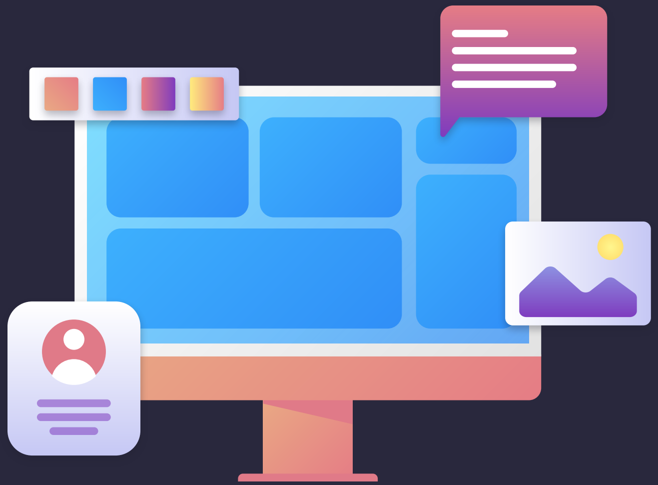
YOUR LOGO HERE

Project 1: Merge Sort + Insertion Sort

<ins>SDDB Group 3</ins>

Chiang Qin Zhi
Donovan Goh Jun Heng
Gao Wen Jie



INDEX.HTML

YOUR LOGO HERE

/TABLE OF CONTENTS

/01 Implementation & Input Data

Part (a) & (b)

```python
def insertion_sort(arr, left, right):
    comparisons = 0
    for i in range(left + 1, right + 1):
        key = arr[i]
        j = i - 1
        while j >= left:
            comparisons += 1
            if key < arr[j]:
                arr[j + 1] = arr[j]
                j -= 1
            else:
                break
        arr[j + 1] = key
    return comparisons
```

Insertion Sort

- Array is split into a sorted (left) & an unsorted (right) part.
- Values from the unsorted part are picked and inserted into its correct position in the sorted part

```
def insertion_sort(arr, left, right):
    comparisons = 0
    for i in range(left + 1, right + 1):
        key = arr[i]
        j = i - 1
        while j >= left:
            comparisons += 1
            if key < arr[j]:
                arr[j + 1] = arr[j]
                j -= 1
            else:
                break
        arr[j + 1] = key
    return comparisons
```

Insertion Sort

Time Complexity

- Worst Case: $O(n^2)$
- Best Case: $\Omega(n)$
- Average Case: $\Theta(n^2)$

Merge Sort

- Recursively split the unsorted array into smaller subarrays
- Sort and merge subarrays back together to form sorted array

```
def merge(arr, left, middle, right):
    if left > right:
        return 0

    comparisons = 0
    n1 = middle - left + 1
    L = arr[left:middle+1]
    n2 = right - middle
    R = arr[middle+1:right+1]
    i, j, k = 0, 0, left
    while i < n1 and j < n2:
        comparisons += 1
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    arr[k:right+1] = L[i:n1] + R[j:n2]
    return comparisons
```

Merge Sort

Time Complexity

- Worst Case: O(n•log(n))
- Best Case: Ω(n•log(n))
- Average Case: Θ(n•log(n))

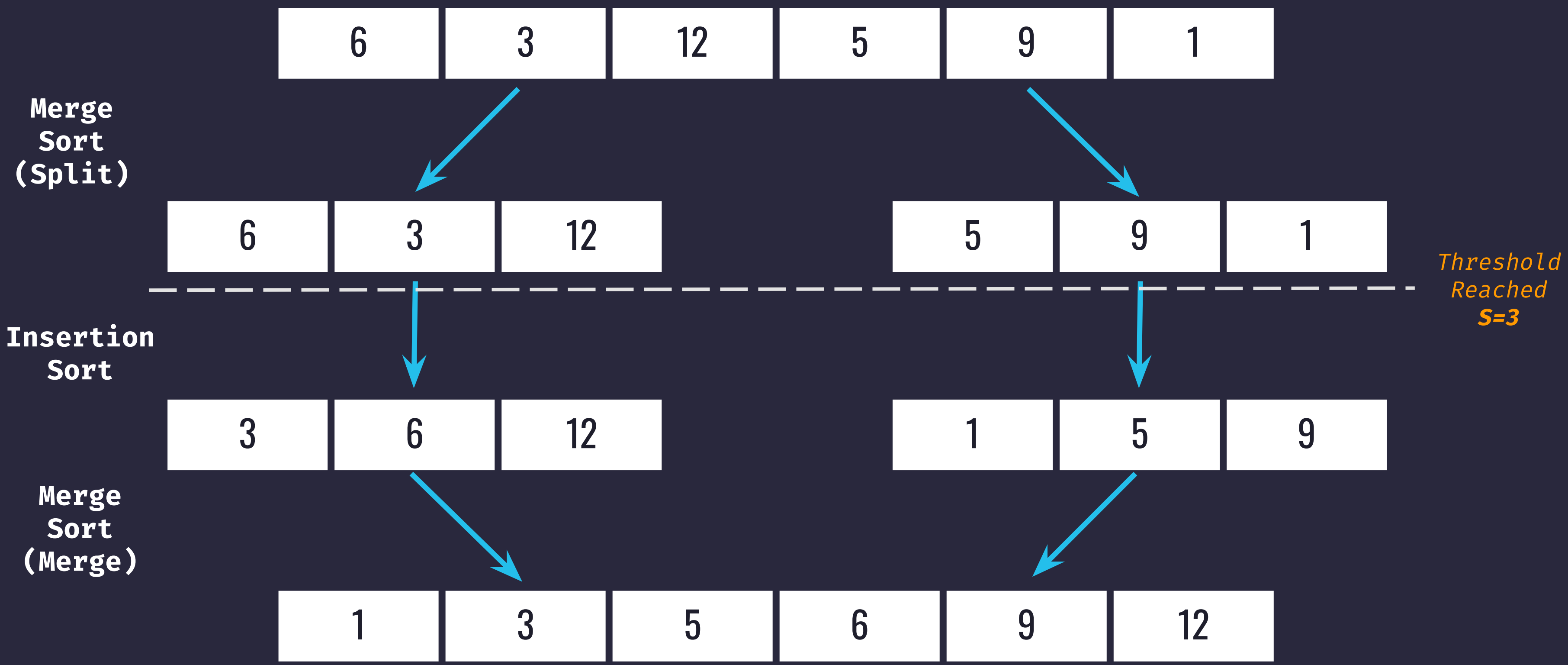
```
def merge(arr, left, middle, right):
    if left > right:
        return 0

    comparisons = 0
    n1 = middle - left + 1
    L = arr[left:middle+1]
    n2 = right - middle
    R = arr[middle+1:right+1]
    i, j, k = 0, 0, left
    while i < n1 and j < n2:
        comparisons += 1
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    arr[k:right+1] = L[i:n1] + R[j:n2]
    return comparisons
```

Hybrid Sort = Merge + Insertion

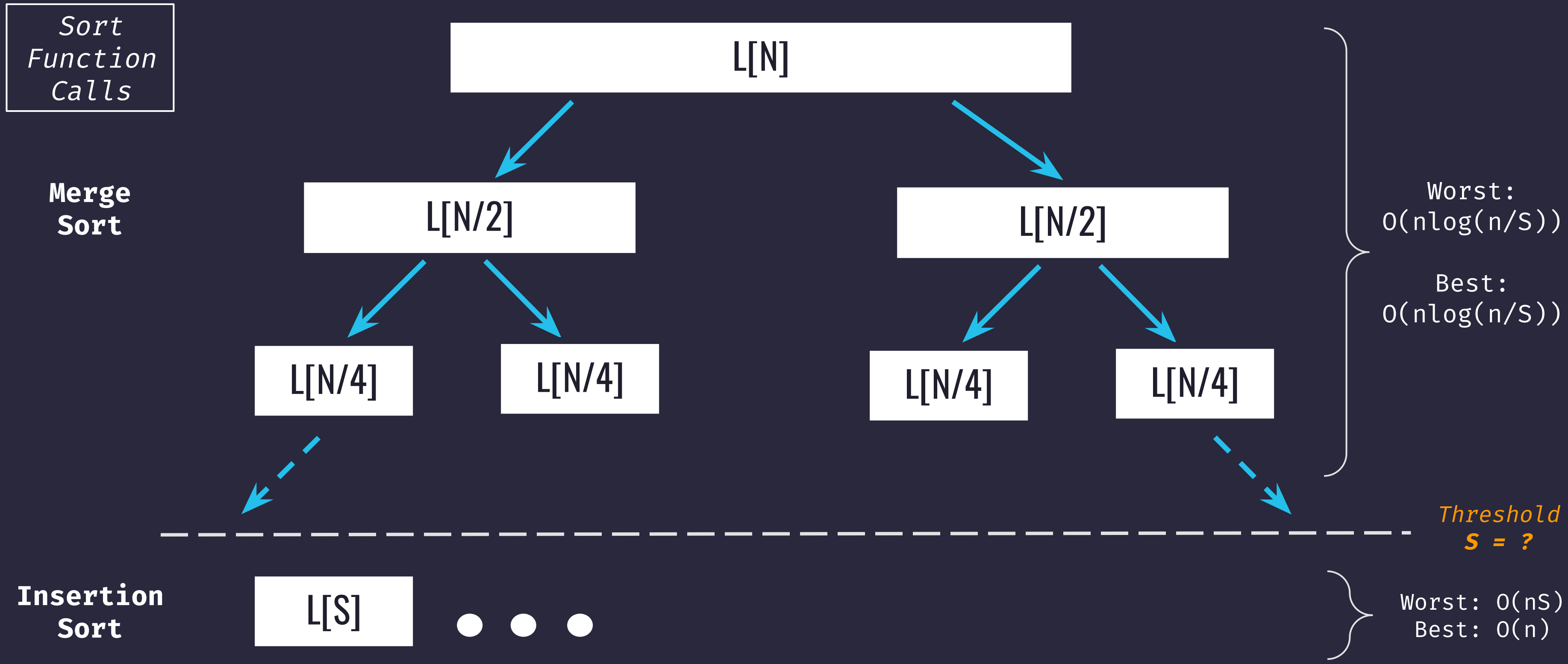
```
def hybrid_sort(arr, left, right, S):
    comparisons = 0
    if left < right:
        if (right-left+1) <= S:
            comparisons += insertion_sort(arr,
left, right)
        else:
            middle = (left+right)//2
            comparisons += hybrid_sort(arr, left,
middle, S)
            comparisons += hybrid_sort(arr,
middle+1, right, S)
            comparisons += merge(arr, left, middle,
right)
    else:
        return 0
    return comparisons
```

- Recursively split the unsorted array into smaller subarrays
- Sort smaller subarrays using Insertion Sort once size ⩽ S (threshold)
- Sort and merge subarrays back together to form sorted array

Hybrid Sort (S = 3)



Time Complexity: Hybrid Sort



Time Complexity: Hybrid Sort (Theory)

Best Case: Best of Merge & Insertion

$\Omega(n + n\log(n/S))$

Worst Case: Worst of Merge & Insertion

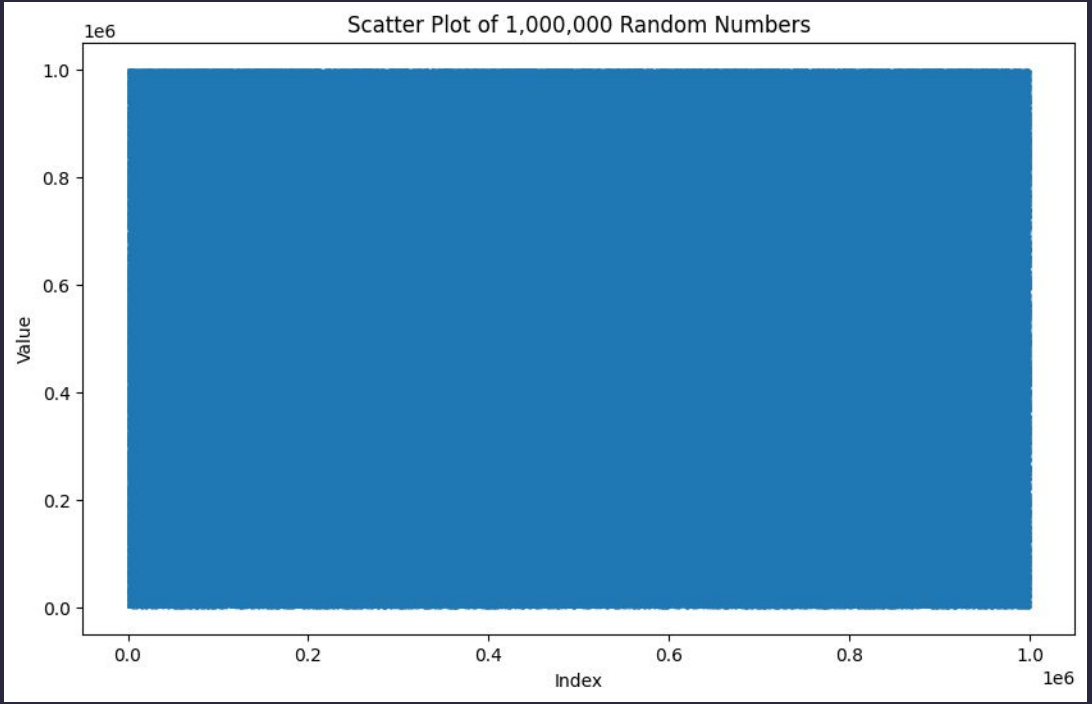
$O(nS + n\log(n/S))$

Input Data Generation

```python
import random

random.seed(10)

def generate_random_array(size, max_val):
   return [random.randint(1, max_val) for _ in range(size)]

# Input Data
sizes = [1000, 10000, 100000, 1000000, 10000000]
datasets = {size: generate_random_array(size, size) for
size in sizes}
```

- Using random module
- Set seed to 10 for reproducible results while testing
- Generates random arrays of sizes = 1k, 10k,100k, 1M & 10M

Input Data Generation



- In the data of 1 million long array, points are uniform and random.

/02

Time Complexity Analysis

Part (c)i. & (c)ii.

Insertion Sort (S = 3)

6	3	12

Best case: O(N)

Each key only compared once against previous key (which they are greater than)

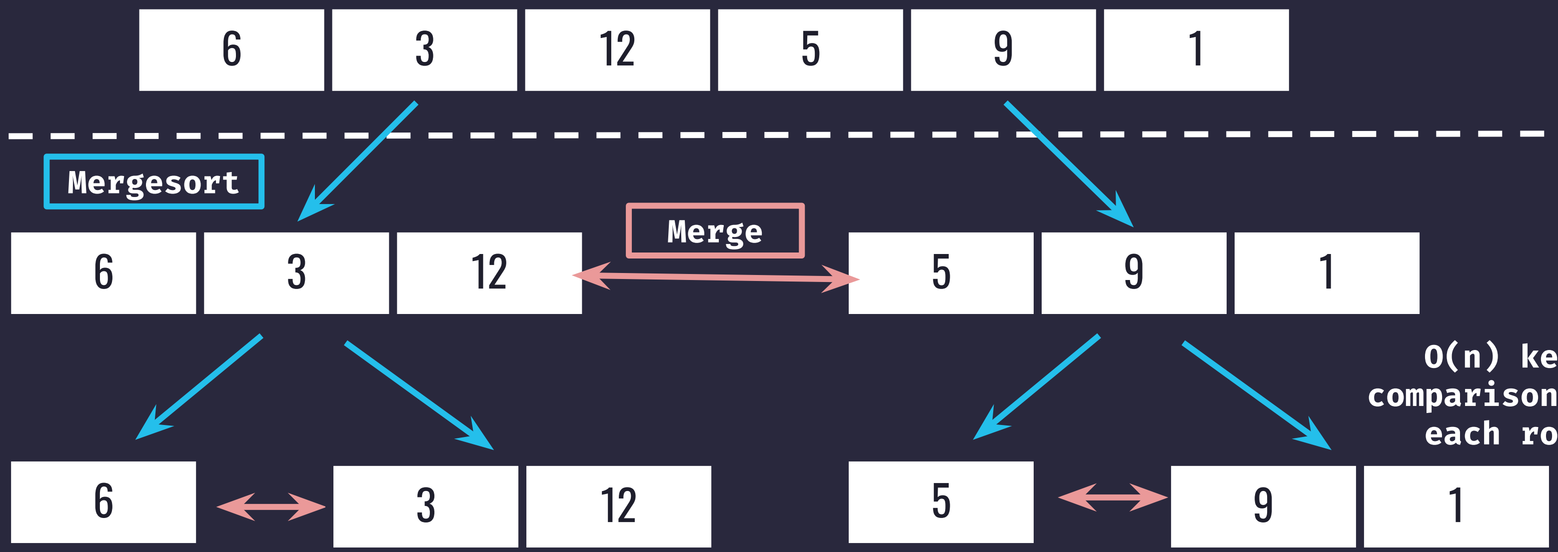
Worst case: $O(N^2)$

Each key has to be compared with all the previous keys (since the next key is always the smallest)

$\boxed{\textbf{Average case: } O(N^2)}$

Each i^{th} key has an i^{th} chance of comparing with the previous keys.

Merge Sort (S = 3)



No. of levels = $\log_2 N$

O(n)*level → time complexity = O(nlogn)

O(n) key comparisons in each row

Time complexity = O(NlogN)

<u>Comparison:</u> At lower N values, N^2 < NlogN.

With a lower array size where it is more likely to be ordered, insertion sort is likely to be more efficient. Moreover, time complexity of mergesort is independent of initial order of the elements.

Key Comparisons with Fixed S

We conducted hybrid sort on array sizes from 10 to 10 million with a fixed value of S=10. Saved the number of key comparisons for each size as a txt file.

Used the data to plot graphs of array sizes taken against magnitude of S.

```
# Random S value.
S = 50
comparisons_SFixed = {size: hybrid_sort(datasets[size], 0, size-1, S) for size in sizes}
print(comparisons_SFixed)
```

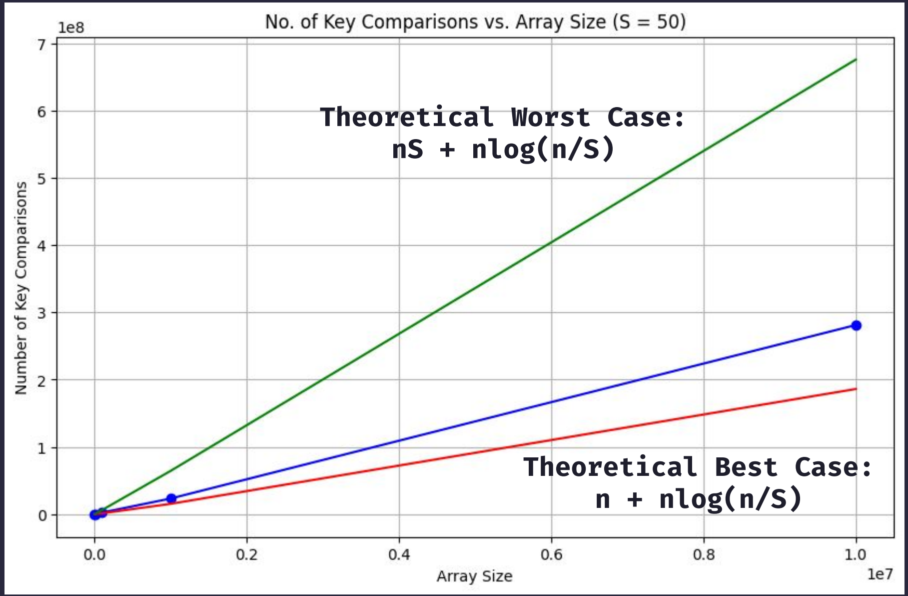
Results:

{1000: 13193, 10000: 183691, 100000: 2384838, 1000000: 23183546, 10000000: 281241446}

Key Comparisons with Fixed S

Code for Graph Plotting

```
import matplotlib.pyplot as plt
import numpy as np

x = sizes
y = [comparisons_SFixed[size] for size in
sizes]
best line = [size + size * np.log2(size/S) for
size in sizes]
worst line = [size * S + size * np.log2(size/S)
for size in sizes]

plt.figure(figsize=(10, 6))
plt.plot(x, y, marker='o', linestyle='-',
color='b')
plt.plot(x, best line, marker='',
linestyle='-', color='r', label='n +
nlog(n/{})'.format(S))
plt.plot(x, worst line, marker='',
linestyle='-', color='g', label='{}n +
nlog(n/{})'.format(S,S))
plt.title('No. of Key Comparisons vs. Array
Size (S = {})'.format(S))
plt.xlabel('Array Size')
plt.ylabel('Number of Key Comparisons' )
plt.grid(True)
plt.show()
```



S = 50

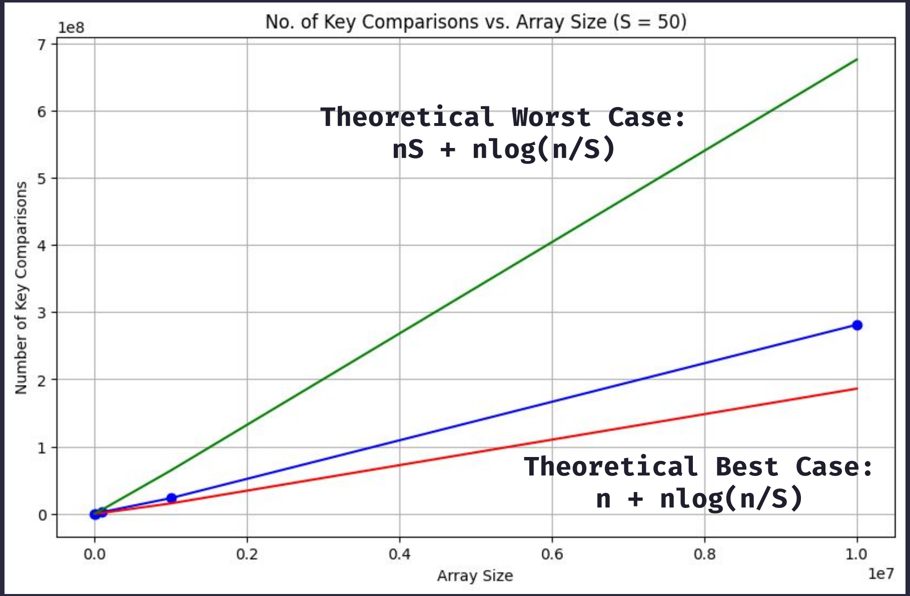
Key Comparisons with Fixed S

Empirical Analysis

As array size increases, the number of key comparisons increases

Comparison With Theoretical Analysis

When array size increases, number of key comparison increases within the bounds of the worst case and best case time complexity.

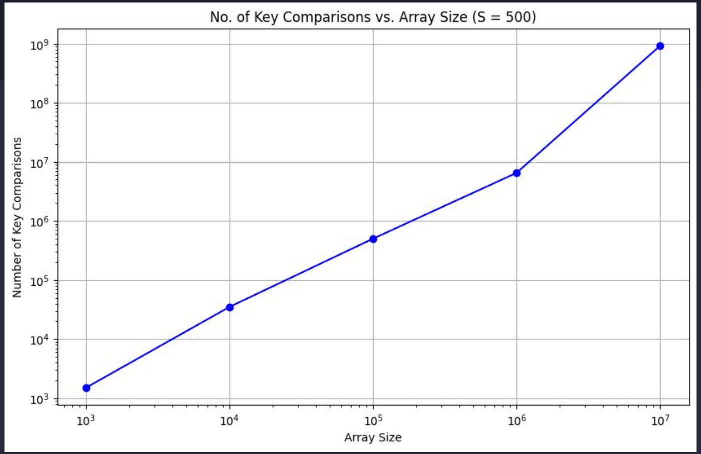


S = 50

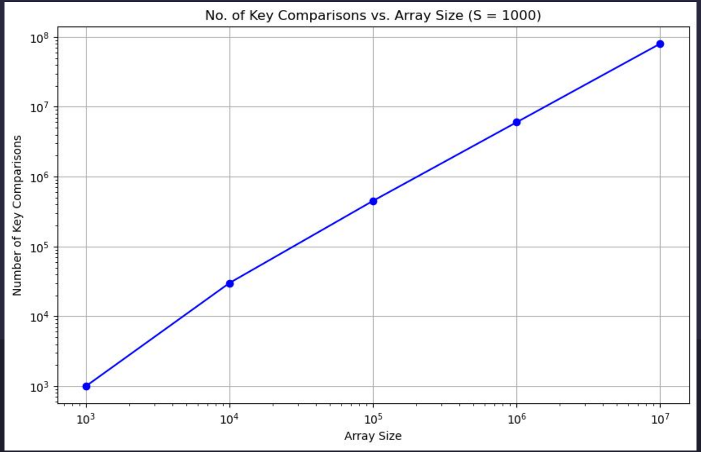
Analysis

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

S = 500



S = 1000



In general, graphs appear linear. Since time complexity is O(nS + nlog(n/S)), we expect that graphs resemble that of an O(nlogn) graph.

Since the axes are logarithmic, the graph performs linearly.

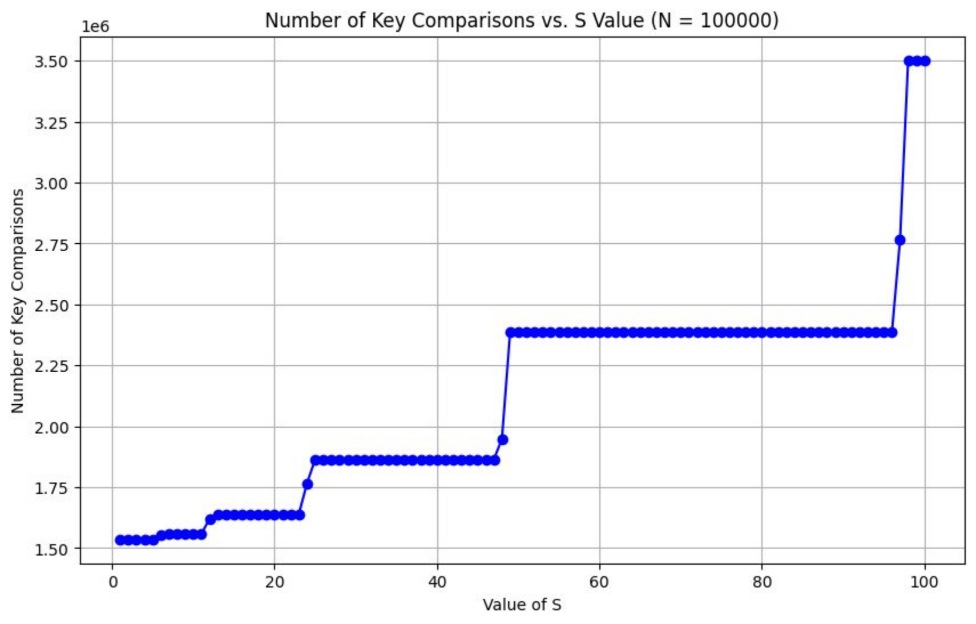
Key Comparisons Against Different S

For analysing number of key comparisons using hybrid sort on array of size 10 million, we used values of S ranging from 1 to 100.

```python
# Different Values of S
S_List= [i for i in range(1,101)]

# Input size n = 100,000
n = 100000
comparisons SList = {S: hybrid_sort(copy.deepcopy(datasets[n]), 0, n-1, S) for S in S_List}
print(comparisons_SList)
```

Key Comparisons Against Different S

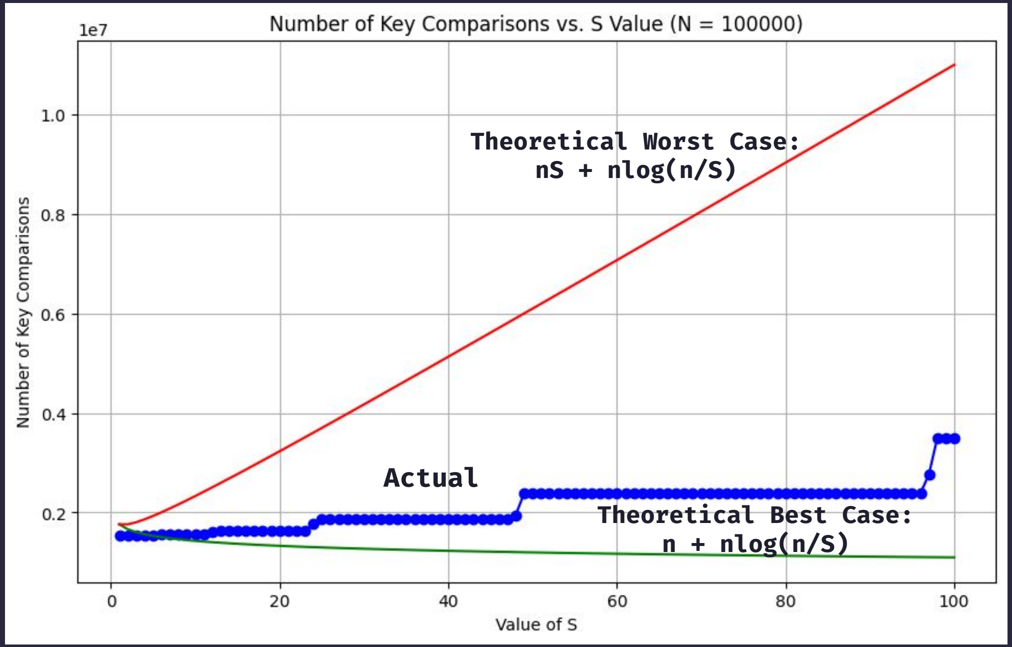
```
x = S_List
y = [comparisons_SList[S] for S in
S_List]

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(x, y, marker='o',
linestyle='-', color='b')
plt.title('Number of Key
Comparisons vs. S Value (N =
{})'.format(n))
plt.xlabel('Value of S')
plt.ylabel('Number of Key
Comparisons')
#plt.xscale('log')
plt.grid(True)
plt.show()
```



Key Comparisons Against Different S

<u>Empirical Analysis</u>
As S increases from 1 to 100, the number of key comparisons increases in a stepwise manner.

<u>Comparison With Theoretical Analysis</u>
When S increases sufficiently to reduce number of merge sort iterations, number of key comparisons increase due to earlier switch to insertion sort.



Optimal Value of S

Solution 1:
Run Hybrid Sort algorithm with a several differently sized input array and adjust the S value for each array and plot the run time against it. But it may be very time consuming to do.

Solution 2 (More Time Efficient):
S value is the threshold for Hybrid Sort to **switch to Insertion Sort when it becomes faster than Merge Sort**. By comparing the runtime between Insertion Sort and Merge Sort, optimal S value should be the S value where Insertion Sort is fastest relative to Merge Sort.

To sort an array with Merge Sort without reimplementing the original Merge Sort, we can set S = 0. This causes the Hybrid Sort algorithm to never switch to Insertion Sort mode as all arrays that need to be sorted > 0 and go through the Merge Sort branch.

Optimal Value of S

```
insertion_sort_comparisons = []
merge_sort_comparisons = []


for size in sizes2:
    # Generate a random array of 'size' integers
    random.seed(10)
    arr2 = [random.randint(1, size) for _ in range(size)]

    # Insertion Sort
    arr2_insertion = arr2.copy()
    comparisons = insertion_sort(arr2_insertion, 0, size - 1)
    insertion_sort_comparisons.append(comparisons)

    # Merge Sort
    arr2_merge = arr2.copy()
    comparisons = hybrid_sort(arr2_merge, 0, size - 1, 0)
    merge_sort_comparisons.append(comparisons)
```

We created two lists and recorded key comparisons taken by insertion sort and merge sort as the value of S increases.

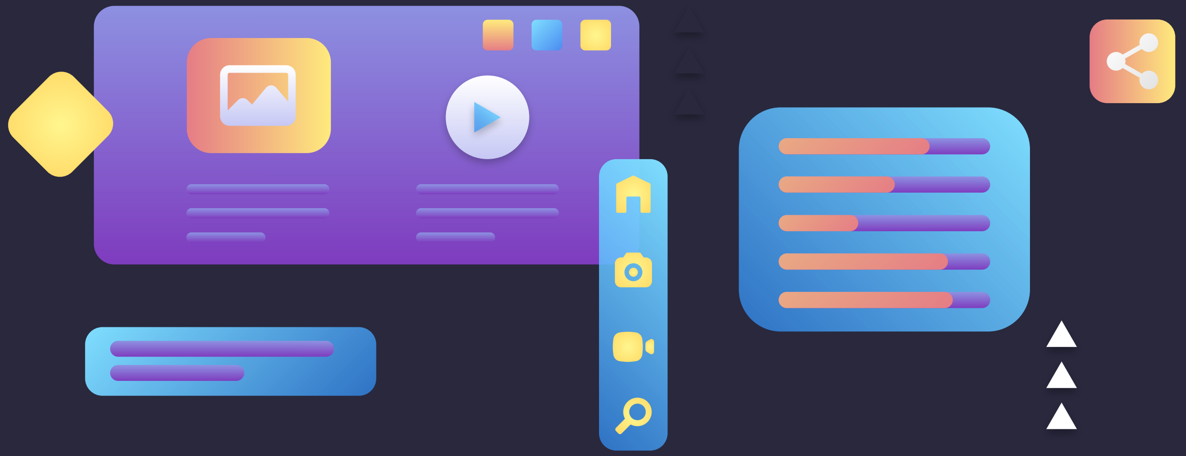
```
S_Optimal = 0
for i in range(len(sizes2) - 1):
    if merge_sort_comparisons[i] > insertion_sort_comparisons[i]:
        S_Optimal = i + 1

print("Merge    :", merge_sort_comparisons)
print("Insertion:", insertion_sort_comparisons)
print("Optimal S:", S_Optimal)
```

Whenever merge sort timing is greater than insertion sort, we increase S_optimal by 1. Insertion sort timing exceeds mergesort timing after S reaches 20.

Optimal Value of S

We recorded the time taken performing insertion sort and merge sort. Performance timing for insertion sort increases beyond the merge sort after a certain input size since $O(n^2) > O(n\log_2 n)$.

Optimal S:
20



/03 Comparison with Merge Sort

(c)i. & (d)

MergeSort vs Hybrid Sort

```python
def merge_sort(arr, l, r):
    comparisons = 0
    if l < r:
        m = (l + r) // 2
        # Sort left and right halves
        comparisons += merge_sort(arr, l, m)
        comparisons += merge_sort(arr, m + 1, r)
        comparisons += merge(arr, l, m, r)
    else:
        return 0
    return comparisons

start_time = time.time()
comparisons = merge_sort(datasets[10000000], 0, 10000000 - 1)
end_time = time.time()
merge_sort_time_10M = end_time - start_time
print("Key Comparisons (Merge Sort): ", comparisons)
print("Time Taken (Merge Sort): ", merge_sort_time_10M)

start_time = time.time()
comparisons = hybrid_sort(datasets[10000000], 0, 10000000 - 1, S_Optimal)
end_time = time.time()
hybrid_sort_time_10M = end_time - start_time
print("Key Comparisons (Hybrid Sort): ", comparisons)
print("Time Taken (Hybrid Sort): ", hybrid_sort_time_10M)
```

```
Key Comparisons (Merge Sort):  220102936
Time Taken (Merge Sort):  63.66669416427612
Key Comparisons (Hybrid Sort):  242317899
Time Taken (Hybrid Sort):  57.141727924346924
```

Firstly, we used MergeSort and Hybrid Sort on the 10 million dataset we have generated in part b to obtain Key Comparisons and CPU time.

MergeSort vs Hybrid Sort

```
import pandas as pd
df_time = pd.DataFrame(columns=("mergesort_execution_time", "hybridsort_execution_time"))
df_compare = pd.DataFrame(columns=("mergesortkeycomparisons", "hybridsortkeycomparisons"))
df_time.loc[0] = [merge_sort_time_10M, hybrid_sort_time_10M]
df_compare.loc[0] = [comparisons, hybrid_comparisons]

    for i in range (1,10):

        array_size = 10000000
        randomarray = [random.randint(1, 10000000) for _ in range(array_size)]


        start_time = time.time()
        mergesort_comparisons = merge_sort(randomarray, 0, 10000000 - 1)
        end_time = time.time()
        merge_sort_time_10M = end_time - start_time
        print("Key Comparisons (Merge Sort): ", comparisons)
        print("Time Taken (Merge Sort): ", merge_sort_time_10M)

        start_time = time.time()
        hybrid_comparisons = hybrid_sort(randomarray, 0, 10000000 - 1, S_Optimal)
        end_time = time.time()
        hybrid_sort_time_10M = end_time - start_time
        print("Key Comparisons (Hybrid Sort): ", hybrid_comparisons)
        print("Time Taken (Hybrid Sort): ", hybrid_sort_time_10M)

        df_time.loc[i] = [merge_sort_time_10M, hybrid_sort_time_10M]
        df_compare.loc[i] = [comparisons, hybrid_comparisons]
```

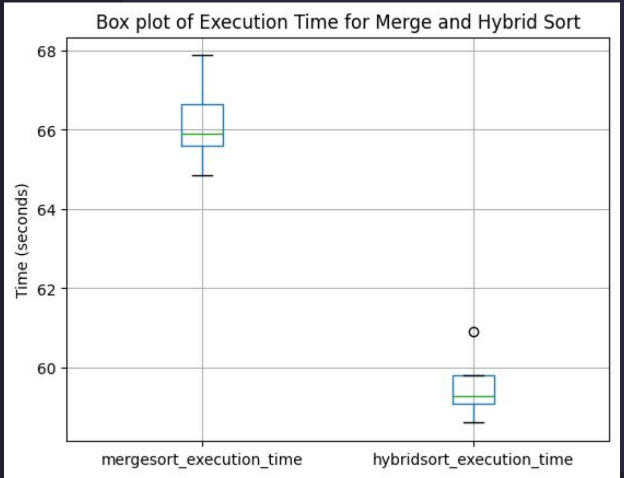
Next, we created 2 data frames to store the CPU time and key comparisons of MergeSort and Hybrid Sort.

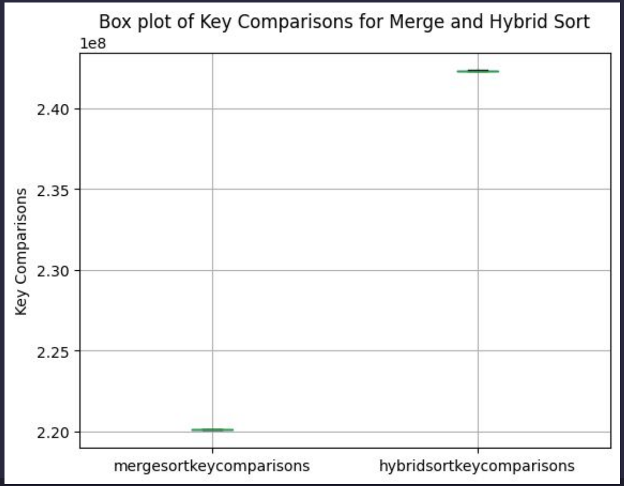
Subsequently, we generated 9 addition 10 million arrays and repeated the process of MergeSort, Hybrid Sort and storing the values into the data frames.

MergeSort vs Hybrid Sort

```python
df_time.boxplot(column=["mergesort_execution_time", "hybridsort_execution_time"])
plt.title('Box plot of Execution Time for Merge and Hybrid Sort')
plt.ylabel('Time (seconds)')
plt.show()

df_compare.boxplot(column=["mergesortkeycomparisons", "hybridsortkeycomparisons"])
plt.title('Box plot of Key Comparisons for Merge and Hybrid Sort')
plt.ylabel('Key Comparisons')
plt.show()
```

Lastly, we created a boxplot for both the data frames.





Using the CPU time, Hybrid Sort has a better performance.

Despite more comparisons which are reflected by it’s time complexity, the decreases computational overhead for recursive calls decrease time overall.

<Thank You>