Project 1:
Integration of Mergesort & Insertion Sort

Team 5
Sydney Teo, Namrah, Saniya

Table of Contents

Hybrid Algorithm 1

```
def hybridSort(arr, n, m, S):
    if (m-n) <= S:
        insertionSort(arr[n:m+1])
    else:
        mid = (n + m) // 2

        if ((m - n) <= 0):
            return

        elif ((m - n) > 1):
            hybridSort(arr, n, mid, S)
            hybridSort(arr, mid + 1, m, S)

        merge(arr, n, mid, m)
```

Insertion Sort

```
1  # performs pure insertion sort
2  def insertionSort(arr):
3      global insertionCount
4      for i in range(1, len(arr)):
5          key = arr[i]
6          j = i-1
7          while j >= 0:
8              insertionCount+=1
9              if(arr[j] > key):
10                 arr[j + 1] = arr[j]
11                 j -= 1
12             else:
13                 break
14         arr[j + 1] = key
```

Merge function

```
def merge(arr, n, mid, m):
    global mergeCount
    a = n               # first element (start of first list)
    b = mid + 1         # start of second list
    i = 0
    temp = 0

    if ((m - n) <= 0): # trivially sorted
        return

    while (a <= mid and b <= m): # not reached the end of respective lists

        if(arr[a] > arr[b]): # move element in second list to the front
            mergeCount += 1       # need to make a comparison
            temp = arr[b]

            i = b
            while i > a:
                arr[i] = arr[i - 1]
                i -= 1

            arr[a] = temp
            a += 1
            b += 1
            mid += 1

        elif arr[a] <= arr[b]: # element is already in the right place
            mergeCount += 1         # need to make a comparison
            a += 1

        else:
            if a == mid and b == m:
                break

            mergeCount += 1         # need to make a comparison
            temp = arr[b]
            a += 1

            i = b
            while i > a:
                arr[i] = arr[i - 1]
                i -= 1

            arr[a] = temp

            a += 1
            b += 1
            mid += 1
```

Hybrid Algorithm 2

```
def mergesertionSort(arr, thresh):
    global mergeCount
    if len(arr) <= thresh:
        insertionSort(arr)
    else:
        mid = len(arr)//2

        L = arr[:mid]
        R = arr[mid:]

        mergesertionSort(L, thresh)
        mergesertionSort(R, thresh)

        i = j = k = 0

        while i < len(L) and j < len(R):
            mergeCount +=1
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

Hybrid Algorithm 1

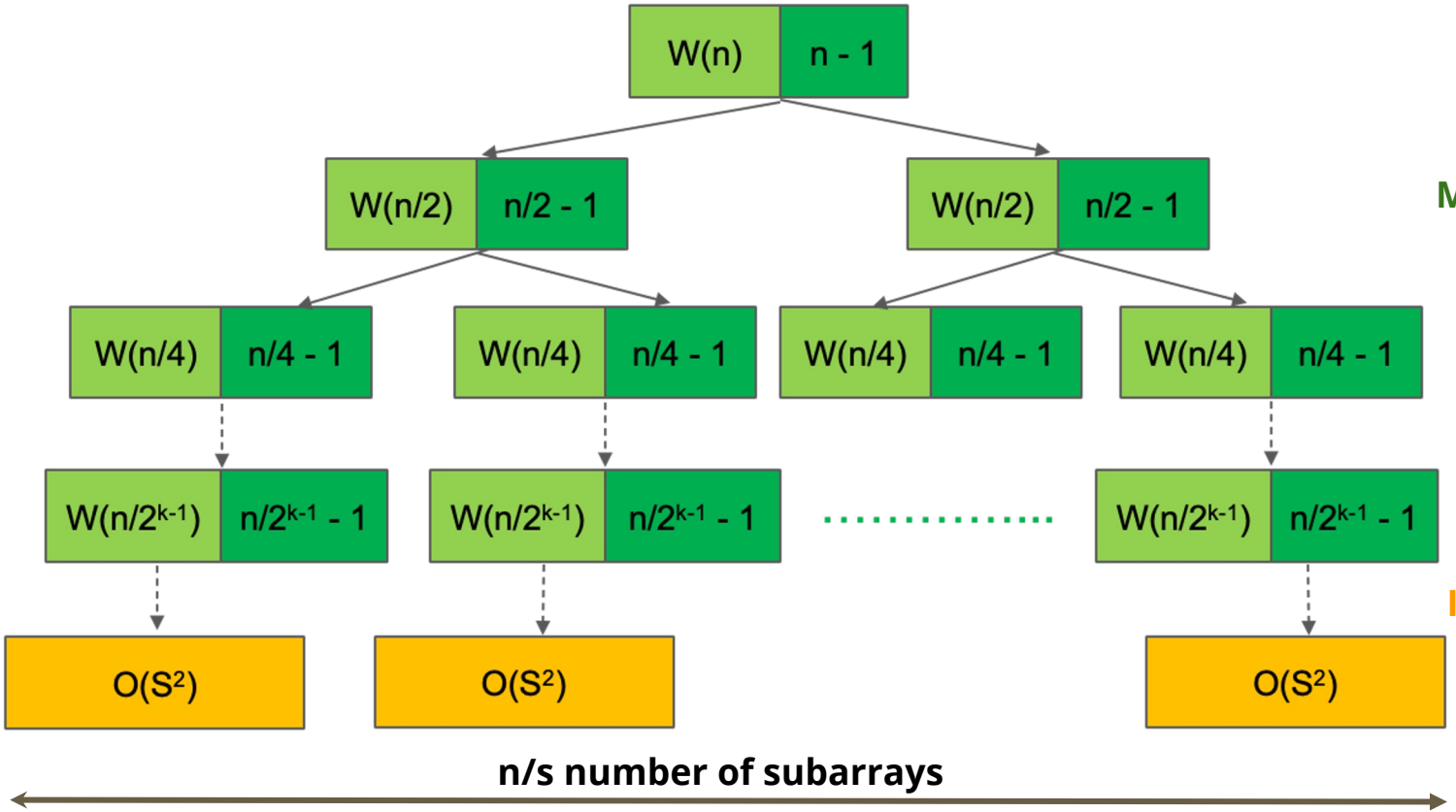
- Unable to run efficiently on array sizes more than 100,000
- For instance, our code took 50 min and was still unable to sort for array size 100,000
- This could be due to the **number of swaps** done in the merge function
- It may be so significant that it resulted in the large time complexity

Hybrid Algorithm 2

- Therefore, we decided to use **auxiliary array** (a copy of the original array) to help with sorting in-place
- This has a trade-off of space complexity of O(n) (size of array)
- This method does not require swapping, which may reduce the time complexity
- Able to run on array size of 10 million efficiently

Analysis of Time Complexity (Theoretical)

Theoretical Analysis



Insertion sort Time Complexity

O(ns)

Insertion Sort

For $s-1$ iterations:

$$1 + \frac{1}{2}(1+2) + \frac{1}{3}(1+2+3) + \cdots + \frac{1}{s-1}(1+2+\cdots+(s-1))$$

$$= \sum_{i=1}^{s-1} \left(\frac{1}{i} \sum_{j=1}^{i} j\right)$$

$$= \frac{1}{2}\left[\frac{s(s+1)}{2} - 1\right]$$

$$= \theta(n^2)$$

Since there are $\frac{n}{s}$ sub arrays

$$\text{Insertion Time Complexity} = \frac{n}{s} \times \frac{1}{2}\left[\frac{s(s+1)}{2} - 1\right]$$

$$= \frac{n}{s} \times \left[\frac{s(s+1)}{4} - \frac{1}{2}\right]$$

$$= \frac{n(s+1)}{4} - \frac{n}{2s}$$

$$= O(ns)$$

Mergesort Time Complexity

O(n log (n/s))

Mergesort

We know that $\frac{n}{2^k} = s \Rightarrow 2^k s = n$

$$2^k = \frac{n}{s}$$

$$\lg 2^k = \lg \frac{n}{s}$$

$$k \lg 2 = \lg \frac{n}{s}$$

$$k = \lg \frac{n}{s} \div \lg 2$$

simply $k = \log \frac{n}{s}$ —①

$$(n-2^0) + (n-2^1) + (n-2^2) + (n-2^3) + \cdots + (n-2^{k-1})$$

$$= kn - (2^0 + 2^1 + \cdots + 2^{k-1})$$

$$= kn - 2^k + 1 \quad —②$$

Substitute ① into ②,

$$n \log \frac{n}{s} - 2^{\log \frac{n}{s}} + 1 = n \log \frac{n}{s} - \frac{n}{s} + 1$$

$$= O(n \log \frac{n}{s})$$

Hybrid Time Complexity

Total complexity = Complexity of Mergesort + Complexity of Insertion sort

$= O(n \log (n/s)) + O(ns)$

Comparison of Time Complexity

	Best	Average	Worst
Insertion Sort	Θ(S)	Θ(S^2)	Θ(S^2)
Merge Sort	O(n log(n/S))	O(n log(n/S))	O(n log(n/S))
Hybrid Sort	(n/S)x(S) + n log(n/S) = **Θ(n log (n/S) + n)**	(n/S)x(S^2) + n log(n/S) = **Θ(n log (n/S) + nS)**	(n/S)x(S^2) + n log(n/S) = **Θ(n log (n/S) + nS)**

Generation of input

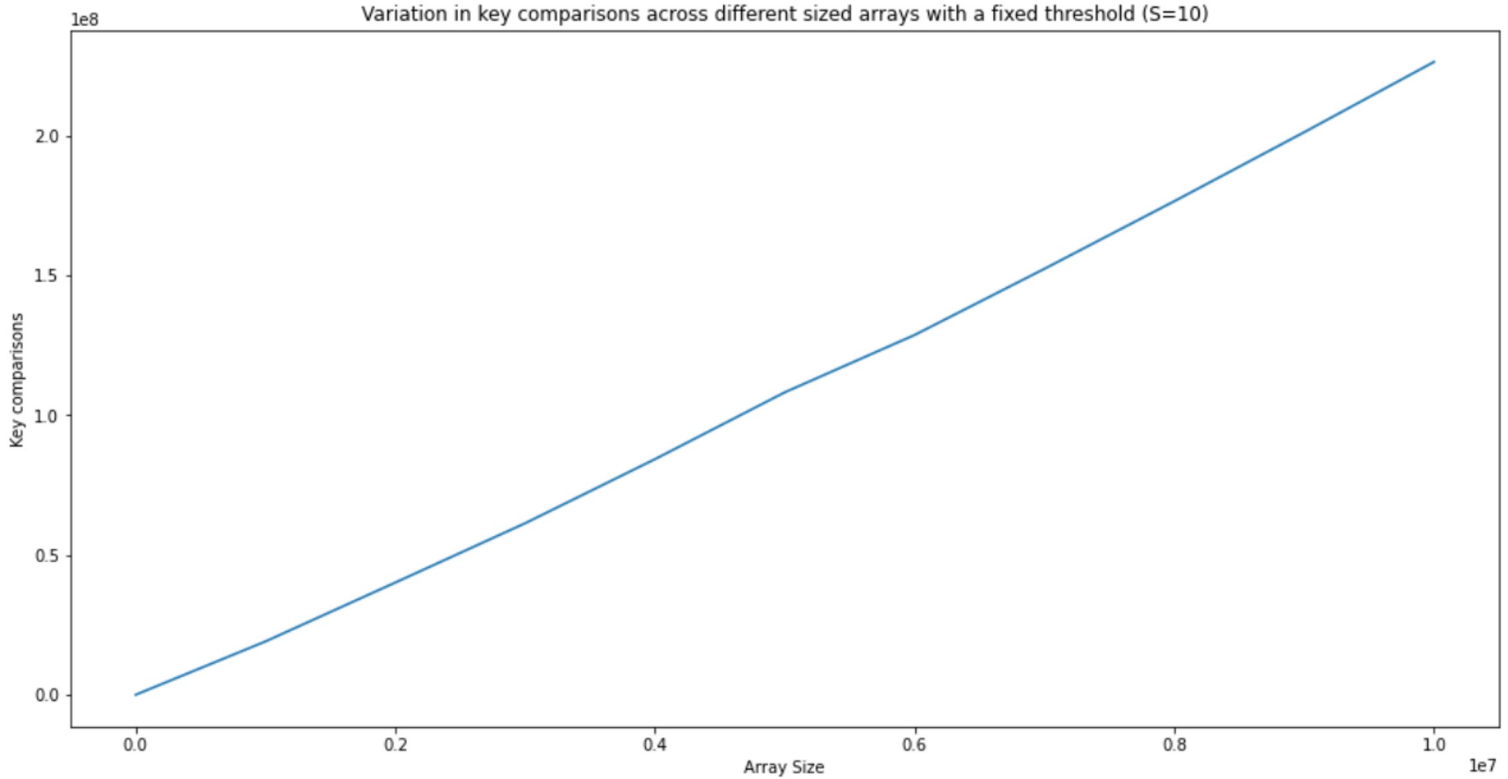
```
# generates a list of input arrays
def inputGenerator():
    arr = []
    np.random.seed(0)
    for i in range (LOWER_N,UPPER_N, STEP_N):
        print(i)
        arr.append(np.random.permutation(i)[:i].tolist())
    arr.append(np.random.permutation(10000000)[:10000000].tolist())
    return arr

inputArray = inputGenerator()

nValues = [*range(LOWER_N,UPPER_N,STEP_N)]
nValues.append(10000000)
sValues = range(LOWER_S,UPPER_S)
```

Analysis of Time Complexity (Empirical)

Fixed S = 10 Across different n values

```
# sorts arrays in a list of arrays, with a fixed threshold
def sortFixedS(inputArray):
    global mergeCount
    global insertionCount
    comparisonCount = []
    cpuTime = []
    for i in inputArray:
        mergeCount = 0
        insertionCount = 0
        arr_size = len(i)
        print(arr_size)
        start_time = time.time()
        mergesertionSort(i, THRESHOLD)
        total_time = (time.time() - start_time)
        comparisonCount.append(mergeCount + insertionCount)
        cpuTime.append(total_time)
    return comparisonCount, cpuTime
```

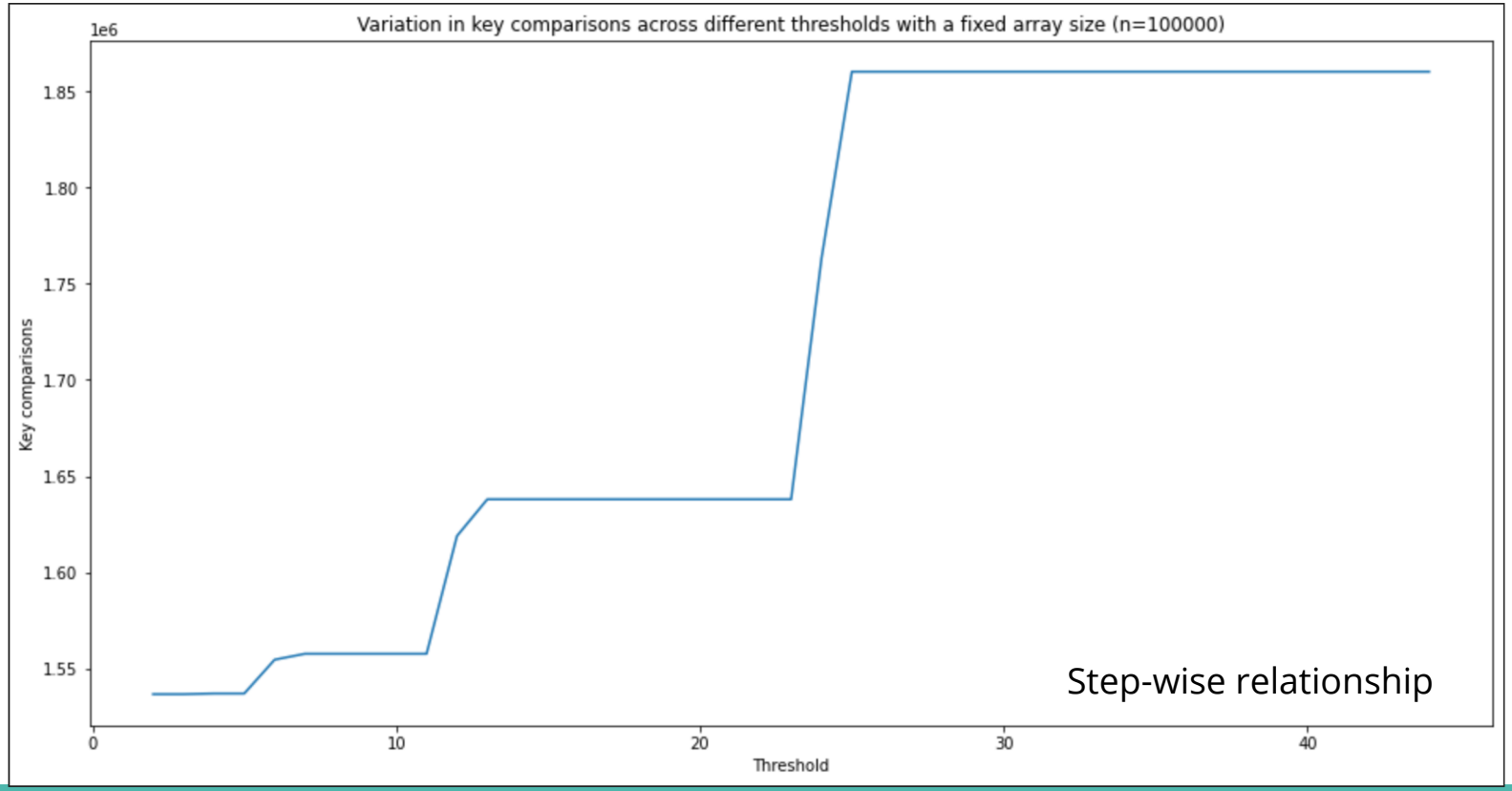
Fixed S = 10 Across different n values



Fixed n = 100,000 Across different S values

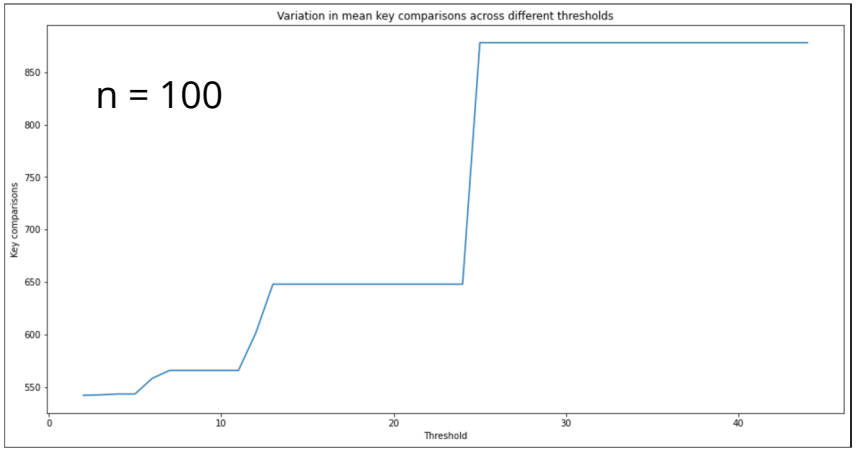
```
#sorts an array with different thresholds
def sortFixedN(arr):
    global mergeCount
    global insertionCount
    comparisonCount = []
    cpuTime = []
    arr_size = len(arr)
    for i in range(LOWER_S,UPPER_S):
        print(i)
        mergeCount = 0
        insertionCount = 0
        start_time = time.time()
        arr1 = arr[:]
        mergesertionSort(arr1, i)
        total_time = (time.time() - start_time)
        comparisonCount.append(mergeCount + insertionCount)
        cpuTime.append(total_time)
    return comparisonCount, cpuTime
```

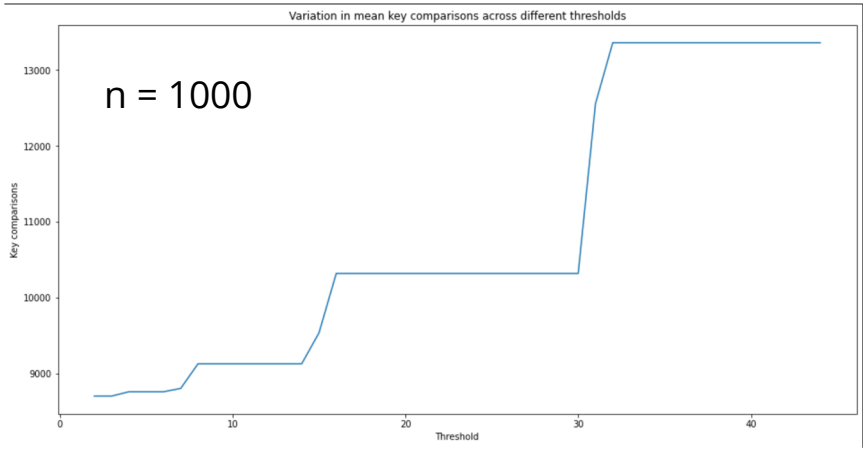
Fixed n = 100,000 Across different S values

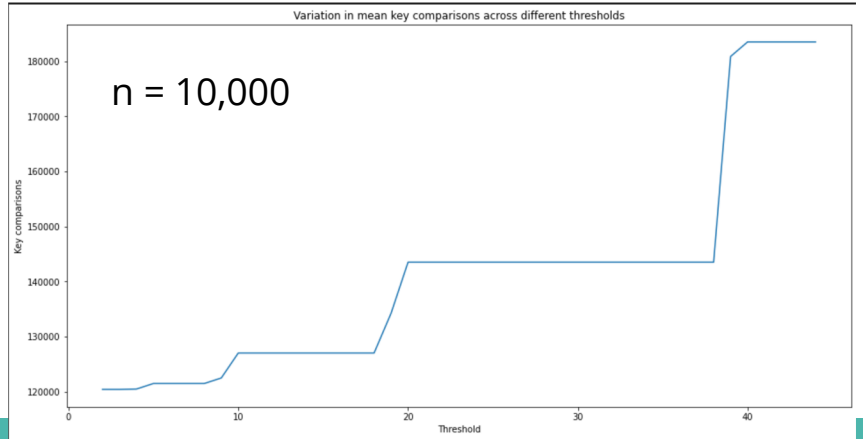


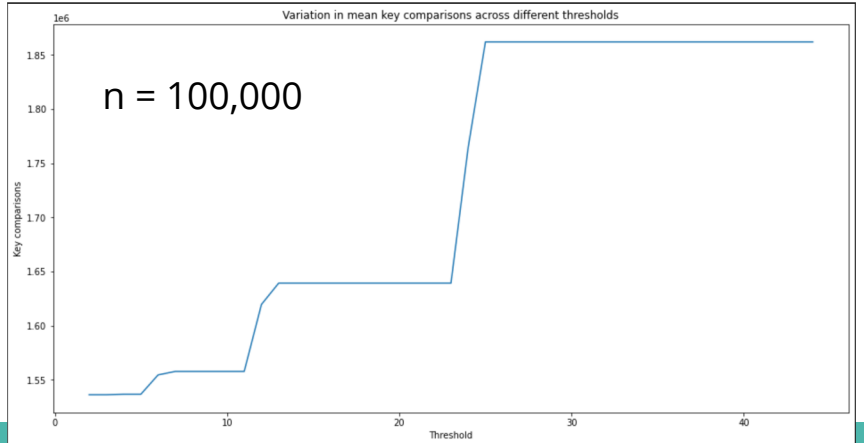
Optimal S

Mean no. of key comparisons at different thresholds

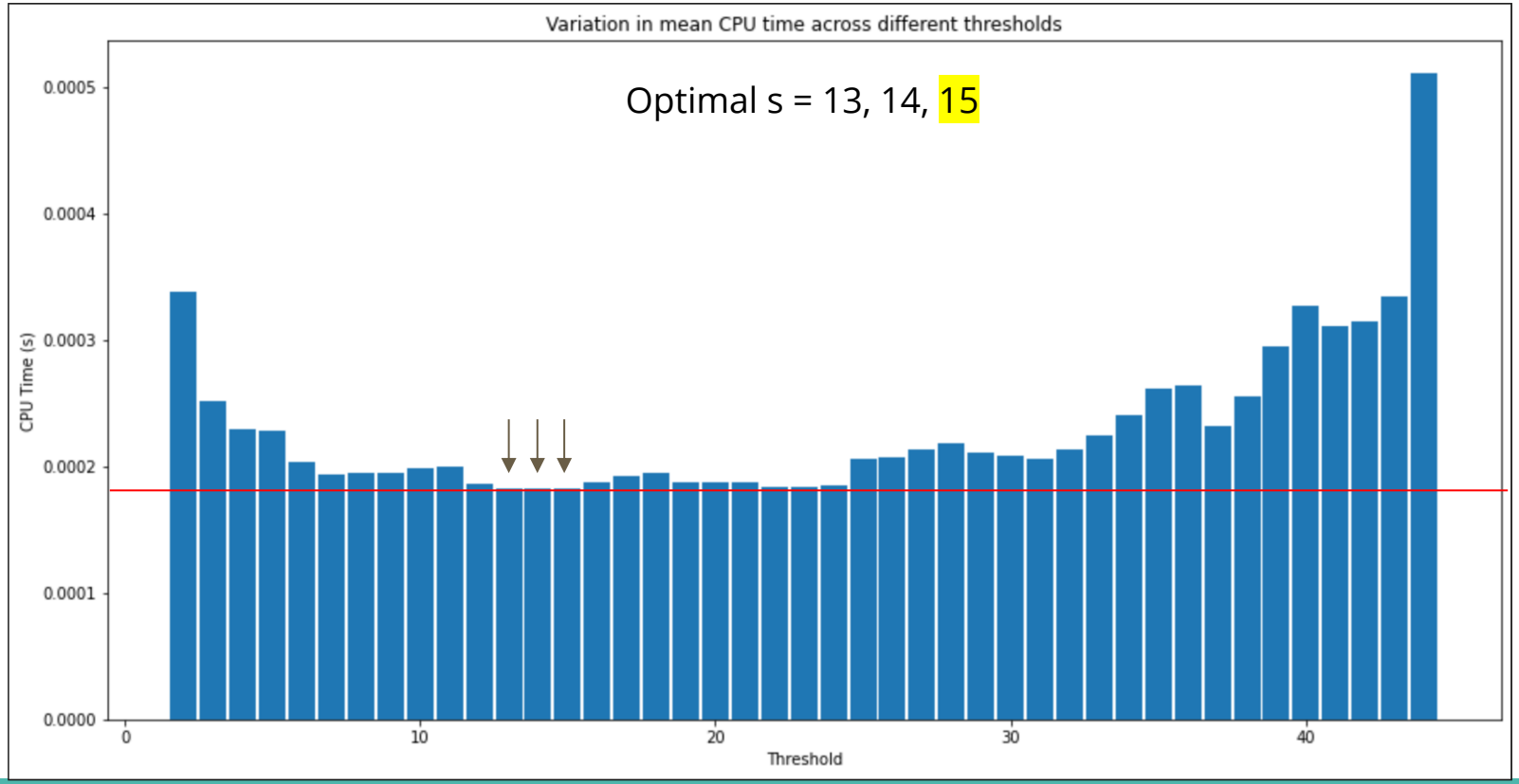




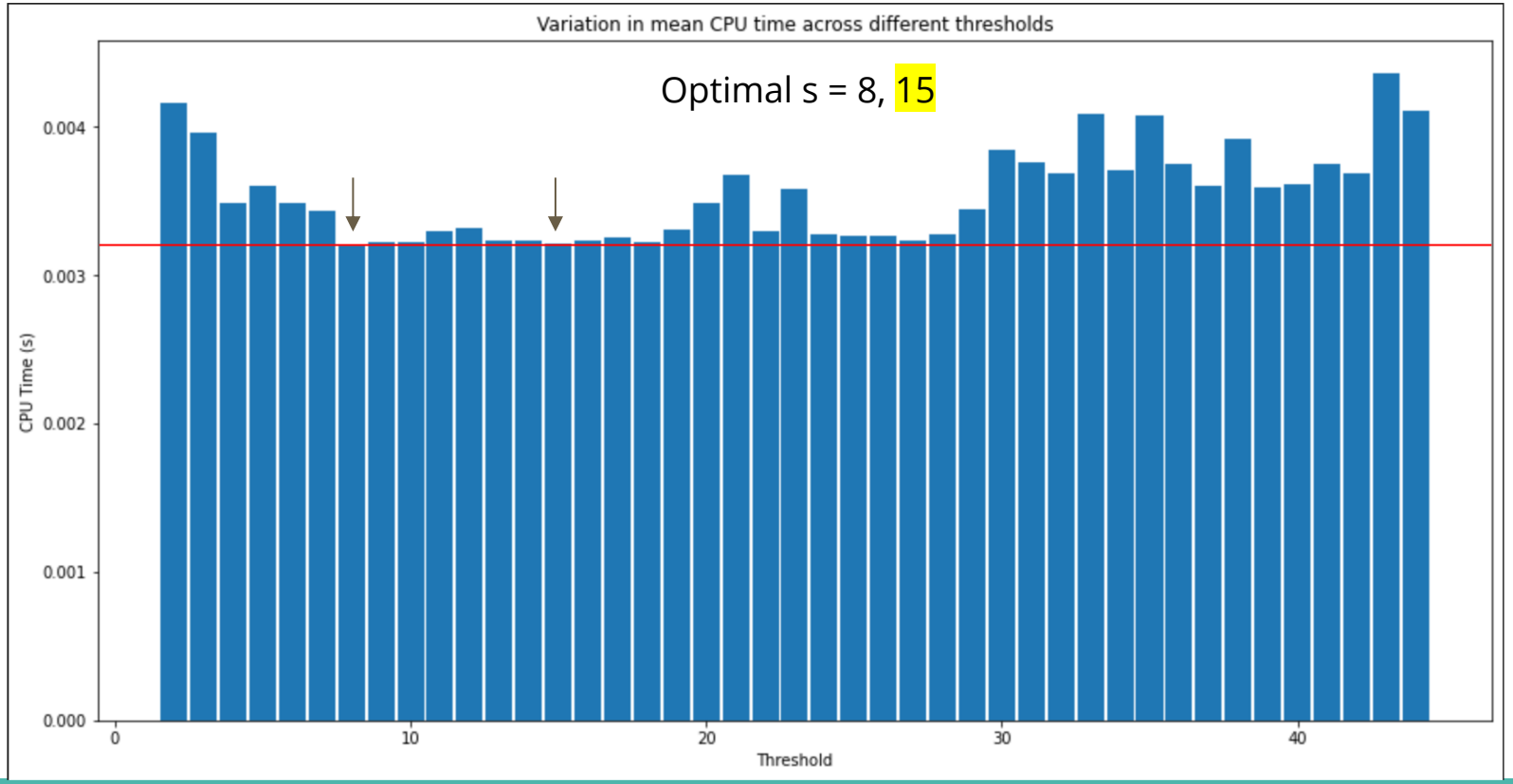




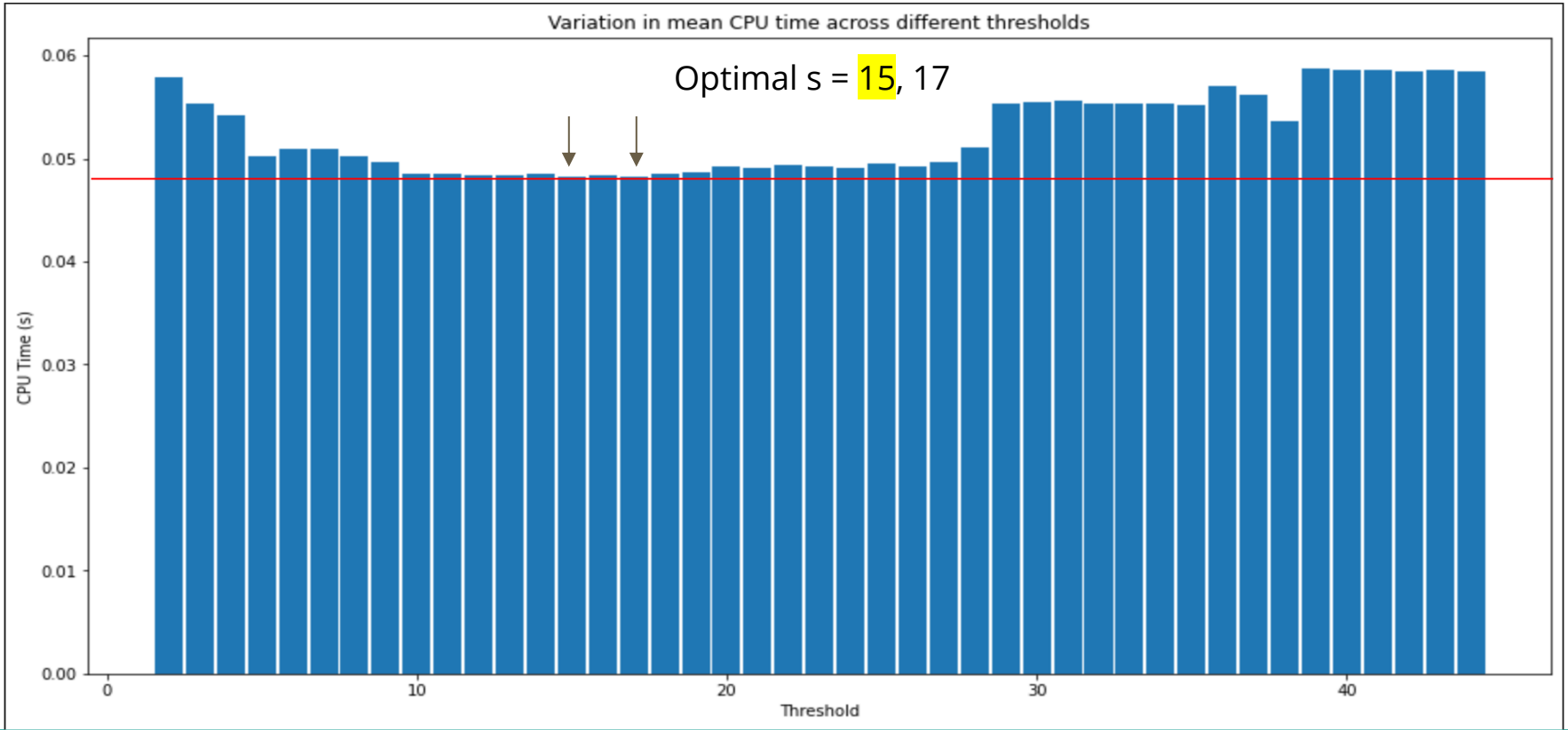
Array size = 100



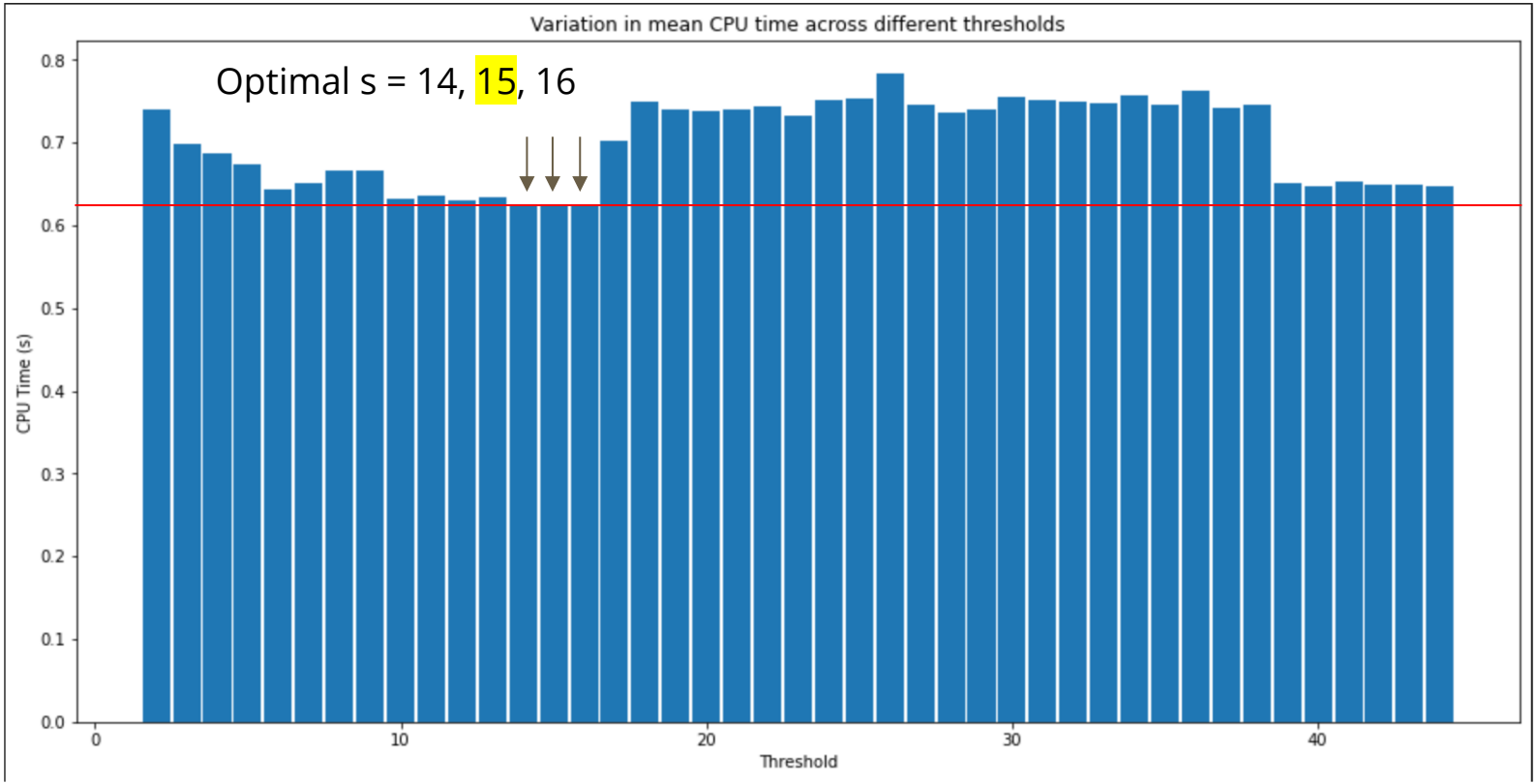
Array size = 1000



Array size = 10,000



Array size = 100,000



Optimal S

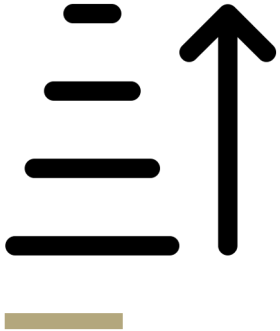
From observation, optimal S = 15

⇒ If we could not find an optimal value from observation alone, we could plot the average CPU time (across different array sizes, n) against each threshold value S

⇒ Then select the S value corresponding to the lowest average CPU time

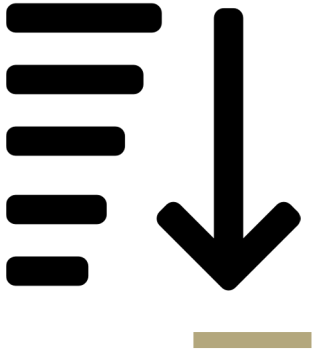
Hybrid Sort vs Merge Sort

Types of Arrays



Sorted

Random



Reverse

Hybrid Sort vs Merge Sort - Key Comparisons

Array Type	Sorted	Random	Reverse
Hybrid Sort	108,628,928	226,413,292	143,136,576
Merge Sort	114,434,624	220,100,166	118,788,160
Hybrid Sort % Change	-5.07%	+2.87%	+20.50%

Higher number of key comparisons in hybrid sort for worst and average cases due to integration of insertion sort

(S = 15)

N = 10mil

Hybrid Sort vs Merge Sort - CPU Time (in seconds)

Array Type	Sorted	Random	Reverse
Hybrid Sort	138.20	168.24	217.96
Merge Sort	215.02	169.43	272.59
Hybrid Sort % Change	-35.73%	-0.70%	-20.04%

Low CPU time in hybrid sort due to lesser recursion overhead

(S = 15)

N = 10mil

Conclusion

Conclusion

1. We believe that the optimal S can range from 5 to 40, depending on type of array and size of array
2. No. of key comparisons is not the only factor we should look at. Instead, there are other factors such as hardware and CPU time that determines how efficient the algorithm is.
3. No. of key comparisons for Hybrid Sort is higher than Merge Sort for average and worst cases due to the Insertion sort implemented within Hybrid Sort
4. CPU Time in Hybrid Sort is consistently lower than Merge Sort due to the decrease in recursion overhead, which has a greater effect than increase in number of key comparisons

Thank you!