Name: Yashwant Chandrakant Bhosale, **MIS**: 612303039, **Div**: SY Comp Div 1 S2

Linear Search and Binary Search Performance Analysis Report

1. What is Linear Search Algorithm?

It works by visiting each element and checking if the element is equal to the key or the target element. If the current element is equal to the target element then we return the index of the first occurrence of the element. Otherwise, if we do not find the element we return -1 to indicate that the element is not present in provided array.
It is very simple and traditional approach.

Time Complexity: O(n)
Best case: The best case is fairly simple. If the key is the first element itself then we do not need to traverse the entire array and time complexity becomes **O(1).**
Worst case: The worst case is also simple to understand. The worst case in this algorithm will be when the element is at the last position of the array and we have to traverse the whole array in order to find the element. So if the array contains **N** elements we have to do N comparisons in order to confirm that the key we are looking for is at the last position. So the worst case time complexity of linear search is **O(N).**

Space Complexity: 0(1)
The space complexity for this algorithm is **O(1)** as only variables for key, iterating variable, etc. are used which are constant so no extra variables or space is used.

2. Pseudo Code for Linear Search

```
linear_search(array, key)
    for each element in array
        if(item == value)
            return item’s index
        end if
    end for
```

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include "helper_functions.h"

int linear_search(int *array, int size, int key) {
    for (int i = 0; i < size; i++) {
        if (array[i] == key) {
            return i;
        }
    }
    return -1;
}

int main() {
    int size, key;
    printf("Enter the size of the array: ");
    scanf("%d", &size);
    int *array = (int *)malloc(size * sizeof(int));
    printf("Enter the elements of the array: ");
    read_array(array, size);
    printf("Enter the key to search: ");
    scanf("%d", &key);
    int index = linear_search(array, size, key);
    if (index == -1) {
        printf("Key not found in the array\n");
    } else {
        printf("Key found at index %d\n", index);
    }
    free(array);
    return 0;
}
```

2. What is Binary Search?
Binary Search is an efficient algorithm used to find the position of a target value within a **Sorted array.** It works by repeatedly dividing the search interval in half starting with the **middle element.** If the middle element matches the target, the search is successful. If the target is smaller than the middle element, the search continues in the left half of the array; if it is larger, the search proceeds in the right half. This process is repeated until the target is found or the interval becomes empty, indicating the target is not in the array. It can be implemented **recursively** as well as **iteratively.**

Time Complexity:
The time complexity of Binary Search varies significantly depending on whether the array is **sorted** or **unsorted.**

For a sorted array:
1. Best Case: **O(1)**
The best case occurs when the middle element of the array is the target alue requiring onlly one comparison.

2. Worst Case: **O(logn)**
The worst case happens when we have to traverse whole array to find out that **the target is not present in the array.** So we need to divide array repeatedly until we reach the leaf node or single element.

For an unsorted array:
Time Complexity becomes: **O(n logn)**
Binary search **Cannot directly work on unsorted arrays.** So we first have to convert unsorted array to sorted one using one of the efficient algorithms like **merge sort** or **quick sort.** So it takes time complexity **O(n logn)**. Then, Worst case time complexity of binary search which is **O(logn).** So overall time complexity becomes of the order **O(n logn).**

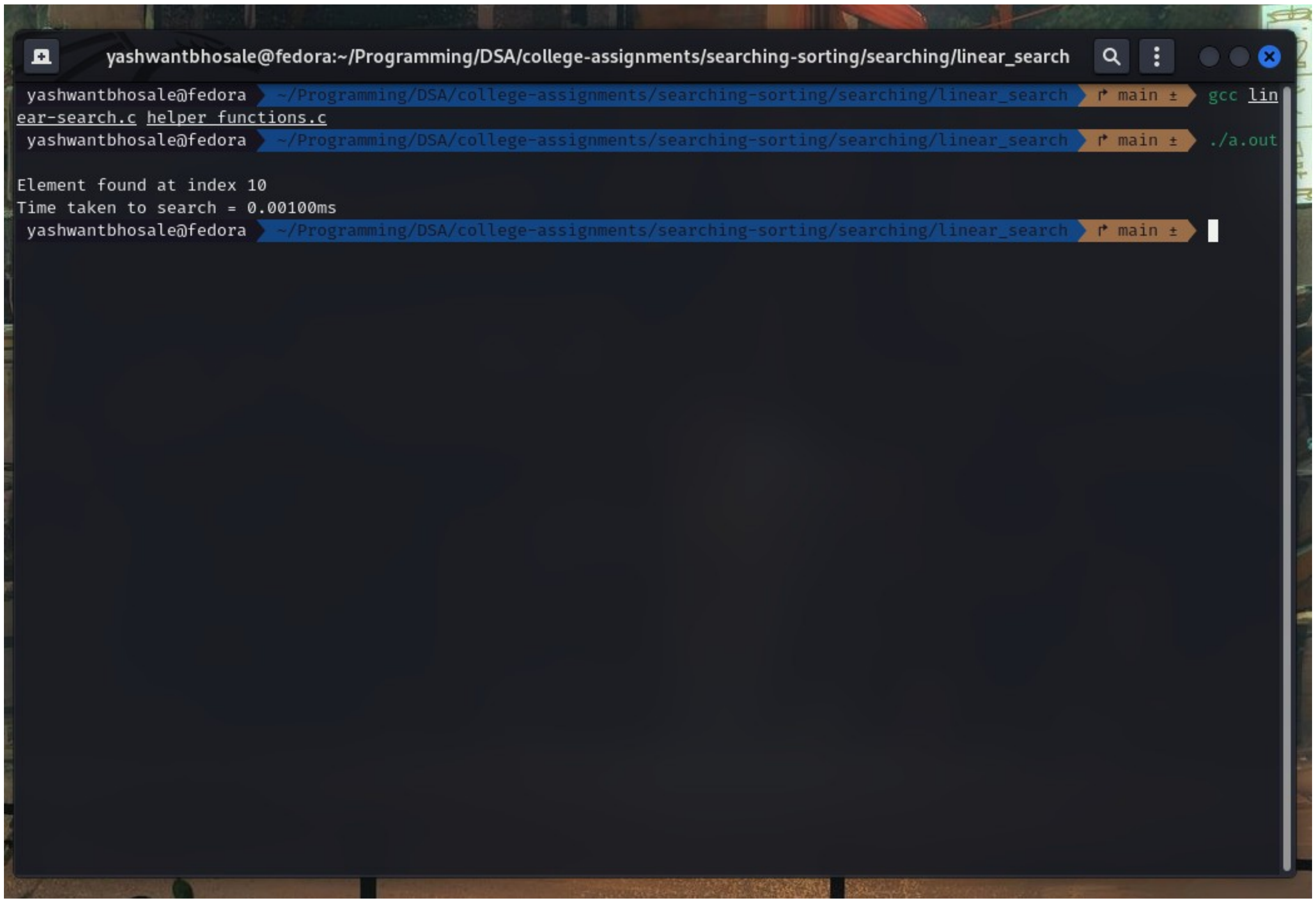
```
binary_search(array, key)
   low = 0
   high = length(array) - 1

   while low ≤ high
       mid = low + (high - low) // 2
       if array[mid] == key
           return mid
       else if array[mid] < key
           low = mid + 1
       else
           high = mid - 1
       end if
   end while

   return -1  // Key not found
```

Code:
```
#include <stdio.h>
#include <stdlib.h>
#include "helper_functions.h"

int binary_search(int *array, int size, int key) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (array[mid] == key) {
            return mid;
        } else if (array[mid] < key) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
}

// recursive binary search
int binary_search_v2(int *array, int left, int right, int key) {
    if (left <= right) {
        int mid = left + (right - left) / 2;

        if (array[mid] == key) {
            return mid;
        } else if (array[mid] < key) {
            return binary_search_v2(array, mid + 1, right, key);
        } else {
            return binary_search_v2(array, left, mid - 1, key);
        }
    }
    return -1;
}
```

```c
int main() {
    int size, key;
    printf("Enter the size of the array: ");
    scanf("%d", &size);
    int *array = (int *)malloc(size * sizeof(int));
    printf("Enter the elements of the array: ");
    read_array(array, size);
    printf("Enter the key to search: ");
    scanf("%d", &key);
    int index = binary_search(array, size, key);
    if (index == -1) {
        printf("Key not found in the array\n");
    } else {
        printf("Key found at index %d\n", index);
    }
    free(array);
    return 0;
}
```

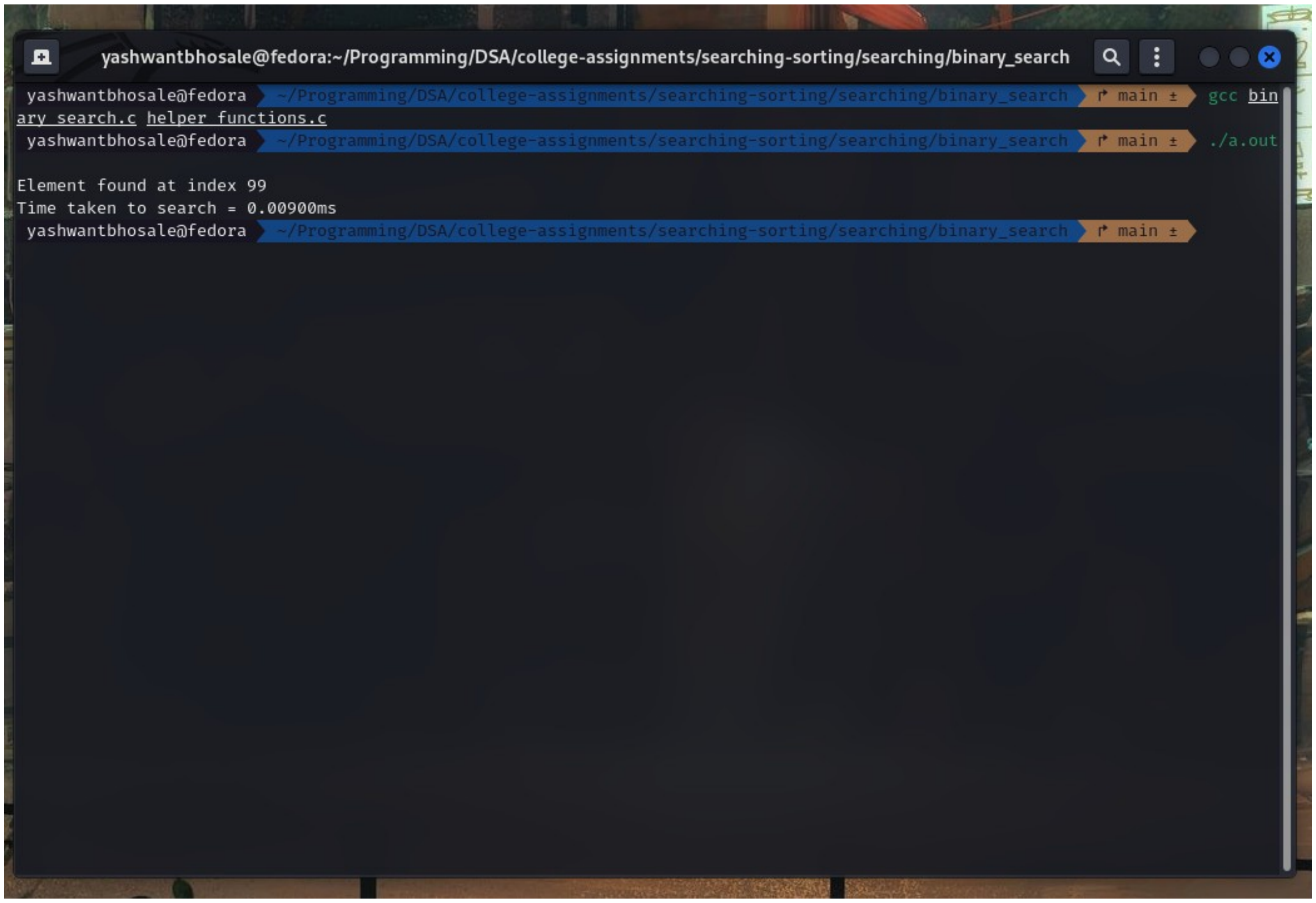
Linear Search	**Binary Search**
Data need not be sorted	Data needs to be sorted
Time Complexity: O(n)	Time complexity: O(log n)
It is comparatively slow.	It is comparatively fast.
It works by visiting each element and comparing with target.	It works by repeatedly dividing array and comparing target and middle element.

```
Array =[
34,74,15,55,40,36,84,11,83,18,100,27,75,8,85,66,70,64,12,58,96,43,93,68,7
2,81,90,62,31,65,59,89,48,3,24,52,50,13,57,33,76,26,87,79,29,23,60,78,37,
99,25,95,98,86,1,4,30,53,56,71,77,73,51,54,6,44,21,42,92,19,17,14,39,9,41
,69,5,28,22,20,16,97,2,7,91,38,80,88,45,46,61,32,67,82,94,47,49,63,35,
10
]
Key = 100

output:
Linear Search: time taken = 0.00100ms
```



```
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/searching-sorting/searching/linear_search
 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/searching/linear_search  ↱ main ±  gcc linear-search.c helper_functions.c
 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/searching/linear_search  ↱ main ±  ./a.out

Element found at index 10
Time taken to search = 0.00100ms
 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/searching/linear_search  ↱ main ± 
```

Binary Search: **time taken = 0.00900ms**


```
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/searching-sorting/searching/binary_search
 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/searching/binary_search  ↱ main ±  gcc binary_search.c helper_functions.c
 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/searching/binary_search  ↱ main ±  ./a.out

Element found at index 99
Time taken to search = 0.00900ms
 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/searching/binary_search  ↱ main ±
```

Conclusion:
1. time taken for binary search is more because the data is unsorted.
2. hence data was sorted using inbuilt qsort function and then searched using binary search.

Code for reference:
linear_search.c

```c
/*
    LINEAR SEARCH ALGORITHM

    Time Complexity: O(n)
    Space Complexity: O(1)
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "helper_functions.h"

int linear_search(int *array, int size, int key) {
    for (int i = 0; i < size; i++) {
        if (array[i] == key) {
            return i;
        }
    }
    return -1;
}

/*
int array[] = {

34,74,15,55,40,36,84,11,83,18,100,27,75,8,85,66,70,64,12,58,96,43,93,68,7
2,81,90,62,31,65,59,89,48,3,24,52,50,13,57,33,76,26,87,79,29,23,60,78,37,
99,25,95,98,86,1,4,30,53,56,71,77,73,51,54,6,44,21,42,92,19,17,14,39,9,41
,69,5,28,22,20,16,97,2,7,91,38,80,88,45,46,61,32,67,82,94,47,49,63,35,10
};
*/

int array[] = {
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
    11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
    21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
    31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
    41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
    51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
    61, 62, 63, 64, 65, 66, 67, 68, 69, 70,
    71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
    81, 82, 83, 84, 85, 86, 87, 88, 89, 90,
    91, 92, 93, 94, 95, 96, 97, 98, 99, 100
};


int main() {
    int len = sizeof(array) / sizeof(array[0]);
    int key = 88;

    clock_t start, end;

    start = clock();
    int index = linear_search(array, len, key);
    end = clock();

    double time_taken = ((double)(end - start)) * 1000 / CLOCKS_PER_SEC;

    if (index ≠ -1) {
        printf("Element found at index %d\n", index);
        printf("Time taken to search = %.5lfms\n", time_taken);
    } else {
        printf("Element not found\n");
    }

    return 0;
}
```

```c
binary_search.c
/*
    BINARY SEARCH ALGORITHM

    If the array is sorted:
    Time Complexity: O(log n)
    Space Complexity: O(1)

*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "helper_functions.h"

// Function to compare two integers for qsort
int compare_int(const void *a, const void *b) {
    return (*(int*)a - *(int*)b);
}

int binary_search(int *array, int size, int key) {
    if (size ≤ 0) return -1;

    int left = 0, right = size - 1;
    while (left ≤ right) {
        int mid = left + (right - left) / 2;

        if (array[mid] == key) {
            return mid; // Key found
        } else if (array[mid] < key) {
            left = mid + 1; // Search in the right half
        } else {
            right = mid - 1; // Search in the left half
        }
    }
    return -1; // Key not found
}


// recursive binary search
int binary_search_v2(int *array, int left, int right, int key) {
    if (left ⩽ right) {
        int mid = left + (right - left) / 2;

        if (array[mid] == key) {
            return mid;
        } else if (array[mid] < key) {
            return binary_search_v2(array, mid + 1, right, key);
        } else {
            return binary_search_v2(array, left, mid - 1, key);
        }
    }
    return -1;
}


int array[] = {

34,74,15,55,40,36,84,11,83,18,100,27,75,8,85,66,70,64,12,58,96,43,93,68,7
2,81,90,62,31,65,59,89,48,3,24,52,50,13,57,33,76,26,87,79,29,23,60,78,37,
99,25,95,98,86,1,4,30,53,56,71,77,73,51,54,6,44,21,42,92,19,17,14,39,9,41
,69,5,28,22,20,16,97,2,7,91,38,80,88,45,46,61,32,67,82,94,47,49,63,35,10
};


int main() {
    int len = sizeof(array) / sizeof(array[0]);
    int key = 100;

    qsort(array, len, sizeof(int), compare_int);

    clock_t start, end;

    start = clock();
    int index = binary_search(array, len, key);
    end = clock();

    double time_taken = ((double)(end - start)) * 1000 / CLOCKS_PER_SEC;
    if (index ≠ -1) {
        printf("Element found at index %d\n", index);
        printf("Time taken to search = %.5lfms\n", time_taken);
    } else {
        printf("Element not found\n");
    }
    return 0;
}
```

Name: Yashwant Chandrakant Bhosale, **MIS**: 612303039 **Div**: SY Comp Div 1, **Batch** S2

Bubble Sort Algorithm

```
void bubble_sort_v2(int *arr, int len) {
    for (int i = 0; i < len; i++) {
        int swapped = 0; // Flag to track if any swaps happened in this pass
        for (int j = 0; j < len - (i + 1); j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr, j, j + 1);
                swapped = 1; // Mark as swapped
            }
        }
        if (!swapped) {
            break; // Array is sorted, exit early
        }
    }
}
```

1. Best case Time Complexity for Bubble Sort : **O(N)**

The best case in bubble sort occurs when array is sorted and when **there are no swaps in first pass** we understand that the array is sorted. So, the number of comparisons required are N-1 and number of swaps required are **N-1.**
So, the best case time complexity for Bubble sort is **O(N).**

2. Worst Case Time Complexity for Bubble Sort: **O(N^2)**

The worst case in bubble sort occurs when array is sorted in reverse order i.e. increasing order when we are sorting in decreasing order or decreasing order when we are sorting in increasing order.
In worst case the Total number of passes required are **N-1** (After each pass the largest element in the selected part of the array gets placed in the last position). So if we place N-1 elements to their correct position last element will automatically be placed at the correct position i.e. first position.
So the total number of comparisons will be:
N-1 comparisons and N-1 swaps ---- 1st pass
N-2 comparisons and N-2 swaps ---- 2nd pass
.
.
.
.
1 comparison and 1 swap ---- (N-1)th pass

---

so, total comparisons
= (N-1) + (N-2) + (N-3) + ……. + 1

$$= \frac{N\cdot(N-1)}{2}$$

3. Average case time complexity
At first pass (n-1) comparisons are made and (n-2) in second and so on.
To total number of comparisons are $\frac{N \cdot (N-1)}{2}$. as demonstrated above. On average we may say that about half pairs are out of order so we have to do those many swaps. So its time complexity is also about O(n^2). So as complexity of comparisons is O(n^2) and also swaps is O(n^2) the overall time complexity is O(n^2).
Even though the exact number of swaps depends on the input, the number of comparisons remains consistent and dominates the time complexity.
The nested loop structure ensures that every pair of elements is considered, O(n^2) time complexity.

Insertion Sort Algorithm

```
void insert(int key, int arr[], int j) {
    // insert key into the sorted subarray arr[0] to arr[j-1]
    // by shifting elements to the right until the correct position is found

    while (j ⩾ 0 && arr[j] > key) {
        arr[j+1] = arr[j];
        j = j-1;
    }
    arr[j+1] = key;
}

void insertion_sort(int arr[], int n) {
    // sort a[0] to a[n-1] into nondecreasing order

    for (int i = 1; i < n; i++) {
        // insert a[i] into the sorted subarray a[0] to a[i-1]
        int key = arr[i];
        int j = i-1;
        insert(key, arr, j);
    }
}
```

1. Best case Time Complexity for Bubble Sort : **O(N)**

The best case is when array is already sorted and we have to traverse array once in order to confirm. So the best case time complexity is **O(N).**

Line	No of times	Cost	total
`for i = 0 to n`	n	C_1	n × C_1
`    key = A[i]`	n-1	C_2	(n-1) × C_2
`    j = i -1`	n-1	C_3	(n-1) × C_3
`    While j ⩾ 0 and A[j] > key`	$\sum_{j=1}^{n-1} t_j$	C_4	$(\sum_{j=1}^{n-1} t_j)$ × C_4
`        A[j+1] = A[j]`	$\sum_{j=1}^{n-1} t_{(j-1)}$	C_5	$(\sum_{j=1}^{n-1} t_{(j-1)})$ × C_5
`        j = j-1`	$\sum_{j=1}^{n-1} t_{(j-1)}$	C_6	$\sum_{j=1}^{n-1} t_{(j-1)}$ × C_6
`    End while`			
`    A[j+1] = key`	n-1	C_7	(n-1) × C_7
`End for`			

t_j = time taken for j comparisons, t_{j-1} = time taken for j-1 comparisons, for simplicity let us assume t_j = j i.e each comparison takes 1 unit time

Total time

$$= (n \times C_1) + [(n-1) \times C_2] + [(n-1) \times C_3] + [(\sum_{j=1}^{n-1} t_j) \times C_4] + [(\sum_{j=1}^{n-1} t_{(j-1)}) \times C_5] + [\sum_{j=1}^{n-1} t_{(j-1)} \times C_6] + [(n-1) \times C_7]$$

$$= (n \times C_1) + [(n-1) \times C_2] + [(n-1) \times C_3] + [(\frac{n\cdot(n-1)}{2}) \times C_4] + [(C_5 + C_6) \times (\frac{n\cdot(n-1)}{2}) - 1] + [(n-1) \times C_7]$$

Which on further simplification gives dominating factor of $\mathbf{n^2}$**.** So the worst case time complexity of insertion sort is **O(n^2).**

Average Case Time Complexity of insertion sort: O(n^2)

Insertion sort builds the sorted portion of the array one element at a time. At every step current element is compared to the sorted portion and inserted into the correct position. The number of comparisons or shifts depends on how unsorted the array is. For an average case we may say that a given element at index i has to be compared with i/2 elements.
(The way we can understand this is by looking at the fact that for some elements comparisons are going to be almost i and for some elements they are going to be less than half so on an average we may consider there are going to be i/2 comparisons at each step.)
so total comparisons in the end are $\sum_{i=1}^{n-1} (\frac{i}{2}) = \frac{n\cdot(n-1)}{4}$. So the average case time complexity of insertion sort is **O(n^2).**

Selection Sort

Selection Sort algorithm works by repeatedly selecting the smallest element from unsorted portion of the array and swapping it with the first unsorted element.

```
void selection_sort(int *arr, int len) {
    for (int i=0; i < len; i++){

        int min_index = i; // Index of the minimum element

        for (int j = i+1; j < len; j++) {
            if(arr[j] < arr[min_index]) {
                min_index = j;
            }
        }

        swap(arr, i, min_index);
    }
    return;
}
```

1. Best Case Time Complexity : **O(N^2)**

The best case is when the array is already sorted and we have to traverse to find smallest element already at the starting position. Here, the time complexity of best case and worst case is almost same (in terms of order at least). The difference is in the swaps. In worst case we have to keep swapping elements until array is sorted in best case we don't have to swap elements as array is already sorted.

2. Worst Case Time Complexity : **O(N^2)**

Line	No. of times	Cost	total
`for i = 0 to n-1`	n	C_1	n $\times C_1$
`    minIndex = i`	n-1	C_2	(n-1) $\times C_2$
`    for j = i+1 to n-1`	$\sum_{j=1}^{n-2}(n-1-i)$	C_3	$(\sum_{j=1}^{n-2}(n-1-i))\times C_3$
`        if array[j] < array[minIndex]`	$\sum_{j=1}^{n-2}(n-1-i)$	C_4	$(\sum_{j=1}^{n-2}(n-1-i))\times C_4$
`            minIndex = j`			
`        end of if`			
`    end of for`			
`    Swap(array[i], array[minIndex])`	n	C_5	n $\times C_5$
`end of for`			

Total time

$$= (n \times C_1) + [(n-1) \times C_2] + [(\sum_{j=1}^{n-2} (n-1-i)) \times C_3] + [(\sum_{j=1}^{n-2} (n-1-i)) \times C_4] + (n \times C_5)$$

$$\because \sum_{j=1}^{n-2} (n-1-i) = \frac{n \cdot (n-1)}{2}$$

The dominant term in above expression is **n^2**, therefore the overall time complexity of selection sort algorithm is **$O(n^2)$.**

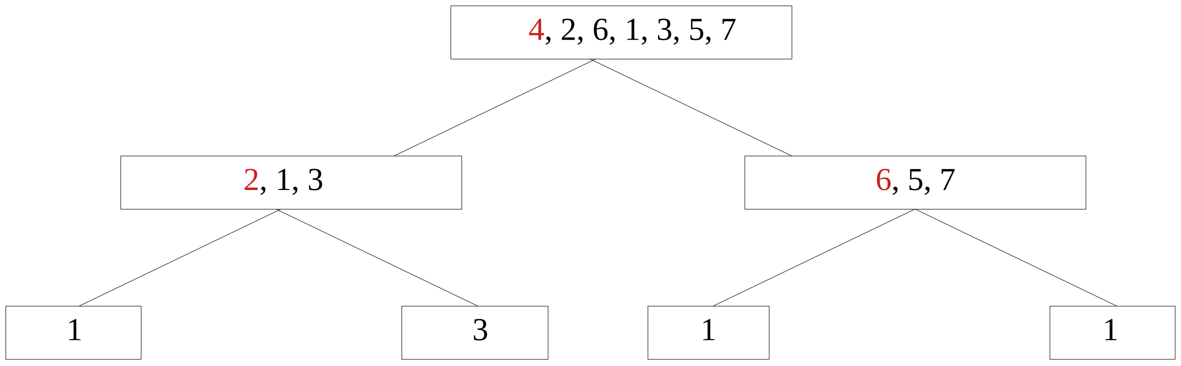
Quick Sort

```
void quick_sort(int *arr, int left, int right) {
     // base case
     if (left ⩾ right) {
          return;
     }

     int pivot = arr[(left + right) / 2];// pick the middle element as
                                           pivot
     int i = left, j = right;

     /*
         quick sort works by partitioning the array into two subarrays
         such that all elements in the left subarray are less than the
         pivot
         and all elements in the right subarray are greater than the
         pivot
     */

     while (i ⩽ j) {
          // find the first element from the left that is greater than
          // the pivot
          while (arr[i] < pivot) {
               i++;
          }

          // find the first element from the right that is less than the
          // pivot
          while (arr[j] > pivot) {
               j--;
          }

          // swap the elements at i and j
          if (i ⩽ j) {
               swap(arr, i, j);
               i++;
               j--;
          }
     }

     // recursively sort the left and right subarrays
     quick_sort(arr, left, j);
     quick_sort(arr, i, right);
     return;
}
```

Quick Sort Time Complexity analysis:

1. Best Case Time Complexity : **O($N \cdot \log_2 N$)**

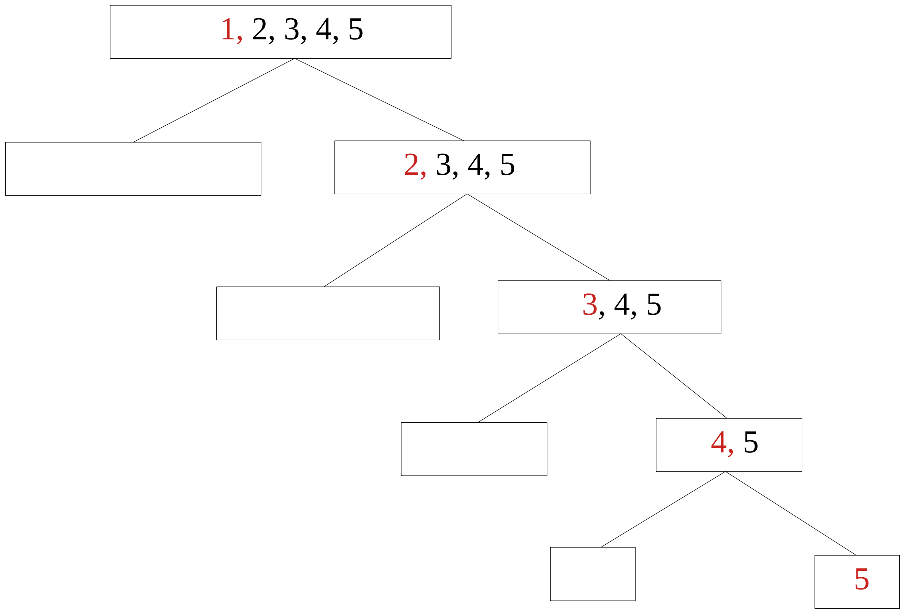
Following recursion tree which is representation of recursive calls to quick sort function on subarrays demonstrates the best case time complexity when the height of the tree is minimum.
The element in red color denotes the pivot around which array is partitioned.



As we can see the best case is when the pivot is median every time and we get a **complete binary tree**. It has height of almost **$\log_2$N** which is minimum. Also, at each step array is **partitioned around the pivot.** It involves scanning array once i.e **O(N).** So, overall complexity at best case is **O(N$\log_2$N).**

2. Worst Case Time Complexity: **O(N^2)**

The Worst case in quick sort is when **array is already sorted** and **every time minimum or maximum element of the array is chosen as the pivot.** This creates and empty partition at each step and we have to scan almost entire array at each step.



The worst case in quicksort when pivot always results in extremely unbalanced partitions. For example, the pivot might consistently divide the array such that one side has all the elements except the pivot itself, and the other side is empty. This leads to the recursion depth being equal to the number of elements, resulting in a time complexity of **O(n^2)**.

To address the worst-case time complexity of Quicksort, a common workaround is to randomize the pivot selection. In this approach, the pivot is chosen randomly at each step. While this may not guarantee the best-case scenario, it ensures that, on average, the time complexity remains O(n $\log_2$ n), which aligns with the algorithm's average-case performance. Probabilistically, random pivot selection significantly reduces the likelihood of encountering the worst-case scenario by avoiding consistently unbalanced partitions. As a result, this approach ensures the robustness of Quicksort without compromising the efficiency.

Average Case Time Complexity of Quick Sort: **O(** $N \cdot \log_2 N$ **)**

When the pivot is chosen randomly (or effectively balances the array), on average, it splits the array into two subarrays of roughly equal size. For simplicity, let us assume the split creates subarrays of sizes n/2 each.
At each level of recursion, the array is divided into smaller subarrays. Since the array size reduces approximately by half at each level, the total number of levels in the recursion tree is approximately $\log_2$ n.
as there are comparisons of order O(n) at each step the average time complexity becomes O(n) x O(log n) I.e O(n log n).

Heap Sort

```
void heap_sort(int *arr, int len) {
    Heap heap;
    init_heap(&heap, len);
    free(heap.h);

    heap.h = arr;
    heap.size = len;
    heap.rear = len-1;

    build_max_heap(&heap);
    int old_rear = heap.rear;

    for(int i = heap.rear; i >= 0; i--) {
        swap_heap(&heap, 0, i);
        heap.rear--;
        heapify(&heap, 0);
    }

    heap.rear = old_rear;
    return;
}
```

The three main functions involved in heap sort are:
1. Heapify
2. Build Max Heap
3. HeapSort

The time complexity of heapify involves two components:
1. time taken to decide the root element that is maximum of h[i], h[left], h[right]
2. time taken to heapify a child of i^{th} node.

The time required for 1^{st} task is constant as it involves constant number of comparisons in any case.
Maximum number of nodes in a subtree can be (2n/3) when n is number of nodes in a tree.
So the time complexity of heapify function is O(log n).

The time complexity of build max heap function comes out to be O(n).

So in conclusion the time complexity of heap sort is **O(nlogn).**

Best Case:
Even if the array is already sorted, Heapsort still performs the same operations

Since no part of the algorithm can skip or optimize for sorted inputs, the best-case time complexity remains **O(nlogn)**

Average Case:
As explained for best case for even randomly ordered input the heapsort still performs same operations so the number of comparisons are consistent and hence time complexity in average case is **O(nlogn).**

```c
// Main driver program
#include <stdio.h>
#include <stdlib.h>
#include <syscall.h>
#include <time.h>
#include <string.h>
#include "helper_functions.h"
#include "heap.h"

// 1. Bubble Sort
void bubble_sort(int *arr, int len) {
     for (int i = 0; i < len; i++) {
          for (int j = 0; j < len - (i + 1); j++) {
               if (arr[j] > arr[j + 1]) {
                    swap(arr, j, j + 1);
               }
          }
     }
}

// 2. Bubble sort V2
/*
    there is one obvious optimization that we can do in
    the bubble sort algorithm. If in any iteration, we do not
    swap any elements, then the array is already sorted.
    So, we can break out of the loop and return.
*/
void bubble_sort_v2(int *arr, int len) {
    for (int i = 0; i < len; i++) {
        int swapped = 0; // Flag to track if any swaps happened in this
pass
        for (int j = 0; j < len - (i + 1); j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr, j, j + 1);
                swapped = 1; // Mark as swapped
            }
        }
        if (!swapped) {
            break; // Array is sorted, exit early
        }
    }
}


// 3. Insertion Sort (swapping)
void insertion_sort_swapping(int *arr, int len) {
    for(int i=1; i<len; i++) {
        int j = i;
        while(j > 0 && arr[j] < arr[j-1]) {
            swap(arr, j, j-1);
            j--;
        }
    }
    return;
}

// 4. Insertion Sort (Shifting)
void insert(int key, int arr[], int j) {
    // insert key into the sorted subarray arr[0] to arr[j-1]
    // by shifting elements to the right until the correct position is
found

    while (j >= 0 && arr[j] > key) {
        arr[j+1] = arr[j];
        j = j-1;
    }
    arr[j+1] = key;
}

void insertion_sort(int arr[], int n) {
    // sort a[0] to a[n-1] into nondecreasing order

    for (int i = 1; i < n; i++) {
        // insert a[i] into the sorted subarray a[0] to a[i-1]
        int key = arr[i];
        int j = i-1;
        insert(key, arr, j);
    }
}

// 5. Selection Sort
void selection_sort(int *arr, int len) {
    for (int i=0; i < len; i++){

        int min_index = i; // Index of the minimum element

        for (int j = i+1; j < len; j++) {
            if(arr[j] < arr[min_index]) {
                min_index = j;
            }
        }

        swap(arr, i, min_index);
    }
    return;
}

// 6. Quick Sort
void quick_sort(int *arr, int left, int right) {
     // base case
     if (left ⩾ right) {
         return;
     }

     int pivot = arr[(left + right) / 2];  // pick the middle element as
pivot
     int i = left, j = right;

     /*
         quick sort works by partitioning the array into two subarrays
         such that all elements in the left subarray are less than the
pivot
         and all elements in the right subarray are greater than the
pivot
     */

     while (i ⩽ j) {
         // find the first element from the left that is greater than
the pivot
         while (arr[i] < pivot) {
                i++;
         }

         // find the first element from the right that is less than the
pivot
         while (arr[j] > pivot) {
                j--;
         }

         // swap the elements at i and j
         if (i ⩽ j) {
                swap(arr, i, j);
                i++;
                j--;
         }
     }

     // recursively sort the left and right subarrays
     quick_sort(arr, left, j);
     quick_sort(arr, i, right);
     return;
}

// 7. Heap Sort
void heap_sort(int *arr, int len) {
    Heap heap;
    init_heap(&heap, len);
    free(heap.h);

    heap.h = arr;
```

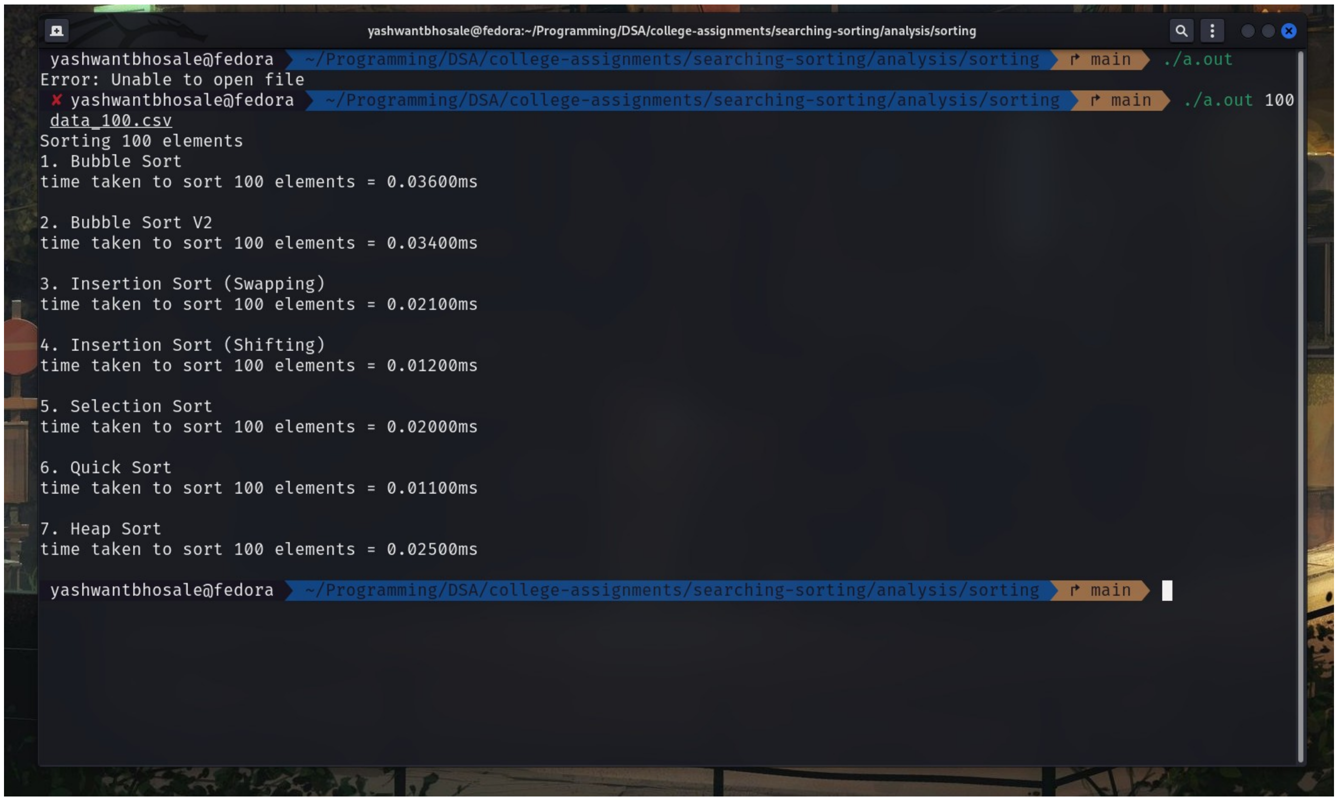
```c
    heap.size = len;
    heap.rear = len-1;

    build_max_heap(&heap);
    int old_rear = heap.rear;

    for(int i = heap.rear; i >= 0; i--) {
        swap_heap(&heap, 0, i);
        heap.rear--;
        heapify(&heap, 0);
    }

    heap.rear = old_rear;
    return;
}

int main(int argc, char *argv[]) {
    int len = argc > 1 ? atoi(argv[1]) : 1000;
    int *arr_original = read_array_from_csv(argv[2], len);
    if (arr_original == NULL) {
        return 1;
    }

    int *arr = (int *)malloc(len * sizeof(int));
    if (arr == NULL) {
        printf("Error: Memory allocation failed\n");
        free(arr_original);
        return 1;
    }

    printf("Sorting %d elements\n", len);

    clock_t start, end;
    double time_ms;

    // Bubble Sort
    printf("1. Bubble Sort\n");
    memcpy(arr, arr_original, len * sizeof(int));
    start = clock();
    bubble_sort(arr, len);
    end = clock();
    time_ms = ((double)(end - start)) * 1000 / CLOCKS_PER_SEC;
    printf("time taken to sort %d elements = %.5lfms\n\n", len, time_ms);


    printf("2. Bubble Sort V2\n");
    memcpy(arr, arr_original, len * sizeof(int));
    start = clock();
    bubble_sort_v2(arr, len);
    end = clock();
    time_ms = ((double)(end - start)) * 1000 / CLOCKS_PER_SEC;
    printf("time taken to sort %d elements = %.5lfms\n\n", len, time_ms);

    // Insertion Sort (Swapping)
    printf("3. Insertion Sort (Swapping)\n");
    memcpy(arr, arr_original, len * sizeof(int));
    start = clock();
    insertion_sort_swapping(arr, len);
    end = clock();
    time_ms = ((double)(end - start)) * 1000 / CLOCKS_PER_SEC;
    printf("time taken to sort %d elements = %.5lfms\n\n", len, time_ms);

    // Insertion Sort (Shifting)
    printf("4. Insertion Sort (Shifting)\n");
    memcpy(arr, arr_original, len * sizeof(int));
    start = clock();
    insertion_sort(arr, len);
    end = clock();
    time_ms = ((double)(end - start)) * 1000 / CLOCKS_PER_SEC;
    printf("time taken to sort %d elements = %.5lfms\n\n", len, time_ms);

    // Selection Sort
    printf("5. Selection Sort\n");
    memcpy(arr, arr_original, len * sizeof(int));
    start = clock();
    selection_sort(arr, len);
    end = clock();
    time_ms = ((double)(end - start)) * 1000 / CLOCKS_PER_SEC;
    printf("time taken to sort %d elements = %.5lfms\n\n", len, time_ms);

    // Quick Sort
    printf("6. Quick Sort\n");
    memcpy(arr, arr_original, len * sizeof(int));
    start = clock();
    quick_sort(arr, 0, len - 1);
    end = clock();
    time_ms = ((double)(end - start)) * 1000 / CLOCKS_PER_SEC;
    printf("time taken to sort %d elements = %.5lfms\n\n", len, time_ms);

    // Heap Sort
    printf("7. Heap Sort\n");
    memcpy(arr, arr_original, len * sizeof(int));
    start = clock();
    heap_sort(arr, len);
    end = clock();
    time_ms = ((double)(end - start)) * 1000 / CLOCKS_PER_SEC;
    printf("time taken to sort %d elements = %.5lfms\n\n", len, time_ms);

    free(arr_original);
    free(arr);
    return 0;
}

// Heap Functions


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "heap.h"
#include "helper_functions.h"

// utility functions used throughout the program
int get_parent_index(int index) {
     return (index - 1) / 2;
}
int lchild_index(int index) {
     return 2 * index + 1;
}
int rchild_index(int index) {
     return 2 * index + 2;
}

void init_heap(Heap *heap, int size) {
     heap->h = (int *)malloc(sizeof(int) * size);
     heap->size = size;
     heap->rear = size-1;
     return;
}

void swap_heap(Heap *heap, int i, int j) {
     // Make sure that the indices are valid
     if (i >= heap->size || i < 0 || j >= heap->size || j < 0) {
          return;
     }

     int temp = heap->h[i];
     heap->h[i] = heap->h[j];
     heap->h[j] = temp;

     return;
}
```

```c
/*
    * Heapify function
    It takes two arguments:
    1. Heap pointer
    2. Index of the node to heapify

    The function recursively heapifies the subtree formed by the node at
the given index.
    It assumes that the subtrees rooted at the left and right children of
the node are already max-heaps.

    Why this is useful?
    this is useful when we want to build a max heap from an array of
elements in O(n) time complexity.
    We can start from the last non-leaf node and heapify all the nodes in
reverse order.
    This way, we can build a max heap in O(n) time complexity.
*/
void heapify(Heap *heap, int index) {
    int left = lchild_index(index);
    int right = rchild_index(index);
    int largest = index;

    // Find the largest element among the current node, left child, and
right child
    if (left <= heap->rear && heap->h[left] > heap->h[largest]) {
        largest = left;
    }
    if (right <= heap->rear && heap->h[right] > heap->h[largest]) {
        largest = right;
    }

    // If the largest element is not the current node, swap the current
node with the largest element
    if (largest != index) {
        swap_heap(heap, index, largest);
        heapify(heap, largest);
    }
    return;
}
```

```
void build_max_heap(Heap *heap) {
    /*
        Why heap->rear / 2?
        So generally heap array is divided into two parts:
        1. internal nodes
        2. leaf nodes
        leaf nodes are already max-heaps, so we don't need to heapify
them.
        We only need to heapify the internal nodes.
        The last internal node is at index heap->rear / 2.
        So we start from this node and heapify all the nodes in reverse
order.

        Why reverse order?
        Because we are building a max heap, and the heapify function
assumes that the left and right subtrees are already max-heaps.
        So we need to start from the last internal node and move towards
the root.
    */
    for (int i = heap->rear / 2; i >= 0; i--) {
        heapify(heap, i);
    }
    return;
}
```

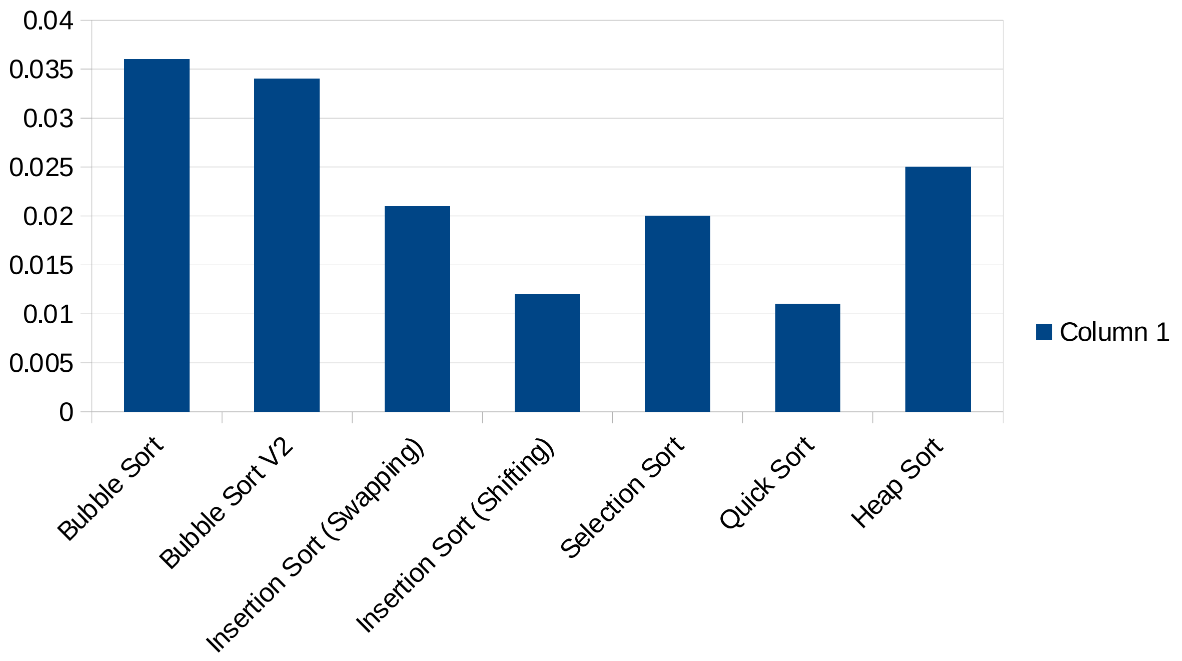
Results:
1. Following are the results I.e time taken to sort in ms by different algorithms.
2. in following results: bubble sort v2 exits early if no swaps occur in first pass I.e optimized.
3. also, insertion sort is implemented in two ways: 1. by shifting 2. by swapping to compare performance of both.

1. Data of 100 unsorted elements [Link to dataset: Data of 100 elements]
Lowest time taken by **Quick Sort :** 0.01100ms

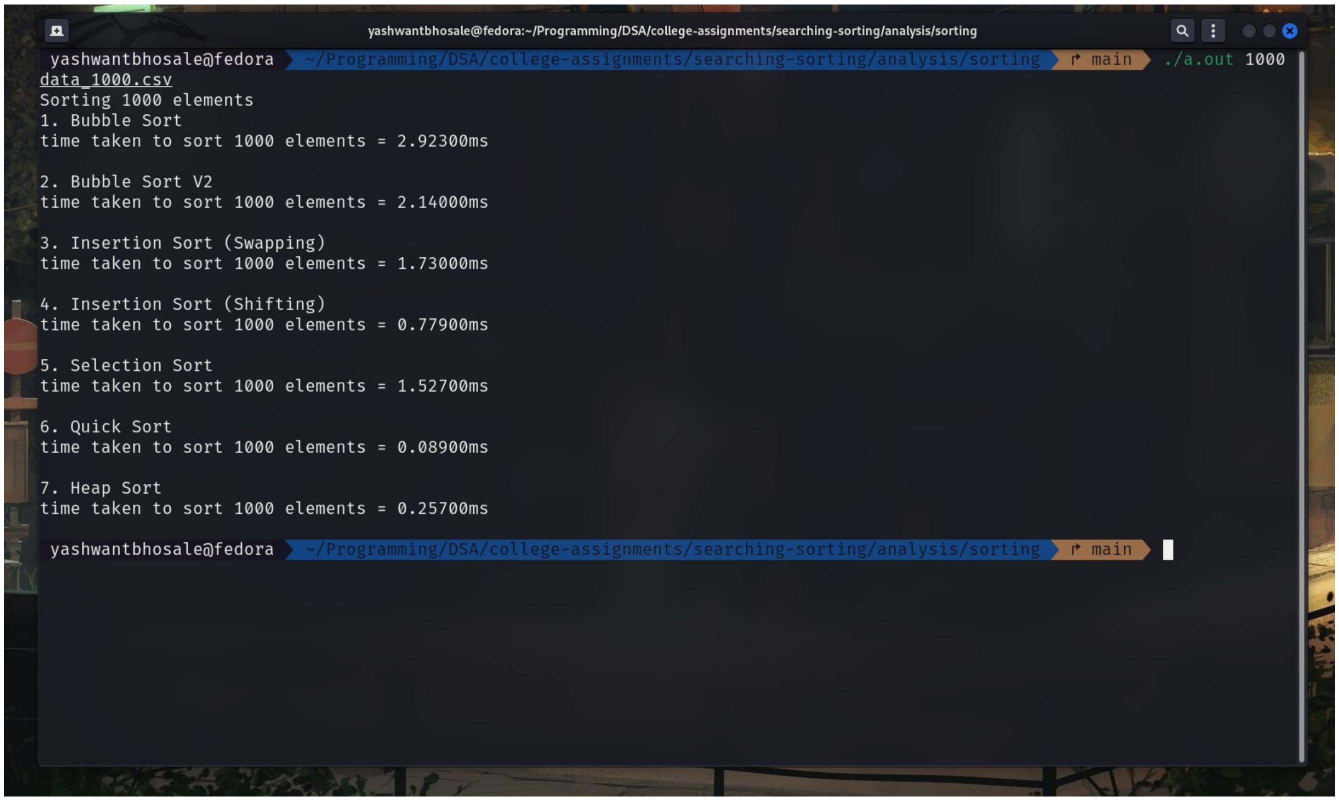


yashwantbhosale@fedora:~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting

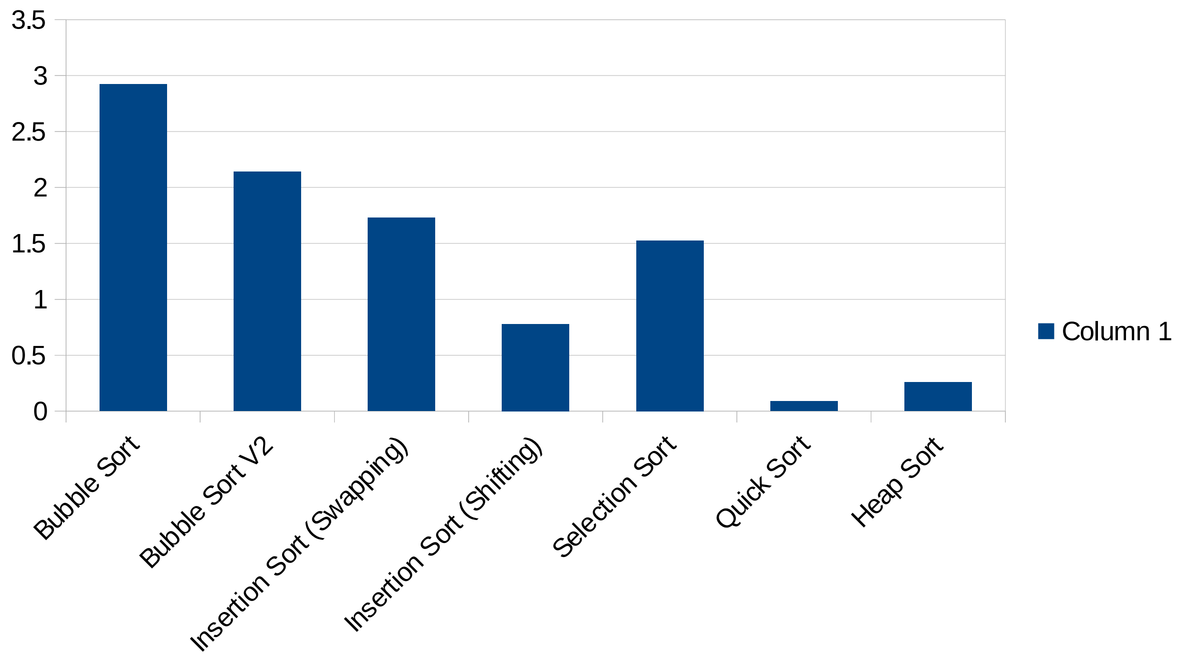
```
 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting  ↱ main  ./a.out
Error: Unable to open file
 ✘ yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting  ↱ main  ./a.out 100
 data_100.csv
Sorting 100 elements
1. Bubble Sort
time taken to sort 100 elements = 0.03600ms

2. Bubble Sort V2
time taken to sort 100 elements = 0.03400ms

3. Insertion Sort (Swapping)
time taken to sort 100 elements = 0.02100ms

4. Insertion Sort (Shifting)
time taken to sort 100 elements = 0.01200ms

5. Selection Sort
time taken to sort 100 elements = 0.02000ms

6. Quick Sort
time taken to sort 100 elements = 0.01100ms

7. Heap Sort
time taken to sort 100 elements = 0.02500ms

 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting  ↱ main 
```



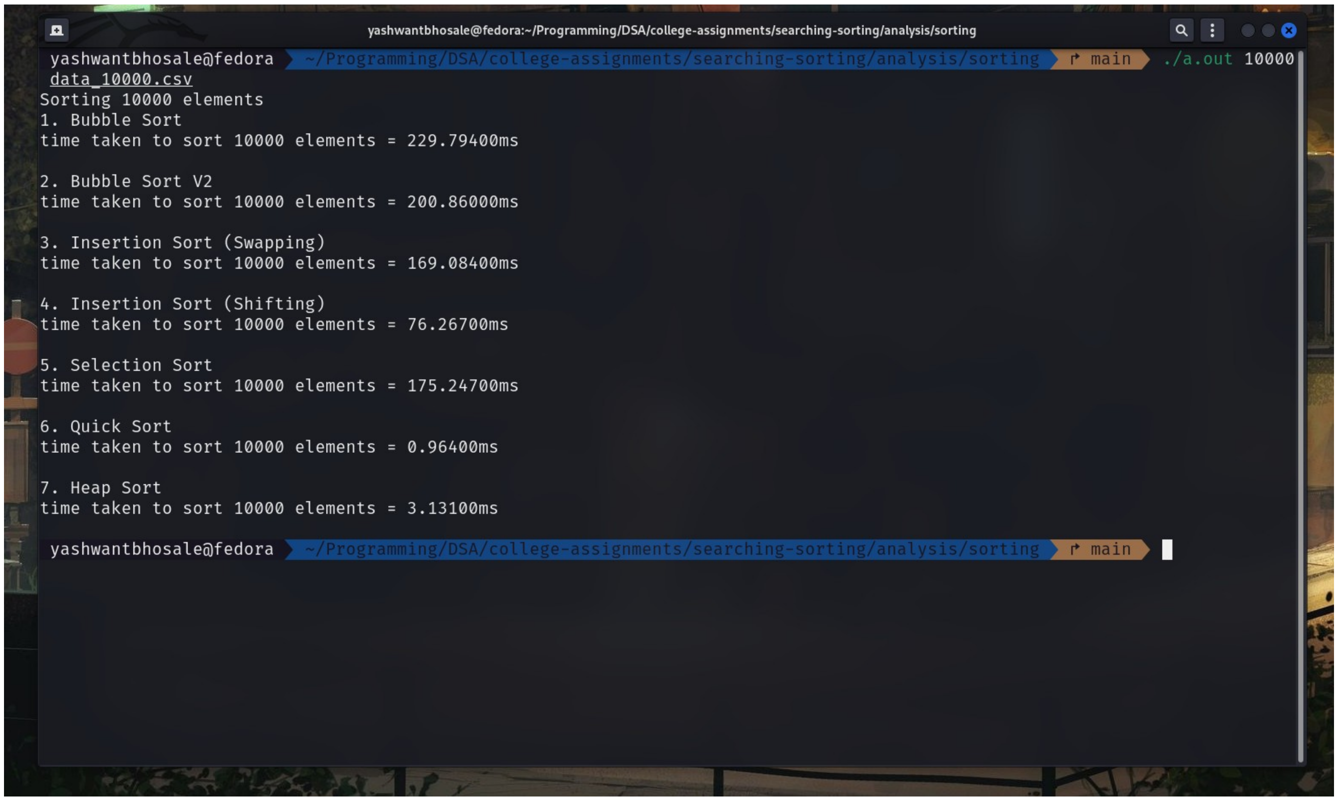
2. Data of 1000 unsorted elements [Link to dataset: Data set of 1000 unsorted elements]
Shortest time taken by **Quick Sort : 0.08900ms**



```
yashwantbhosale@fedora ~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting ↱ main ./a.out 1000
data_1000.csv
Sorting 1000 elements
1. Bubble Sort
time taken to sort 1000 elements = 2.92300ms

2. Bubble Sort V2
time taken to sort 1000 elements = 2.14000ms

3. Insertion Sort (Swapping)
time taken to sort 1000 elements = 1.73000ms

4. Insertion Sort (Shifting)
time taken to sort 1000 elements = 0.77900ms

5. Selection Sort
time taken to sort 1000 elements = 1.52700ms

6. Quick Sort
time taken to sort 1000 elements = 0.08900ms

7. Heap Sort
time taken to sort 1000 elements = 0.25700ms

yashwantbhosale@fedora ~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting ↱ main
```

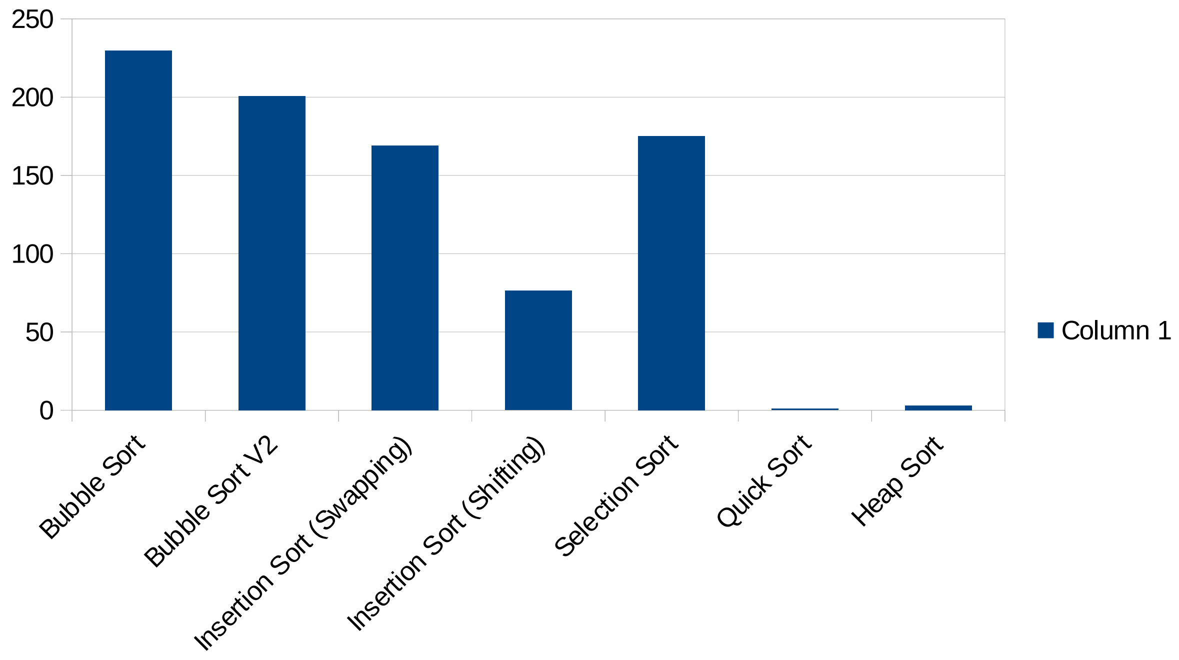


3. Dataset of 10000 Unsorted elements: [Link to dataset : Dataset of 10000 unsorted elements]



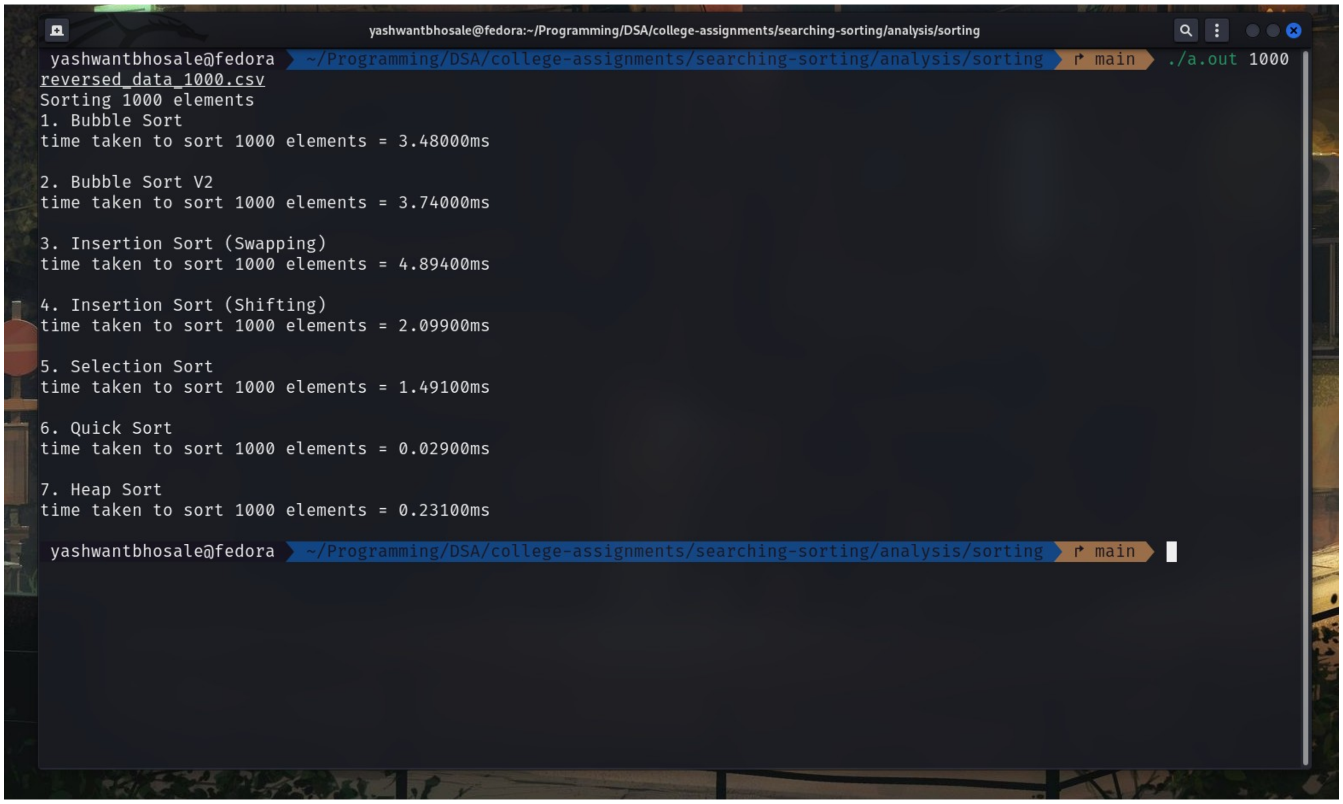
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting

```
 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting  ↱ main  ./a.out 10000
 data_10000.csv
Sorting 10000 elements
1. Bubble Sort
time taken to sort 10000 elements = 229.79400ms

2. Bubble Sort V2
time taken to sort 10000 elements = 200.86000ms

3. Insertion Sort (Swapping)
time taken to sort 10000 elements = 169.08400ms

4. Insertion Sort (Shifting)
time taken to sort 10000 elements = 76.26700ms

5. Selection Sort
time taken to sort 10000 elements = 175.24700ms

6. Quick Sort
time taken to sort 10000 elements = 0.96400ms

7. Heap Sort
time taken to sort 10000 elements = 3.13100ms

 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting  ↱ main 
```

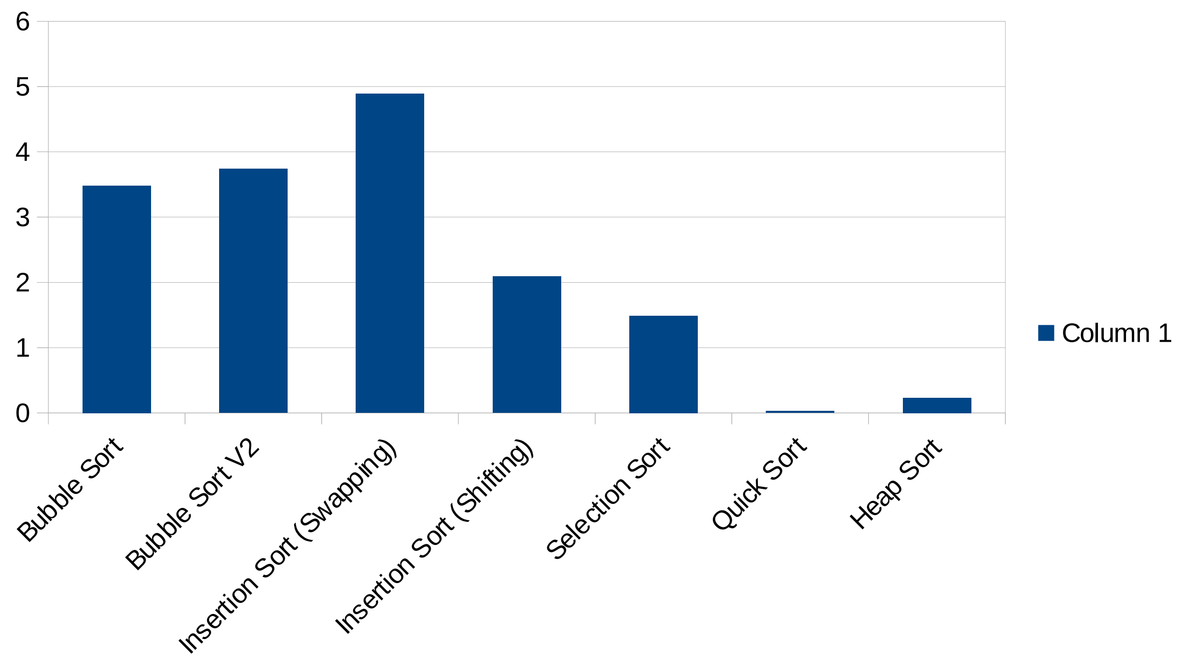


3. Data of 1000 reversed elements [Link to dataset: Dataset of 1000 reversed elements]

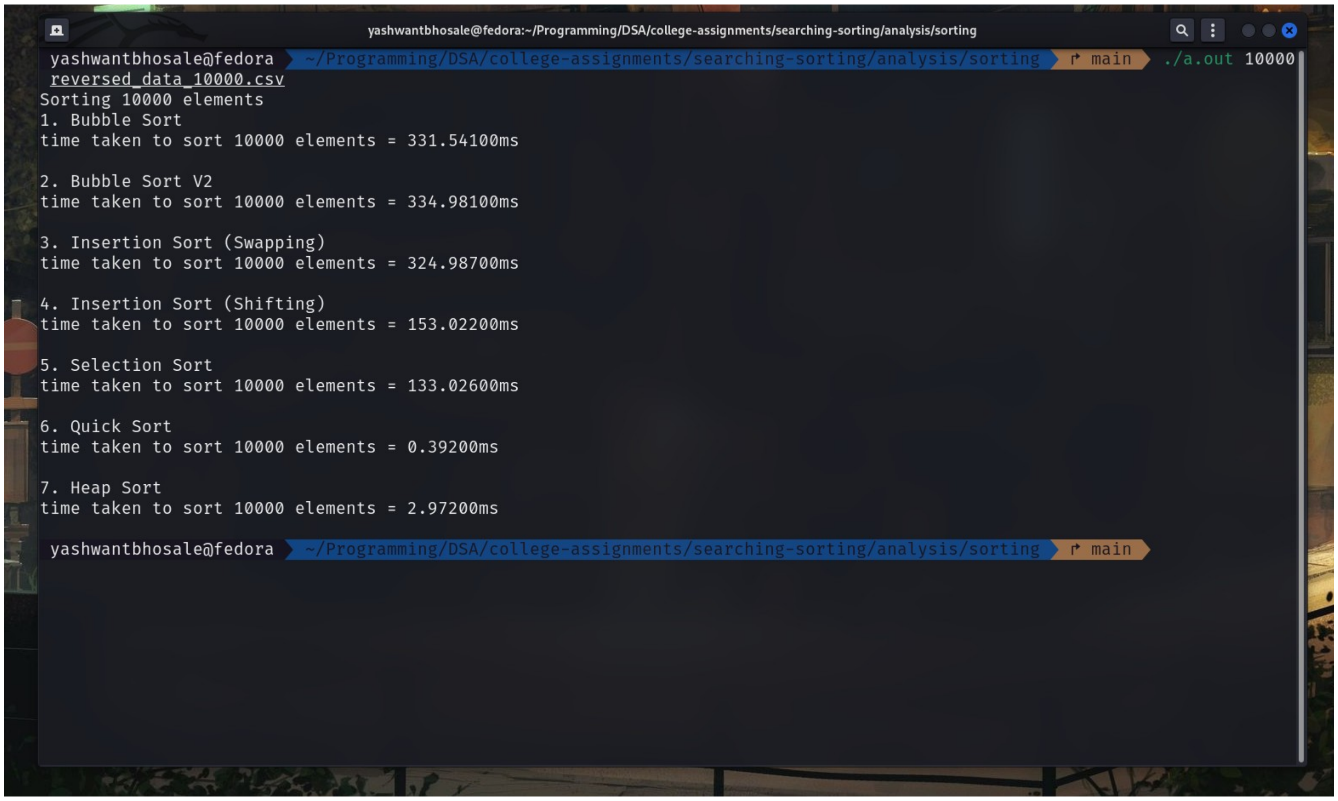
Shortest time taken by **Quick Sort : 0.02900ms**



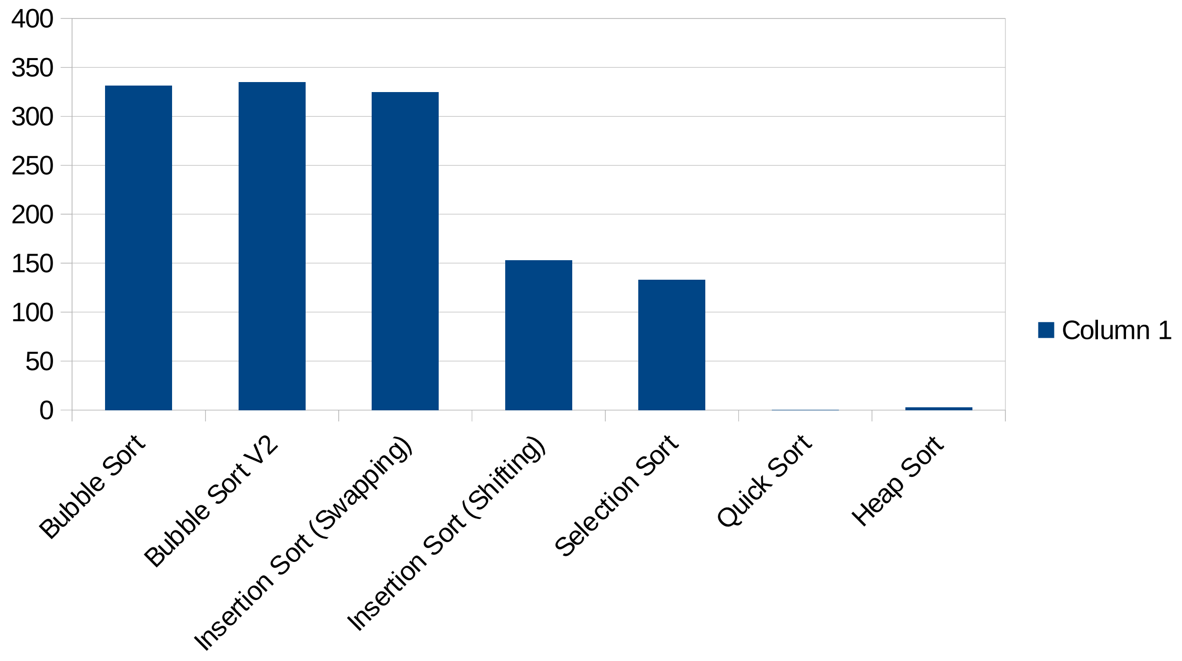
```
yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting  ↱ main  ./a.out 1000
reversed_data_1000.csv
Sorting 1000 elements
1. Bubble Sort
time taken to sort 1000 elements = 3.48000ms

2. Bubble Sort V2
time taken to sort 1000 elements = 3.74000ms

3. Insertion Sort (Swapping)
time taken to sort 1000 elements = 4.89400ms

4. Insertion Sort (Shifting)
time taken to sort 1000 elements = 2.09900ms

5. Selection Sort
time taken to sort 1000 elements = 1.49100ms

6. Quick Sort
time taken to sort 1000 elements = 0.02900ms

7. Heap Sort
time taken to sort 1000 elements = 0.23100ms

yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting  ↱ main
```



4. Data of 10000 reversed elements: [Link to dataset: Data of 10000 reversed elements]
Shortest time taken by **Quick Sort: 0.39299ms**



```
yashwantbhosale@fedora:~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting
 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting  ↱ main  ./a.out 10000
 reversed_data_10000.csv
Sorting 10000 elements
1. Bubble Sort
time taken to sort 10000 elements = 331.54100ms

2. Bubble Sort V2
time taken to sort 10000 elements = 334.98100ms

3. Insertion Sort (Swapping)
time taken to sort 10000 elements = 324.98700ms

4. Insertion Sort (Shifting)
time taken to sort 10000 elements = 153.02200ms

5. Selection Sort
time taken to sort 10000 elements = 133.02600ms

6. Quick Sort
time taken to sort 10000 elements = 0.39200ms

7. Heap Sort
time taken to sort 10000 elements = 2.97200ms

 yashwantbhosale@fedora  ~/Programming/DSA/college-assignments/searching-sorting/analysis/sorting  ↱ main
```



Conclusion:
1. Heap sort and Quick sort perform better in all the datasets due to their O(nlogn) time complexity.
2. Quick sort performs better in all the datasets.
3. Insertion sort works better with shifting than swapping
4. bubble sort v2 -(early exit if data is already sorted) works better in larger datasets no significant difference otherwise.