Project 1

Integration of Merge Sort & Insertion Sort

Hybrid Algorithm

hybridSort

```python
def hybridSort(arr, S):
    c = 0
    if len(arr) <= S:  # 1
        return insertionSort(arr)
    l = arr[:len(arr)//2]
    r = arr[len(arr)//2:]
    l, l_c = hybridSort(l, S)  # 2
    r, r_c = hybridSort(r, S)
    arr, c = merge(l, r)  # 3
    total = l_c + r_c + c
    return arr, total
```

1. Insertion Sort will be performed if the array length is less than or equal to S
2. Recursively partitioning arrays into subarrays until subarrays are of S-sized
3. Merges two sub-arrays of elements between index l and middle element and between middle element and index r

insertionSort

```python
def insertionSort(arr):
    c = 0
    for i in range(1, len(arr)):
        j = i
        while (j > 0) and (arr[j - 1] > arr[j]):
            c += 1
            arr[j - 1], arr[j] = arr[j], arr[j-1]
            j -= 1
        if j != 0:
            c += 1
    return arr, c
```

1. Insertion sort uses incremental approach
2. Keep comparing elements j - 1 and j until j == 0 or arr[j - 1] < arr[j]
3. Swap elements if j - 1 > j
4. j decreases by 1, and will break from ‘for’ loop if j == 0

merge

```
def merge(l, r):
    i = j = c = 0
    arr = []
    while i < len(l) and j < len(r):
        c += 1
        if l[i] <= r[j]:
            arr.append(l[i])        1
            i += 1
        else:
            arr.append(r[j])        2
            j += 1
    arr += list(l[i:])          2
    arr += list(r[j:])          3
    return arr, c
```

1. If 1st element of 1st half is smaller, put 1st element of 1st half into merged list
2. If 1st element of 1st half is bigger, put 1st element of 2nd half into merged list
3. If 1st elements of 2 halves are equal, put both of them into merged list

Algorithm Demonstration

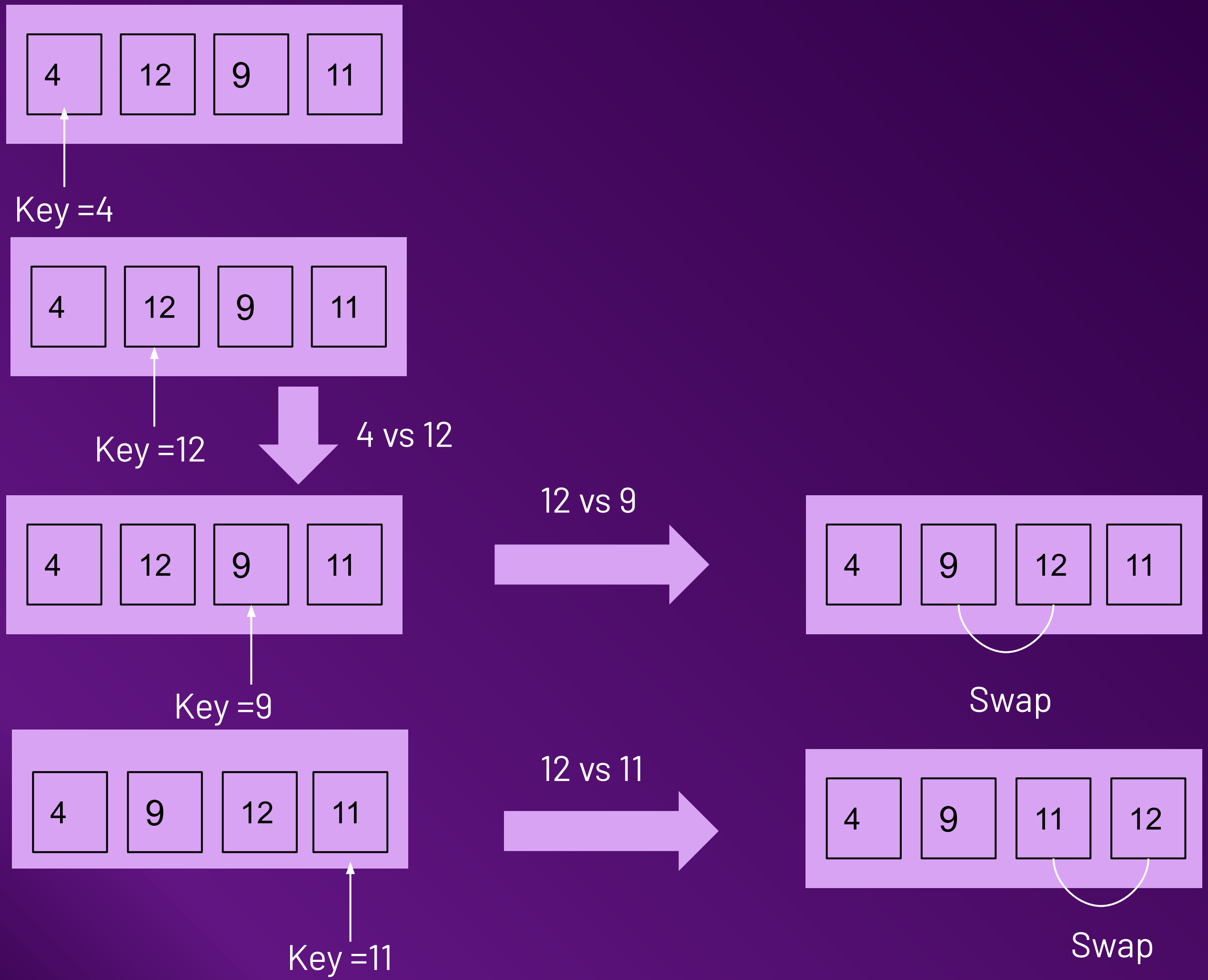
Taking S=4

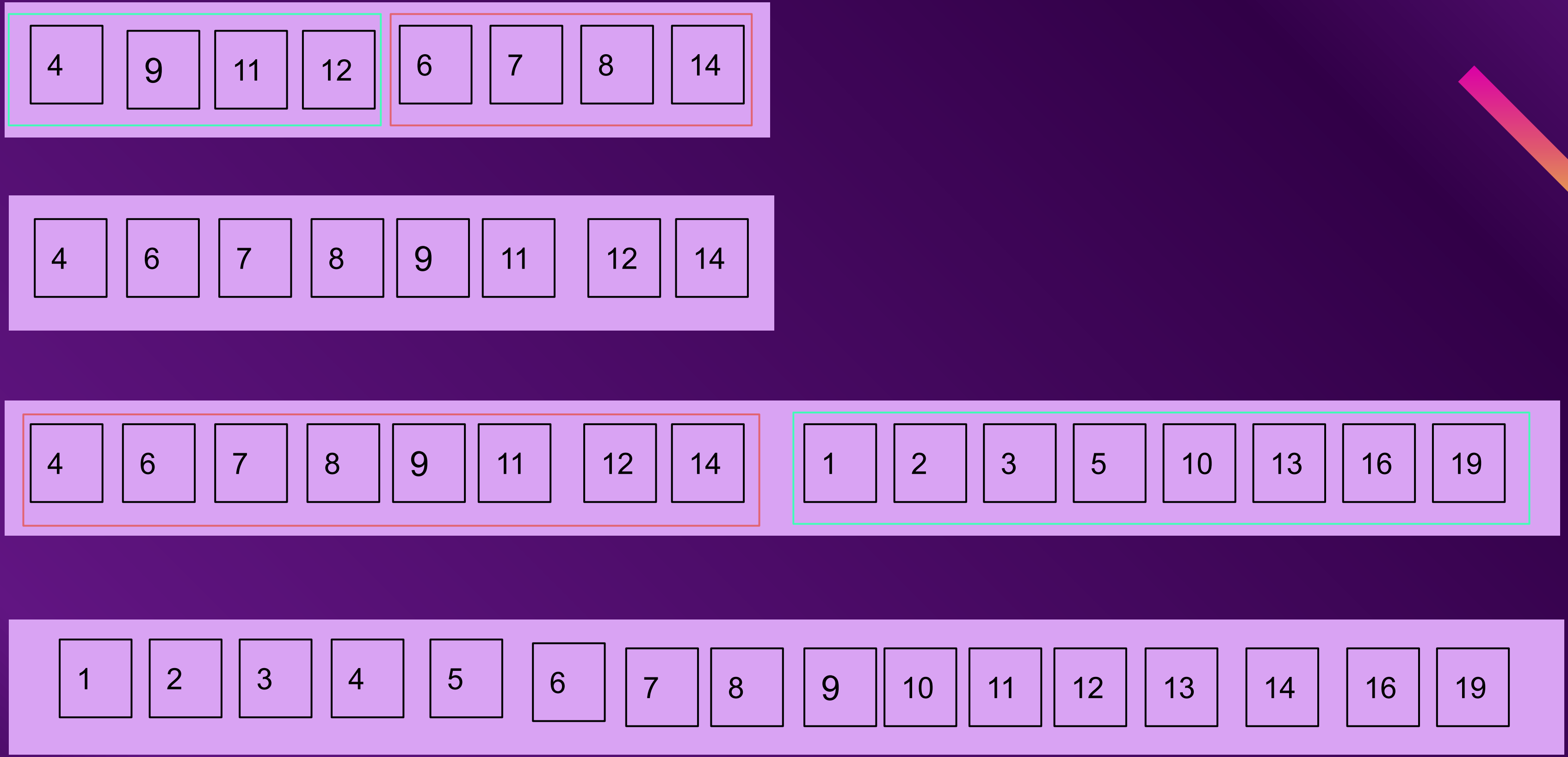
4	12	9	11	6	7	8	14	19	13	1	2	10	3	5	16

4	12	9	11	6	7	8	14

4	12	9	11

Lesser than s=4, carry out insertion sort


4
12
9
11
Key =4
4
12
9
11
Key =12
4 vs 12
4
12
9
11
Key =9
12 vs 9
4
9
12
11
Swap
4
9
12
11
Key =11
12 vs 11
4
9
11
12
Swap


4 9 11 12
6 7 8 14
4 6 7 8 9 11 12 14
4 6 7 8 9 11 12 14
1 2 3 5 10 13 16 19
1 2 3 4 5 6 7 8 9 10 11 12 13 14 16 19

```python
arr=[4,12,9,11,6,7,8,4,19,13,1,2,10,35,16]
print("Given array is: ")
print(arr)
print("\n")
arr, count = hybridSort(arr, S)
print("Sorted Array: ")
print(arr)
print("Number of Key Comparisons: " + str(count))
```

```
Given array is:
[4, 12, 9, 11, 6, 7, 8, 4, 19, 13, 1, 2, 10, 35, 16]


Sorted Array:
[1, 2, 4, 4, 6, 7, 8, 9, 10, 11, 12, 13, 16, 19, 35]
Number of Key Comparisons: 40
```

Finding Best S Value

Generate Input Data

Generate arrays of increasing sizes, in a range from 1,000 to 10 million. For each of the sizes, generate a random dataset of integers in the range of [1, ..., x], where x is the largest number you allow for your datasets.

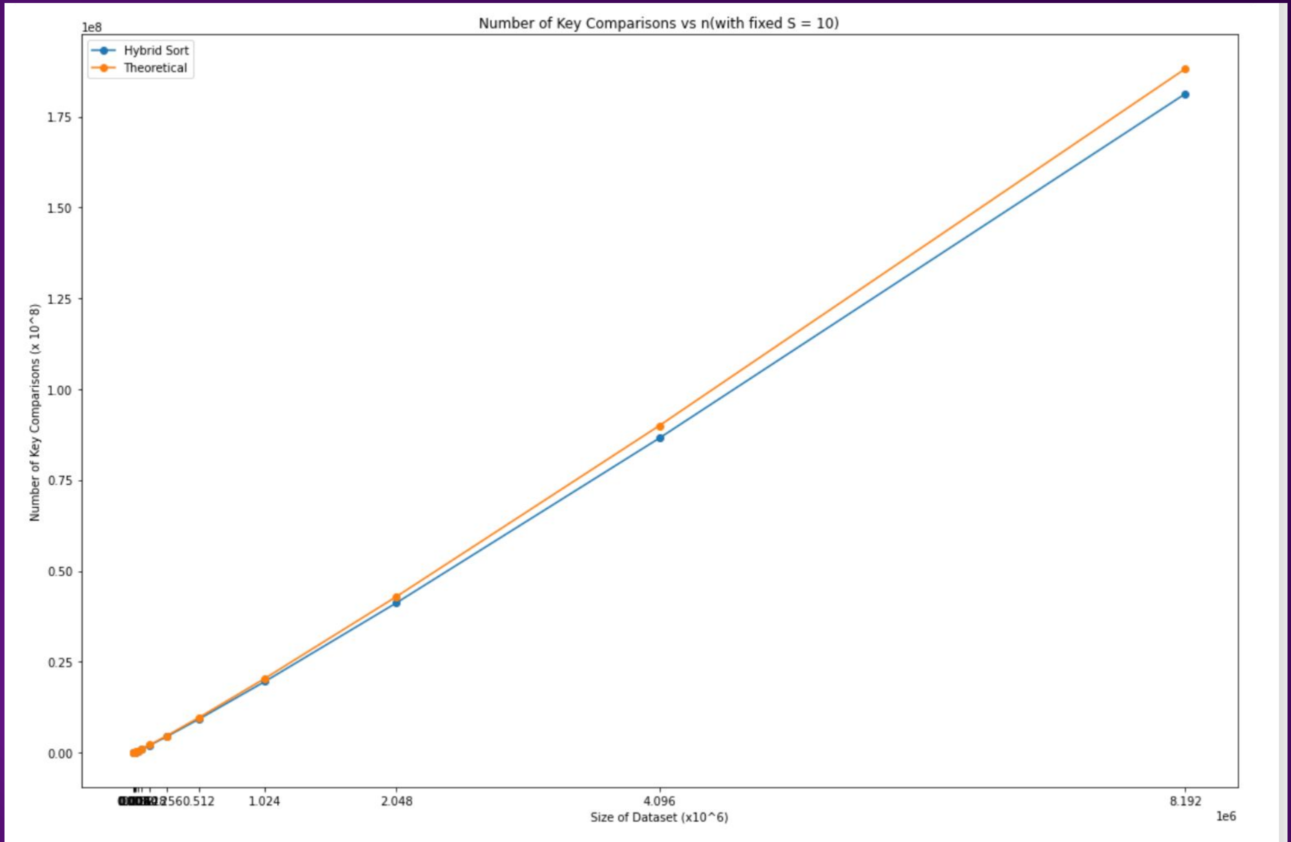
Enter Array Size: 1000
Array generated:
[782, …,255, 181, 150]
Enter Subarray Size: 4
Sorted Array:
[1,..1000]
Number of Key Comparisons: 8730

Enter Array Size: 10,000
Array generated:
[7953, 6690, 9765,...,1063, 9711, 3764]
Enter Subarray Size: 4
Sorted Array:
[1,..10000]
Number of Key Comparisons: 120470

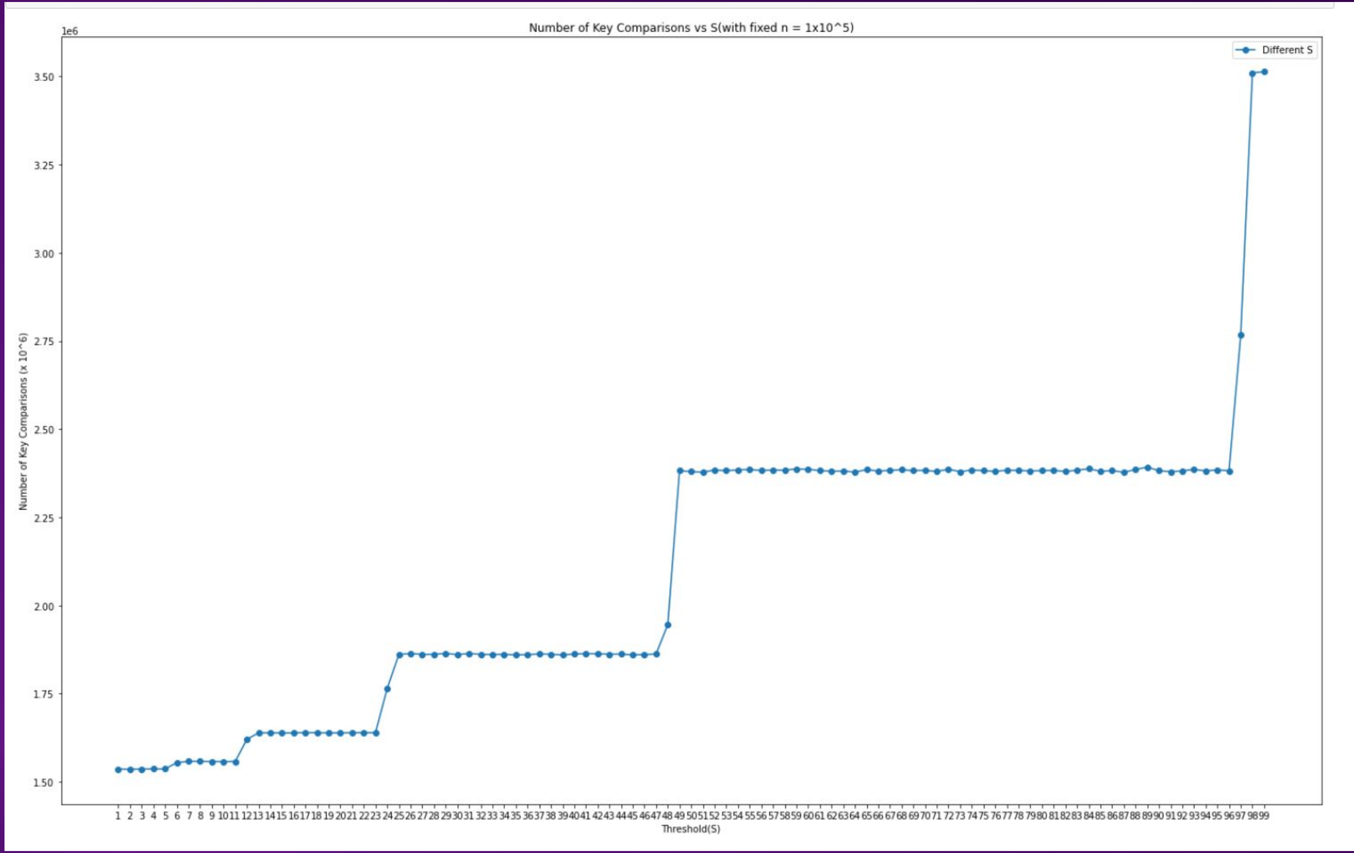
Enter Array Size: 100,000
Array generated:
[16557, 70464, 7647, 81506
,...19215, 1180, 4373]
Enter Subarray Size: 4
Sorted Array:
[1,..100000]
Number of Key Comparisons: 1536454

```
#can generate using the following code
arr_size = 1000; # change the value here to change array size
arr = random.sample(range(1, arr_size+1), arr_size)
```

Key Comparisons vs List Size (S=10)



Key Comparisons vs Subarray Size

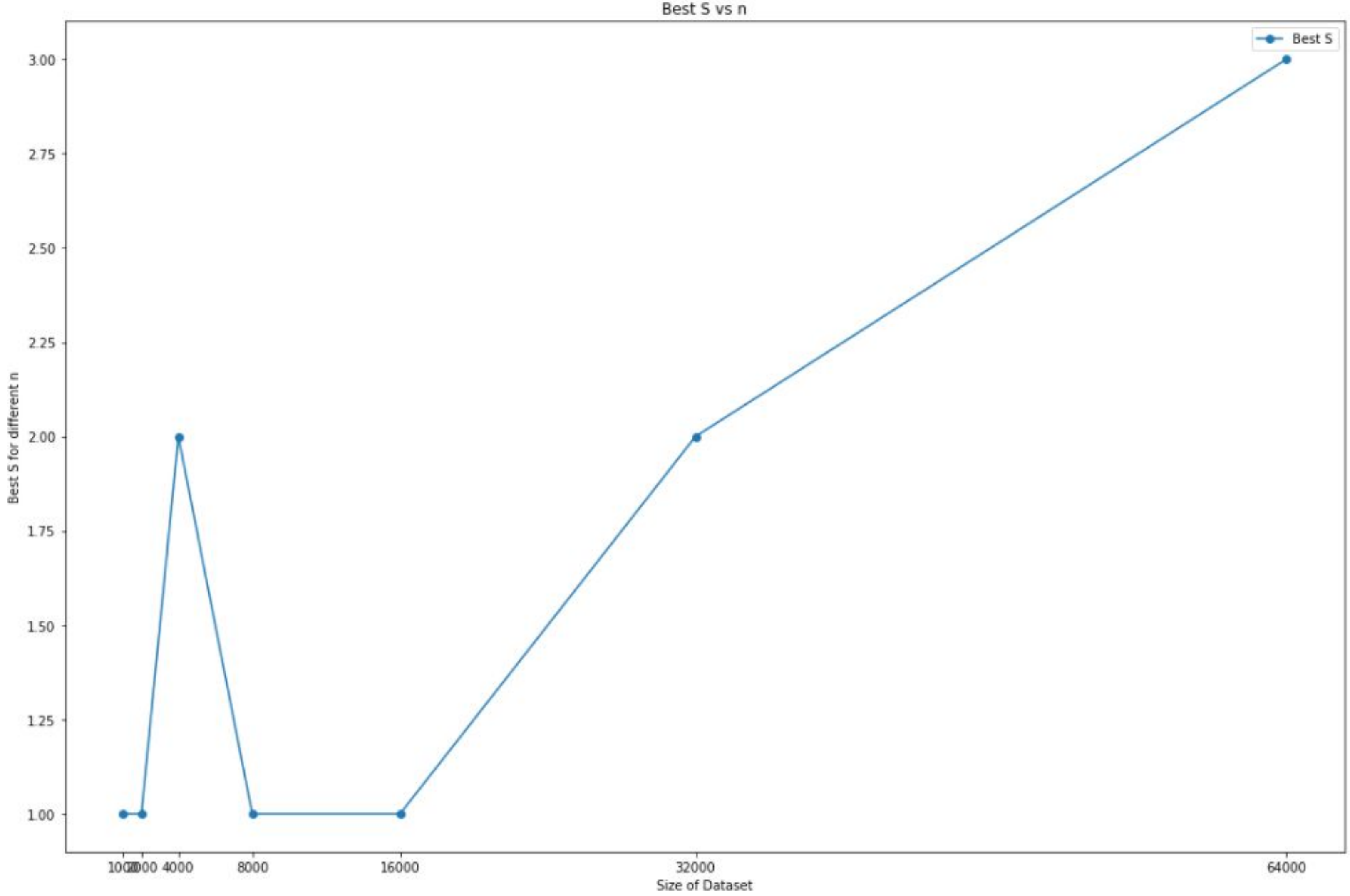


Determining Rough Range Of S

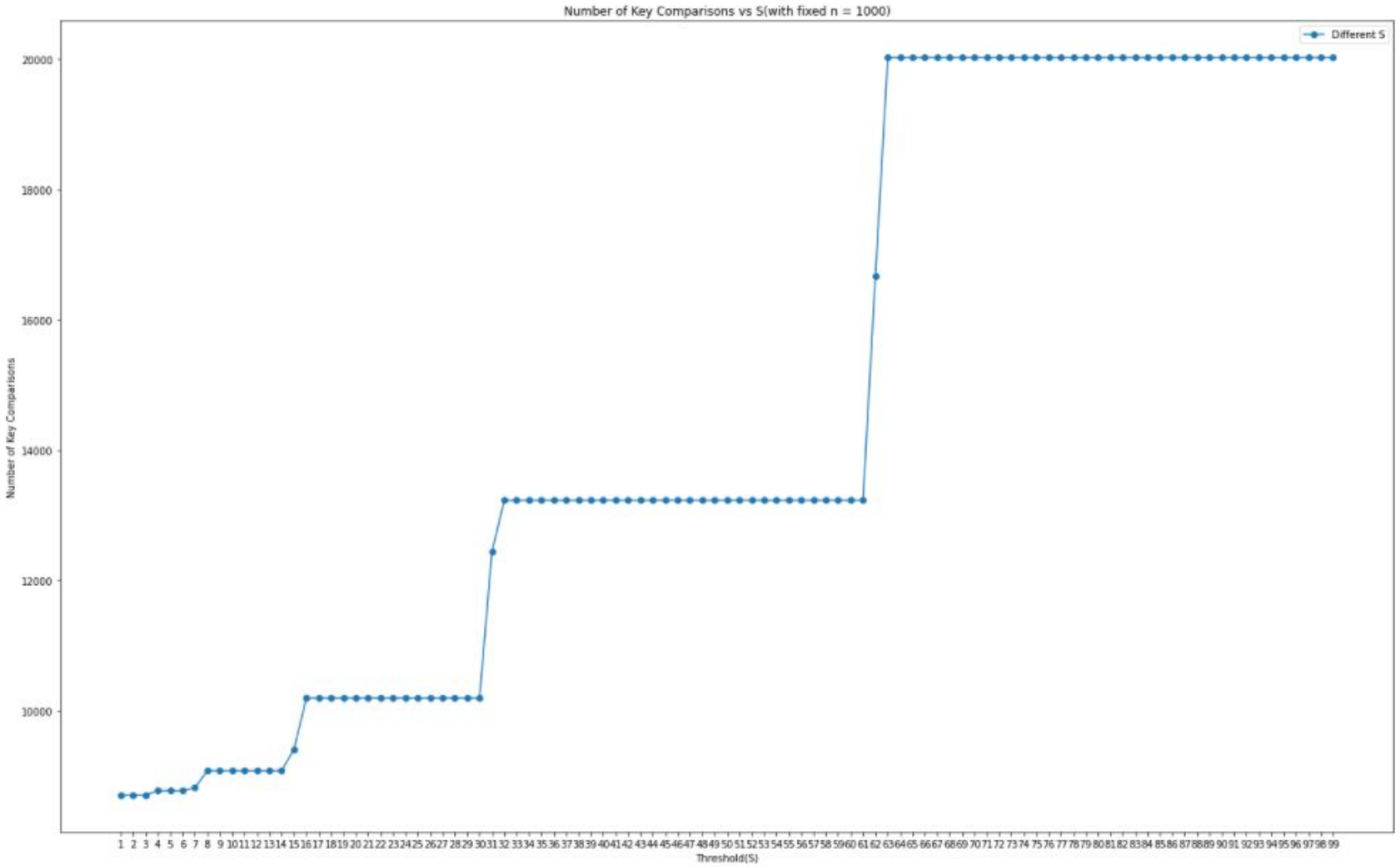
Range for S for any given dataset should be roughly below 100, and leaning towards the smaller side. So we set the range for S to be between [1,25].

Find Best S – Method 1

```
def findBestS (arr_size):
    best_S = [];
    for trial_time in range(50): #to avoid uncertainty, we repeat the procedure for 50 times
        key_cprs = [];
        for subarray_size in range(1,25):
            arr = random.sample(range(1, arr_size+1), arr_size)
            arr, count = hybridSort(arr, subarray_size);
            key_cprs.append(count);
        min_cprs = min(key_cprs)
        min_cprs_size = key_cprs.index(min_cprs) + 1
        best_S.append(min_cprs_size)
    plt.bar(*np.unique(best_S, return_counts=True))
    plt.show()
    df_describe = pd.DataFrame(best_S)
    display(df_describe.describe())
    return max(set(best_S), key = best_S.count) #in the best
```



Best S vs n
Best S
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
Best S for different n
1000
2000
4000
8000
16000
32000
64000
Size of Dataset

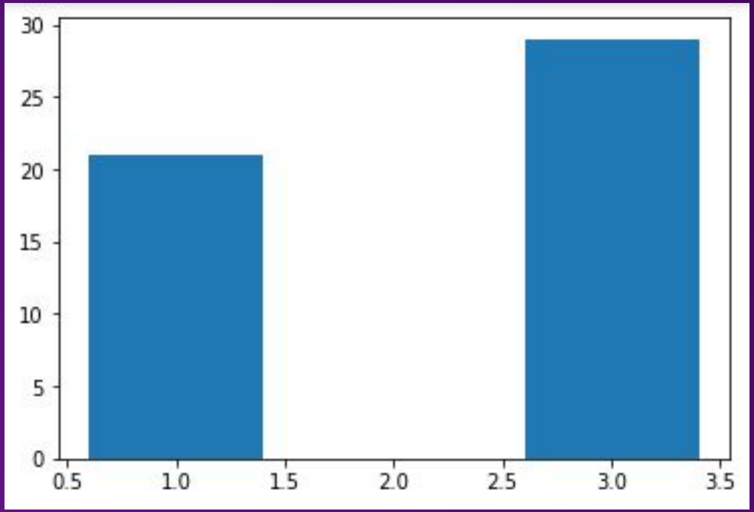


Number of Key Comparisons vs S(with fixed n = 1000)
Different S
Number of Key Comparisons
Threshold(S)
20000
18000
16000
14000
12000
10000

Find Best S – Method 2

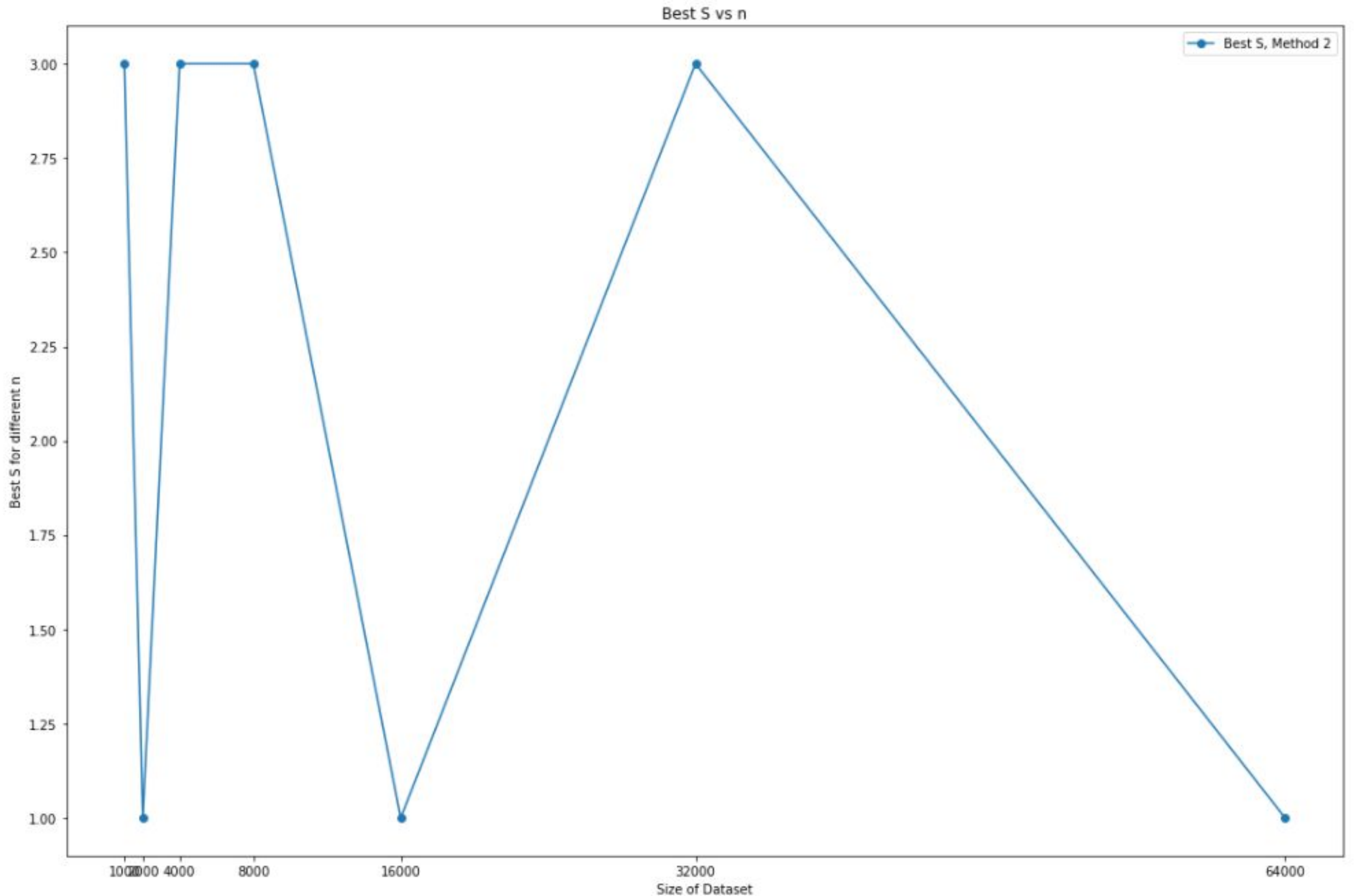
```
def findBestS2 (arr_size):
    best_S2 = [];
    for trial_time in range(50): #to avoid uncertainty, we repeat the procedure for 50 times
        key_cprs2 = [];
        fixed_arr = random.sample(range(1, arr_size+1), arr_size)
        for subarray_size in range(1,25):
            arr = copy.deepcopy(fixed_arr)
            arr, count = hybridSort(arr, subarray_size);
            key_cprs2.append(count);
        min_cprs = min(key_cprs2)
        min_cprs_size = key_cprs2.index(min_cprs) + 1
        best_S2.append(min_cprs_size)
    plt.bar(*np.unique(best_S2, return_counts=True))
    plt.show()
    df_describe2 = pd.DataFrame(best_S2)
    display(df_describe2.describe())
    return max(set(best_S2), key = best_S2.count) #in the best
```

Generating 50 datasets



	0
count	50.000000
mean	2.160000
std	0.997139
min	1.000000
25%	1.000000
50%	3.000000
75%	3.000000
max	3.000000

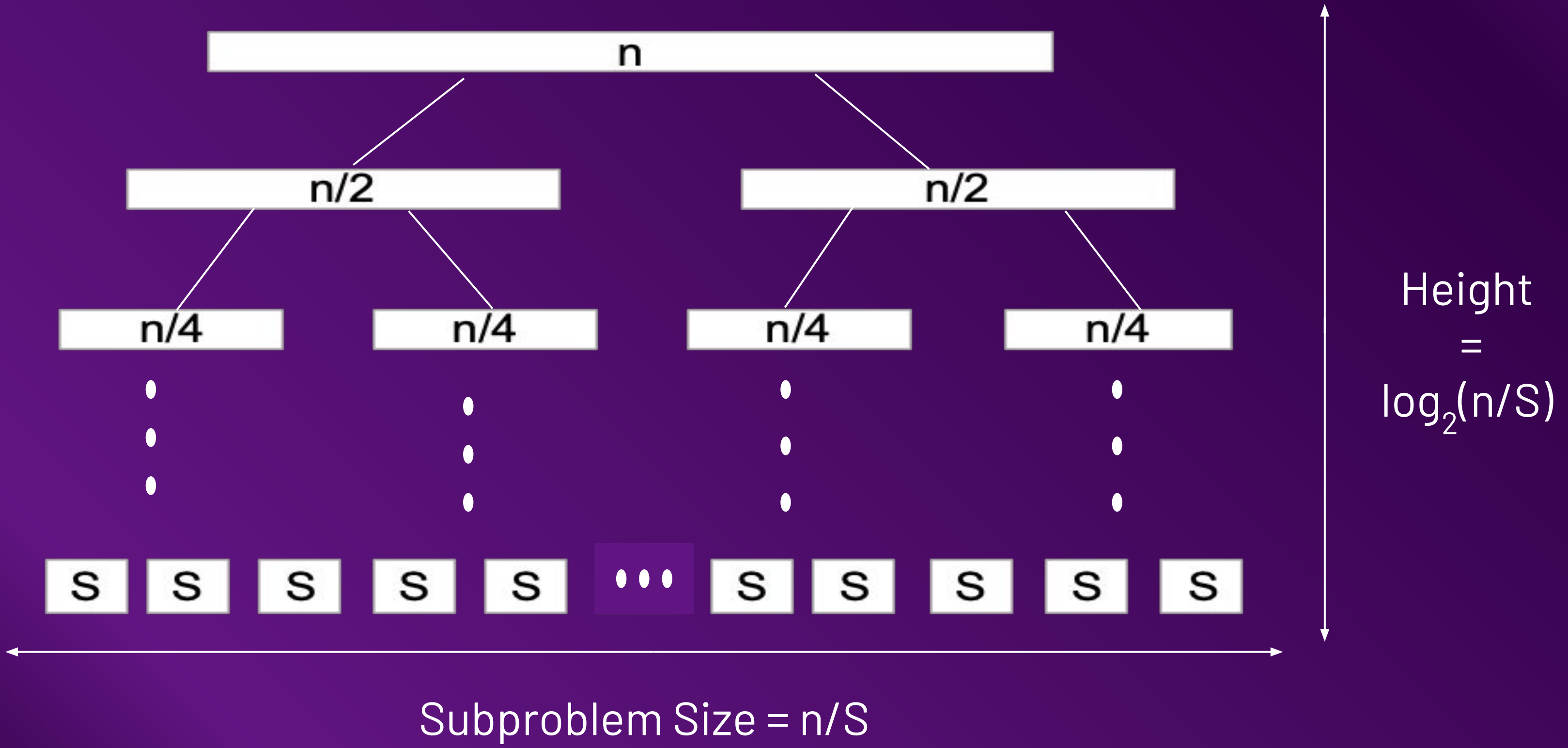
```
arr_size:  1000   best S:  3
```



Best S vs n
Best S, Method 2
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
Best S for different n
1000
2000
4000
8000
16000
32000
64000
Size of Dataset

Performance

Time Complexity



Time Complexity

1. Performs **Recursion** until n/S subarrays → log (n/S)

2. n/S subarrays performs **Insertion Sort**:
   a. **Best Case**: (n/S) * S = n
   b. **Average & Worst Case**: (n/S) * (S^2) = nS

3. Performs **Merge** on all subarrays → n

4. **Hybrid Sort** Time Complexity = **(2) + (3) * (1)**:
   a. **Best Case**: θ (n + n log (n/S))
   b. **Worst Case**: θ (nS + n log (n/S))

mergeSort

```
def mergeSort(arr):
    c = 0
    if len(arr) <= 1:          1
        return arr, c
    l = arr[:len(arr)//2]
    r = arr[len(arr)//2:]
    l, l_c = mergeSort(l)          2
    r, r_c = mergeSort(r)
    arr, c = merge(l, r)          3
    total = l_c + r_c + c
    return arr, total
```

1. Return array and number of comparisons when array length is 0 or 1
2. Recursively partitioning arrays into subarrays until subarrays length are 0 or 1
3. Merges two sub-arrays of elements between index l and middle element and between middle element and index r

1000000 Inputs

	Attempt	hybridSort	mergeSort
Number of Key Comparisons	1	220100740	220101082
	2	220101014	220101183
	3	220101307	220098687
	Approximation	≈ 220100000	≈ 220100000
CPU Time	1	1 min 5.822693 secs	1 min 14.307664 secs
	2	1 min 5.937196 secs	1 min 14.281961 secs
	3	1 min 6.330342 secs	1 min 14.634777 secs
	Approximation	≈ <u>1 min 6 secs</u>	≈ 1 min 14 secs

Conclusion

- **Best S-Value**: 3
  - Recursively divide array until S <= 3
  - Insertion Sort will be performed when S <= 3

- **Performance**
  - **Best Case Time Complexity**: $\theta(n + n \log(n/S))$
  - **Average & Worst Case Time Complexity**: $\theta(nS + n \log(n/S))$
  - Generally performs better than Merge Sort

Thank You!