dataset size 1k, S = 8, key comparisons: 7897
Dataset size 1k, S = 9, key comparisons: 7897
Dataset size 1k, S = 10, key comparisons: 7897
```

Time Complexity Analysis
Part C iii



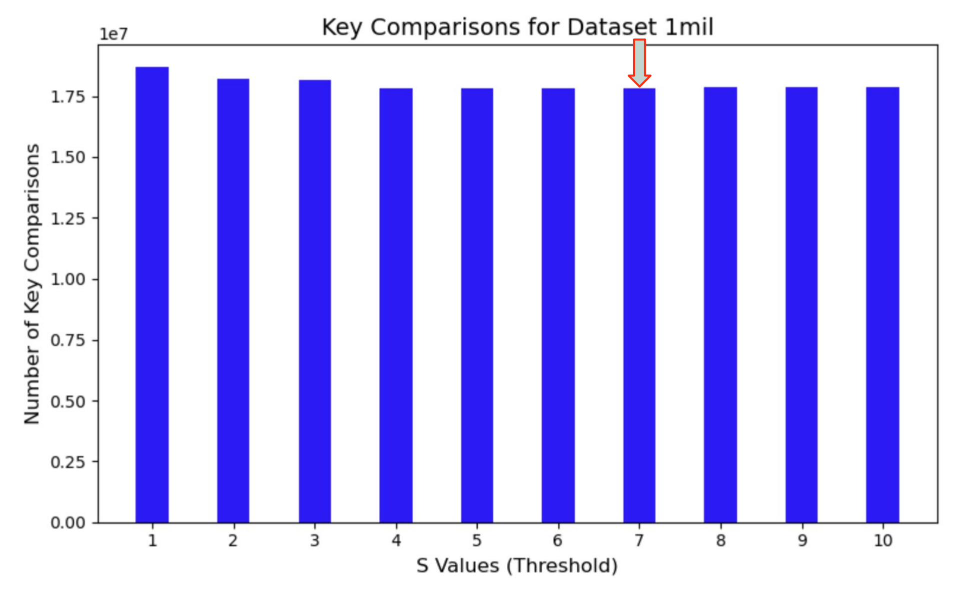

```
Dataset size 10k, S = 1, key comparisons: 120402
Dataset size 10k, S = 2, key comparisons: 116279
Dataset size 10k, S = 3, key comparisons: 114012
Dataset size 10k, S = 4, key comparisons: 113660
Dataset size 10k, S = 5, key comparisons: 111554
Dataset size 10k, S = 6, key comparisons: 111554
Dataset size 10k, S = 7, key comparisons: 111554
Dataset size 10k, S = 8, key comparisons: 111554
Dataset size 10k, S = 9, key comparisons: 111931
Dataset size 10k, S = 10, key comparisons: 114312
```

Time Complexity Analysis
Part C iii




```
Dataset size 100k, S = 1, key comparisons: 1536272
Dataset size 100k, S = 2, key comparisons: 1499609
Dataset size 100k, S = 3, key comparisons: 1462182
Dataset size 100k, S = 4, key comparisons: 1459414
Dataset size 100k, S = 5, key comparisons: 1459414
Dataset size 100k, S = 6, key comparisons: 1446139
Dataset size 100k, S = 7, key comparisons: 1445418
Dataset size 100k, S = 8, key comparisons: 1445418
Dataset size 100k, S = 9, key comparisons: 1445418
Dataset size 100k, S = 10, key comparisons: 1445418
```

Time Complexity Analysis
Part C iii




```
Dataset size 1mil, S = 1, key comparisons: 18674596
Dataset size 1mil, S = 2, key comparisons: 18193855
Dataset size 1mil, S = 3, key comparisons: 18139111
Dataset size 1mil, S = 4, key comparisons: 17827646
Dataset size 1mil, S = 5, key comparisons: 17827646
Dataset size 1mil, S = 6, key comparisons: 17827646
Dataset size 1mil, S = 7, key comparisons: 17818810
Dataset size 1mil, S = 8, key comparisons: 17861151
Dataset size 1mil, S = 9, key comparisons: 17861151
Dataset size 1mil, S = 10, key comparisons: 17861151
```

Time Complexity Analysis
Part C iii

Dataset Size	S Value Corresponding to Smallest Key Comparison
1k	7
10K	5,6,7,8
100K	7,8,9,10
1Mil	7

7 consistently produced smallest key comparison across multiple dataset sizes

Time Complexity Analysis
Part D

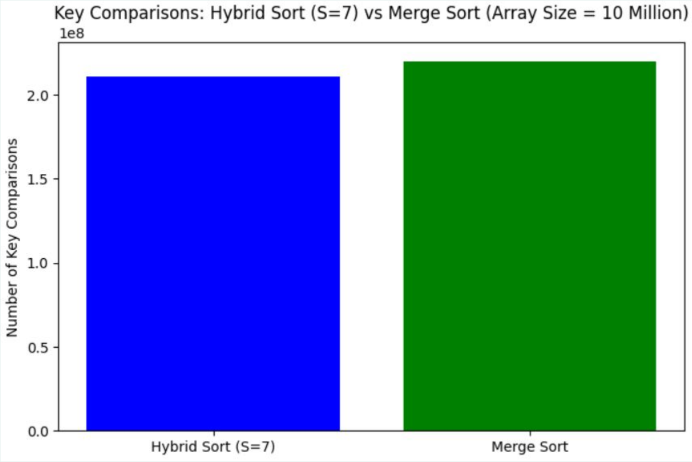
```
# Function to perform normal merge sort and track key comparisons
def merge_sort(arr, left, right):
    global key_comparisons
    if left < right:
        mid = (left + right) // 2
        merge_sort(arr, left, mid)
        merge_sort(arr, mid + 1, right)
        merge(arr, left, mid, right)

# Function to reset the global comparison counter and call the sort function
def sort_and_measure(array, sorting_function, S=None):
    global key_comparisons
    key_comparisons = 0  # Reset comparisons count
    arr_copy = array.copy()  # Work with a copy to keep the original data intact
    if S is not None:
        sorting_function(arr_copy, 0, len(arr_copy) - 1, S)  # For hybrid sort calling earlier function
    else:
        sorting_function(arr_copy, 0, len(arr_copy) - 1)  # For merge sort
    return key_comparisons

# Fetch the 10 million elements array
array_10mil = arrays['10mil']

# Measure key comparisons for hybrid sort with S=7
key_comparisons_hybrid = sort_and_measure(array_10mil, hybrid_sort, S=7)

# Measure key comparisons for normal merge sort
key_comparisons_merge = sort_and_measure(array_10mil, merge_sort)
```

Made a function for normal merge sort

Run to Compare Performance with hybrid sort of S value 7 (Optimal)

Time Complexity Analysis
Part D

```
# Plotting the key comparisons for both algorithms
algorithms = ['Hybrid Sort (S=7)', 'Merge Sort']
key_comparisons = [key_comparisons_hybrid, key_comparisons_merge]

plt.figure(figsize=(8, 5))
plt.bar(algorithms, key_comparisons, color=['blue', 'green'])
plt.ylabel('Number of Key Comparisons')
plt.title('Key Comparisons: Hybrid Sort (S=7) vs Merge Sort (Array Size = 10 Million)')
plt.show()
```



From Key comparison analysis for Merge Sort VS Hybrid Sort we can tell hybrid Sort is more efficient

Thank You