Nil Patel

CSCI 4520

Dr. Wang

Analysis of Algorithms

This project applies the concepts of algorithm analysis to a Java program that performs sorting and searching on a large dataset. The program generates 100,000 random integers, searches for the value 53 using a sequential search, counts how many times 53 appears, sorts the data using Merge Sort, and then counts the occurrences of 53 again using a Binary Search–based counting method.

The main goal is to compare the time cost of counting occurrences using only a sequential scan versus sorting the array and then using binary search to count the same value. This experiment connects the theoretical time complexity of algorithms to practical runtime behavior on a real computer.

Description of Algorithms

Sequential Search

The program first uses a simple sequential (linear) search to find the first occurrence of the number 53 in the unsorted array. It scans the array from index 0 to the end, checking each element:

- If it finds 53, it reports the index and stops.
- If it reaches the end without finding 53, it reports that 53 is not in the array.

For counting occurrences, it again scans all elements and increments a counter every time array[i] == 53.

The time complexity for both finding the first occurrence and counting all occurrences is:

- Best case: $\Theta(1)$ (if 53 is at the beginning)
- Worst / average case: $\Theta(n)$, where n is the size of the array (100,000).

Merge Sort

The array is then sorted using Merge Sort. The method mergeSort(int arr[], int start, int end):

1. Recursively splits the array into two halves:
   - Left half: start to middle
   - Right half: middle + 1 to end
2. Calls itself on both halves until each subarray has one element.

3. Uses the merge(int arr[], int start, int middle, int end) function to merge the two sorted halves back into one sorted segment.

The merge step:

- Creates two temporary arrays: Leftarr and Rightarr.
- Copies the elements into these arrays.
- Walks through both arrays, picking the smaller current element each time and placing it back into the original array.
- Copies any remaining elements from the left or right side.

The recurrence relation for Merge Sort is:

$$T(n) = 2T(n/2) + \Theta(n)$$

Solving this gives a time complexity of:

- Time: $\Theta(n \log n)$
- Space: $\Theta(n)$, due to the temporary arrays used during merging.

Binary Search–Based Counting

After sorting, the program uses a binary search–based method to count how many times 53 appears in the array.

The method binarySearch(int startingArray[], int search):

1. Performs a standard binary search to locate one position where search (53) appears.
2. If the value is found at index middle:
   - It sets left = middle - 1 and scans left while startingArray[left] == search, increasing the count each time.
   - It sets right = middle + 1 and scans right while startingArray[right] == search, increasing the count.
   - It returns the total count of occurrences.
3. If 53 is not found, it returns 0.

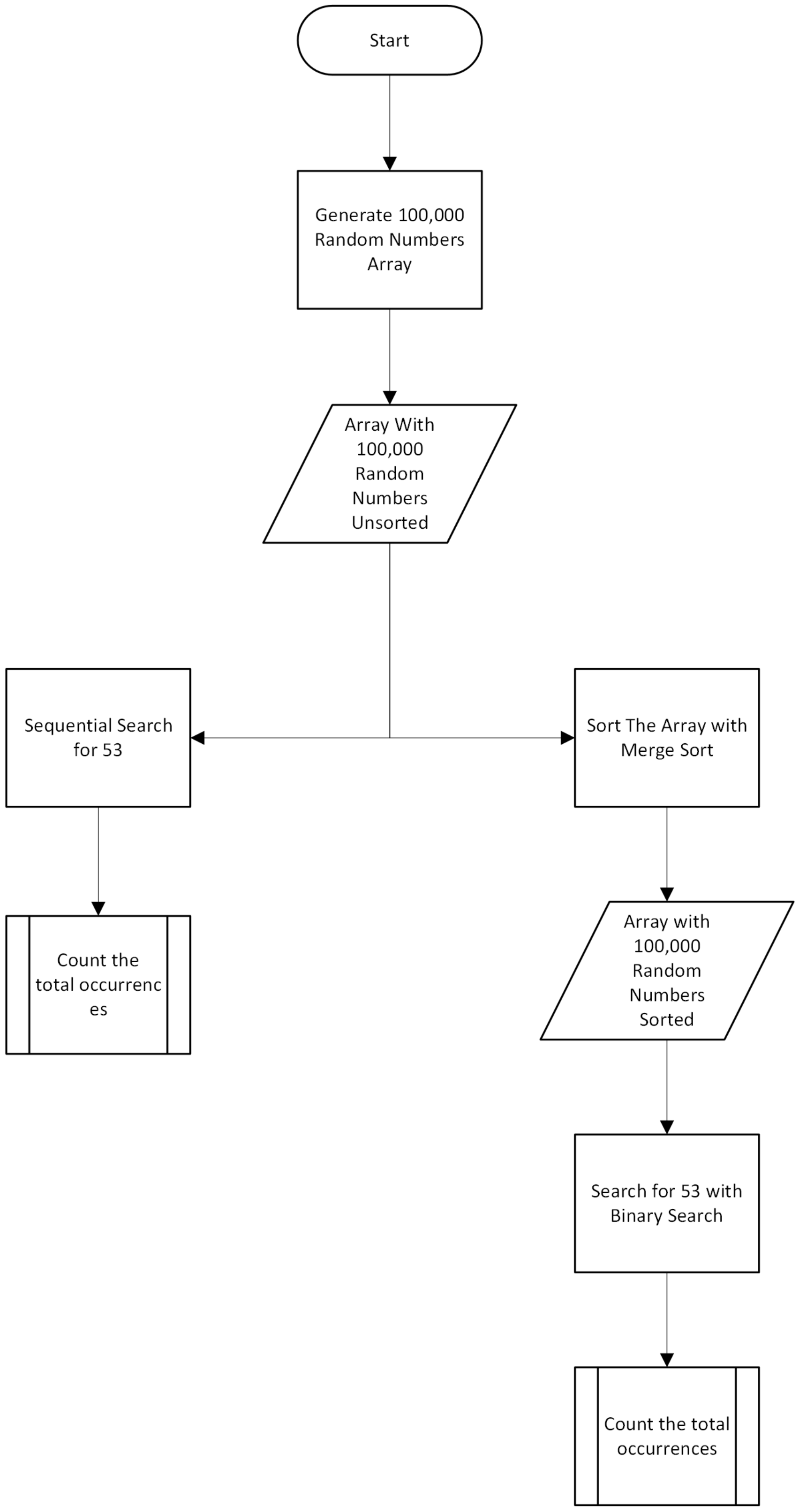
The binary search portion runs in $\Theta(\log n)$ time to find one occurrence, and the additional left/right scan takes $\Theta(k)$ time, where k is the number of repeated occurrences of the value.

Overall complexity for counting this way is:

- Time: $\Theta(\log n + k)$
- Space: $\Theta(1)$, because it only uses a few variables for indices and counters.

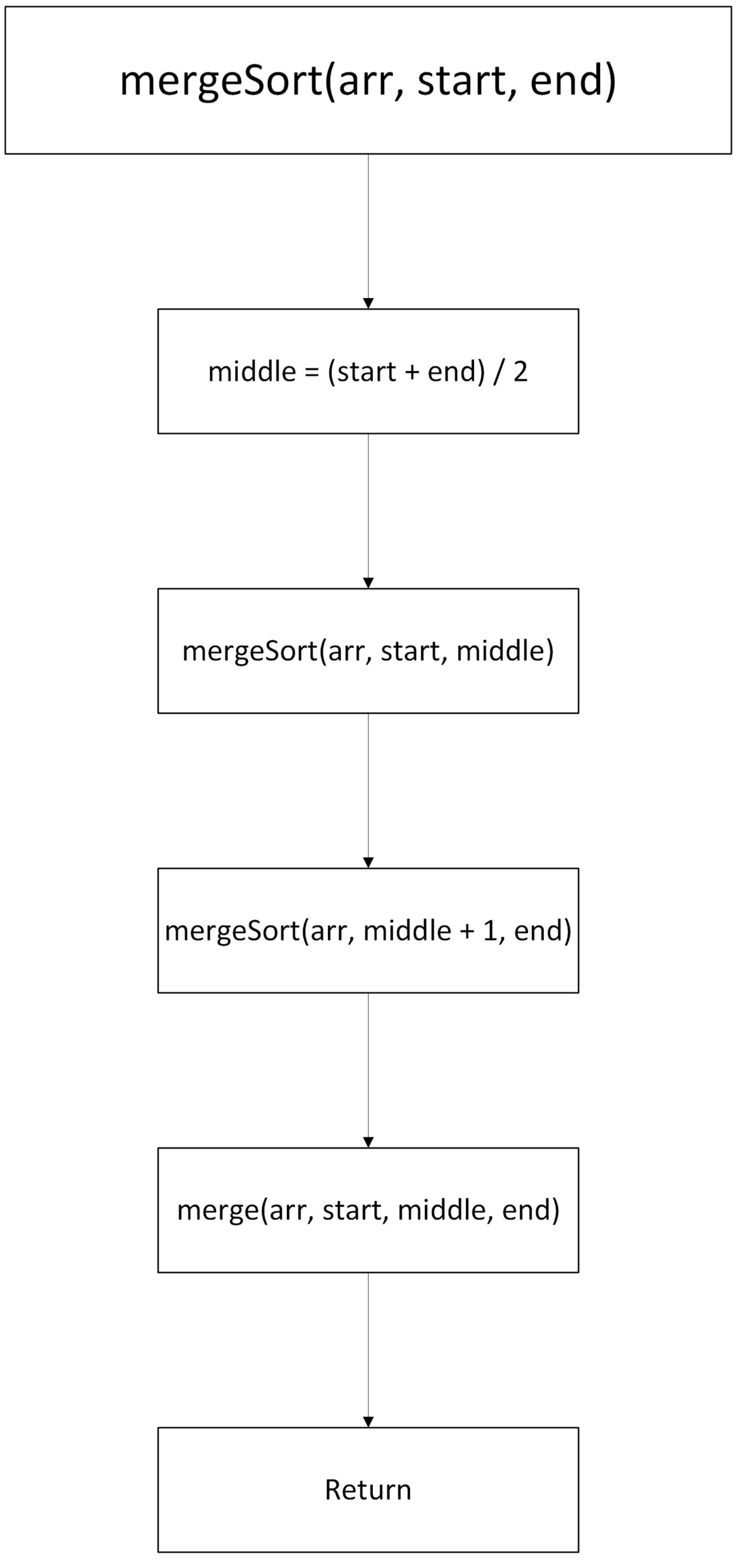
General Process

Main Algorithm

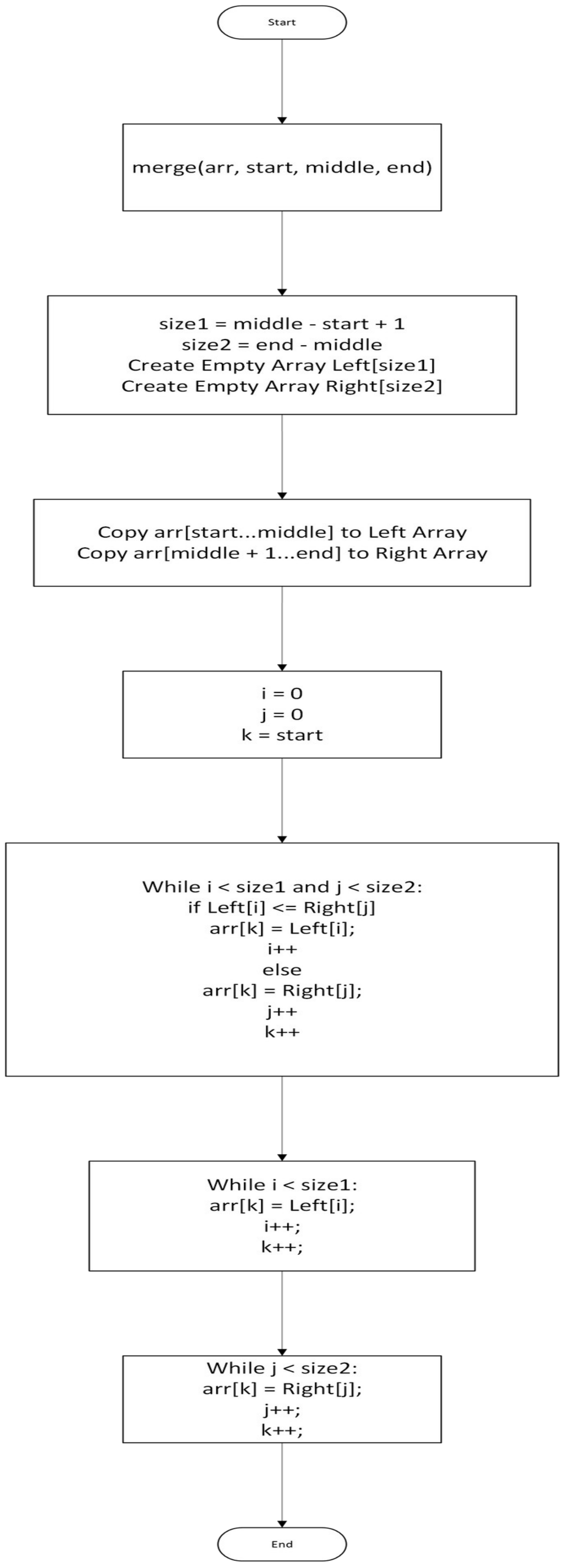


Start
Generate 100,000 Random Numbers Array
Array With 100,000 Random Numbers Unsorted
Sequential Search for 53
Sort The Array with Merge Sort
Count the total occurrenc es
Array with 100,000 Random Numbers Sorted
Search for 53 with Binary Search
Count the total occurrences

MergeSort

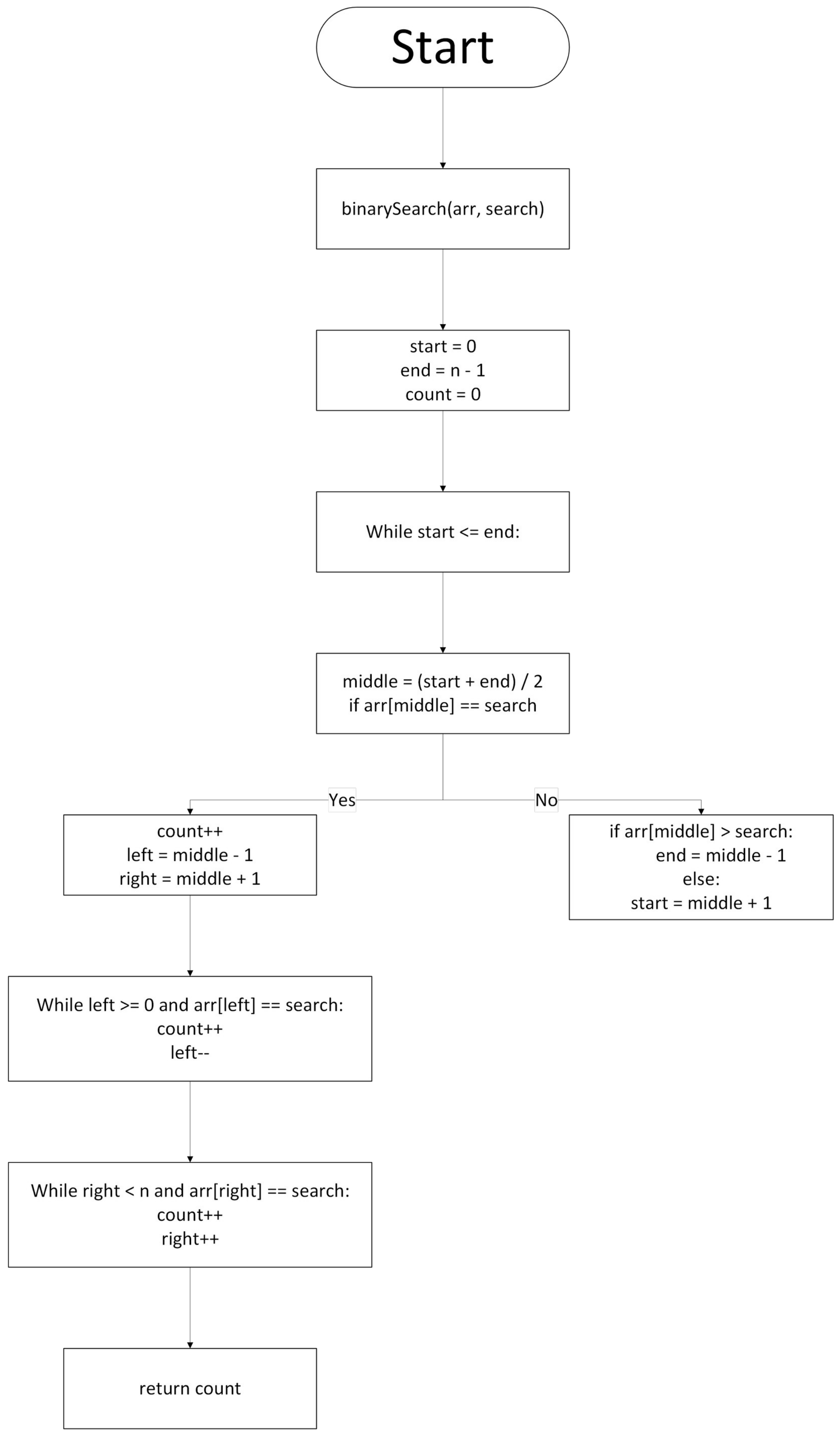

mergeSort(arr, start, end)
middle = (start + end) / 2
mergeSort(arr, start, middle)
mergeSort(arr, middle + 1, end)
merge(arr, start, middle, end)
Return

Merge



Start
merge(arr, start, middle, end)
size1 = middle - start + 1
size2 = end - middle
Create Empty Array Left[size1]
Create Empty Array Right[size2]
Copy arr[start...middle] to Left Array
Copy arr[middle + 1...end] to Right Array
i = 0
j = 0
k = start
While i < size1 and j < size2:
if Left[i] <= Right[j]
arr[k] = Left[i];
i++
else
arr[k] = Right[j];
j++
k++
While i < size1:
arr[k] = Left[i];
i++;
k++;
While j < size2:
arr[k] = Right[j];
j++;
k++;
End

BinarySearch



Results

The Runs were obtained on a PC running OS: CachyOS x85_64, Kernel: Linux 6.17.8-2-cachyos, with a AMD Ryzen 7 7800X3D 8 Core @ 5.05 GHz CPU, with 32 GB 6000MHz DDR5, GPU: AMD Radeon Rx 6950XT.

The program was executed three times. Each run generated 100,000 random integers between 0 and 125, searched for the value 53, sorted the array using Merge Sort, and then counted the occurrences of 53 again using a binary search–based counting method. The detailed timings for each run are shown below.

Timings.

Run 1

- Time to generate 100,000 numbers: 2.787705 ms
- Time to find first occurrence of 53 (sequential): 0.00053 ms
- Time to count occurrences of 53 (sequential): 0.20952 ms
- Time to sort array (Merge Sort): 10.934431 ms
- Time to count occurrences using binary search: 0.014049 ms
- Time for sequential counting only (step 3): 0.20952 ms
- Time for merge sort + binary search counting (steps 4 + 5): 10.94848 ms
- Total program time: 20.776414 ms
- Occurrences of 53: 806

Run 2

- Time to generate 100,000 numbers: 3.036265 ms
- Time to find first occurrence of 53 (sequential): 0.0002 ms
- Time to count occurrences of 53 (sequential): 0.19899 ms
- Time to sort array (Merge Sort): 10.887341 ms
- Time to count occurrences using binary search: 0.015569 ms

- Time for sequential counting only (step 3): 0.19899 ms
- Time for merge sort + binary search counting (steps 4 + 5): 10.90291 ms
- Total program time: 21.238071 ms
- Occurrences of 53: 788

Run 3

- Time to generate 100,000 numbers: 2.918074 ms
- Time to find first occurrence of 53 (sequential): 0.00023 ms
- Time to count occurrences of 53 (sequential): 0.200299 ms
- Time to sort array (Merge Sort): 10.798121 ms
- Time to count occurrences using binary search: 0.01603 ms
- Time for sequential counting only (step 3): 0.200299 ms
- Time for merge sort + binary search counting (steps 4 + 5): 10.814151 ms
- Total program time: 20.882112 ms
- Occurrences of 53: 811

Average Timings

Algorithm	**Order of Growth**	**Average Time (ms)**
Generation of Random Numbers	Θ(n)	≈ 2.91
Sequential search 53 first occurrence	Θ(n)	≈ 0.00032
Sequential search 53 Count occurrences	Θ(n)	≈ 0.203
Merge Sort	Θ(n log n)	≈ 10.87
Binary Search Count occurrences of 53	Θ(log n + k)	≈ 0.015

These results show that sequential counting is very fast for a single scan, while Merge Sort dominates the cost when sorting the array. Once the array is sorted, the binary search–based counting algorithm is extremely fast, but the sorting overhead makes the combination more expensive than a single sequential count when only one search is needed.

Conclusion

This project demonstrates the relationship between theoretical algorithm complexity and actual runtime performance. The sequential search and counting steps run in $\Theta(n)$ time and are very efficient for a single pass through the data. Merge Sort, with $\Theta(n \log n)$ time, introduces a noticeable cost to sort the array, but it enables the binary search–based counting algorithm to run in $\Theta(\log n + k)$ time.

The comparison between sequential counting only and merge sort + binary search counting shows that for a single query on the data, it is faster to use a simple linear scan. Sorting becomes more attractive when the same dataset is reused for many different searches. Overall, the project reinforced the importance of choosing appropriate algorithms based on the problem requirements and expected usage patterns.

The full source code for this project can be found on GitHub: https://github.com/Gojira34/Sorting-and-Searching-project.