Imam Abdulrahman Bin Faisal University

College of science and humanities studies - Jubail

جامعة الإمام عبدالرحمن بن فيصل
IMAM ABDULRAHMAN BIN FAISAL UNIVERSITY

Algorithm Report

Heap Sorting & Bubble Sorting

Name	Student ID
Batool Al Salim	2210003514
Aminah Alghazali	2210003639
Shaden al zahrani	2210003675
Fatima Alsalman	2210003663
Lujain Jarwan	2210003645

Instructor: Azza Ahmed Abdo Ali

Table of Contents:

Introduction:

This project discusses the bubble sorting and heap sorting algorithms. We combined both codes to create a hybrid sorting algorithm implemented in the Java programming language. We measured the time it takes for each sorting method for random numbers in the best case, average case, and worst case scenarios in milliseconds. Based on our previous lectures on algorithms, we understand that the time complexity of bubble sort in cases is Θ(n^2) due to its double loop, which is worse than heap sorting with a time complexity of Θ(n log n) achieved through a divide and conquer approach. We aim to compare our empirical results with the theoretical ones and see which one is the most efficient.

bubble sorting:

bubble sort is a basic algorithm for arranging numbers or other elements in the correct order. The method works by examining each set of adjacent elements in the string, from left to right, switching their positions if they are out of order. The algorithm then repeats this process until it can run through the entire string and find no two elements that need to be swapped. The advantage of bubble sort it is popular and easy to implement and doesn't use additional temporary storage. The disadvantage of bubble sort does not deal well with a list containing a huge number of items, not suitable for real-life applications. And like we study in previous lectures in algorithms bubble sort uses loop invariants for sorting algorithms They help in verifying correctness and understanding the behaviour of the code within loops.

bubble sorting algorithm:

```
// Bubble Sort

   public static int[] bubbleSort(int[] arr) {

      int n = arr.length;

      for (int i = 0; i < n - 1; i++) {

         for (int j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {

               int temp = arr[j];

               arr[j] = arr[j + 1];

               arr[j + 1] = temp  } } }

      return arr;  }
```

subsection for bubble sorting:

```
long startTimeBubble = System.currentTimeMillis();

      bubbleSort(arr.clone());

      long endTimeBubble = System.currentTimeMillis();

      float elapsedTimeBubble = ((float) (endTimeBubble - startTimeBubble));

      System.out.println("Bubble Sort - Time taken for " + order + " sorting: " +
elapsedTimeBubble + " Ms");
```

Heap sorting:

Heap sort uses divide and conquer algorithm. Is a comparison-based sorting technique based on Binary Heap data structure. First we find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements until all elements are completely sorted. The advantage of heap requires less memory compared to other data structures. Disadvantage requires dynamic memory allocation. As we took in algorithms heap sort use binary tree techniques for sorting known as binary heap first we build heapify for a max-heap, the parent nodes have values greater than or equal to their children. For a min heap, the parent nodes have values less than or equal to their children. heap construction process typically starts from the last non leaf node and iterates backward through the array, ensuring each subtree satisfies the heap property.

Heap sorting algorithm:

```
// Heap Sort
    public static int[] heapSort(int[] arr) {
        int n = arr.length;
for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);  }
        for (int i = n - 1; i > 0; i--) {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
            heapify(arr, i, 0);  }
        return arr;  }
    public static void heapify(int[] arr, int n, int i) {
        int largest = i;
        int left = 2 * i + 1;
     int right = 2 * i + 2;
        if (left < n && arr[left] > arr[largest]) {
            largest = left;   }
        if (right < n && arr[right] > arr[largest]) {
            largest = right;   }
        if (largest != i) {
```

```
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;
        heapify(arr, n, largest);   } }
```

subsection for heap sorting:

```
long startTimeHeap = System.currentTimeMillis();
    heapSort(arr.clone());
    long endTimeHeap = System.currentTimeMillis();
    float elapsedTimeHeap = ((float) (endTimeHeap - startTimeHeap));
    System.out.println("Heap Sort - Time taken for " + order + " sorting: " +
elapsedTimeHeap + " Ms");
```

Methodology:

For our project we used java programming language for executing codes for sorting and implantation codes for random numbers we used ascending order, random order, descending order. For finding the cases running time in best, and average ,and worst case. It will calculate the execution time in Millie seconds for every sorting and the start and the end time of the execution.

The user enters the number of elements and it would calculate the time it take for every algorithm implementing for 150000 elements from random numbers from 1000 to 100000 in milliseconds that the programmer enters before execution the code.

```
the minimum number: 1000
the maximum number: 100000
number of elements:
150000
```

implementation for bubble sorting:

We take ascending order for best case because it takes the less time in executing for sorting the array and random sort for the average case and descending sorting for the worst case.

```
Bubble Sort - Time taken for Best Case sorting: 2155.0 Ms

Bubble Sort - Time taken for Average Case sorting: 35391.0 Ms

Bubble Sort - Time taken for Worst Case sorting: 16427.0 Ms
```

implementation for Heap sorting:

```
Heap Sort - Time taken for Best Case sorting: 9.0 Ms

Heap Sort - Time taken for Average Case sorting: 16.0 Ms

Heap Sort - Time taken for Worst Case sorting: 9.0 Ms
```

Execution time:

Execution time can be different for every device and are interconnected in programming, but their relationship is influenced by multiple factors including algorithm choice, data size, optimizations, resource constraints, programming language, and hardware performance. and for the CPU we used for the implementation is intel i5 and the execution time can be less in higher CPU devices.

Start time for execution algorithm and run time of the code:

```
    long startTime = System.currentTimeMillis();
    System.out.println("Start time: " + startTime + " Ms");
```

```
--- exec:3.1.0:exec (default-cli) @ algorithms
Start time: 1702065031109 Ms
```

End time for execution all of the code:

```
    //prints end time of execution
    long endTime = System.currentTimeMillis();
    System.out.println("\nEnd time: " + endTime + " Ms");

    float elapsedTime = ((float) (endTime - startTime));
    System.out.println("Total execution time: " + elapsedTime + " Ms");
```

```
End time: 1702065057354 Ms
Total execution time: 26245.0 Ms
------------------------------------------
```

Hybrid algorithm:

We utilized Merge Sort to combine both the Bubble Sort and Merge Sort algorithms within our hybrid sorting approach. Merge Sort employs a divide and conquer methodology, making it a recursive algorithm similar to Heap Sort. We used Merge Sort due to its compatibility with the chosen algorithms and its efficiency in handling the sorting tasks.

The code that combine both sorting :

```
public static void hybridMergeSort(int[] arr, int left, int right) {
    if (right - left <= 10) {
        bubbleSortPartial(arr, left, right); // For small subarrays, use Bubble Sort
    } else {
        if (left < right) {
            int mid = left + (right - left) / 2;
            hybridMergeSort(arr, left, right:mid); // Recursively sort the left half
            hybridMergeSort(arr, mid + 1, right); // Recursively sort the right hali
            merge(arr, left, mid, right); // Merge the sorted halves
        }
    }
}

public static void bubbleSortPartial(int[] arr, int left, int right) {
    for (int i = left; i <= right; i++) {
        for (int j = left; j <= right - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
```

```
public static void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int[] L = new int[n1];
    int[] R = new int[n2];
    for (int i = 0; i < n1; ++i) {
        L[i] = arr[left + i];
    }
    for (int j = 0; j < n2; ++j) {
        R[j] = arr[mid + 1 + j];
    }
    int i = 0, j = 0;
    int k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
```

implementation for hybrid sorting:

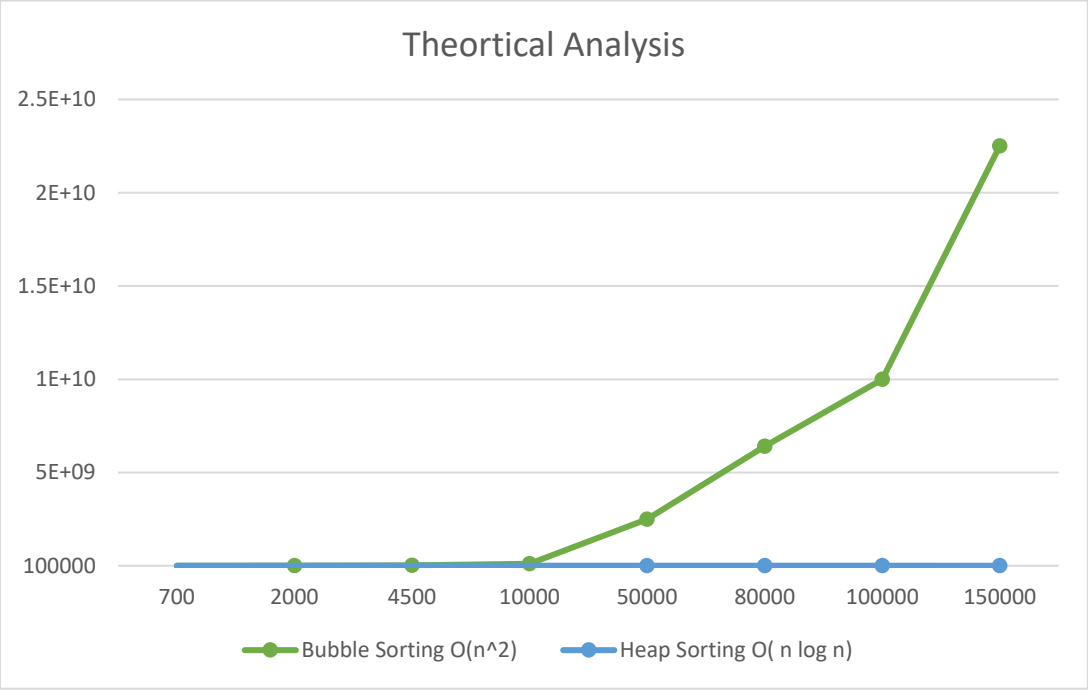
```
Hybrid Merge Sort - Time taken for Best Case sorting: 17.0 Ms

Hybrid Merge Sort - Time taken for Average Case sorting: 26.0 Ms

Hybrid Merge Sort - Time taken for Worst Case sorting: 10.0 Ms
```

Excel implementation :

We did theoretical and empirical analysis that we calculate using time complexity in milliseconds for bubble sort and heap sort and hybrid sorting.

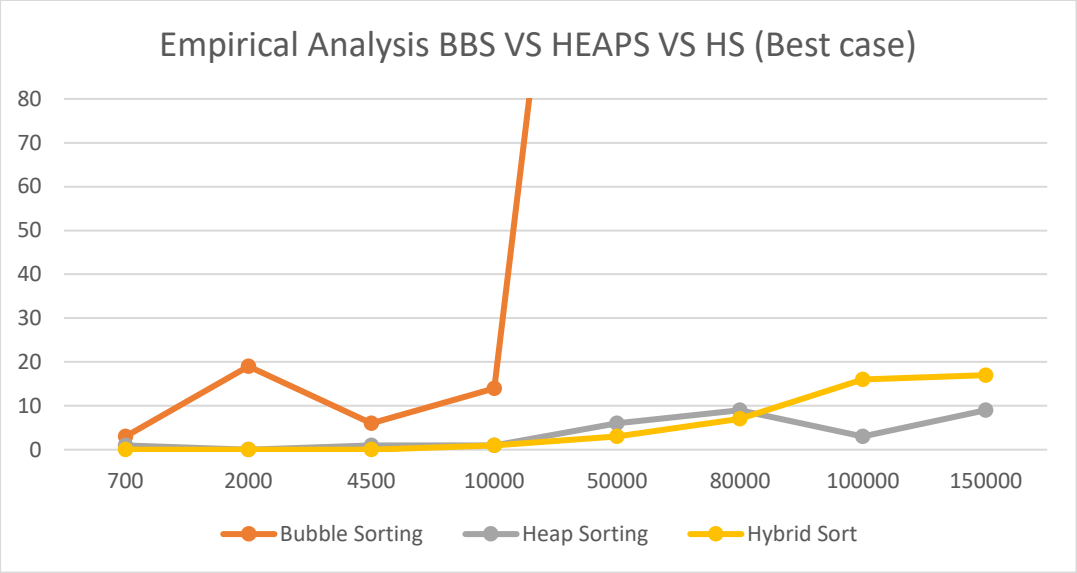
theoretical Analysis:

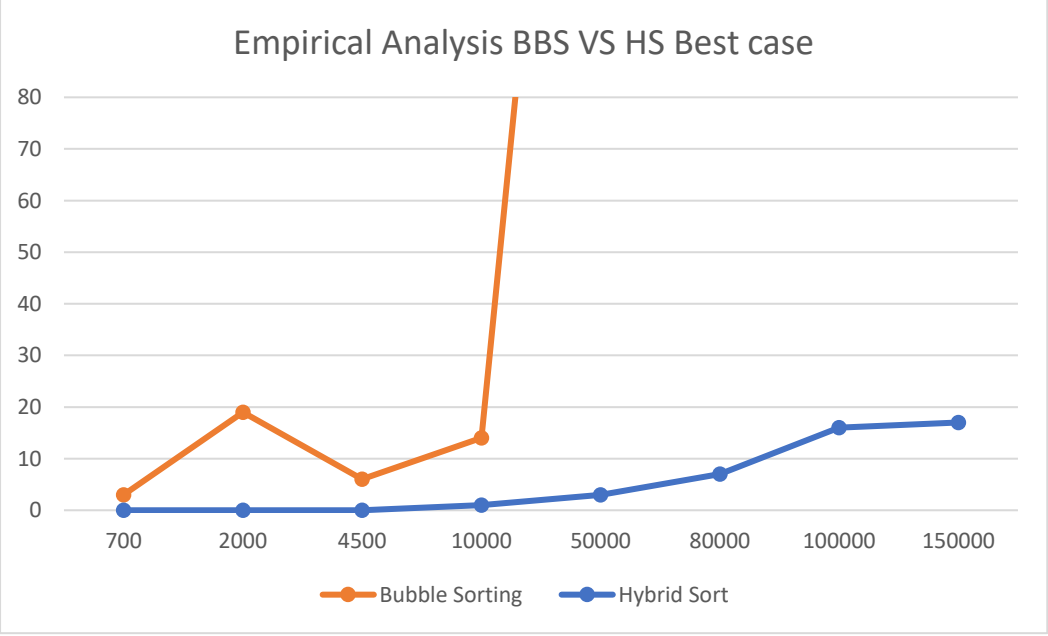
theoretical Analysis Time complexity		
n	Bubble Sorting O(n^2)	Heap Sorting O(n log n)
700	70000	1991.56
2000	4000000	6602.059
4500	20250000	16439.456
10000	100000000	40000
50000	2500000000	234948.5
80000	6400000000	392247.199
100000	10000000000	500000
150000	22500000000	776413.6889

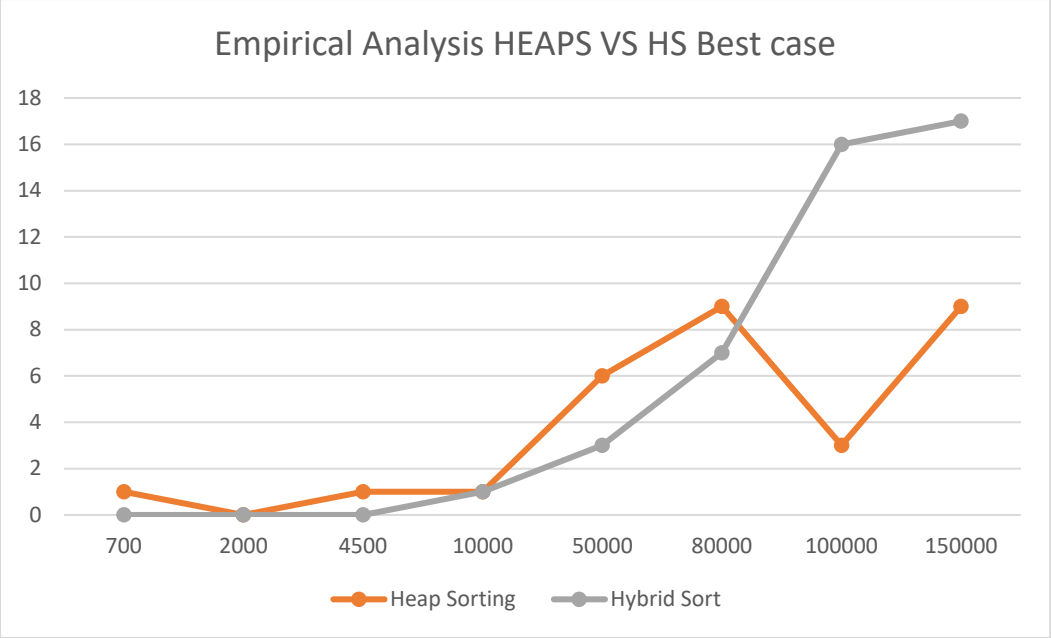


Empirical Analysis Best case:

Empirical Analysis Best case			
n	Bubble Sorting	Heap Sorting	Hybrid Sort
700	3	1	0
2000	19	0	0
4500	6	1	0
10000	14	1	1
50000	246	6	3
80000	598	9	7
100000	941	3	16
150000	2155	9	17



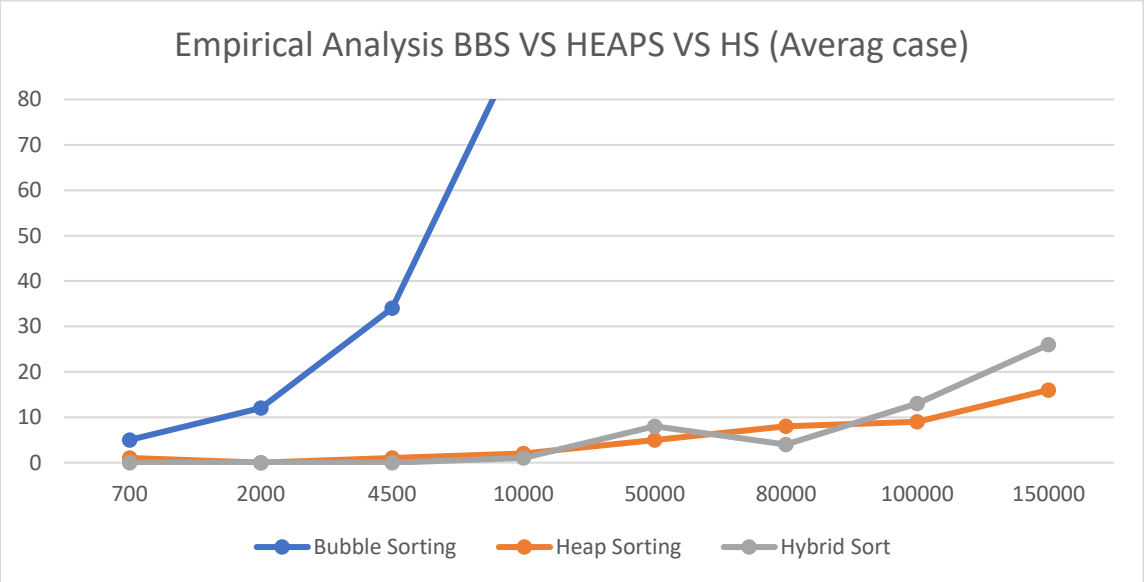


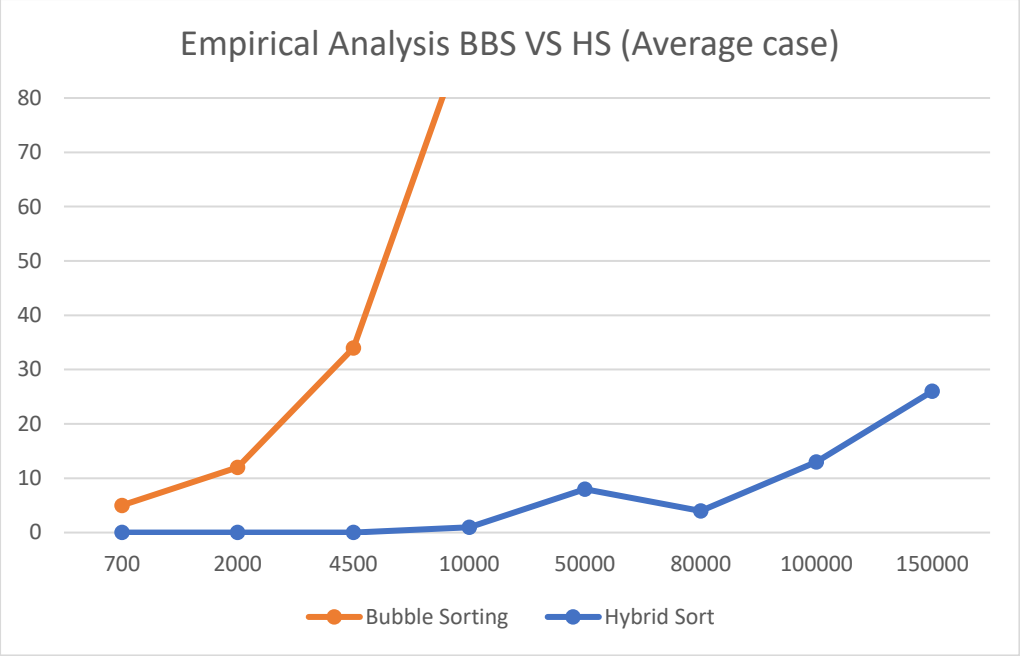


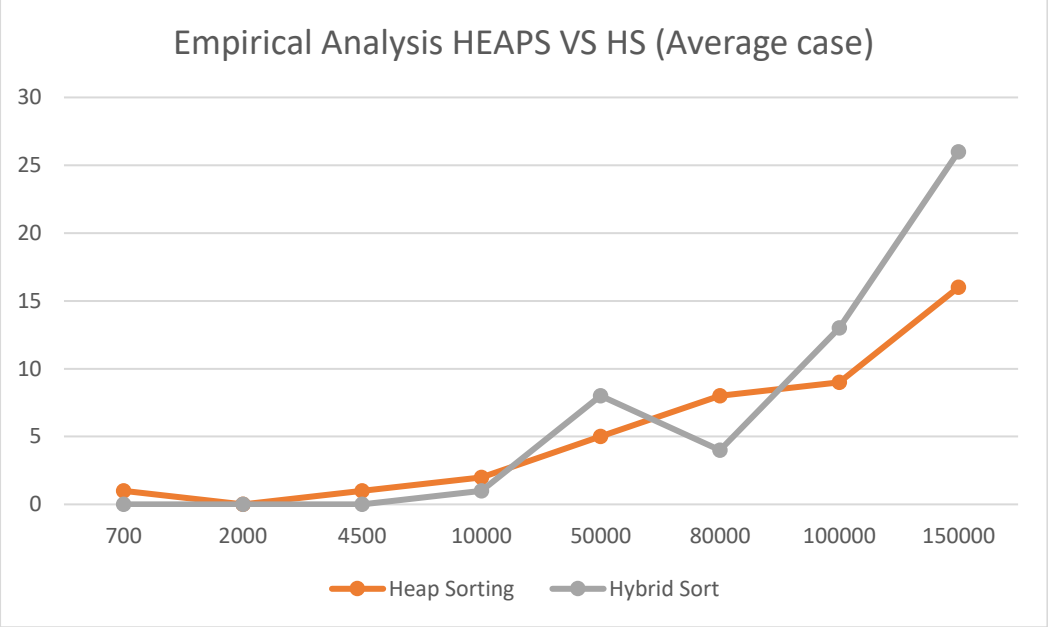
Empirical Analysis Average case:

Empirical Analysis Average case

n	Bubble Sorting	Heap Sorting	Hybrid Sort
700	5	1	0
2000	12	0	0
4500	34	1	0
10000	94	2	1
50000	3356	5	8
80000	8763	8	4
100000	13953	9	13
150000	35391	16	26



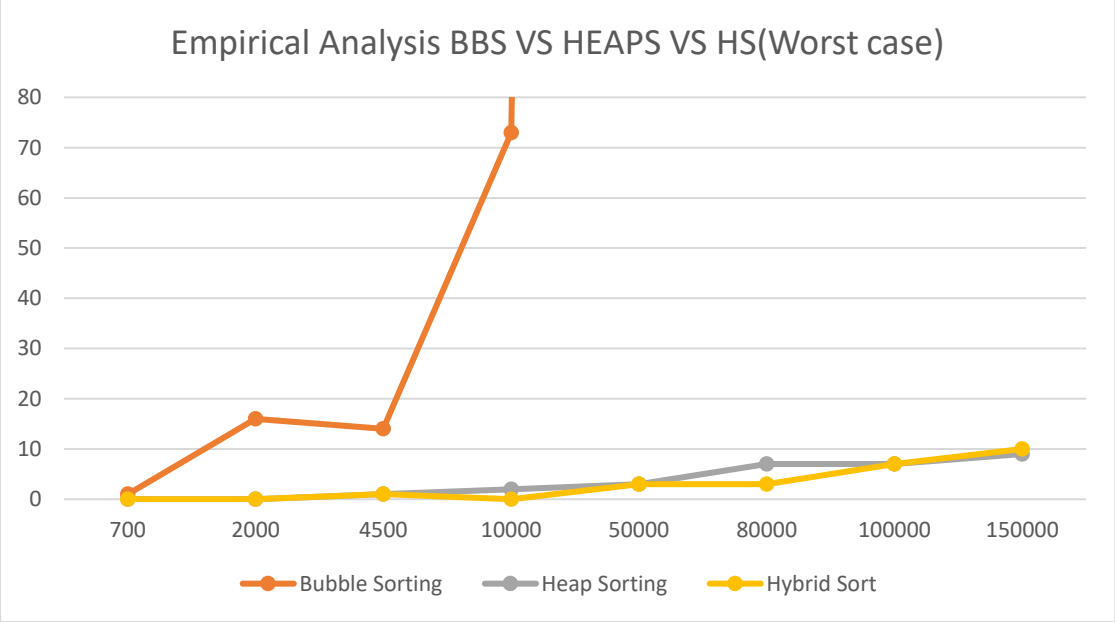

Empirical Analysis BBS VS HS (Average case)
80
70
60
50
40
30
20
10
0
700
2000
4500
10000
50000
80000
100000
150000
Bubble Sorting
Hybrid Sort

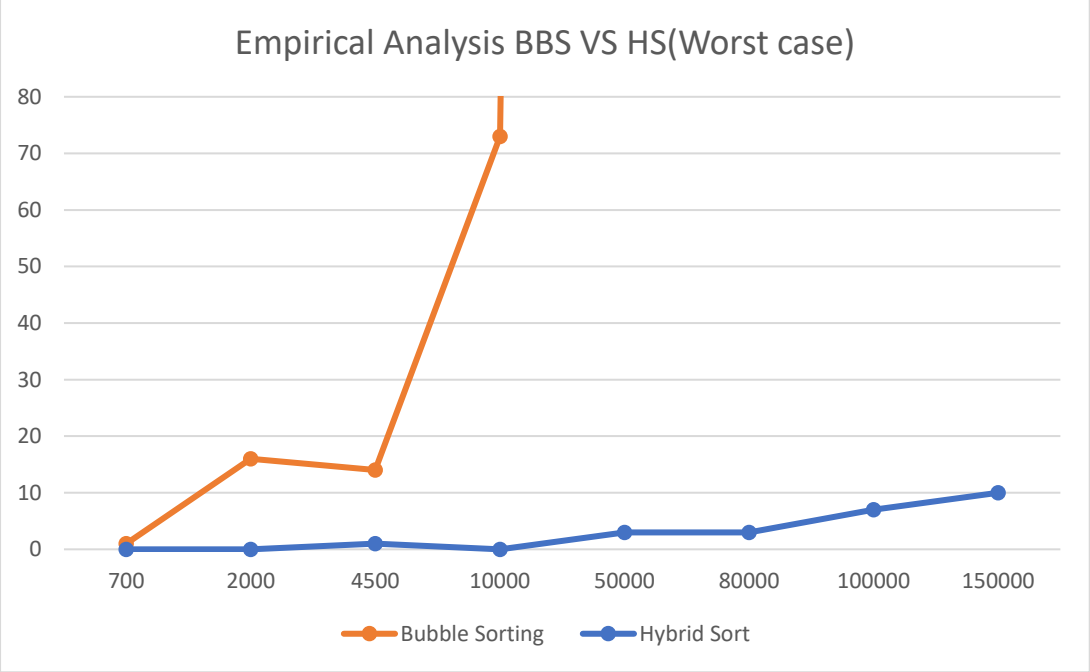

Empirical Analysis HEAPS VS HS (Average case)
30
25
20
15
10
5
0
700
2000
4500
10000
50000
80000
100000
150000
Heap Sorting
Hybrid Sort

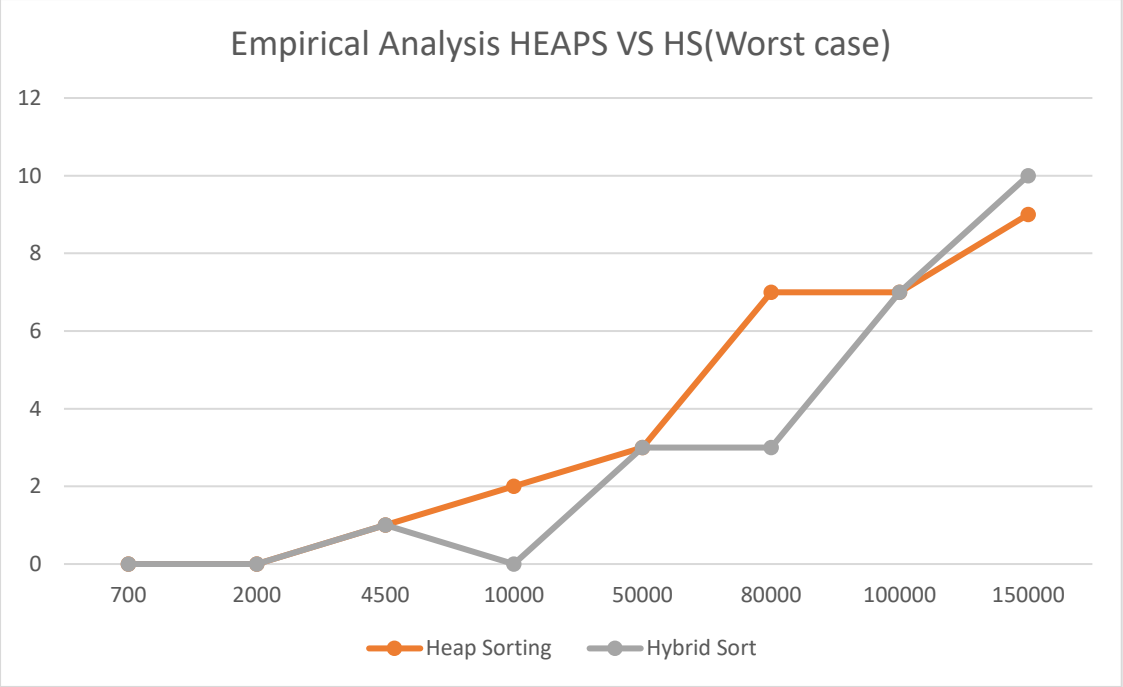
Empirical Analysis Worst case:

Empirical Analysis Worst case

n	Bubble Sorting	Heap Sorting	Hybrid Sort
700	1	0	0
2000	16	0	0
4500	14	1	1
10000	73	2	0
50000	1578	3	3
80000	3924	7	3
100000	6204	7	7
150000	16427	9	10






Empirical Analysis HEAPS VS HS(Worst case)
12
10
8
6
4
2
0
700
2000
4500
10000
50000
80000
100000
150000
Heap Sorting
Hybrid Sort

Conclusion:

In conclusion, we can say that heap is the most efficient among all the sorting algorithms, displaying consistent execution times across different scenarios. And hybrid sort ranking the second. In conclusion T(n) of hybrid sort switching between n log n and n^2 for bubble sorting. with bubble sorting ranking last. It's evident that there's a substantial difference in time complexity between the best-case scenario and the worst/average cases. These observations align with the theoretical analyses we conducted.