SC2001 Example Class 1

An Integration of Merge Sort and Insertion Sort

Group 6
Tan Jin Wei Daniel
Chan Ming Han

Hybrid Sort Implementation

Mergesort

- Takes an array, an upper bound and a lower bound
- Recursively calls itself for the front half and back half of the array
- Merges the results together

Insertionsort

- Takes an array, an upper bound and a lower bound
- Compares each element with preceding elements until there is no swap

```
static void mergeSort(int arr[], int upper, int lower) {
    if(lower < upper) {
        int mid = lower + (upper - lower)/2;
        mergeSort(arr,mid,lower);
        mergeSort(arr,upper,mid+1);
        merge(arr,upper,lower,mid);
        mergecomps += compcounter;
    }
}
static void insertionSort(int arr[], int upper, int lower) {

    int temp = 0;
    for(int i = lower + 1; i <= upper; i++) {
        temp = arr[i];
        int j = i-1;
        while (j >= lower && arr[j] > temp) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = temp;
    }
}
```

Hybrid Sort Implementation

Merge

- Takes an array, an upper bound, a lower bound and middle number
- Instantiates 2 new arrays half the size of the array passed in
- Fills the 2 new arrays with the front half and second half of the array passed in
- Iterates through the 2 new arrays simultaneously, comparing each element and filling the original array with the lower element
- Copies any remaining elements of the new arrays to the original array
- Runs faster than the merge in place algorithm as there is no shifting of elements in the arrays

```
static void merge(int arr[], int upper, int lower, int mid) {
    int s1 = mid - lower + 1;
    int s2 = upper - mid;

    int arr1[] = new int[s1];
    int arr2[] = new int[s2];

    for(int i = 0; i < s1; i++) {
        arr1[i] = arr[lower + i];
    }
    for(int i = 0; i < s2; i++) {
        arr2[i] = arr[mid + 1 + i];
    }

    int m = 0, n = 0;
    int i = lower;
    while(m < s1 && n < s2) {

        if(arr1[m] <= arr2[n]) {
            arr[i] = arr1[m];
            m++;
        }
        else {
            arr[i] = arr2[n];
            n++;
        }
        i++;
    }

    while(m < s1) {
        arr[i] = arr1[m];
        i++;
        m++;
    }
    while(n < s2) {
        arr[i] = arr2[n];
        i++;
        n++;
    }
}
```

Hybrid Sort Implementation

Hybridsort

- Takes an array, an upper bound and a lower bound
- If the size of the array is smaller than a threshold, call insertionSort on the array
- If the size of the array is bigger than the threshold, recursively call itself for the front and back half of the array, then merges the results together

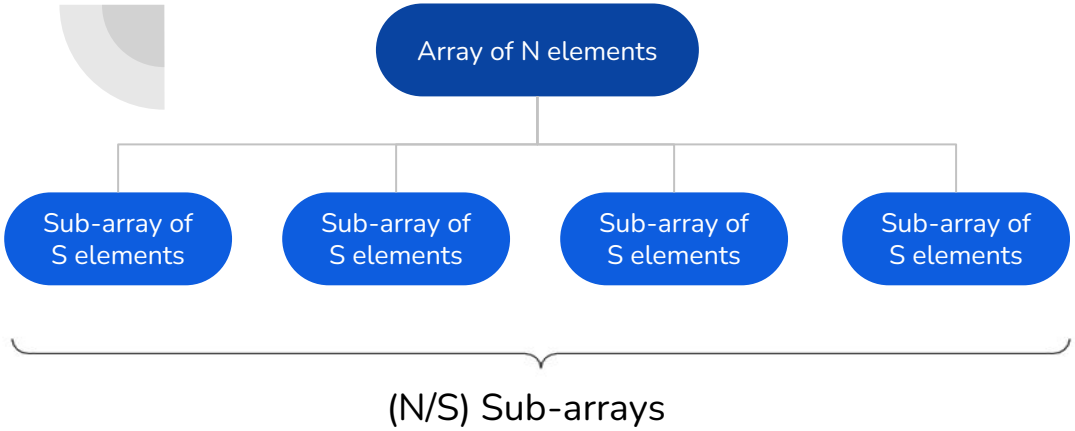
```
static void hybridSort(int arr[], int upper, int lower) {
    if (upper - lower <= THRESHOLD) {
        insertionSort(arr, upper, lower);
    }
    else {

        int mid = lower + (upper - lower)/2;
        hybridSort(arr,mid,lower);
        hybridSort(arr,upper,mid+1);
        merge(arr,upper,lower,mid);

    }
}
```

Generation of Test Arrays

- Arrays generated using the rand function in C++, up to a maximum value of INT64_MAX, which is 92233720368547758O7LL (according to documentation)
- Arrays were then copied using a simple for loop. This makes sure that the arrays used to test the sorting algorithms were identical.
- srand(time(0)); called to make sure that the array of random numbers generated are different with each C++ run.
- Sizes of the arrays generated are then controlled by the for loop. In this case, from 1,000,000 to 10,000,000, with a step of 1,000,000

```
// GENERATE RANDOM NUMBERS
int* arrGenerator(int n){
    int *randArr = new int[n];
    for (int i=0; i<n; i++){
        randArr[i] = rand()%INT64_MAX;
    }
    return randArr;
}


// DRIVER CODE


int main(){
    srand(time(0)); // create a time seed so that the array of random numbers generated is different with each c++ run
    ofstream fout("hybridSortComparison_10kto110k3.csv");

    fout<<"n,Merge Sort, Hybrid Sort\n";



    for (int i=1000000; i<=10000000; i+=1000000){
    // copying the arrays twice, and then running these 3 arrays through insertion sort, merge sort, hybrid sort respectively
        int *arrI = arrGenerator(i);
        int *arrM = new int[i];
        int *arrH = new int[i];
        for (int s=0; s<i; s++){
            arrM[s] = arrI[s];
            arrH[s] = arrI[s];
        }
```

Theoretical Analysis of Time Complexity

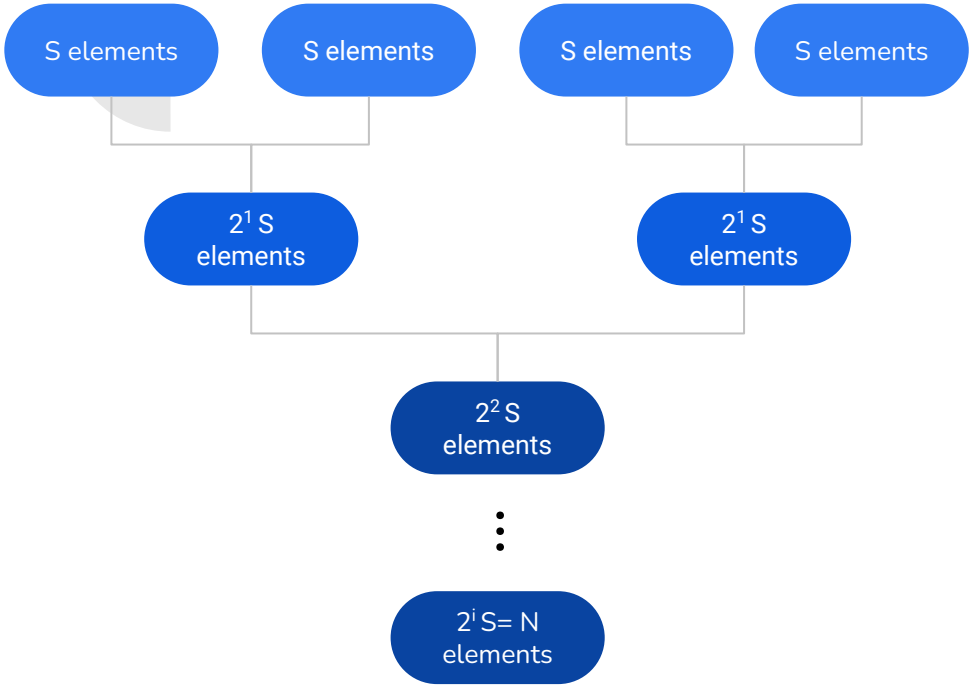


The array of N elements is split into (N/S) Sub-arrays of S elements each, on which insertion sort is called.

(Note: S is the threshold)

Insertion Sorting	Best Case	Worst Case
Sub-array of K elements	O(S)	$O(S^2)$
Array of N elements	O(S*(N/K)) = O(N)	$O(S^2*(N/S))$ = O(SN)

Theoretical Analysis of Time Complexity



The sub-arrays of K elements are now sorted and must be merged back

Every merge iteration doubles the number of elements in the array, until it reaches N, i.e. $2^i S = N$

Hence the number of merge iterations, i, is $i = \log_2(N/S)$

Since the cost for each iteration of merging is O(N), the total cost is $O(N \log_2(N/S))$

Theoretical Analysis of Time Complexity

Best Case	Insertion: O(N) Cost of merging: O(N $\log_2$(N/S)) Hybrid: N + N $\log_2$(N/S)
Worst Case	Insertion: O(SN) Cost of merging: O(N $\log_2$(N/S)) Hybrid: SN + N $\log_2$(N/S)

Now we just need to combine the two to get the time complexity of our hybrid sorting algorithm

Theoretical Comparison with MergeSort

	Small S (S approaches 1)	Large S (S approaches N)
HybridSort Best Case	Approaches $N + N \log_2(N)$	Approaches $N + N \log_2(1) = N$
HybridSort Worst Case	Approaches $N + N \log_2(N)$	Approaches $N^2 + N \log_2(1) = N^2$

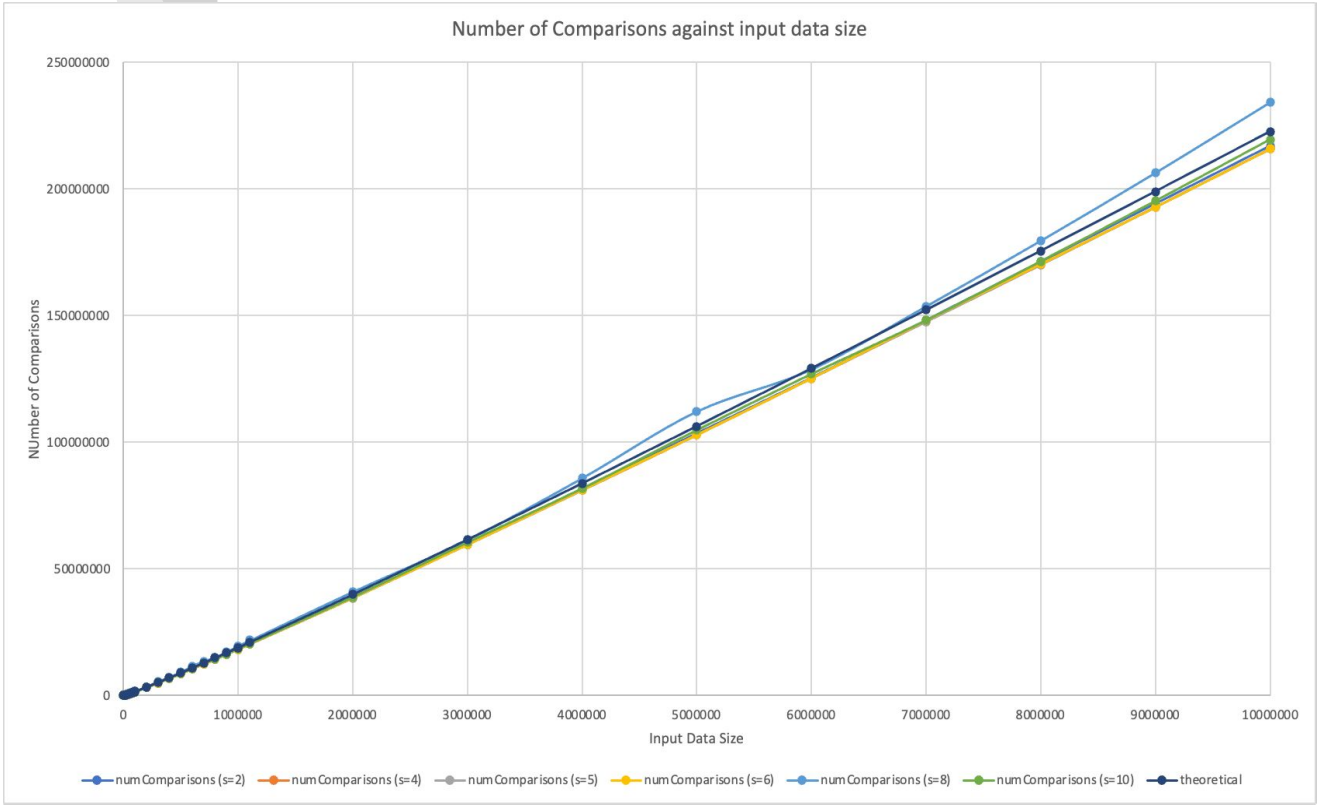
Conclusion: With small S, HybridSort approaches MergeSort's time complexity. With large S, HybridSort approaches InsertionSort time complexity.

Theoretical Comparison with MergeSort

Merge vs Hybrid best case	Merge slower than Hybrid	Merge Faster than Hybrid
	$N \log_2(N) > N + N \log_2(N/K)$ $\log_2(N) > 1 + \log_2(N) - \log_2(K)$ $\log_2(K) > 1$ $K > 2$	$N \log_2(N) < N + N \log_2(N/K)$ $\log_2(N) < 1 + \log_2(N) - \log_2(K)$ $\log_2(K) < 1$ $K < 2$
Merge vs Hybrid worst case	$N \log_2(N) > KN + N \log_2(N/K)$ $\log_2(N) > K + \log_2(N) - \log_2(K)$ $\log_2(K) > K$ No real solutions for K	$N \log_2(N) < KN + N \log_2(N/K)$ $\log_2(N) < K + \log_2(N) - \log_2(K)$ $\log_2(K) < K$ $K \geq 0$

Conclusion: MergeSort is always faster than HybridSort's worst case, but is slower than HybridSort's best case when threshold K is greater than 2 and faster than HybridSort's best case when threshold K is less than 2.

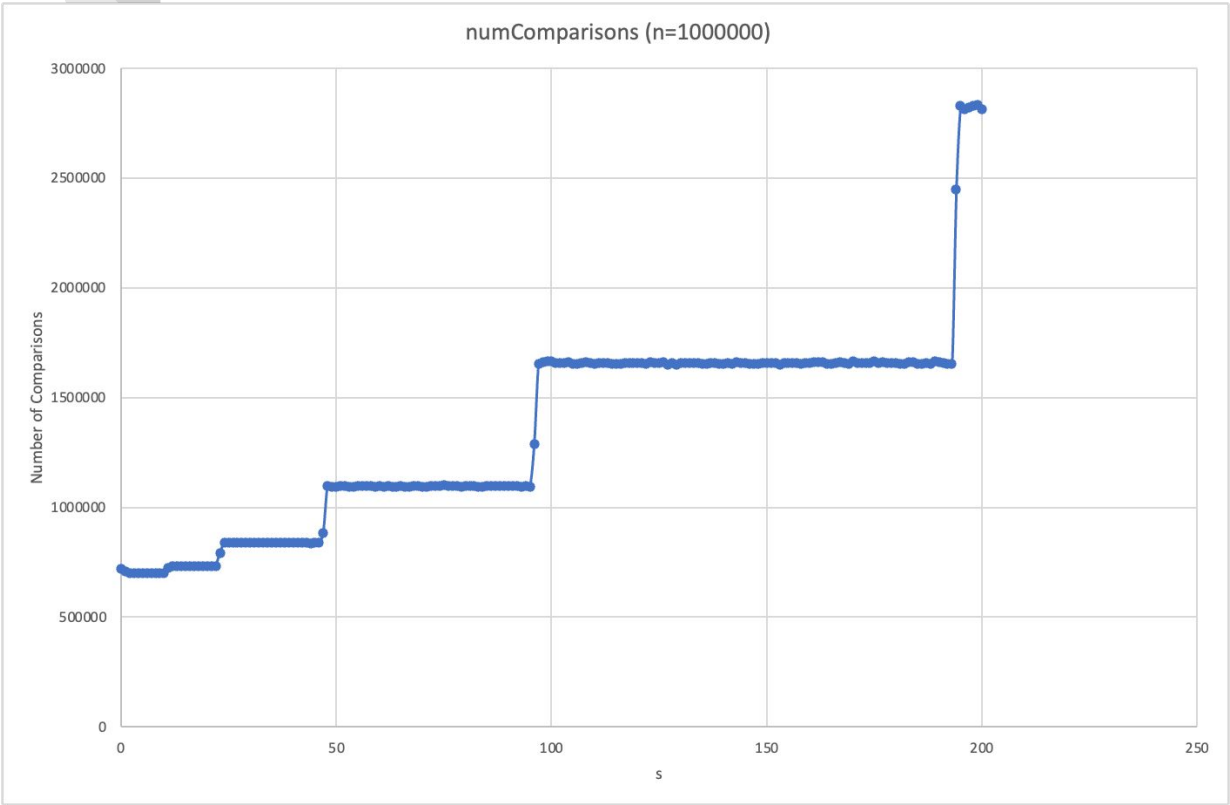
Empirical Analysis of Time Complexity



- Agrees with theoretical data, where O(nlgn).
- Theoretical graphed by the black line, which is the equation of the worst case of merge sort

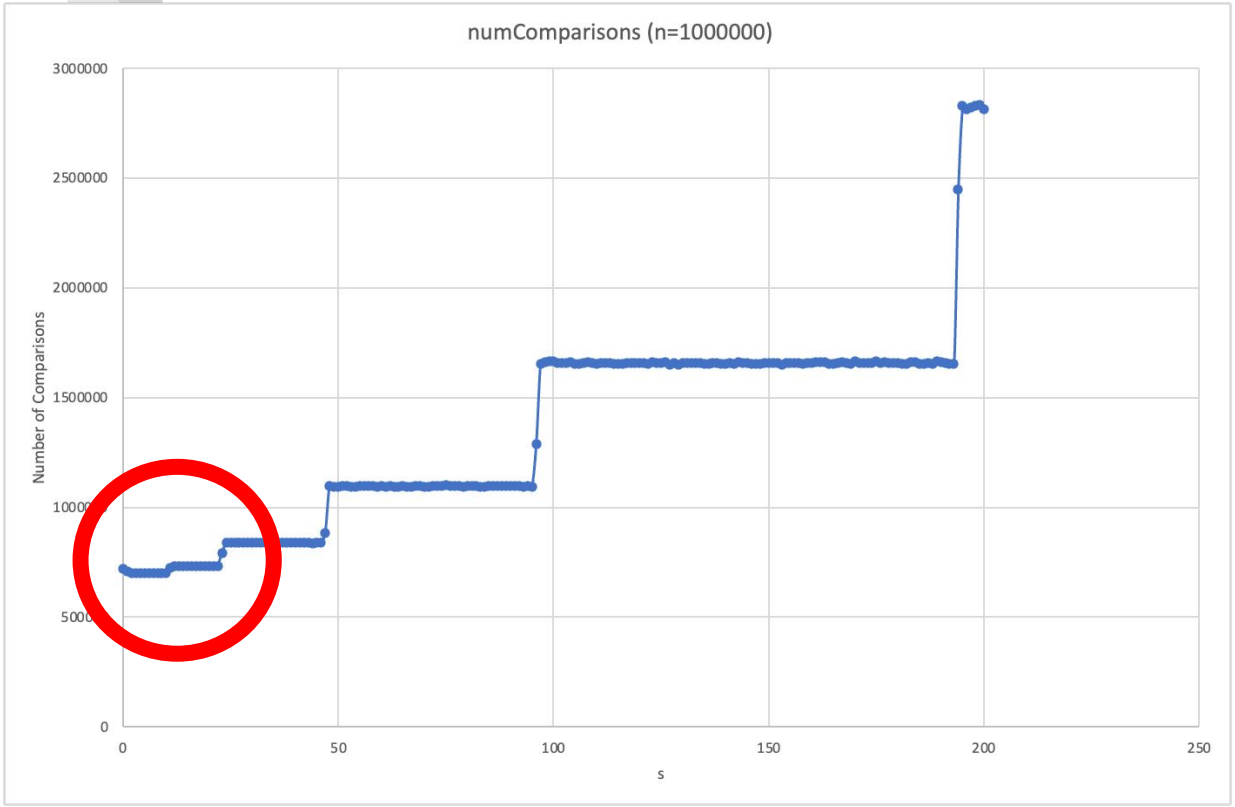
$$n * log_2 n - n + 1$$

Determining the Optimal Value of S, n=1,000,000



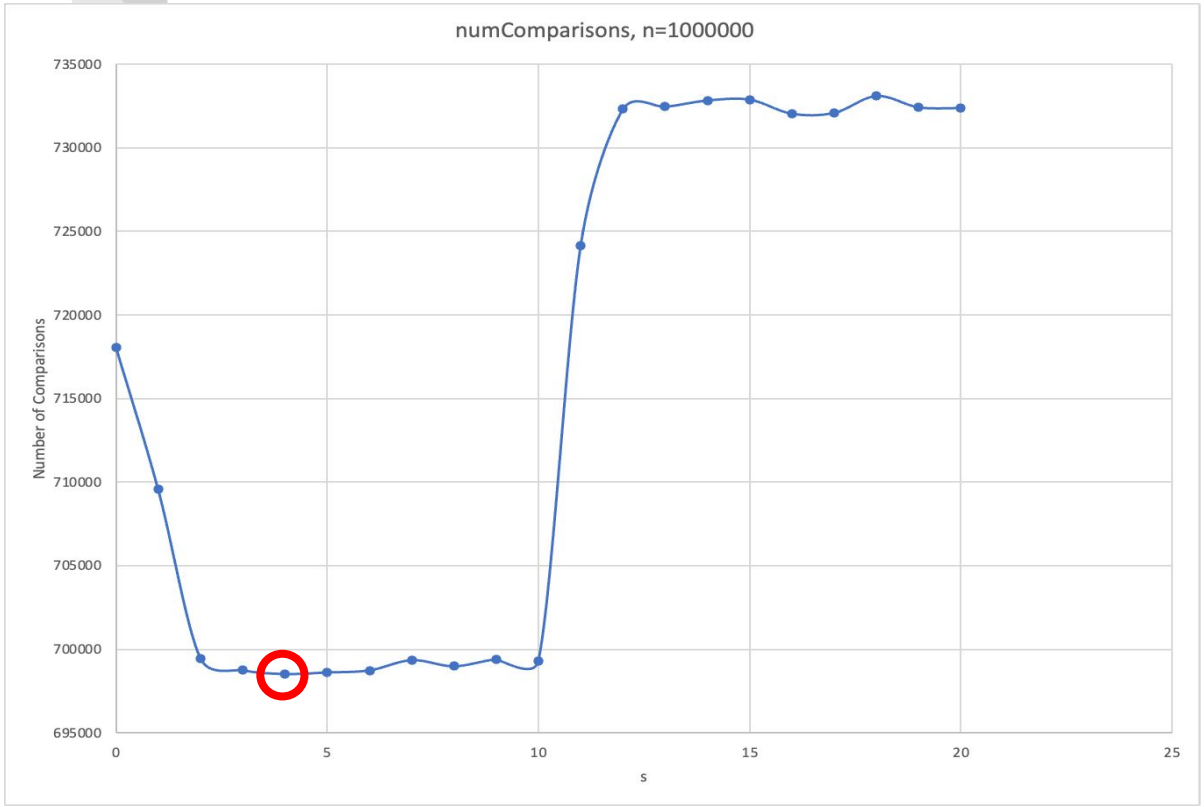
- Used 1,000,000 as our chosen input data size.
- Number of comparisons went up in steps as s ranged from 0 to 200.
- Looking for the least number of comparisons.

Determining the Optimal Value of S, n=1,000,000



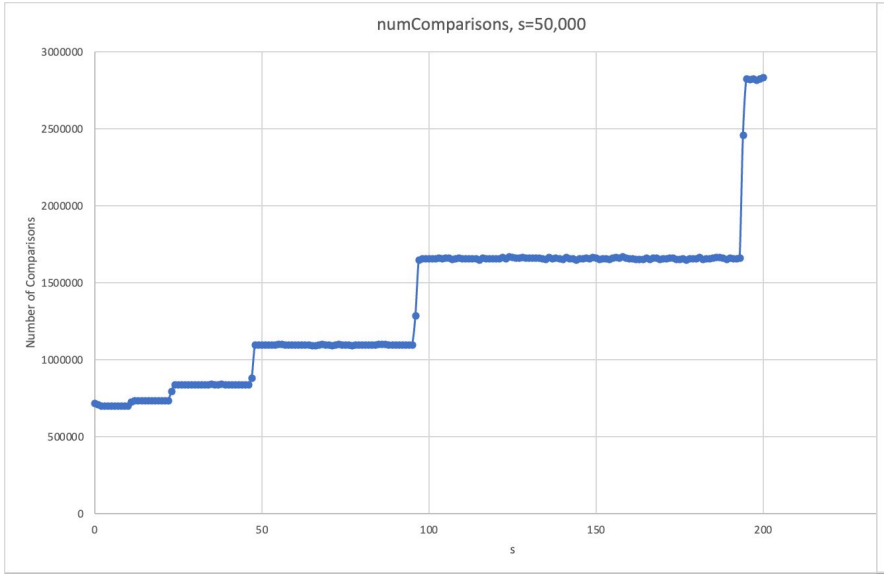
- Zoom in on the first step up, and graph s from 0 to 20.

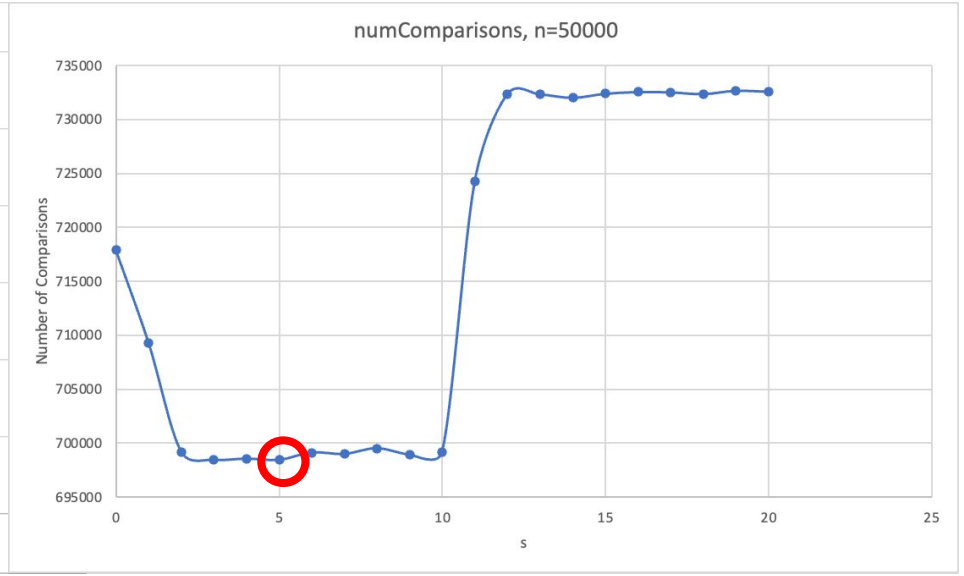
Determining the Optimal Value of S, n=1,000,000



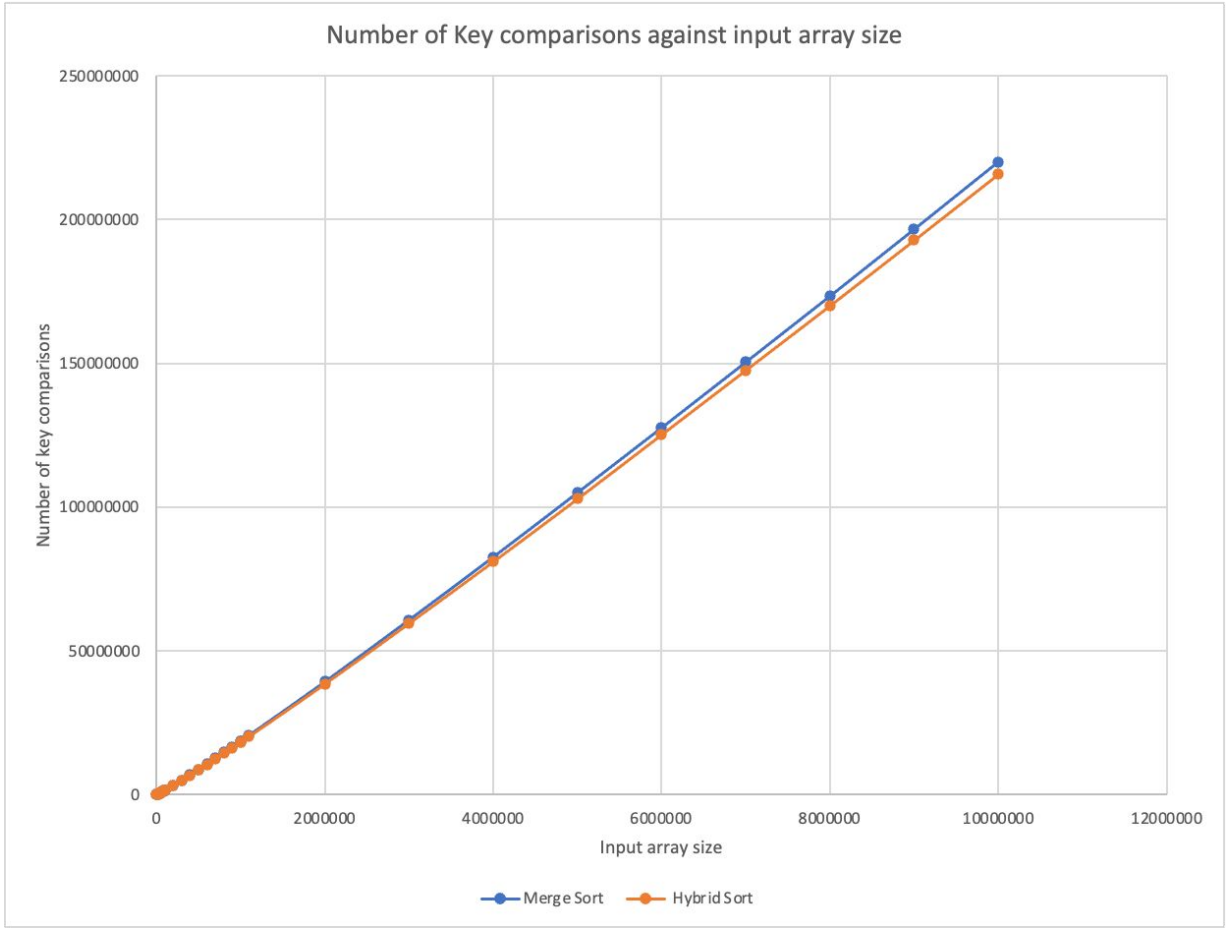
- Number of comparisons from in the range of s=3 to s=10, with the lowest point being s=4.

Determining the Optimal Value of S, n=50,000

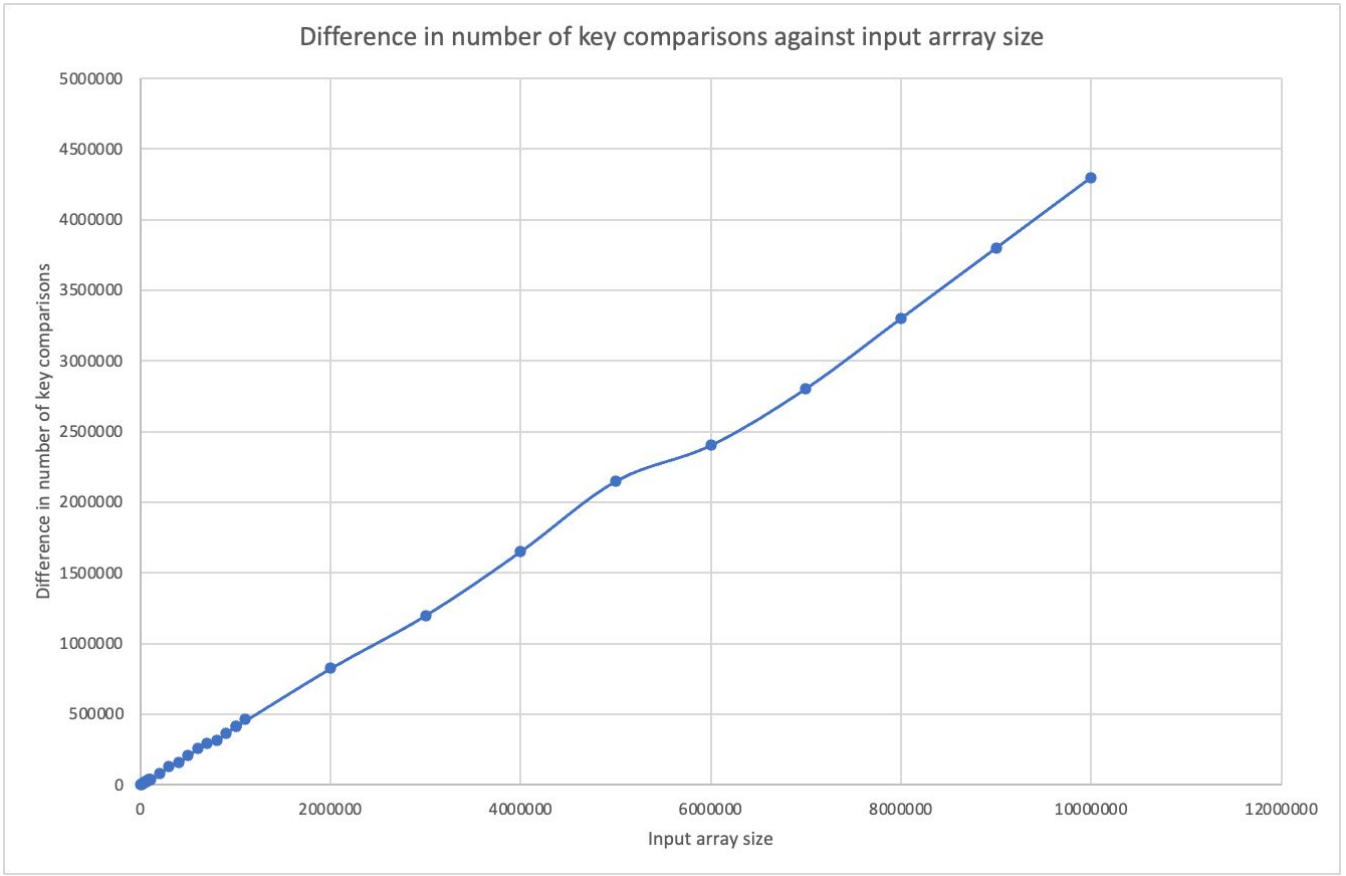




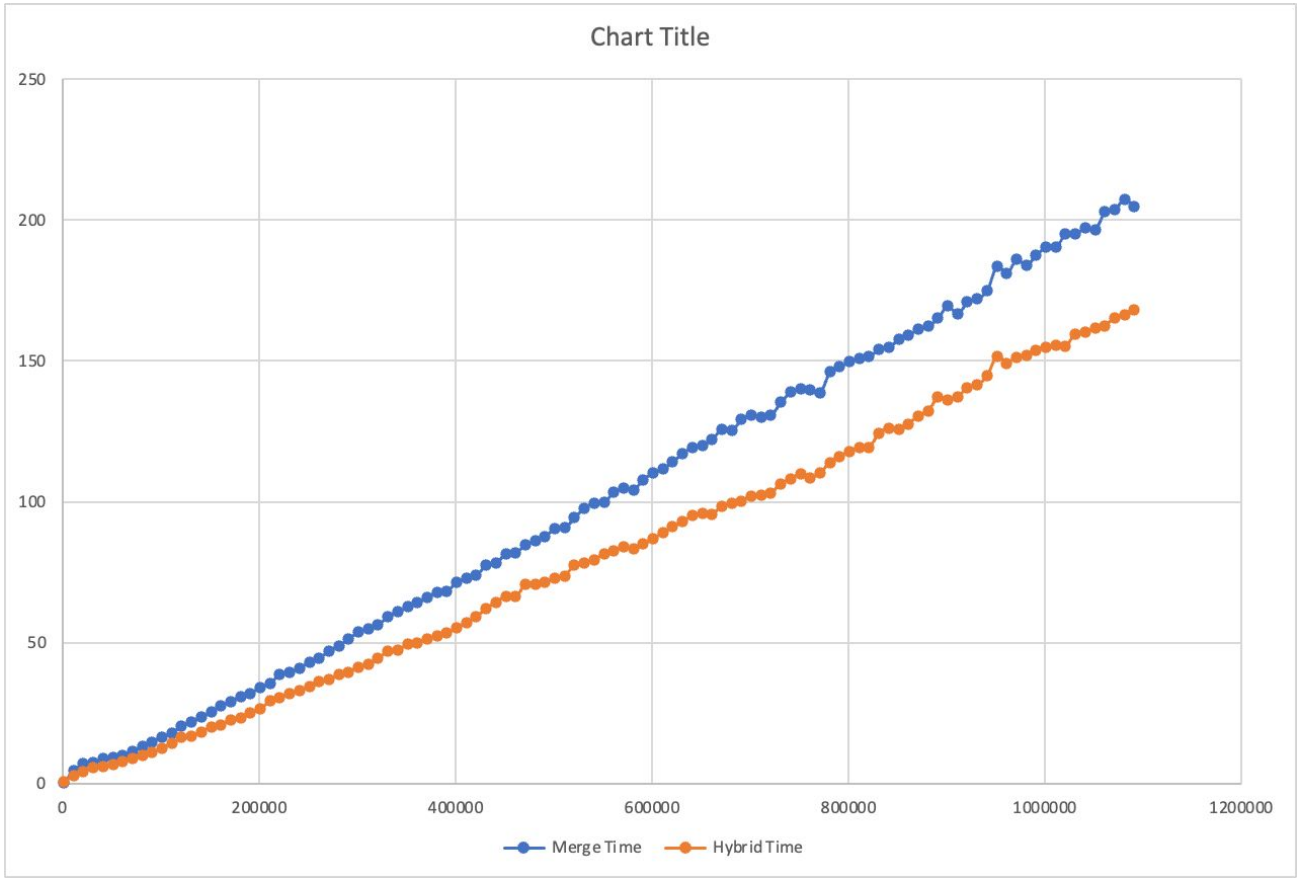
Comparison Against Mergesort, S=5



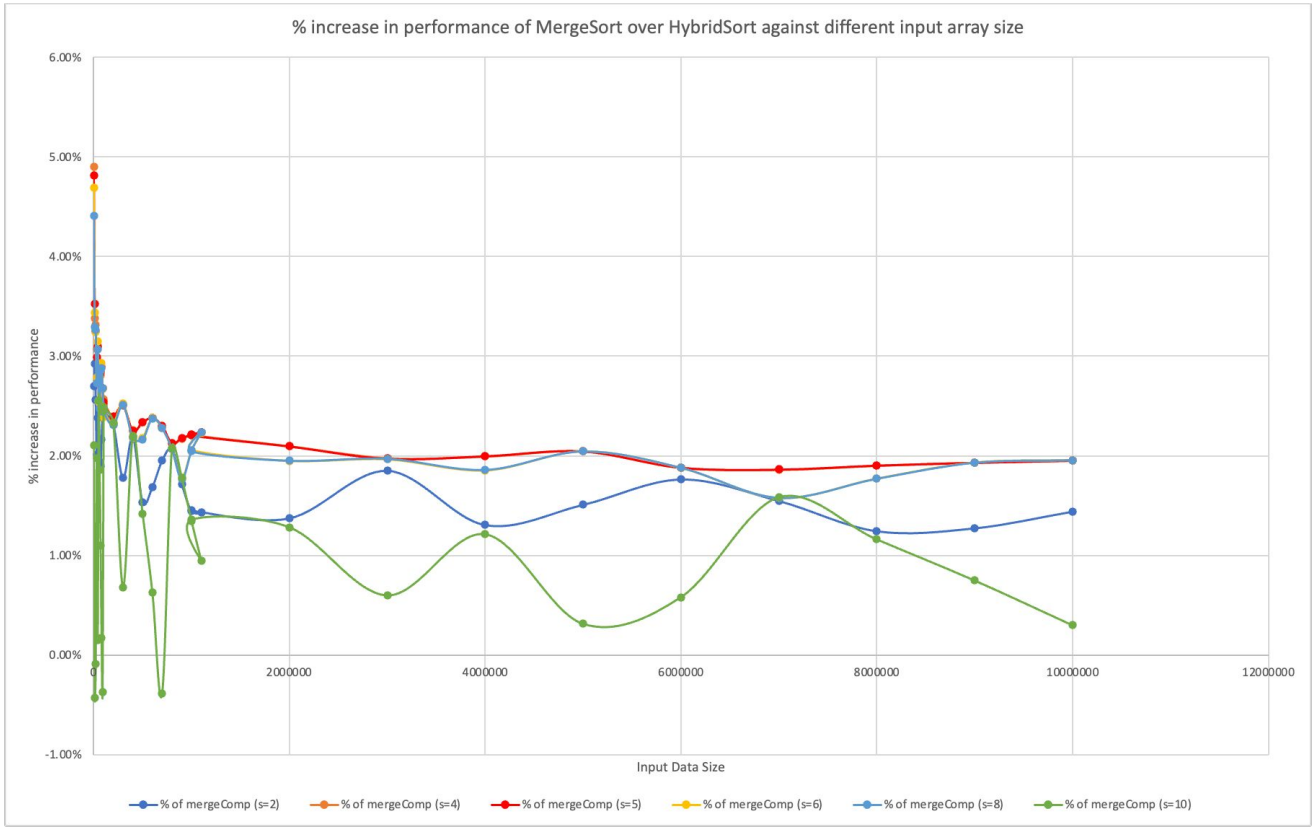
Comparison Against Mergesort, S=5, Absolute Difference



Comparison Against Mergesort (runtime), S=5, Absolute



Comparison Against Mergesort, Different S



- **Increase** in performance as s goes from 2 to 5. **Decrease** in performance thereafter.
- **Optimal value of s = 5**
- **Decrease** in performance as n increases.
- Constraints:
  - $2<s<10$
  - $1000<n<10{,}000{,}000$