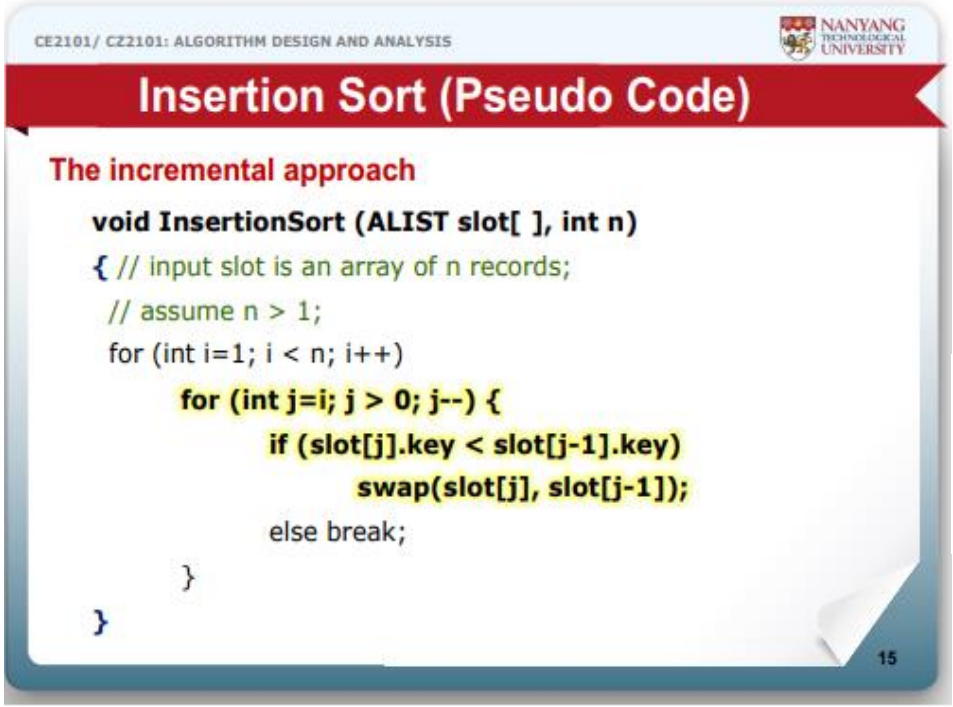
SC2001 Algorithm Design and Analysis Project 1: Integration of Merge Sort & Insertion Sort

Presented by: Zheng Wei, Felicia and Qi Yang

a) Implementation of Hybrid Algorithm

In Mergesort, when the sizes of subarrays are small, the overhead of many recursive calls makes the algorithm inefficient. Therefore, in real use, we often combine Mergesort with Insertion Sort to come up with a hybrid sorting algorithm for better efficiency. The idea is to set a small integer **S** as a threshold for the size of subarrays. Once the size of a subarray in a recursive call of Mergesort is less than or equal to **S**, the algorithm will switch to Insertion Sort, which is efficient for small-sized input.

a) Implementation of Hybrid Algorithm: Insertion Sort Part



- **Time complexity analysis:**
  - Best case: $\Theta(n)$, when input array is already sorted.
  - Worst case: $\Theta(n^2)$, when input array is reversely sorted.
  - Average case: $\Theta(n^2)$.

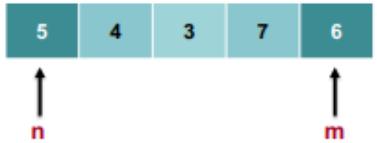
a) Implementation of Hybrid Algorithm: Insertion Sort Part

```java
public static int insertionSort(int arr[],int beg, int end){
    int comp = 0, temp = 0;
    for (int i = beg; i <= end; i++) {
        for (int j = i; j > beg; j--) {
            comp++;
            if (arr[j]<arr[j-1]){
                temp = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = temp;
            }
            else{
                break;
            }
        }
    }
    return comp;
}


}
```

a) Implementation of Hybrid Algorithm: MergeSort Part

Mergesort

```
void mergesort(int n, int m)
{   int mid = (n+m)/2;
    if (m-n <= 0)
        return;
    else if (m-n > 1) {
        mergesort(n, mid);
        mergesort(mid+1, m);
    }
    merge(n, m);
}
```



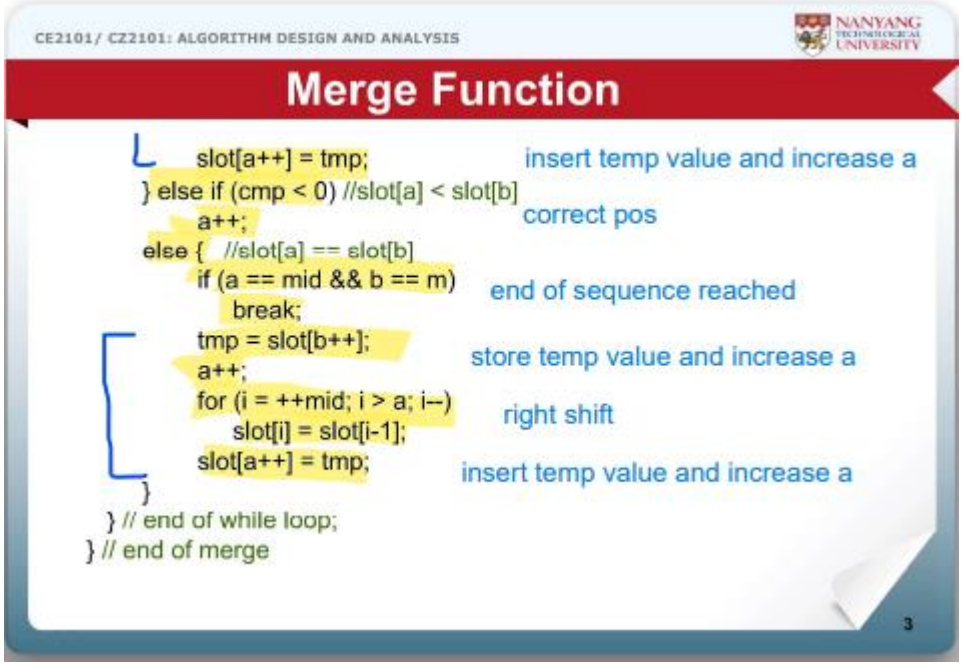
Merge Function

```
void merge(int n, int m)
{
    int mid = (n+m)/2;
    int a = n, b = mid+1, i, tmp;
    if (m-n <= 0) return;
    while (a <= mid && b <= m) {
        cmp = compare(slot[a], slot[b]);
        if (cmp > 0) { //slot[a] > slot[b]
            tmp = slot[b++];
            for (i = ++mid; i > a; i--)
                slot[i] = slot[i-1];
```

store temp value and increase b

right shift

a) Implementation of Hybrid Algorithm: MergeSort Part



Average Time Complexity [Big-theta]:
O(n*log n)

Space Complexity: **O(n)**

- Time complexity of Merge Sort is `O(n*Log n)` in all the 3 cases (worst, average and best) as merge sort always **divides** the array in two halves and takes linear time to **merge** two halves.

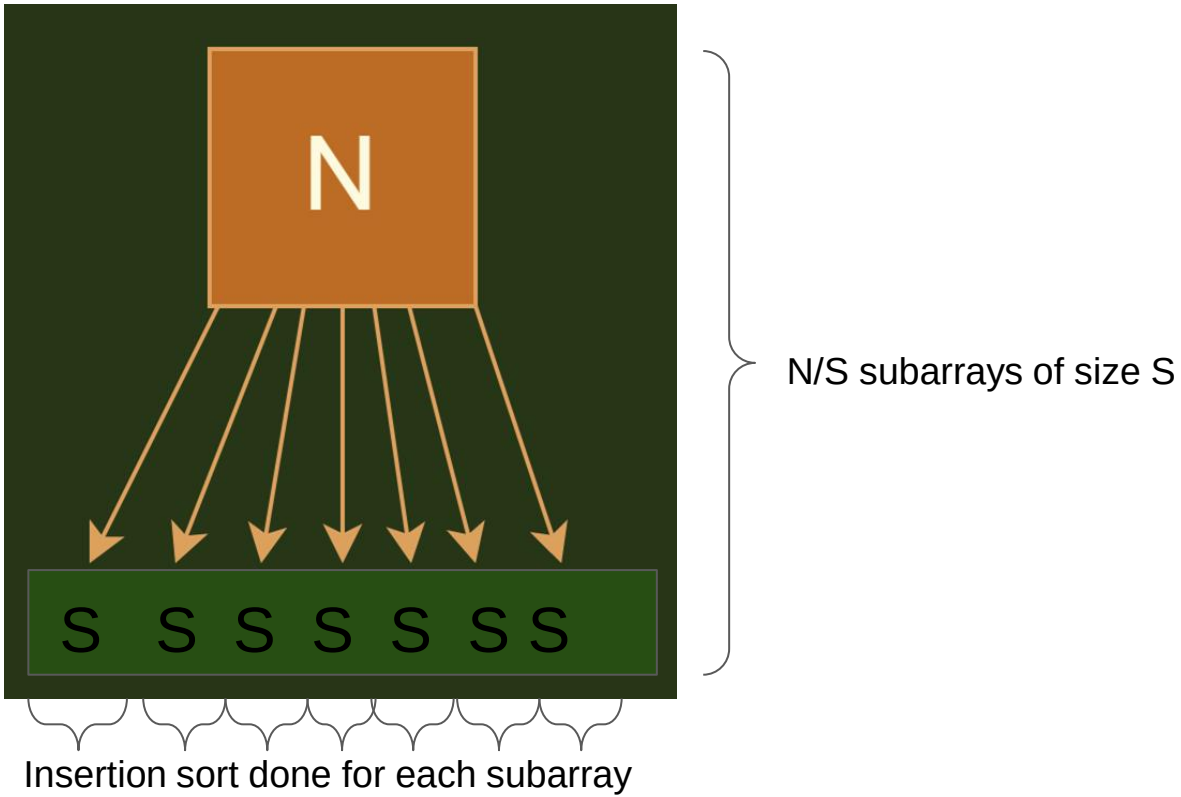
a) Implementation of Hybrid Algorithm: MergeSort Part

```java
public static int merge(int arr[],int beg,int end,int mid){
    int comCount = 0;
    int leftArrSize = mid - beg + 1;  //start to mid
    int rightArrSize = end - mid;     //mid+1 to end

    int leftArr[] = new int[leftArrSize];
    int rightArr[] = new int[rightArrSize];

    for (int i = 0; i < leftArrSize; i++) {
        leftArr[i] = arr[beg+ i];
    }

    for (int j = 0; j < rightArrSize; j++) {
        rightArr[j] = arr[mid + 1+ j];
    }
    int i=0,j=0;
    int k=beg;
    while(i< leftArrSize && j< rightArrSize) {
        comCount++;
        if (leftArr[i] == rightArr[j]){
            arr[k++] = leftArr[i++];
            arr[k++] = rightArr[j++];
        }
        else if(leftArr[i] < rightArr[j]){
            arr[k++] = leftArr[i++];
        }
        else{
            arr[k++] = rightArr[j++];
        }
    }
    while(i < leftArrSize){
        arr[k++] = leftArr[i++];
    }
    while(j < rightArrSize){
        arr[k++] = rightArr[j++];
    }
    return comCount;
}
```

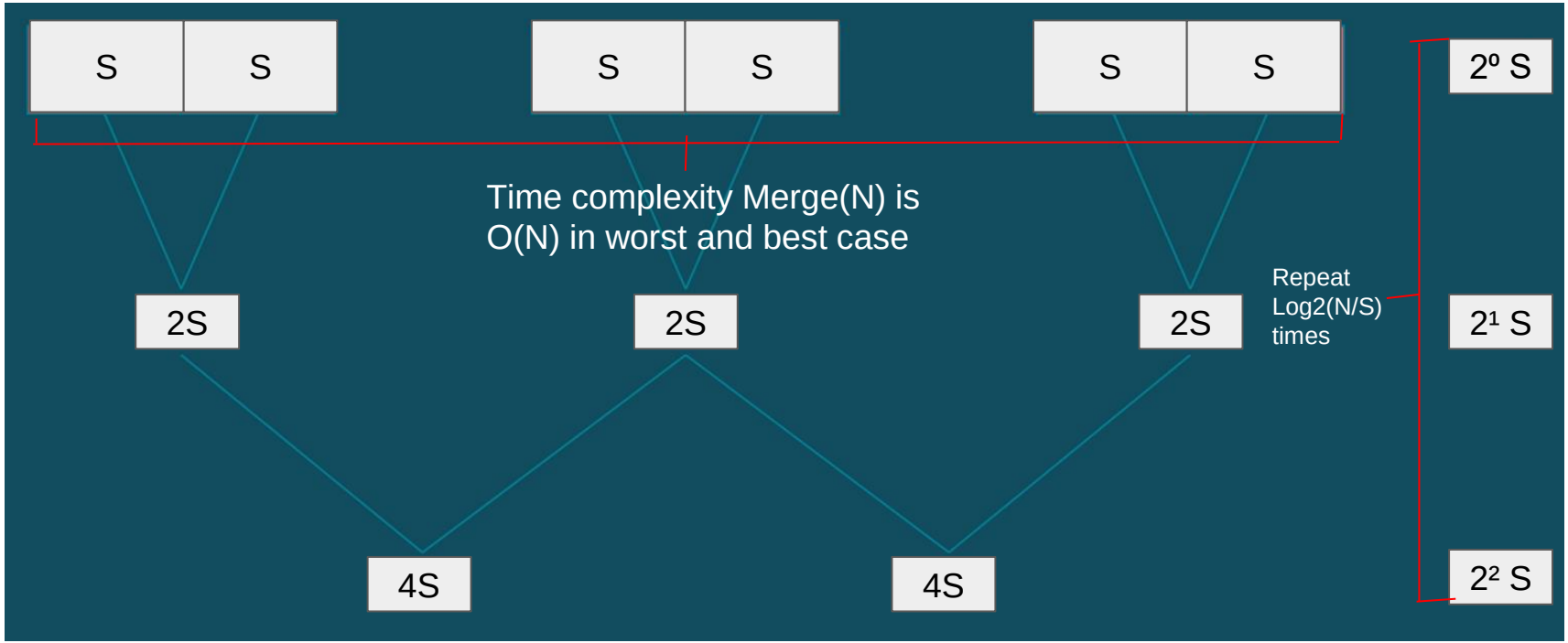
a) Implementation of Hybrid Algorithm

```
public static int hybridSort(int arr[],int beg, int end,int S){
    int mid = (beg+end)/2;
    if((end-beg) <= S){
        return insertionSort(arr,beg,end);
    }
    else{
        return hybridSort(arr,beg,mid,S) + hybridSort(arr,mid+1, end,S)+
        merge(arr, beg, end, mid);
    }
}
```

a) Implementation of Hybrid Algorithm: Division



a) Implementation of Hybrid Algorithm: Merging



Time Complexity of Hybrid Sort

Best Case: $O((n/S)(S) + n * \log(n/S)) = O(n + n \log(n/S))$

Worst Case: $O((n/S)(S^2) + n * \log(n/S)) = O(nS + n \log(n/S))$

Average Case: $O((n/S)(S^2) + n * \log(n/S)) = O(nS + n \log(n/S))$

b) Generate Input Data

```java
public static int[] generateInput(int n) {
    Random rand = new Random();
    int arr[] = new int[n];
    for(int i=0;i< arr.length;i++){
        arr[i] = rand.nextInt(n);
    }
    return arr;
}
```

c) Time complexity analysis

insertionSort():

```java
public static int insertionSort(int arr[],int beg, int end){
    int comp = 0, temp = 0;
    for (int i = beg; i <= end; i++) {
        for (int j = i; j > beg; j--) {
            comp++;
            if (arr[j]<arr[j-1]){
                temp = arr[j];
                arr[j] = arr[j-1];
                arr[j-1] = temp;
            }
            else{
                break;
            }
        }
    }
    return comp;
}
```

merge():

```java
while(i< leftArrSize && j< rightArrSize) {
    comCount++;
    if (leftArr[i] == rightArr[j]){
        arr[k++] = leftArr[i++];
        arr[k++] = rightArr[j++];
    }
    else if(leftArr[i] < rightArr[j]){
        arr[k++] = leftArr[i++];
    }
    else{
        arr[k++] = rightArr[j++];
    }
}
while(i < leftArrSize){
    arr[k++] = leftArr[i++];
}
while(j < rightArrSize){
    arr[k++] = rightArr[j++];
}
return comCount;
```

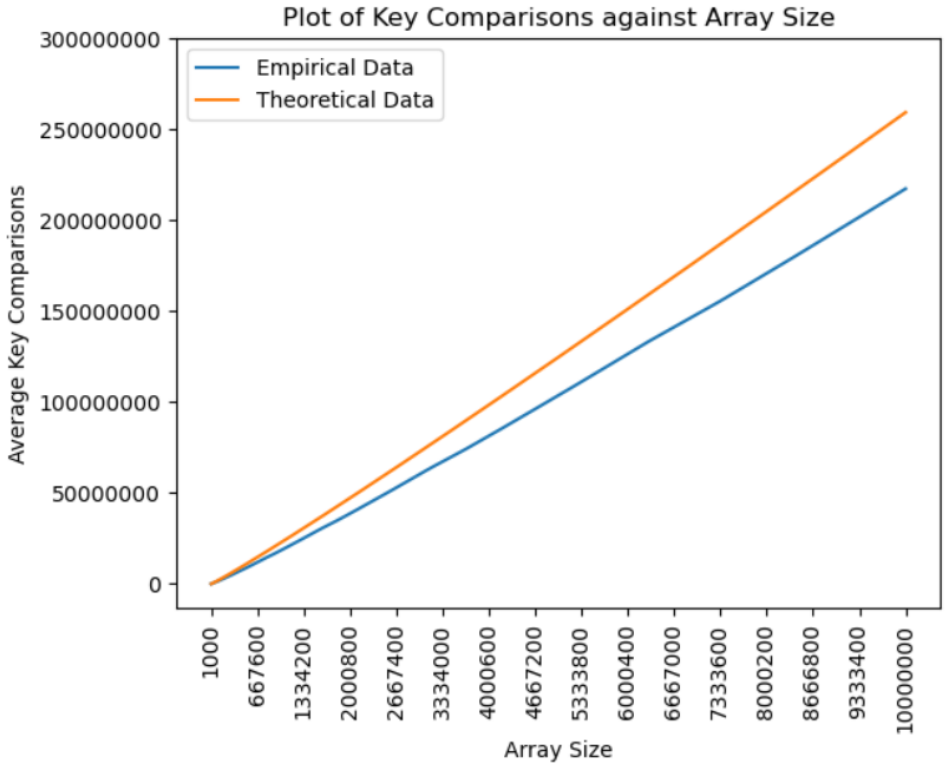
c) Time complexity analysis

Best Case:	$O(n + n \log(n/S))$
Worst Case:	$O(nS + n \log(n/S))$
Average Case:	$O(nS + n \log(n/S))$

ci) Time complexity analysis: Key Comparisons vs Input size

```
df=pd.read_csv("Proj1\ci\ci.csv")
df['Average Key Comparisons'] = df['Average Key Comparisons'].astype(int)

tsize = np.array(range(1000,10000000))
tkeycomp = tsize*5 + tsize*(np.log2(tsize/5))

fig, ax = plt.subplots()
ax.ticklabel_format(useOffset=False, style='plain')
plt.plot(df["Array Size"].values.tolist(), df["Average Key Comparisons"].values.tolist(), label = 'Empirical Data')
plt.plot(tsize, tkeycomp, label = 'Theoretical Data')

plt.title('Plot of Key Comparisons against Array Size')
plt.xlabel('Array Size')
plt.ylabel('Average Key Comparisons')
plt.xticks(range(min(df["Array Size"].values.tolist()), max(df["Array Size"].values.tolist()) + 1, 666600),rotation='vertical')
plt.yticks(range(0, 350000000,50000000 ))


plt.legend()
plt.show()
```

ci) Time complexity analysis: Key Comparisons vs Input size

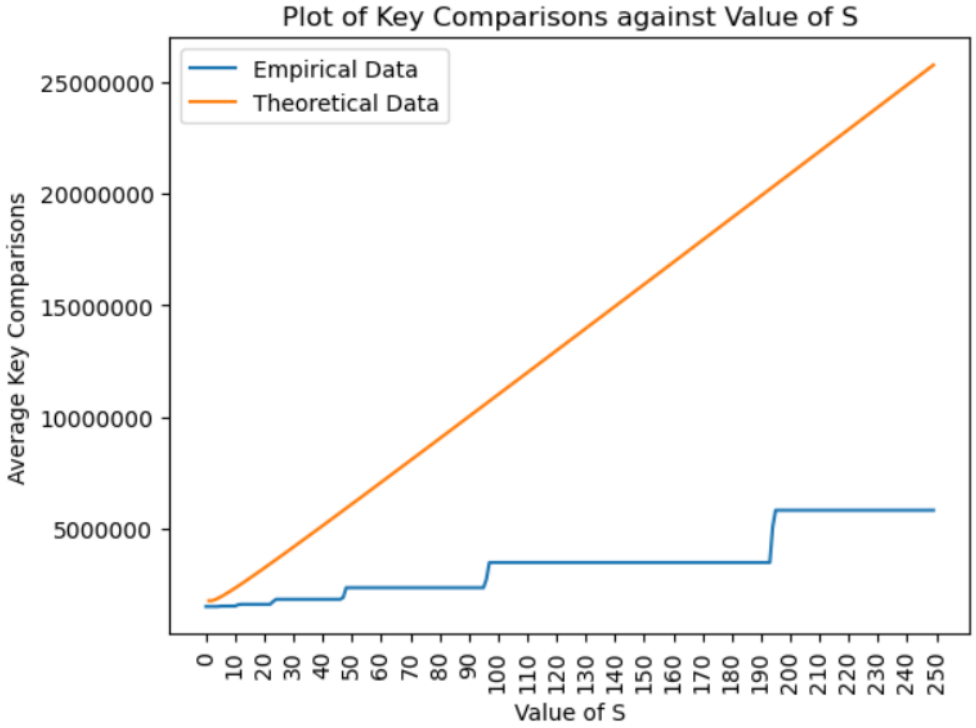


- S=5
- Input range: 1000 to 10,000,000
- Each point on the graph is the average of 100 random tests
- Conclusion
  - As S increase, average key comparisons increase
  - Graphs grow at relatively same rate

cii) Time complexity analysis: Key Comparisons vs S

```
df=pd.read_csv("Proj1\cii\cii.csv")
df['Average Key Comparisons'] = df['Average Key Comparisons'].astype(int)

s = np.array(range(1,250))
tkeycomp = 100000*s + 100000*(np.log2(100000/s))

fig, ax = plt.subplots()
ax.ticklabel_format(useOffset=False, style='plain')

plt.plot(df["Value of S"].values.tolist(), df["Average Key Comparisons"].values.tolist(), label = 'Empirical Data')
plt.plot(s, tkeycomp, label = 'Theoretical Data')

plt.title('Plot of Key Comparisons against Value of S')
plt.xlabel('Value of S')
plt.ylabel('Average Key Comparisons')
plt.xticks(range(min(df["Value of S"].values.tolist()), 260,10),rotation='vertical')  # Change the step value as needed

plt.legend()
plt.show()
```

cii) Time complexity analysis: Key Comparisons vs S



- n = 100,000
- S range: 1 to 250
- Each point on the graph is the average of 100 random tests
- Conclusion
  - As S increase, average key comparisons increase
  - Graphs grow at different rate

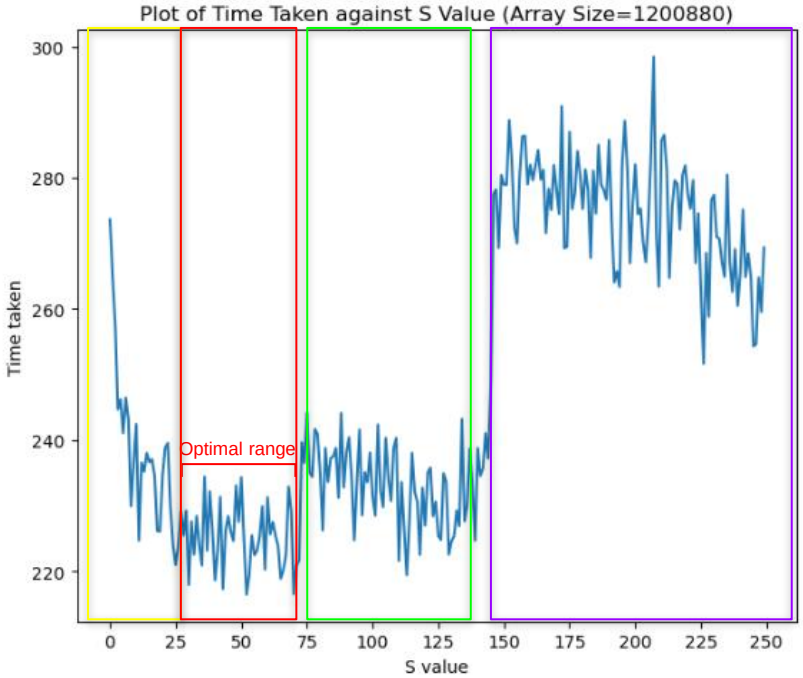
c) (iii) Finding the Optimal Value of S

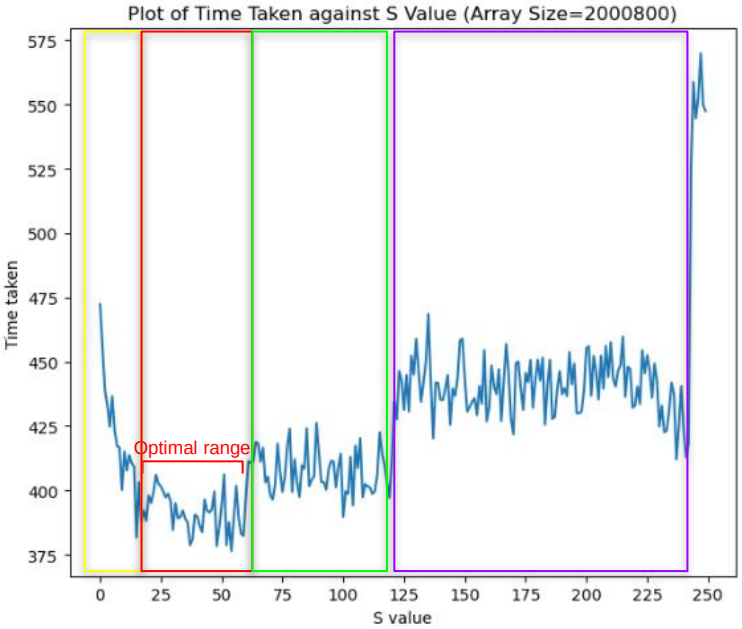
- Our Optimal Range of S is **S[30,55]**
- Collect data on arraySizes(i) from 1000 to 1,000,000 at increments of 399960

```
for(i=1000;i<=1,000,000;i+=399960)//iterates 26 times
{
        //collect data
}
```

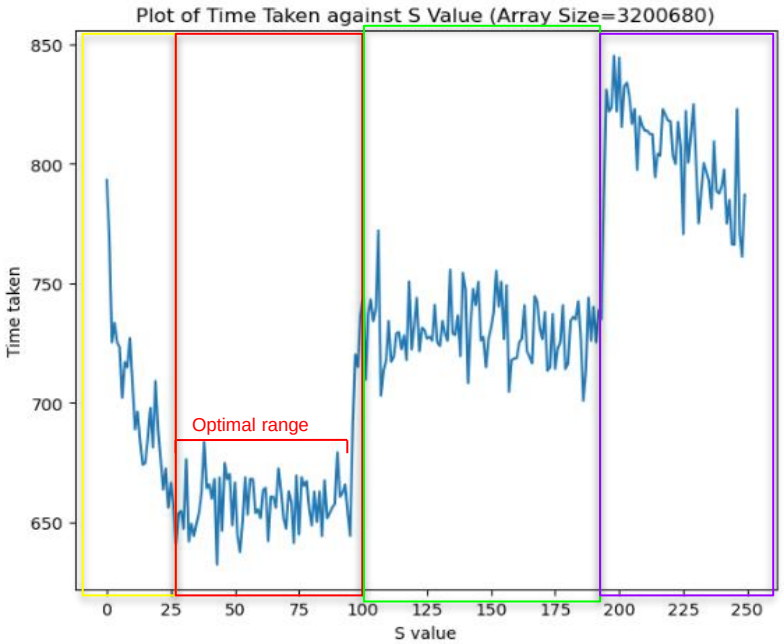
- We test performance of hybridSort for range S[0,250]
- Every data point is the average result of 20 random tests
- Try to identify a range of S values that would consistently perform optimally across all our array sizes

Example graph: Optimal Range S[25,75]





Optimal Range: [13,60]

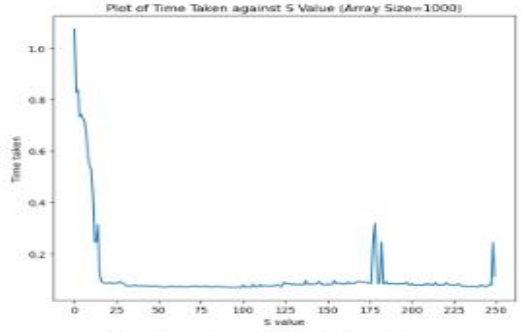


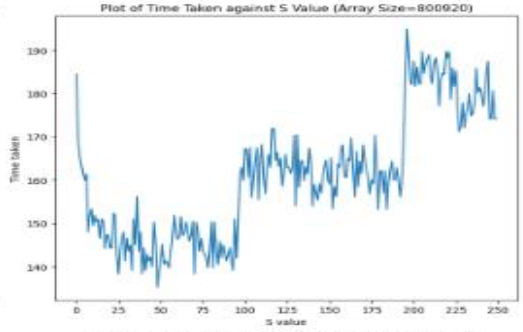
Optimal Range: [25,95]

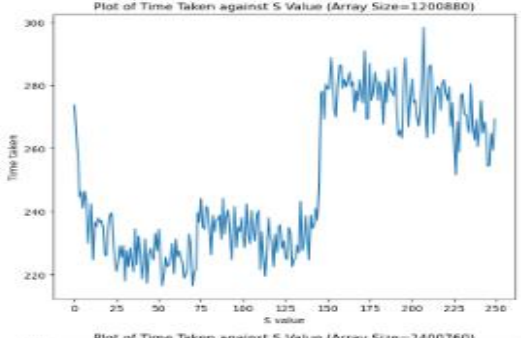
In this example if we pick a value of S from [25,60], then hybridsort would have optimal performance in both array sizes.
This range is essentially [Highest LowerBound, Lowest UpperBound]

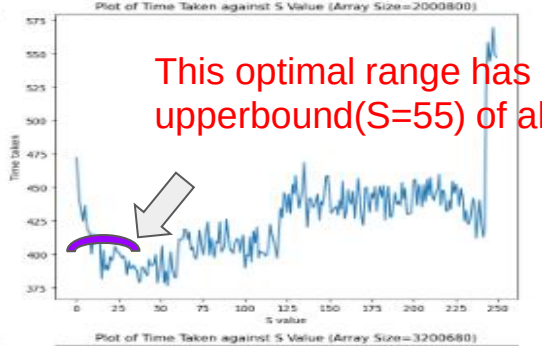
We repeat this process for all 26 array sizes and find that the Optimal range is S[30,55].

Array sizes 1000 to 3,200,680

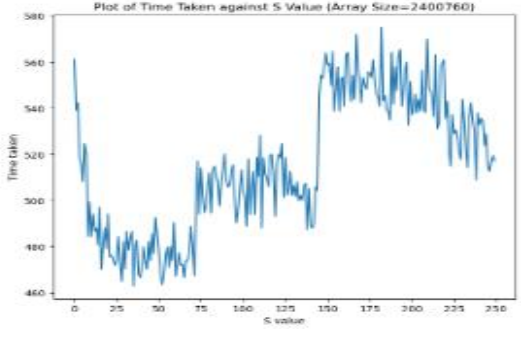


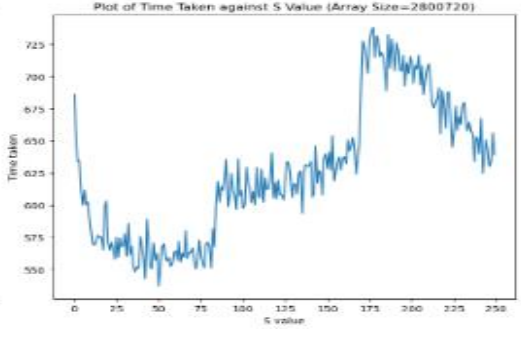


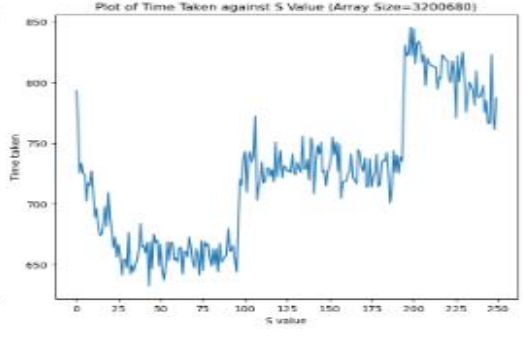




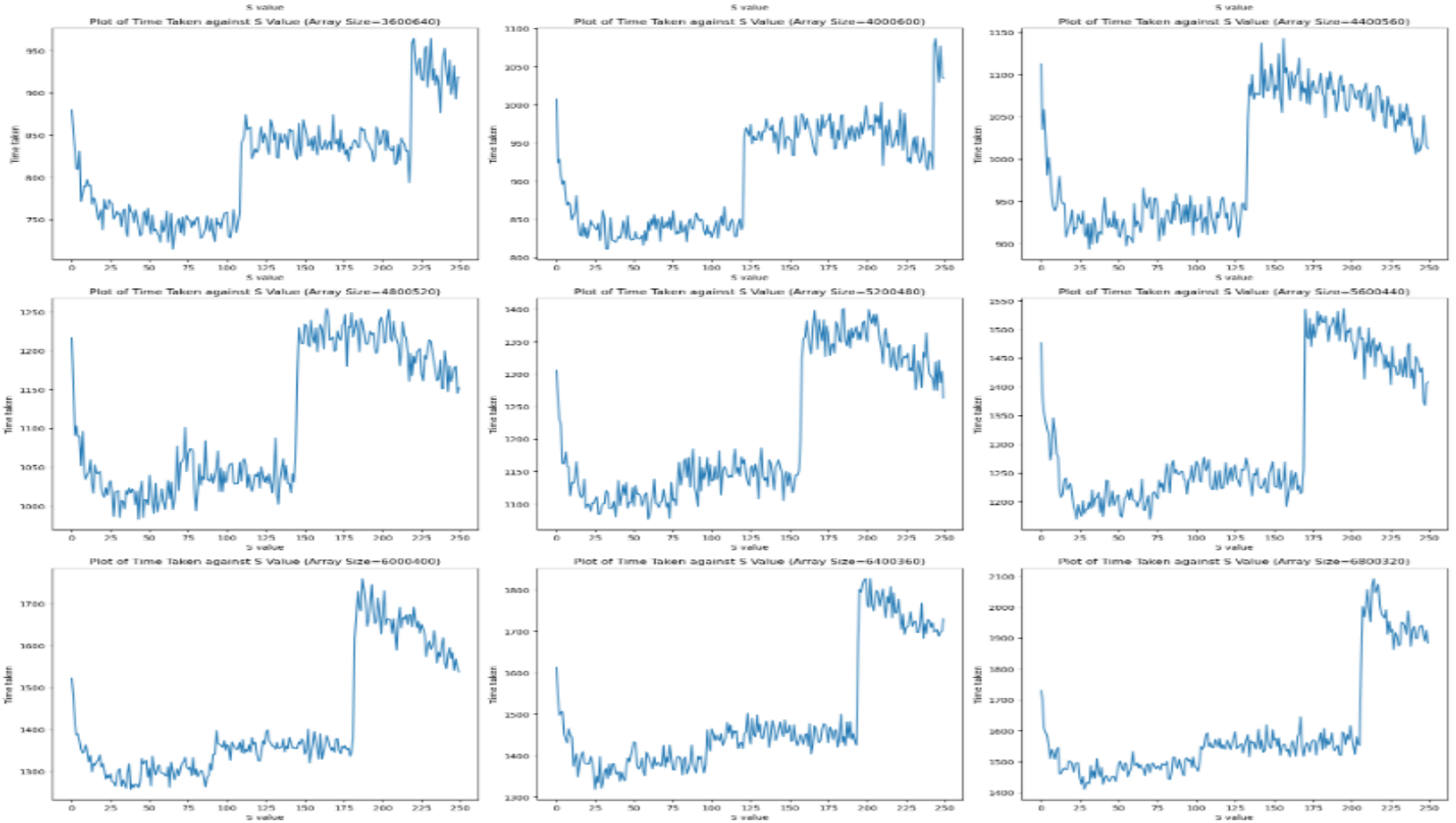
This optimal range has lowest upperbound(S=55) of all graphs



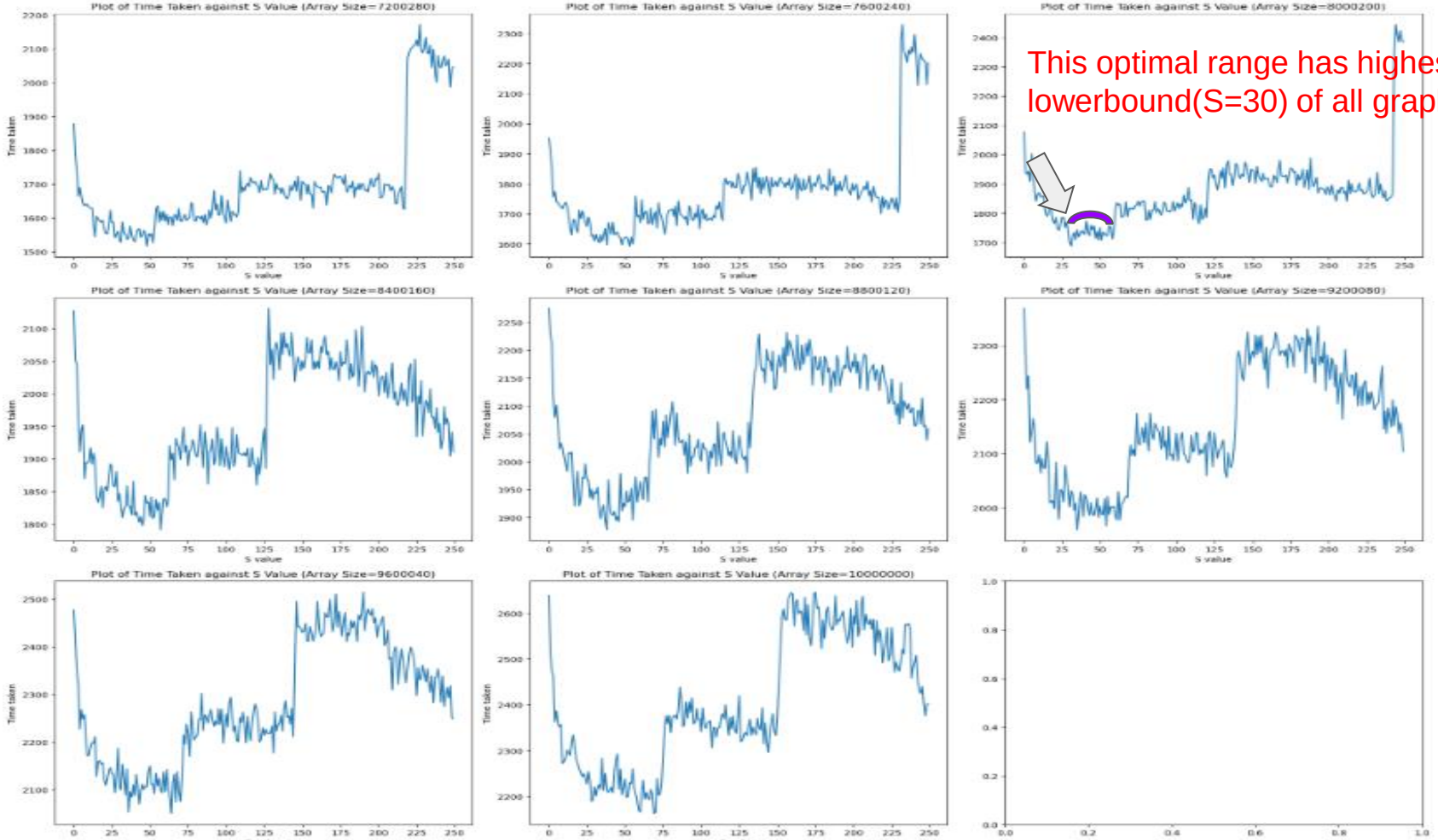




Array sizes 3,600,640 to 6,800,320



Array sizes 7,200,280 to 10,000,000



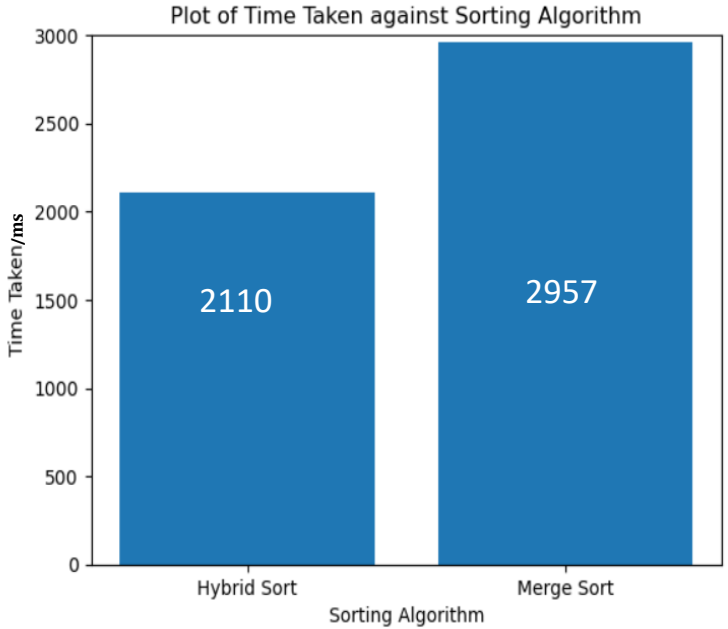
This optimal range has highest lowerbound(S=30) of all graphs

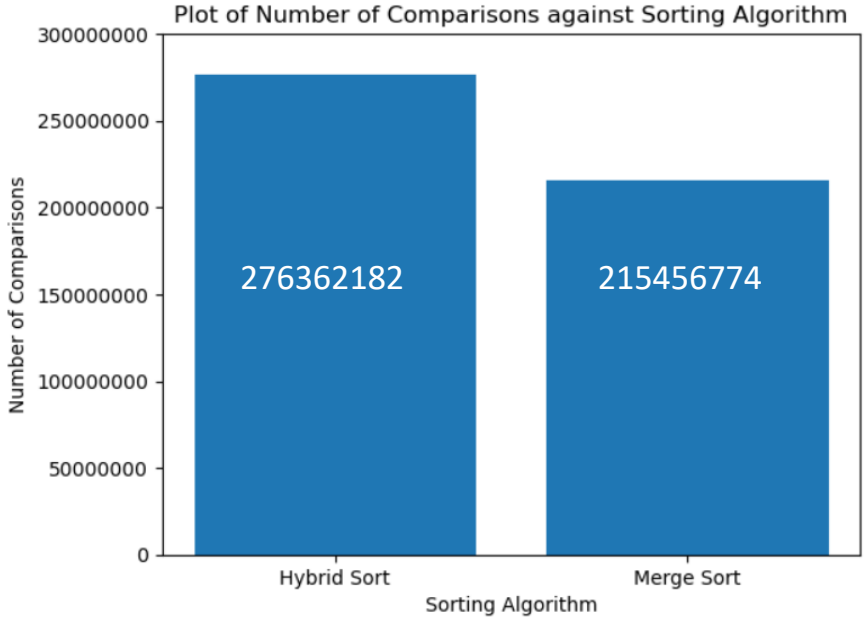
d) Comparing Optimal HybridSort with MergeSort

- We choose S=48. This value is within our optimal range of S[30,55].
- Data is the average collected from 500 random tests

Results:

- Hybrid sort has approximately 28.2% more key comparisons than Merge Sort (60905408)
- Despite this, Hybrid sort is still significantly faster than Merge sort, at approximately 28.6% faster (847 milliseconds)





Thank you! :D