SC2001

Lab 1

Nisha, Zi Xiao, Maru, Shruti

(a) Algorithm Implementation

```python
# Insertion Sort
def insertion_sort(arr, left, right, counter):
    for i in range(left + 1, right + 1):
        key = arr[i]
        j = i - 1
        while j >= left:
            counter[0] += 1  # comparison
            if arr[j] > key:
                arr[j + 1] = arr[j]
                j -= 1
            else:
                break
        arr[j + 1] = key
```

INSERTION SORT

```python
# Merge Function
def merge(arr, left, mid, right, counter):
    n1 = mid - left + 1
    n2 = right - mid

    L = arr[left:left + n1]
    R = arr[mid + 1:mid + 1 + n2]

    i = j = 0
    k = left

    while i < n1 and j < n2:
        counter[0] += 1  # comparison
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
```

```python
while i < n1:
    arr[k] = L[i]
    i += 1
    k += 1
while j < n2:
    arr[k] = R[j]
    j += 1
    k += 1
```

Merge Function

(a) Algorithm Implementation

```python
# Standard MergeSort
def merge_sort(arr, left, right, counter):
    if left < right:
        mid = (left + right) // 2
        merge_sort(arr, left, mid, counter)
        merge_sort(arr, mid + 1, right, counter)
        merge(arr, left, mid, right, counter)
```

STANDARD MERGE SORT

```python
# Hybrid MergeSort (threshold S)
def hybrid_merge_sort(arr, left, right, counter, S):
    if right - left + 1 <= S:
        insertion_sort(arr, left, right, counter)
    else:
        if left < right:
            mid = (left + right) // 2
            hybrid_merge_sort(arr, left, mid, counter, S)
            hybrid_merge_sort(arr, mid + 1, right, counter, S)
            merge(arr, left, mid, right, counter)
```

HYBRID MERGE SORT

(b) Experiment Implementation

```
# Experiment
def experiment_vary_n(ns, fixed_s, trials):
    results = []
    for n in ns:
        hybrid_counts, hybrid_times = [], []
        for _ in range(trials):
            arr = [random.randint(0, x) for _ in range(n)]
            count_hybrid = [0]
            start = time.perf_counter()
            hybrid_merge_sort(arr.copy(), 0, n-1, count_hybrid, fixed_s)
            end = time.perf_counter()
            hybrid_counts.append(count_hybrid[0])
            hybrid_times.append(end - start)

        results.append({
            "n": n,
            "s": fixed_s,
            "hybrid_count": sum(hybrid_counts) / trials,
            "hybrid_time": sum(hybrid_times) / trials
        })
    return results
```

collect data from each experiment

run experiment this no. of times

run experiment this no. of times

final data = avg of the trials saved as results

Experiment 1: Varying n, fixed s

```
def experiment_vary_s(ss, fixed_n, trials):
    results = []
    for s in ss:
        hybrid_counts, hybrid_times = [], []
        for _ in range(trials):
            arr = [random.randint(0, x) for _ in range(fixed_n)]
            count_hybrid = [0]
            start = time.perf_counter()
            hybrid_merge_sort(arr.copy(), 0, fixed_n-1, count_hybrid, s)
            end = time.perf_counter()
            hybrid_counts.append(count_hybrid[0])
            hybrid_times.append(end - start)

        results.append({
            "n": fixed_n,
            "s": s,
            "hybrid_count": sum(hybrid_counts) / trials,
            "hybrid_time": sum(hybrid_times) / trials
        })
    return results
```

Experiment 2: Varying s, fixed n

(b) Experiment Implementation

```
# Export results to csv file
def save_results_to_csv(filename, results):
    keys = results[0].keys()
    with open(filename, "w", newline="") as f:
        writer = csv.DictWriter(f, fieldnames=keys)
        writer.writeheader()
        writer.writerows(results)
```

```
# Generate datasets
n = [1000, 10000, 100000, 1000000, 10000000] # sizes for n
x = 1000 # max value allowed in array
s = list(range(1,30)) # threshold s for hybrid sort (if subarr<S, use insertion sort instead of recrusive mergesort)
n_fixed = 50000
s_fixed = 10
```

```
# Run code
res_n = experiment_vary_n(n, s_fixed, trials=10)
res_s = experiment_vary_s(s, n_fixed, trials=10)

save_results_to_csv("results_vary_n.csv", res_n)
save_results_to_csv("results_vary_s.csv", res_s)
```

(b) Experiment Implementation

```python
# Read Experiment fixed s, varying n
df = pd.read_csv("results_vary_n.csv")
df.head()
```

	n	s	hybrid_count	hybrid_time
0	1000	10	9011.0	0.003458
1	10000	10	127231.0	0.034132
2	100000	10	1557722.0	0.336038
3	1000000	10	19067217.0	5.342286
4	10000000	10	226346001.0	64.177780

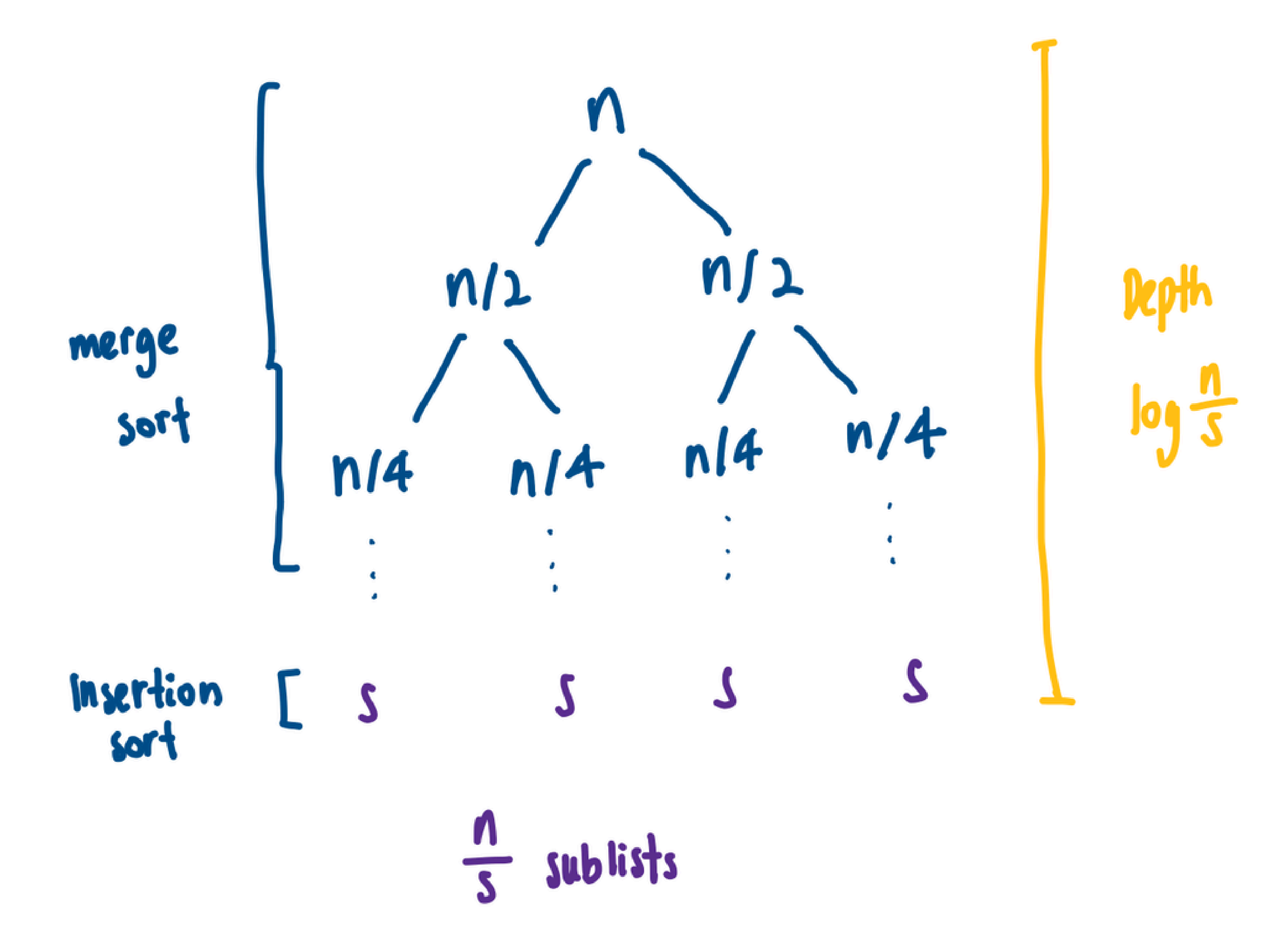
Experiment 1: Varying n, set s

```python
# Read Experiment fixed n, varying s
df = pd.read_csv("results_vary_s.csv")
df.head()
```

	n	s	hybrid_count	hybrid_time
0	50000	5	718343.0	0.151705
1	50000	10	728451.0	0.142924
2	50000	15	769300.0	0.148074
3	50000	20	769497.0	0.142903
4	50000	30	879733.0	0.178232

Experiment 2: Varying s, set n

(c) Theoretical Time Complexity



- Stop split when array size <= S, switching to insertion sort
- At the lowest level of splitting, each leaf sublist has size S
- To cover the whole array of size n, you need roughly n/s sublists
- Each split divides the size by 2 until we reach S, log2(n/s) would be depth

(c) Analysis of time complexity

Merge Sort

time complexity: $O(n \log n)$

$\downarrow$

$n * \log(n/s)$

Implementing Hybrid Algorithm

Insertion Sort

time complexity: $O(n^2)$

$\downarrow$

$(n/s) * s^2 = O(ns)$

Hybrid Sort time complexity = $n \log(n/s) + ns$

Smaller s → more merge recursion, less insertion sort.

Larger s → fewer merge steps, more costly insertion sorts.

(c) Theoretical Optimal S

treat n as constant,

$$\frac{dT}{dS} = n - \frac{n}{S \ln 2}$$

set derivative to 0 to find S that minimizes $T(n, S)$,

$$n - \frac{n}{S \ln 2} = 0 \Rightarrow S = \frac{1}{\ln 2}$$

for avg case constants,

$$\frac{dT}{dS} = cn - \frac{n}{S \ln 2} = 0$$

$$= n\left(c - \frac{n}{S \ln 2}\right) = 0$$

$$\Rightarrow S = \frac{1}{c \ln 2}$$

$$S \approx \frac{1}{1/4 \ln 2} \approx 6$$

real avg. no. comparison in insertion $= \frac{1}{4} S^2$

- Optimal sublist size S does not depend on the total input size n
- The best value of S does not depend on how large the array is
- S only depends on the relative cost of insertion vs merge (the constant c), not on n

theoretical optimum around S ≈ 6.

(c)ii Experiment 1 (Varying n, fixed s) Plot

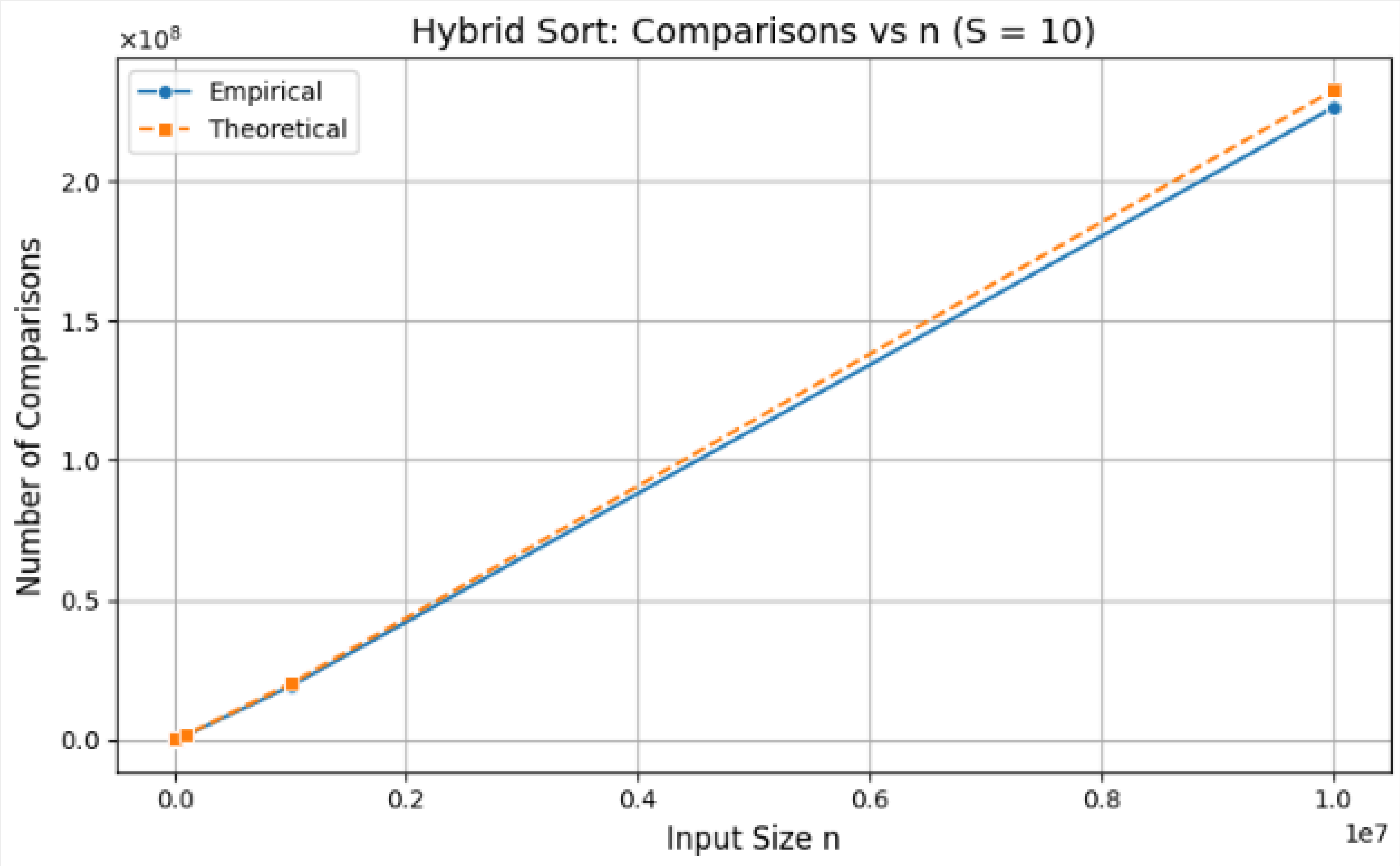
Objective:

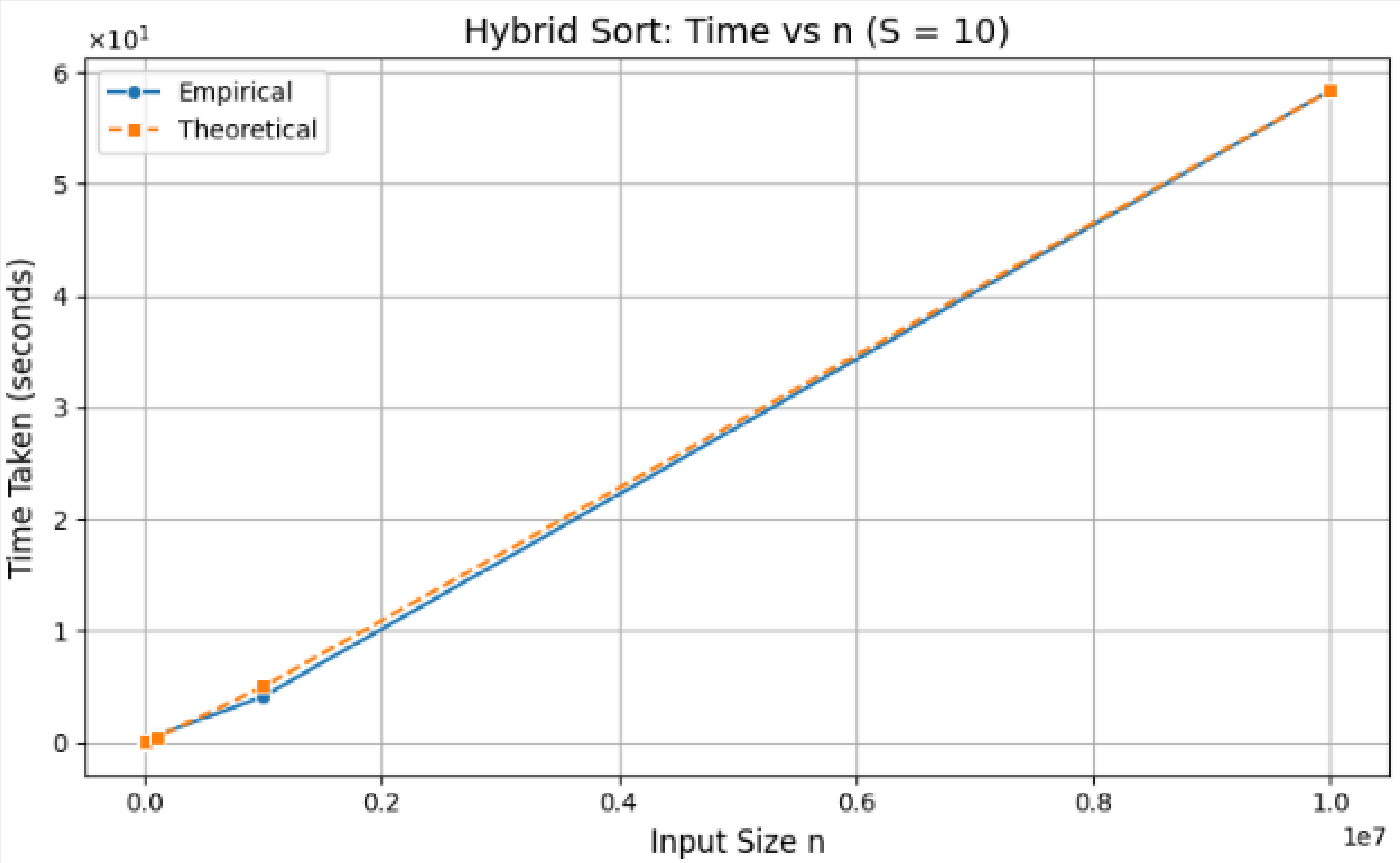
To study how the hybrid sorting algorithm scales with increasing input size n, while keeping the threshold S = 10 constant.

Method:

We varied n across a wide range and recorded both the number of comparisons and the time taken by the algorithm.

(c)i Experiment 1 (Varying n, fixed s) Plot





(c)i Experiment 1 Analysis

- Both comparisons and runtime increase with n, which aligns with the expected time complexity of $O(n \log n)$.
- This confirms that the hybrid approach maintains efficiency as the problem size scales.
- Scientific notation was used to format both axes for consistency across all plots.

(c)ii Experiment 2 (Varying s, fixed n) Plot

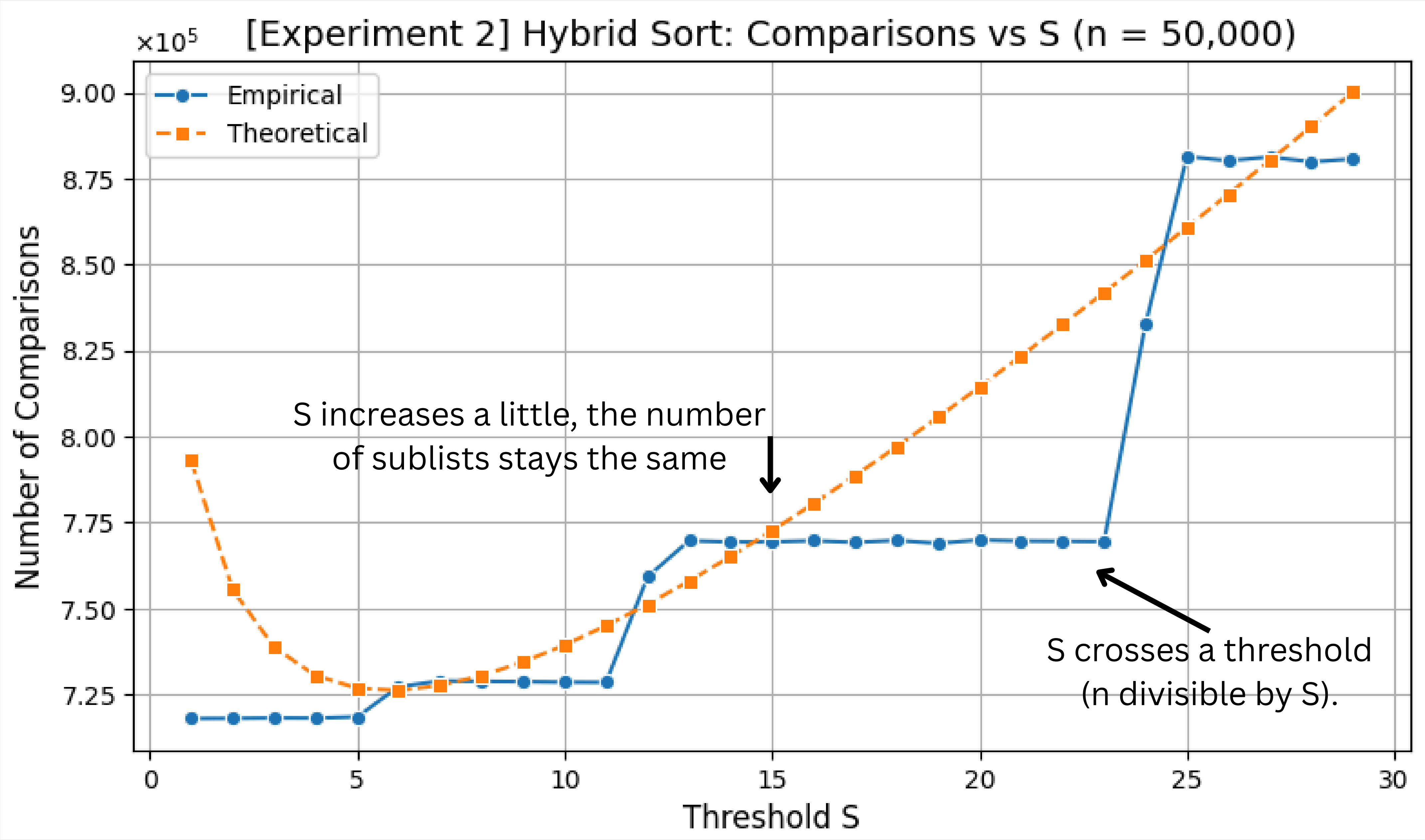
Objective:

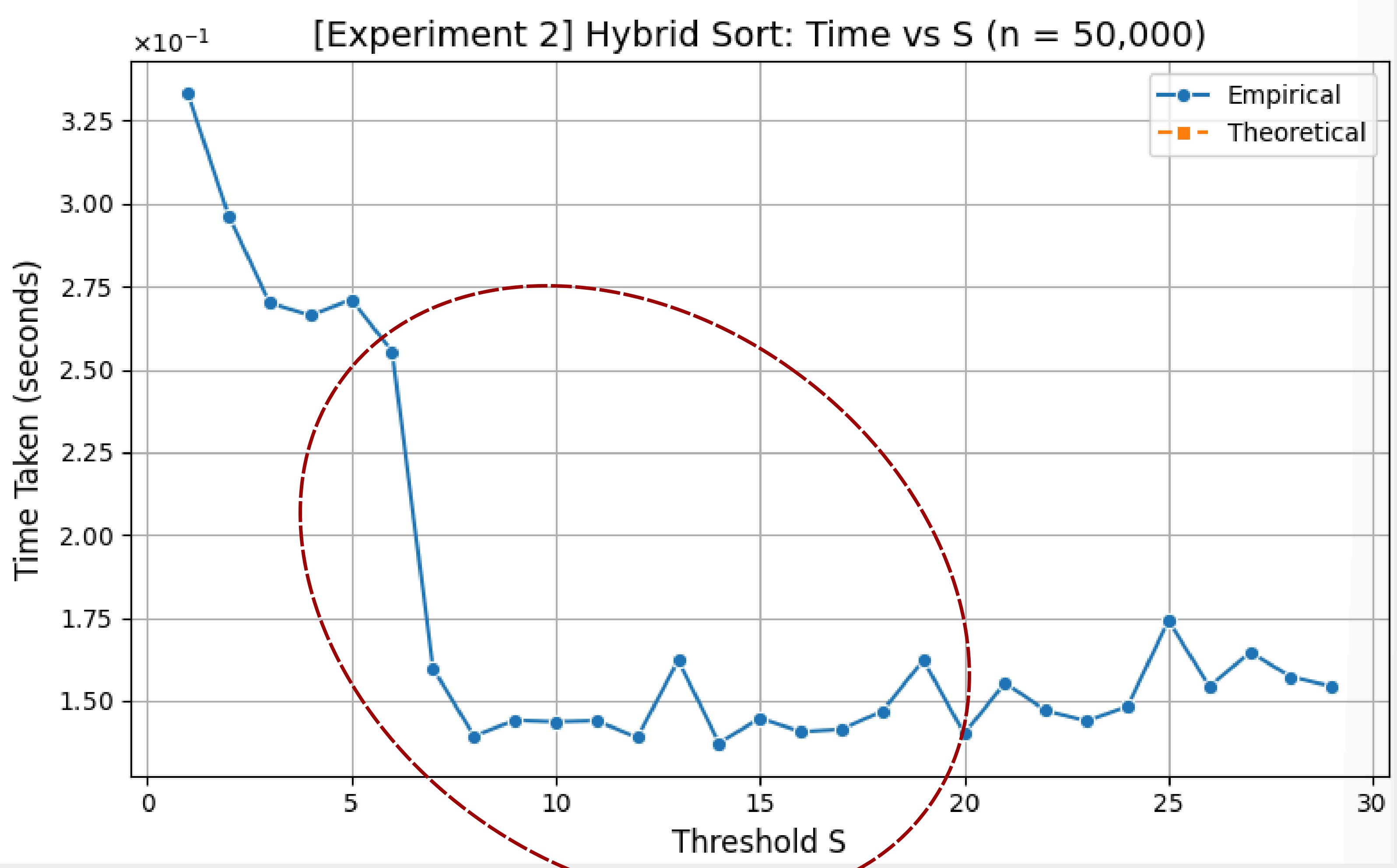
To determine the effect of the hybrid threshold S on performance, with a fixed input size n = 50,000.

Method:

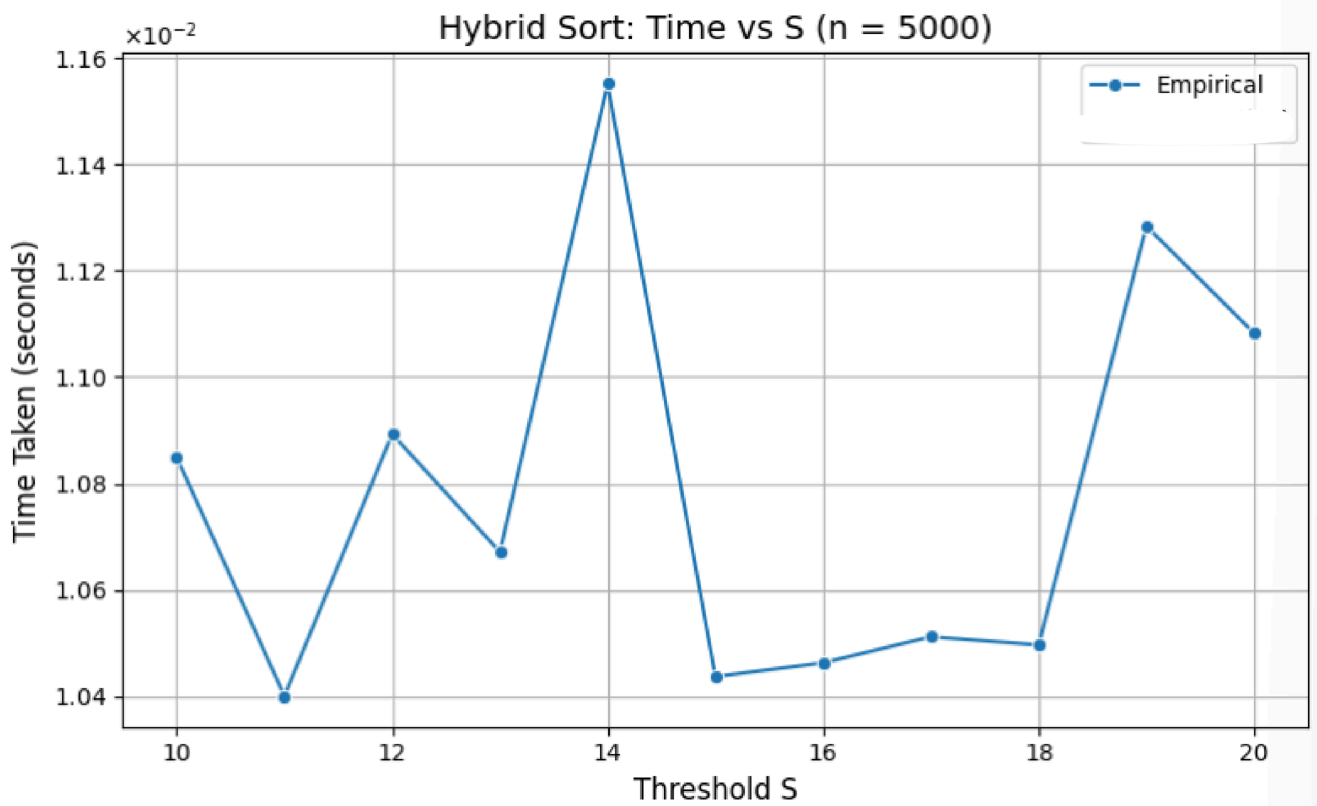
We varied S and plotted both comparisons and runtime for each value.

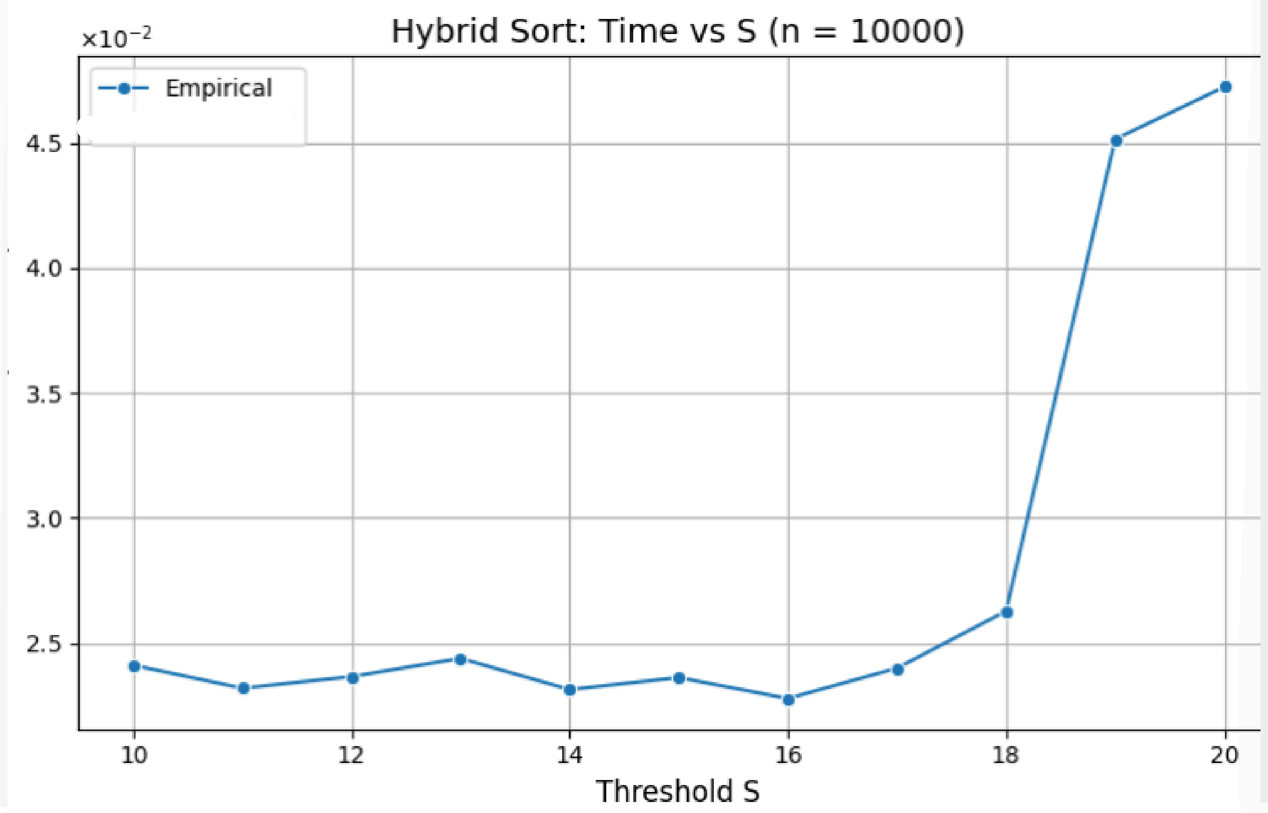
(c)ii Experiment 2 (Varying s, fixed n) Plot

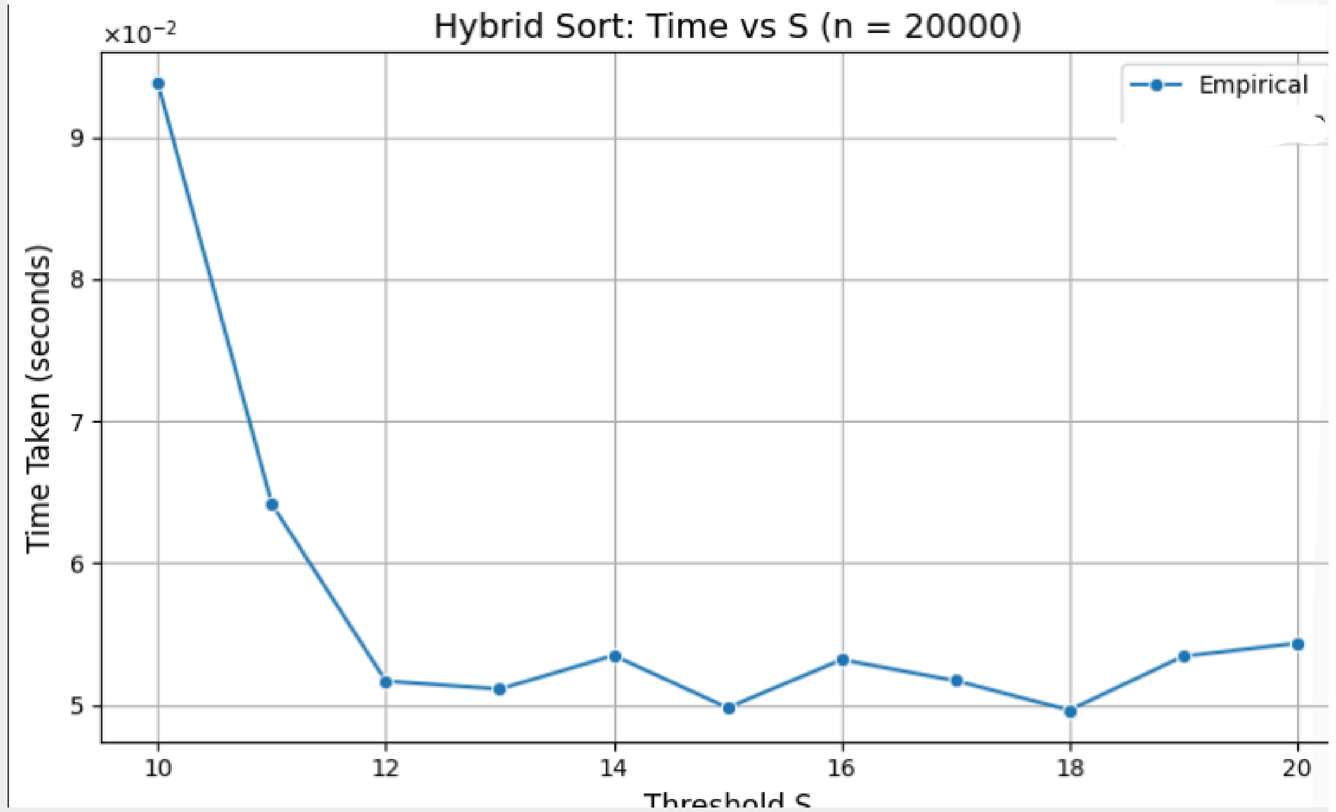


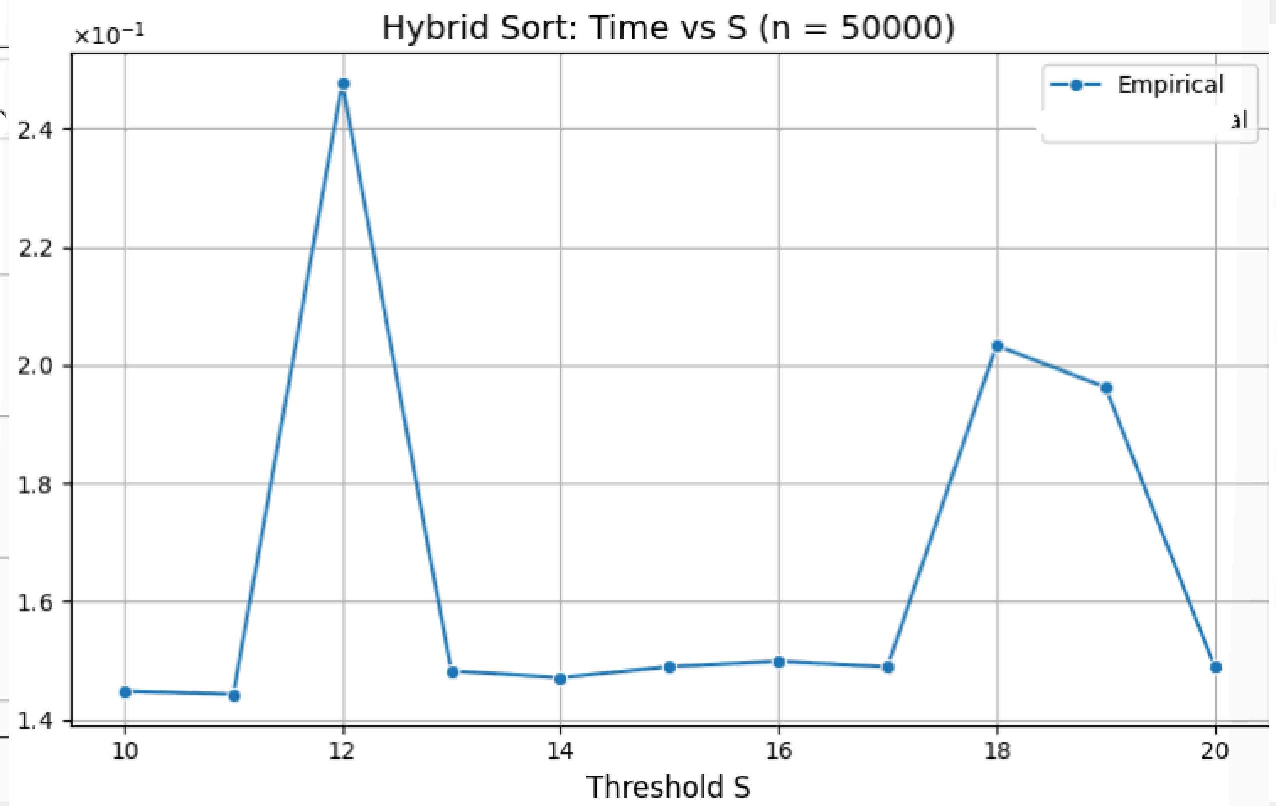


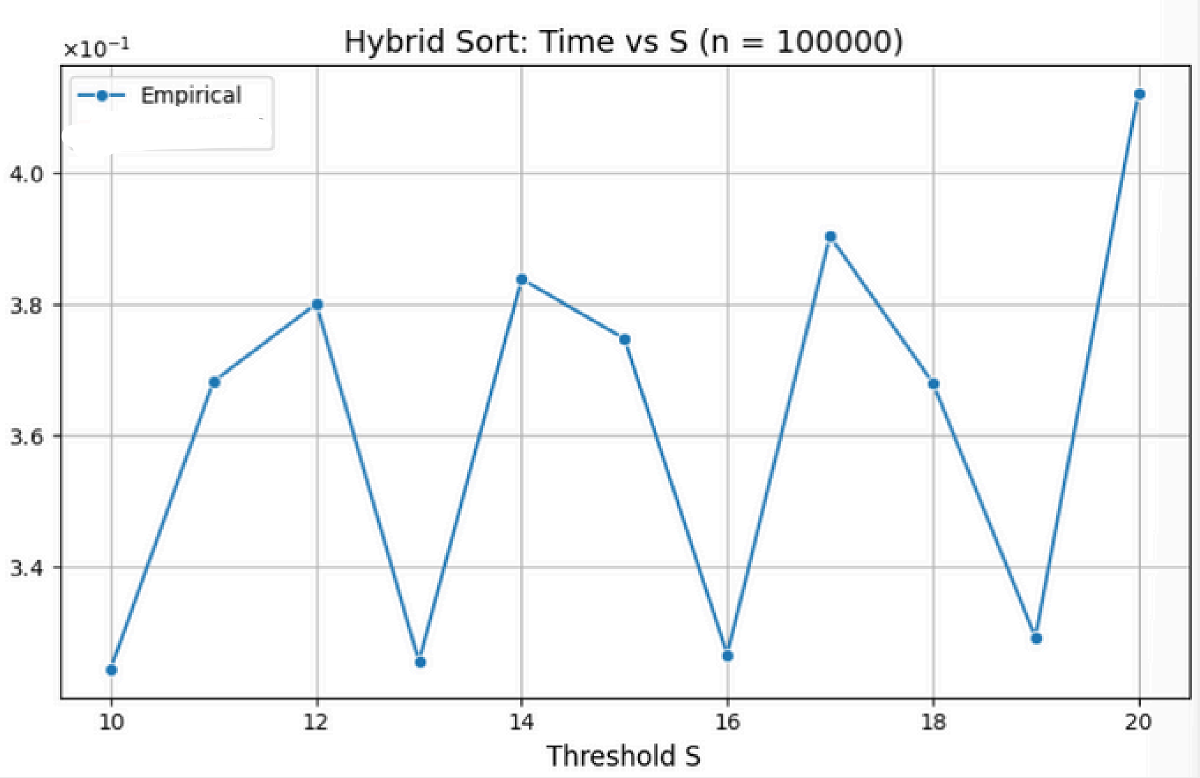
(c)iii Experiment Plot: Zoomed in S









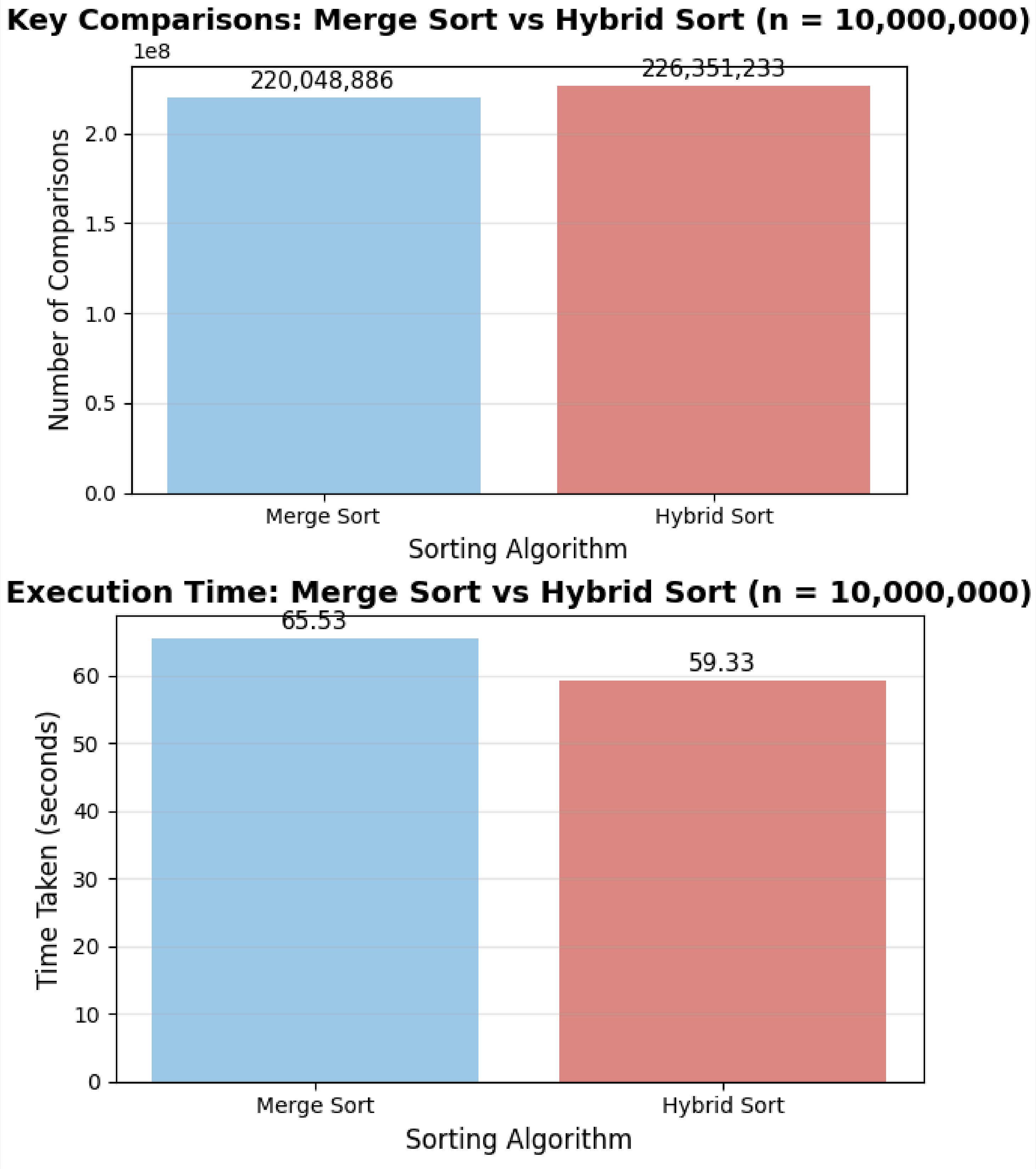


(c) (iii) Experiment Plot: Zoomed in S

Insights:

- This zoom-in removes outliers and improves accuracy by averaging results across trials=10
- The optimal threshold s for switching from merge sort to insertion sort was observed to be around 16 for the input sizes tested.
- Results varied slightly across trials due to the randomness of input arrays, but s = 16 consistently yielded the lowest average runtime
- Indicating it effectively balances the cost of recursive merging with the efficiency of insertion sort on small subarrays

(d) Original merge vs Hybrid Plot



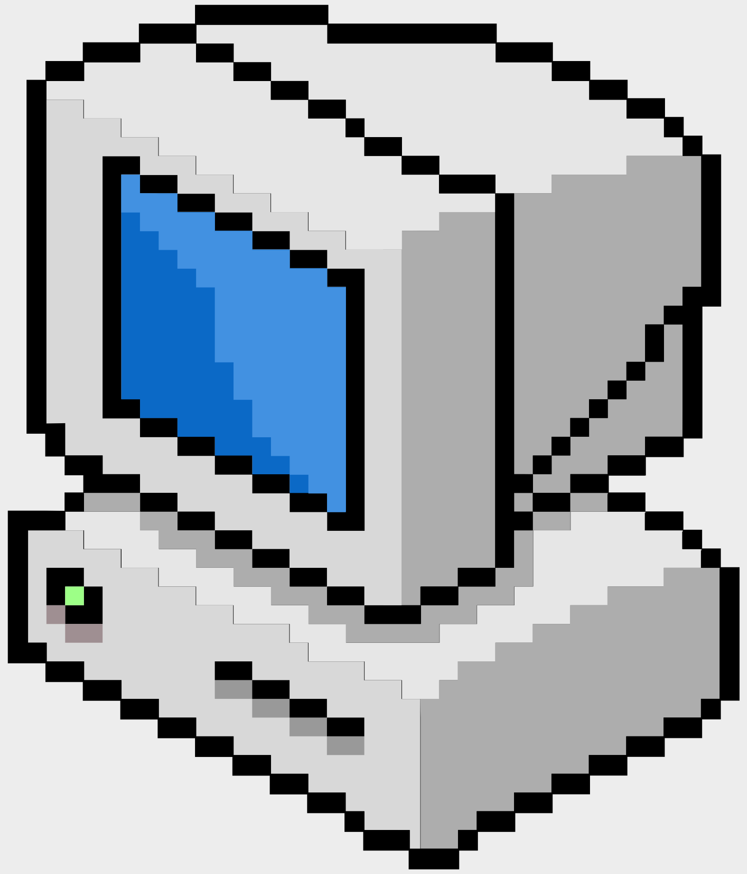
Result: Hybrid does ~2.9% more comparisons.

Result: Hybrid is ~9% faster despite more comparisons.

(d) Merge vs Hybrid Analysis

Why the is the hybrid sort faster?

- Skips ~$\log_2(S)$ merge levels → fewer recursive calls
- Insertion sort at leaves = tight loops, in-place, cache-friendly
- Fewer/lighter merges near the bottom
- Less Python function/recursion overhead



Result: Slightly more comparisons, but lower wall-clock time

(d) Further Analysis

Why Pure Merge Uses Fewer Comparisons

- Merge's comparison count is tightly bounded
- Insertion sort adds extra comparisons on random small subarrays
- Comparisons are cheap vs. allocations, copies, and recursion costs
- Result: Pure merge wins in comparisons, but not in runtime

Thank You