SC2001 SSP1 Lab 1

Team 9: Nicholas Png,
Marcus Soh,
Vignesh Ezhil,
Yap Shen Hwei

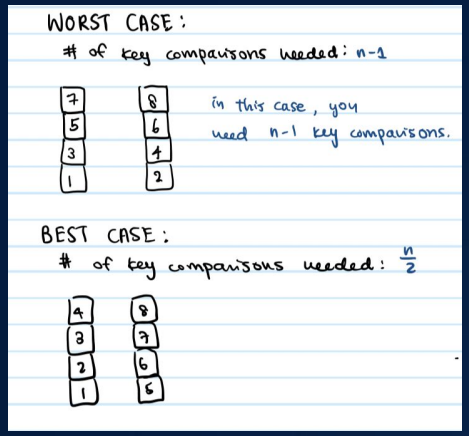
01

Algorithm Implementation

Insertion Sort

```
public static int[] insertionSort(int[] arr, int start, int end) {
    for (int i = start + 1; i <= end; i++) {
        for (int j = i; j > start; j--) {
            keyComparisons++;
            if (arr[j] < arr[j - 1]) {
                arr[j] = arr[j - 1] + arr[j];
                arr[j - 1] = arr[j] - arr[j - 1];
                arr[j] = arr[j] - arr[j - 1];
            } else
                break;

        }
    }

    int size = end - start + 1;
    int[] sorted = new int[size];
    System.arraycopy(arr, start, sorted, 0, size);
    return sorted;
}
```

Merge Sort




```java
public static int[] mergeSort(int[] arr, int start, int end) {
    int size = end - start + 1;
    int[] sorted = new int[end - start + 1];
    if (size <= 1) {
        sorted[0] = arr[start];
        return sorted;
    }

    int mid = (start + end) / 2;
    int s1 = mid - start + 1;
    int s2 = end - mid;
    int[] arr1 = new int[s1];
    arr1 = mergeSort(arr, start, mid);
    int[] arr2 = new int[s2];
    arr2 = mergeSort(arr, mid + 1, end);
    sorted = merge(arr1, arr2, s1, s2);
    return sorted;
}
```

```java
public static int[] merge(int[] arr1, int[] arr2, int s1, int s2) {
    int[] sorted = new int[s2 + s1];
    if (s1 + s2 == 0) {
        return null;
    } else if (s1 == 0) {
        return arr2;
    } else if (s2 == 0) {
        return arr1;
    }

    int cmp, i = 0, j = 0, a = 0;
    while (i < s1 && j < s2) {
        cmp = arr1[i] - arr2[j];
        keyComparisons++;
        if (cmp < 0) {
            sorted[a++] = arr1[i++];
        } else if (cmp > 0) {
            sorted[a++] = arr2[j++];
        } else {
            sorted[a++] = arr1[i++];
            sorted[a++] = arr2[j++];
        }
    }
    int diff1 = s1 - i;
    int diff2 = s2 - j;
    // if there are leftovers in any, then just add to arr
    while (diff1 > 0) {
        sorted[a++] = arr1[i++];
        diff1--;
    }
    while (diff2 > 0) {
        sorted[a++] = arr2[j++];
        diff2--;
    }
    return sorted;
}
```

Hybrid Sort Algorithm Implementation

```
// PSEUDOCODE

hybridSort(arr[], start, end) {
  if (size <= 1) {
          return the array // array already sorted
      }
  if (size <= S) {
      perform insertionSort on the array.
      return the sorted array.
  }
  else {
      perform hybridSort on left half of array.
      perform hybridSort on right half of array.
      merge(both left and right)
      return the sorted array.
  }

}
```

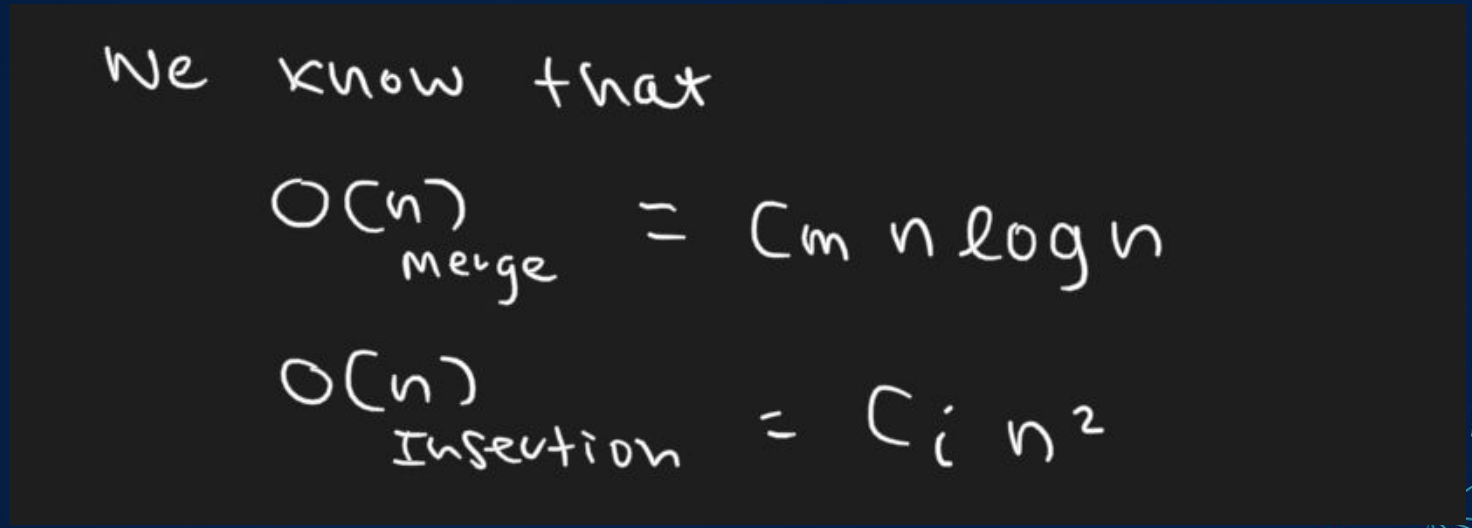
Hybrid Sort

```java
public static int[] hybridSort(int[] arr, int start, int end) {
    int size = end - start + 1;

    if (size <= 1)
        return arr;
    int[] sorted = new int[size];
    if (size <= S) {
        System.arraycopy(arr, start, sorted, 0, size);
        insertionSort(sorted, 0, size - 1);

    } else {
        int mid = (start + end) / 2;
        int s1 = mid - start + 1;
        int s2 = end - mid;
        int[] arr1 = new int[s1];
        arr1 = hybridSort(arr, start, mid);
        int[] arr2 = new int[s2];
        arr2 = hybridSort(arr, mid + 1, end);
        sorted = merge(arr1, arr2, s1, s2);
    }
    return sorted;
}
```

INSERTION

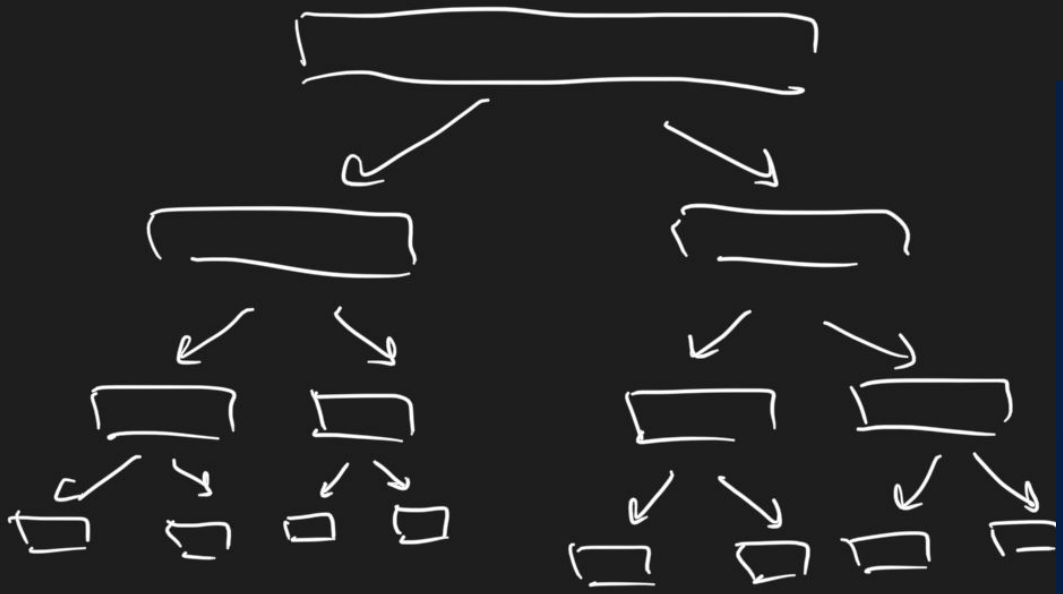
MERGE

Intuitive understanding of hybridSort



Then

For merge sort, your recursion tree will look like this



Suppose you stop when

$S = 32$ [this is only an example of S]

perform insertion sort

then, you save $\log_2 32 - 1$ = 5 levels of recursion

so

$= C_m \cdot n\,(\log(n) - 5) + \frac{n}{32} C_i(32^2)$

$\log 32 - 1$ ⇒ the height

# of subarrays of size 32

time complexity of insertion sort (worst case)

02

Time Complexity

Summary on Time Complexity of Hybrid Sort

- Best case : n+nlog(n/S)
  - Since we apply insertion sort on n/s groups of size S and the best case for insertion sort is O(n) , best case for the insertion sort on n/s groups would be O[(n/s) * S] = O(n)
  - The best case for merge sort is O(n log n)
  - Hence when for Hybrid sort, the best case time complexity would be O[n + n log(n/s)]
- Worst Case : nS+ n log(n/S)
  - Since we apply insertion sort on n/s groups of size S and the worst case for insertion sort is O(n^2) , worst case for the insertion sort on n/s groups would be O[(n/S)* S*2] = O(nS)
  - The worst case for merge sort is O(n log n)
  - Hence when for Hybrid sort, the worst case time complexity would be O[ns + n log(n/s)]

Summary of Time Complexity

	Merge Sort	Insertion Sort	Hybrid Sort
Best Case	nlog(n)	n	n + nlog(n/S)
Worst Case	nlog(n)	n^2	nS + nlog(n/S)
Average Case	nlog(n)	n^2	nS + nlog(n/S)

Fixed “S” & Varying “n”

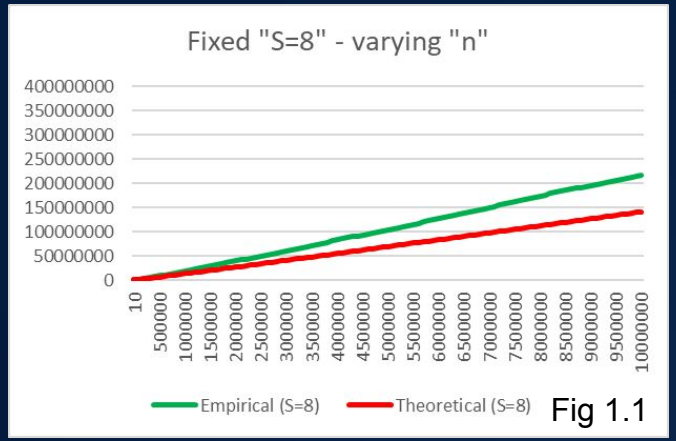


Fig 1.1

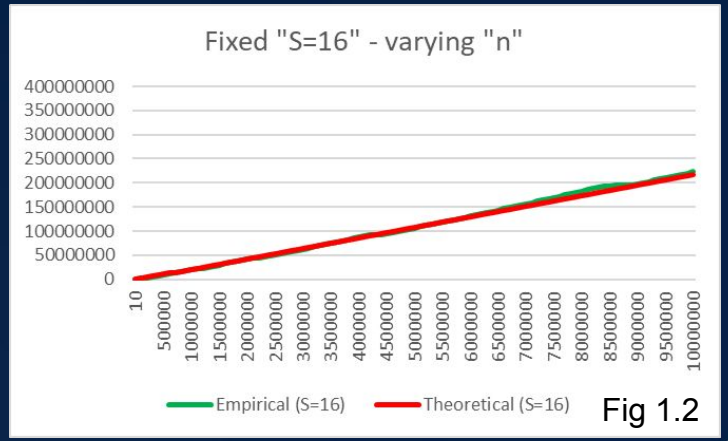


Fig 1.2

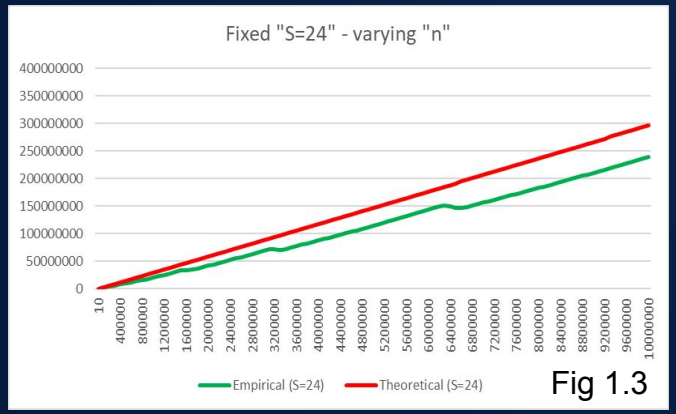


Fig 1.3

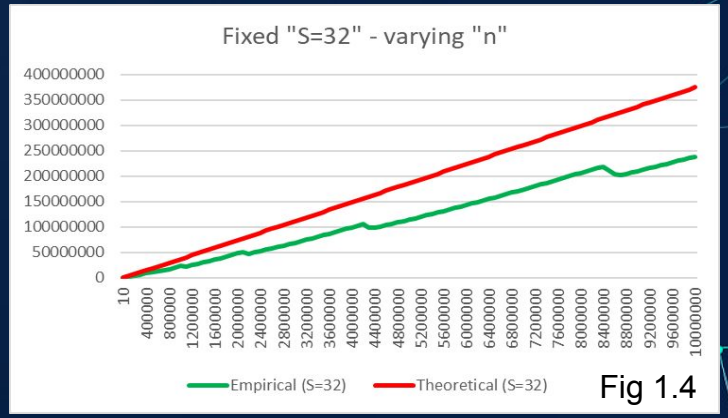


Fig 1.4

Legend

- Empirical
- Theoretical

Theoretical average case:

$nS + nLog_2(n/S)$

Fixed “S”, Varying “n”

Legend

- Empirical
- Theoretical

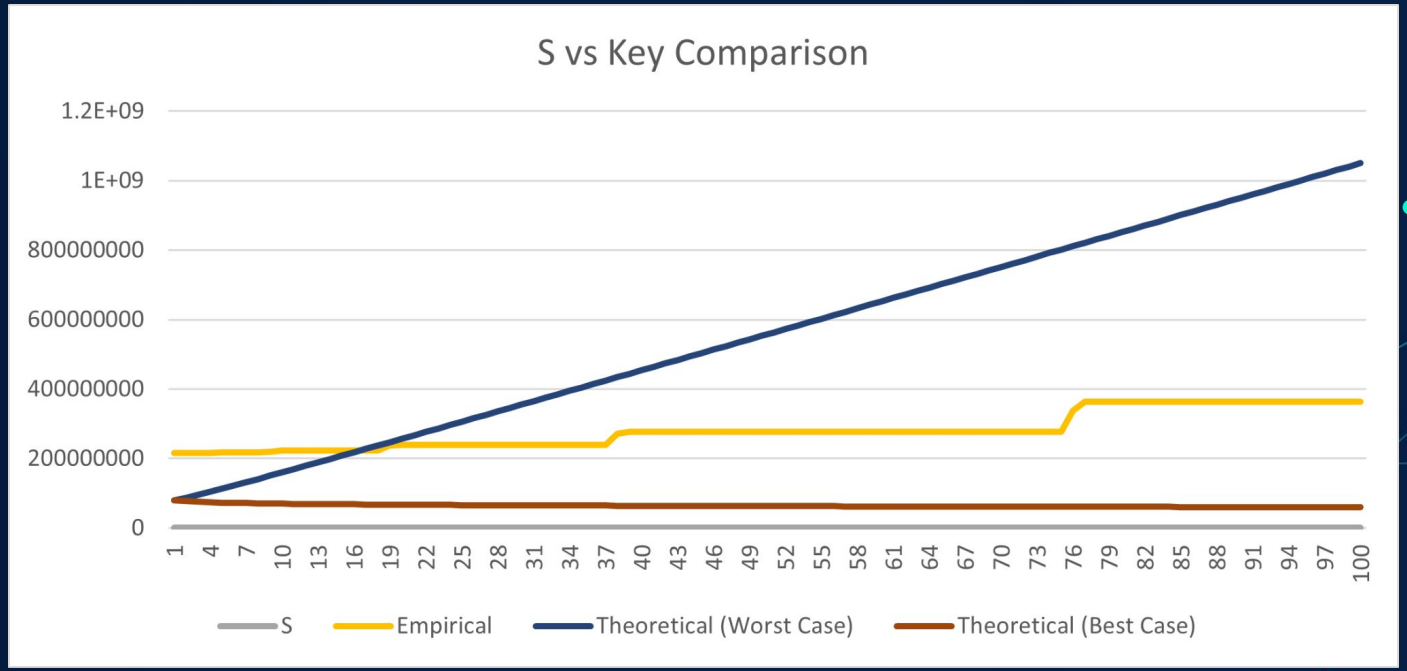
Fig 1.1

Fig 1.2

Theoretical average case:
nS + nLog(n/S)

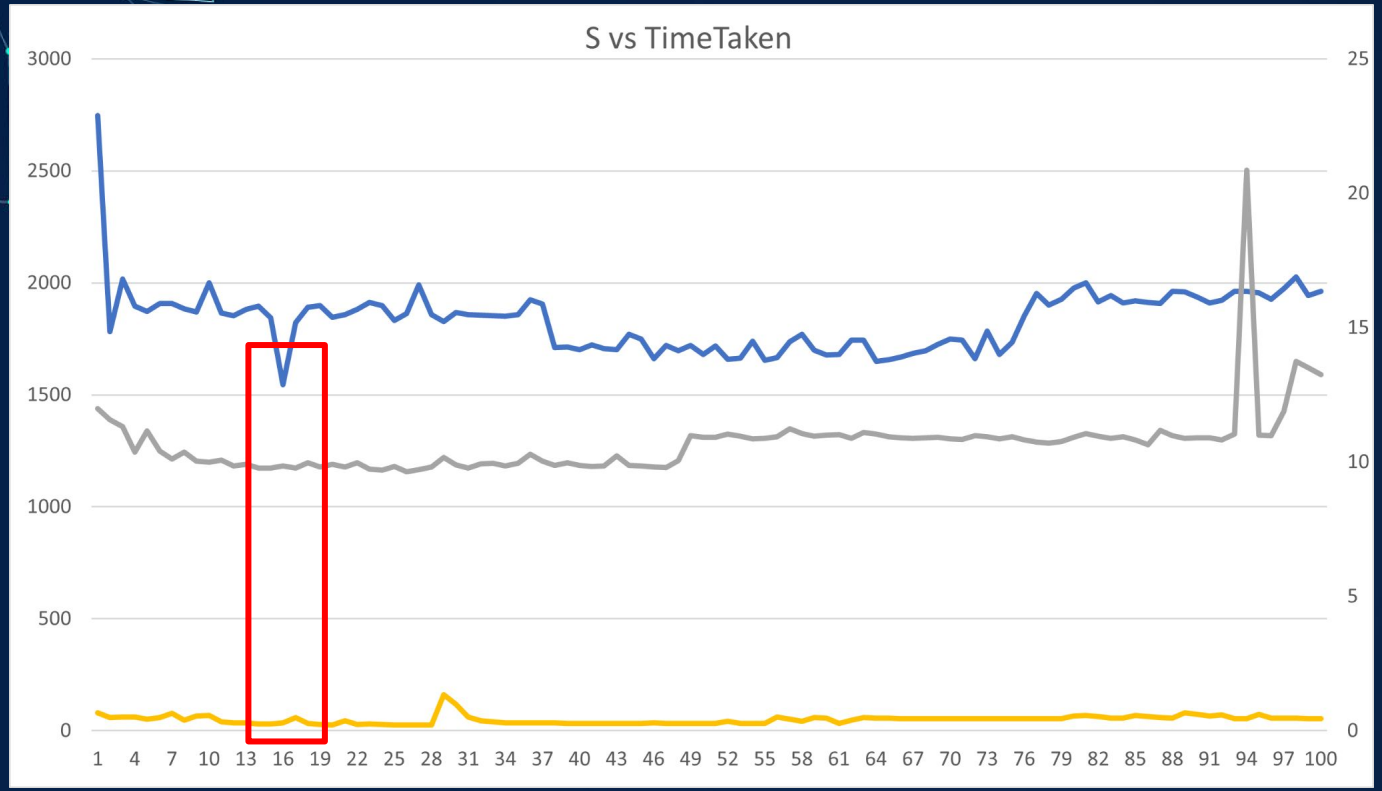
Fig 1.3

Fixed “n” & Varying “S”



- Best case : n+nlog(n/S)
- Worst Case : ns+ n log(n/s)

Varying “S” & varying “n” value



Legend

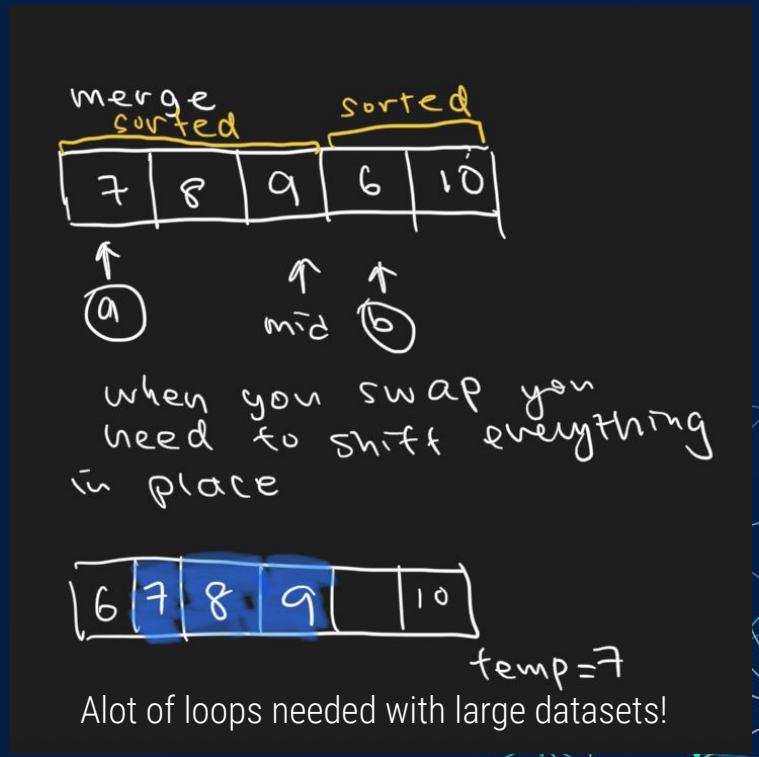
Blue - 10 M
Grey - 10 K
Yellow - 1 K

Key Comparison fails to account number of recursion calls required for mergesort for small S

Qd) Compare with Original Mergesort

Qd) Lecture's MergeSort vs our HybridSort

```java
public static int[] merge(int[] arr, int start, int end) {
    // remember that a and b will keep going right, so it doesn't make sense to
    // initialize b = m;
    int mid = (start + end) / 2, a = start, b = mid + 1, i, tmp, cmp;
    if (end - start <= 0)
        return arr;

    while (a <= mid && b <= end) {
        cmp = arr[a] - arr[b];

        // case 1: when slot[a] already < slot[b]
        if (cmp < 0) {
            a++;
        } else if (cmp > 0) {
            tmp = arr[b++];
            // shift everything and put the first ele of right array into the merged list.

            for (i = ++mid; i > a; i--) {
                arr[i] = arr[i - 1];
            }
            arr[a++] = tmp;
        } else {
            // if it's a single element
            if (a == mid && b == end)
                break;
            a++;
            tmp = arr[b++];
            for (i = ++mid; i > a; i--) {
                arr[i] = arr[i - 1];
            }
            arr[a++] = tmp;
        }
    }
    return arr;
}
```



Comparison continued (Lecture Merge vs Hybrid Sort)

	S = 16	Size = 1000			
	Merge			Hybrid	
	Key Comps	Run Time (ms)		Key Comps	Runtime (ms)
	8335	2.287333		9990	1.002667
	8323	1.332708		9909	1.212584
	8345	0.289292		10125	0.244209
	8346	0.238167		9960	0.135875
	8315	0.280417		9847	0.122208
avg	8332.8	0.8855834		9966.2	0.5435086

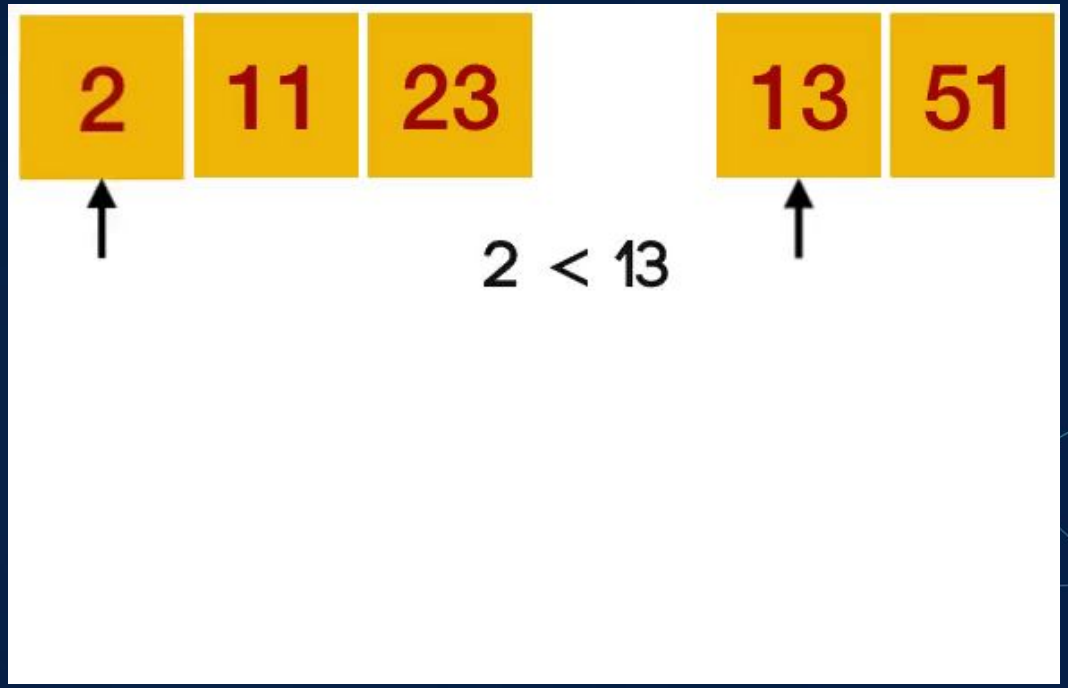
	S = 16	Size = 100000			
	Merge			Hybrid	
	Key Comps	Run Time (ms)		Key Comps	Runtime (ms)
	1498766	571.459208		1601935	14.116875
	1498348	552.889667		1602828	33.360167
	1498221	551.867375		1600634	8.98154
	1498614	577.32466		1602118	14.195292
	1498441	555.4724		1600614	11.115333
avg	1498478	561.802662		1601625.8	16.3538414

	S = 16	Size = 10000			
	Merge			Hybrid	
	Key Comps	Run Time (ms)		Key Comps	Runtime (ms)
	116865	8.46725		122867	0.913458
	116625	8.058458		123279	0.932208
	116701	13.028625		123133	2.907542
	116758	8.27875		123029	3.797083
	116651	8.5215		123382	1.065875
avg	116720	9.2709166		123138	1.9232332

Results:
Key Comparisons: Hybrid > Lect Merge Sort

Runtime: Hybrid takes shorter Runtime than Lect MergeSort

Our Hybrid Sort's Merge()



Use Auxiliary Space - Outplace sorting

Comparison 2.0 (Auxiliary Merge vs Hybrid Sort)

For example: n = 10000, S = 16

```
Auxiliary MS Key Comparisons: 116708---  time taken: 2.1405ms
```

```
HS Key Comparisons: 123421---  time taken: 1.122417ms
```

Results:
Key Comparisons: Hybrid > Auxiliary Merge Sort

Runtime: Hybrid takes shorter Runtime than Auxiliary MergeSort

We conclude that:

1. Recursion overhead will increase run time, so much so that its effects over power larger key comparisons.

2. For our case, we found S =16 to be our optimal S.

What we also learnt:

1. **Since insertion sort will be better when arrays are small and almost sorted,**
2. **Hybrid sort will be good when the subarrays are almost sorted, this reduces the runtime of insertion sort for these subarrays.**
3. **Using insertion sort with merge sort will also greatly reduce the recursion overhead.**

THANK YOU :)

References:

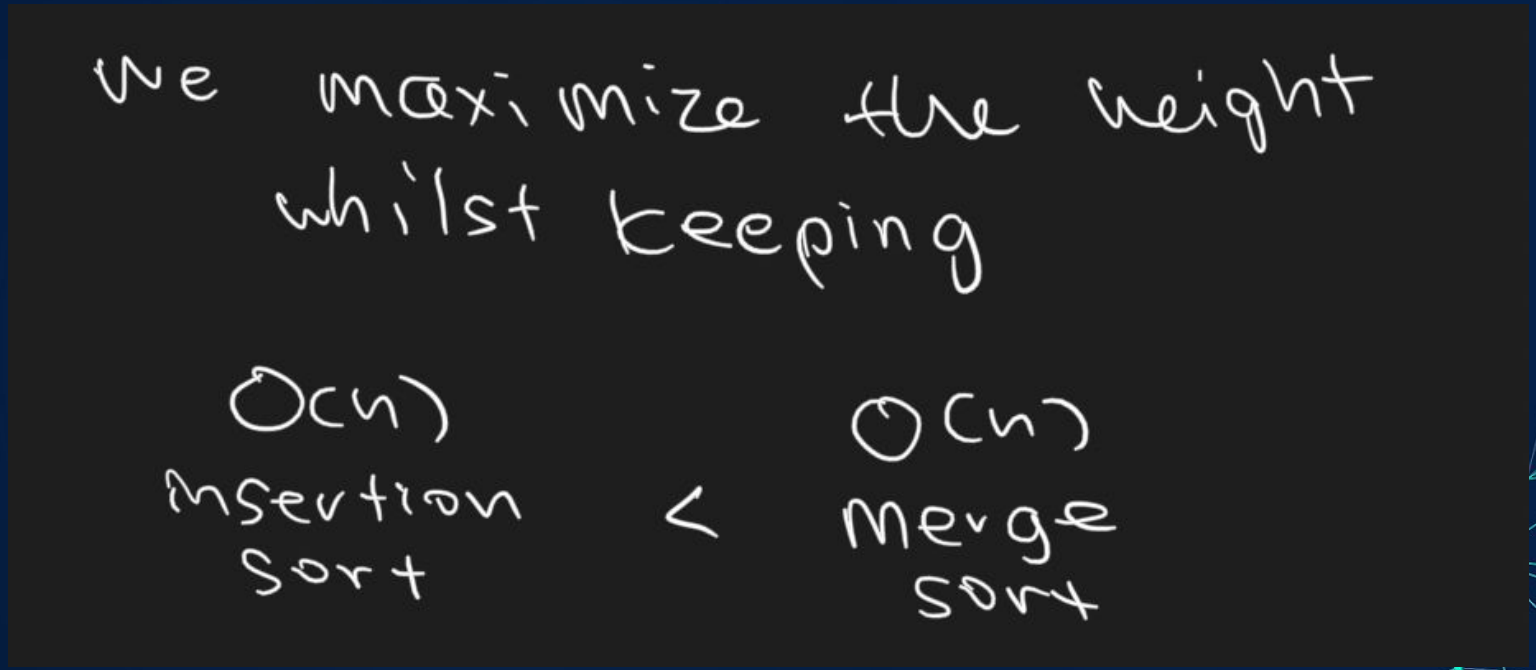
https://www.youtube.com/watch?v=emeME__917E

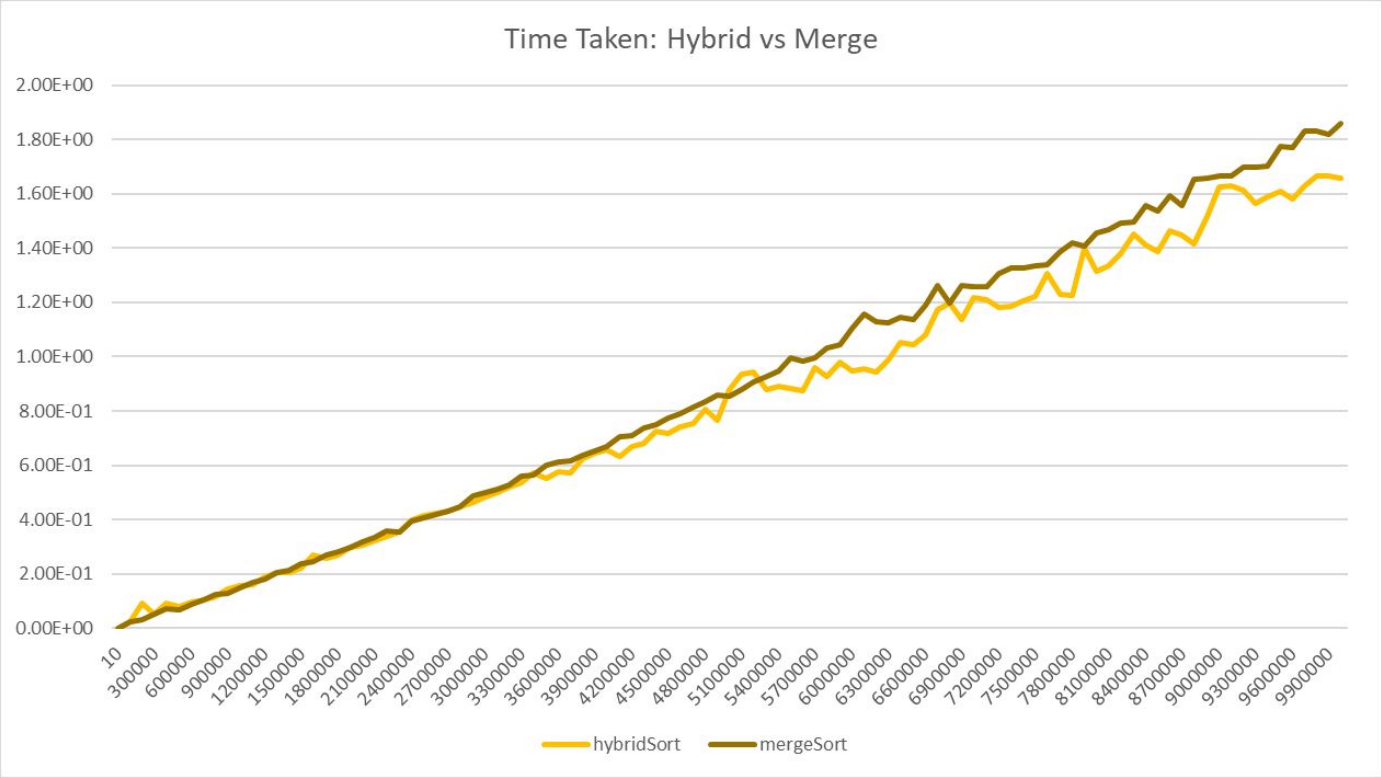
Analysis of Insertion Sort

Best Case : 1 key comparison/ iteration , total (n - 1) key comparisons
Worst Case: i key comparisons for the ith iteration
(1 + 2 + 3… + i-1 + i) → total : n(n-1)/2

```java
public static int[] insertionSort(int[] arr, int start, int end) {
    int j;
    for (int i = start + 1; i <= end; i++) {
        j = i;
        while (j > start) {
            keyComparisons++;
            if (arr[j] < arr[j - 1]) {
                arr[j] = arr[j - 1] + arr[j];
                arr[j - 1] = arr[j] - arr[j - 1];
                arr[j] = arr[j] - arr[j - 1];
            } else {
                break;
            }
            j--;
        }
    }

    int size = end - start + 1;
    int[] sorted = new int[size];
    System.arraycopy(arr, start, sorted, 0, size);
    return sorted;
}
```

Ultimate goal: find the maximum size of Array such that




Time Taken: Hybrid vs Merge
2.00E+00
1.80E+00
1.60E+00
1.40E+00
1.20E+00
1.00E+00
8.00E-01
6.00E-01
4.00E-01
2.00E-01
0.00E+00
10
300000
600000
900000
1200000
1500000
1800000
2100000
2400000
2700000
3000000
3300000
3600000
3900000
4200000
4500000
4800000
5100000
5400000
5700000
6000000
6300000
6600000
6900000
7200000
7500000
7800000
8100000
8400000
8700000
9000000
9300000
9600000
9900000
hybridSort
mergeSort

Theoretical Optimal S