Data Structures and Algorithms

COEP Technological University Pune

NAME: Aman Bipin Morghade

MIS: 612303017

Division: Div-1, SY B.Tech, Computer Science and Engineering

Type: Searching Sorting Assignment

Course: DSA – (Data Structures and Algorithms)

SEARCHING:

Linear Search:

```c
include <stdio.h>
include <stdlib.h>

int linearSearch(int* A, int len, int target) {
    for (int i = 0; i < len; i++) {
        if (A[i] == target) return i;
    }
    return -1;
}

int main() {
    int len, target;
    scanf("%d %d", &len, &target);

    int* A = (int*) malloc(len * sizeof(int));
    for (int i = 0; i < len; i++) {
        scanf("%d", &A[i]);
    }

    int index = linearSearch(A, len, target);
    printf("%d\n", index);

    free(A);
    return 0;
}
```

Binary Search:

```c
include <stdio.h>
include <stdlib.h>

int binarySearch(int* A, int len, int target) {
    int mid;
    int l = 0;
    int r = len - 1;

    while(l <= r) {
        mid = (l + r) / 2;

        if(mid == target) {
            return mid;
        } else if(mid < target) {
```

```
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    };
    return -1;
}

int main() {
    int len, target;
    scanf("%d %d", &len, &target);

    int* A = (int*) malloc(len * sizeof(int));
    for (int i = 0; i < len; i++) {
        scanf("%d", &A[i]);
    }

    int index = binarySearch(A, len, target);
    printf("%d\n", index);

    free(A);
    return 0;
}
```

Asymptotic Analysis of Linear Search vs. Binary Search

LINEAR SEARCH:

- Best Case: O(1)

- Worst Case: O(n)

- Average Case: O(n)

BINARY SEARCH:

- Best Case: O(1)

- Worst Case: O(logn)

- Average Case: O(logn)

Advantages of Linear Search:

1. Versatility: A general-purpose search algorithm that can operate on any dataset, regardless of order.

2. No Sorting Requirement: Works on both sorted and unsorted data structures.

3. Simplicity: Linear asymptotic upper bound with straightforward implementation.

Disadvantages of Linear Search:

1. Lower Efficiency: Slower than binary search for larger datasets.

2. Inefficient Worst Case: In the worst case, requires a complete iteration over the entire dataset.

Advantages of Binary Search:

1. Higher Efficiency: Significantly faster than linear search, especially for large datasets.

2. Divide and Conquer: Repeatedly divides the dataset in half based on comparisons with the median, reducing search space logarithmically.

3. Optimized Complexity: Worst-case time complexity is (O(log n), offering superior performance for sorted data.

Disadvantages of Binary Search:

1. Sorting Requirement: Applicable only on datasets sorted in ascending order, which can add preprocessing time if the data is unsorted.

SORTING:

Selection Sort:

```
include <stdio.h>
include <stdlib.h>
include <string.h>
define MAX_SIZE 256

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
void selectionSort(int* A, int len) {
    for(int i = 0; i < len; i++) {
        for(int j = i; j < len; j++) {
            if(A[i] > A[j]) swap(&A[i], &A[j]);
        }
    };
}
```

Bubble Sort:

```
include <stdio.h>
include <stdlib.h>
include <string.h>
define MAX_SIZE 256

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void bubbleSort(int* A, int len) {
    int swapped;
    for(int i = 0; i < len - 1; i++) {
        swapped = 0;
        for(int j = 0; j < len - i - 1; j++) {
            if(A[j] > A[j + 1]) {
                swap(&A[j], &A[j + 1]);
                swapped = 1;
            }
        }
        if(!swapped) break;
    }
    return;
}
```

Insertion Sort:

```
include <stdio.h>
include <stdlib.h>
include <string.h>
define MAX_SIZE 256

void insertionSort(int* A, int len) {
    int key;
    for(int i = 1; i < len; i++) {
```

```
        key = A[i];
        int j = i - 1;
        while(j >= 0 && A[j] > key) {
            A[j + 1] = A[j];
            j--;
        }
        A[j + 1] = key;
    }
}
```

Heap Sort:

```
include <stdio.h>
include <stdlib.h>
include <string.h>
define MAX_SIZE 256

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void heapify(int* A, int len, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;

    if (l < len && A[l] > A[largest]) largest = l;
    if (r < len && A[r] > A[largest]) largest = r;

    if (largest != i) {
        swap(&A[i], &A[largest]);
        heapify(A, len, largest);
    }
}

void heapSort(int* A, int len) {

    for (int i = len / 2 - 1; i >= 0; i--) heapify(A, len, i);

    for (int i = len - 1; i > 0; i--) {
        swap(&A[0], &A[i]);
        heapify(A, i, 0);
    }
}
```

Quick Sort:

```
include <stdio.h>
include <stdlib.h>
include <string.h>

define MAX_SIZE 256

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int part(int* A, int low, int high) {
    int pivot = A[(low + high) / 2];
    int i = low - 1;
    int j = high + 1;

    while (1) {
        do {
            i++;
        } while (A[i] < pivot);

        do {
            j--;
        } while (A[j] > pivot);

        if (i >= j) return j;
        swap(&A[i], &A[j]);
    }
}

void quickSort(int* A, int low, int high) {
    if (low < high) {
        int x = part(A, low, high);
        quickSort(A, low, x);
        quickSort(A, x + 1, high);
    }
}
```

NOTE: This is the main function used for testing, which opens a random dataset and sorts it based on some specified algorithm.

```
int main() {
    char buffer[MAX_SIZE];
    int A[DATASET_SIZE];
```

```c
    int len = 0;
    FILE *fd = fopen(FILENAME, "r");
    if (!fd) {
        perror("Error Opening the File");
        return 1;
    }
    fgets(buffer, MAX_SIZE, fd);

    while (fgets(buffer, MAX_SIZE, fd)) {
        int index;
        int value;
        if (sscanf(buffer, "%d,%d", &index, &value) == 2) {
            A[len++] = value;
        }
    }

    fclose(fd);
    Sort(A, 0, len - 1); // Specify the Sorting Algorithm

    for (int i = 0; i < len; i++) {
        printf("%d ", A[i]);
    }
    printf("\n");
    return 0;
}
```

Sorting Algorithms

This section covers five popular sorting algorithms: **Selection Sort**, **Bubble Sort**, **Insertion Sort**, **Heap Sort**, and **Quick Sort**. Sorting is a fundamental task in computer science, and different algorithms offer trade-offs in terms of performance, memory usage, and complexity. Below, you'll find a description of each algorithm, along with an analysis of their time complexities, advantages, and disadvantages.

Selection Sort

Description

Selection Sort is a simple, comparison-based sorting algorithm. It divides the input list into a sorted and an unsorted part. It repeatedly selects the smallest (or largest, depending on the order) element from the unsorted part and moves it to the end of the sorted part.

Time Complexity

- **Best Case**: $O(n^2)$
- **Average Case**: $O(n^2)$
- **Worst Case**: $O(n^2)$

- **Space Complexity**: O(1) (In-place)

Advantages
- Simple to understand and implement.
- Does not require additional memory.
- Useful for small datasets.

Disadvantages
- Inefficient for large datasets.
- Always has a time complexity of $O(n^2)$ regardless of the initial order of elements.
- Not a stable sort (relative order of equal elements is not preserved).

Bubble Sort

Description
Bubble Sort is a comparison-based sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

Time Complexity
- **Best Case**: O(n) (If already sorted)
- **Average Case**: $O(n^2)$
- **Worst Case**: $O(n^2)$
- **Space Complexity**: O(1) (In-place)

Advantages
- Simple to implement.
- Detects if the list is already sorted, which can reduce time complexity to O(n).
- Stable sort (maintains the relative order of equal elements).

Disadvantages
- Very slow for large datasets.
- Inefficient with a high number of comparisons and swaps.

Insertion Sort

Description
Insertion Sort is a comparison-based algorithm that builds a sorted list one element at a time. It takes each element from the input and inserts it into its correct position in the sorted list.

Time Complexity
- **Best Case**: O(n) (If already sorted)
- **Average Case**: $O(n^2)$
- **Worst Case**: $O(n^2)$
- **Space Complexity**: O(1) (In-place)

Advantages
- Efficient for small or nearly sorted datasets.
- Simple and easy to implement.
- Stable sort (maintains the relative order of equal elements).

Disadvantages
- Inefficient for large datasets.
- Time complexity degrades to $O(n^2)$ in the worst case.

Heap Sort

Description
Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It first builds a max-heap (or min-heap) from the input data and then repeatedly extracts the maximum (or minimum) element to get a sorted array.

Time Complexity
- **Best Case**: O(n log n)
- **Average Case**: O(n log n)
- **Worst Case**: O(n log n)
- **Space Complexity**: O(1) (In-place)

Advantages
- Good performance (O(n log n)) even for large datasets.
- Does not require additional memory.
- More consistent performance compared to QuickSort.

Disadvantages
- Not a stable sort.
- Slightly slower than QuickSort in practical scenarios.
- The heap operations can be complex to implement.

Quick Sort

Description
QuickSort is a highly efficient, comparison-based sorting algorithm. It uses the divide-and-conquer strategy, where the input list is partitioned into two sub-lists. A pivot element is chosen, and elements are rearranged such that those smaller than the pivot are on the left, and those larger are on the right. The sub-lists are then sorted recursively.

Time Complexity
- **Best Case**: O(n log n)
- **Average Case**: O(n log n)
- **Worst Case**: $O(n^2)$ (When the pivot is poorly chosen)
- **Space Complexity**: O(log n) (Depends on the recursion depth)

Advantages
- Very fast in practice for large datasets.
- Average case performance is O(n log n).
- In-place sort (requires minimal additional memory).

Disadvantages
- Not stable (relative order of equal elements may not be preserved).
- Worst-case performance degrades to $O(n^2)$ if a poor pivot is chosen.
- Recursive nature may lead to stack overflow for very large datasets.

Summary Table

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)
QuickSort	O(n log n)	O(n log n)	$O(n^2)$	O(log n)

Conclusion
Each sorting algorithm has its strengths and weaknesses. Simple algorithms like **Selection Sort**, **Bubble Sort**, and **Insertion Sort** are easy to implement but become inefficient with large datasets. **Heap Sort** and **Quick Sort** are more efficient for larger datasets but come with their own trade-offs, like complexity in implementation and stability issues. Choosing the right algorithm depends on the specific requirements of the task, such as dataset size, memory constraints, and whether stability is needed.

TESTING

DATASET-1:

Particulars: 1000 Floating Point Values, Randomly Arranged

Mean: Over 5 Cycles

Performance:

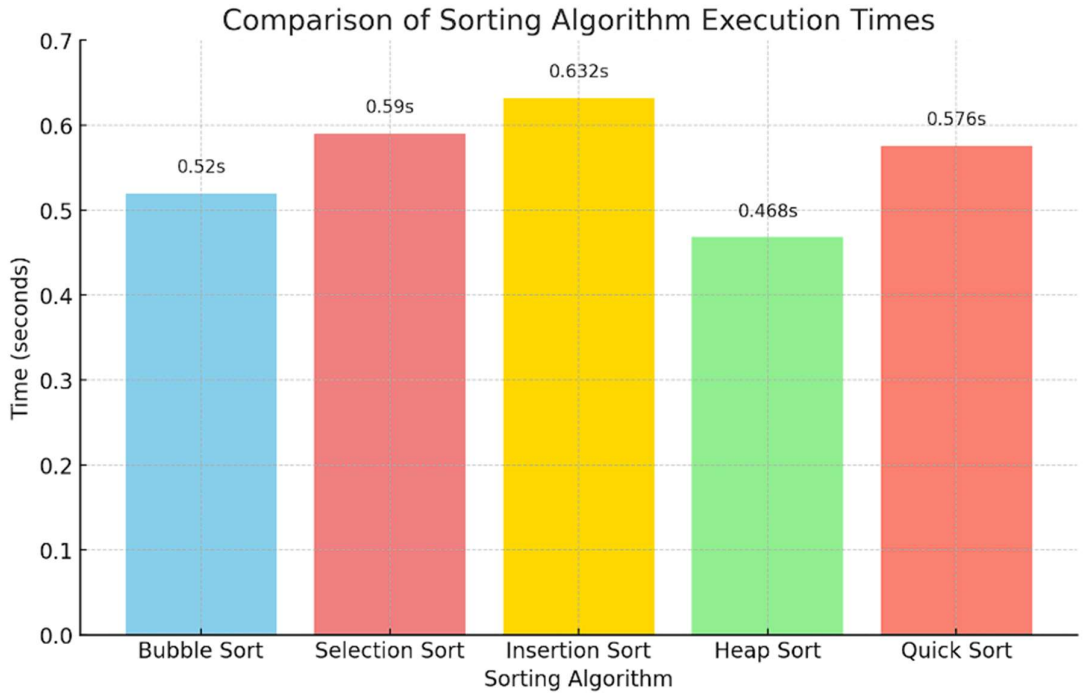
Selection Sort: *0.59s*

Bubble Sort: *0.52s*

Insertion Sort: *0.632s*

Heap Sort: *0.468s*

Quick Sort: *0.576s*



Observations:

- **Heap Sort outperformed all the sorting algorithms with a high speed of 0.468s.**
- **Insertion Sort was the slowest algorithm for this dataset.**
- **All the algorithms followed the Abstract Time Complexity bounds.**

DATASET-2:

Particulars: 50000 Integers, Randomly Arranged

Mean: Over 5 Cycles

Performance:

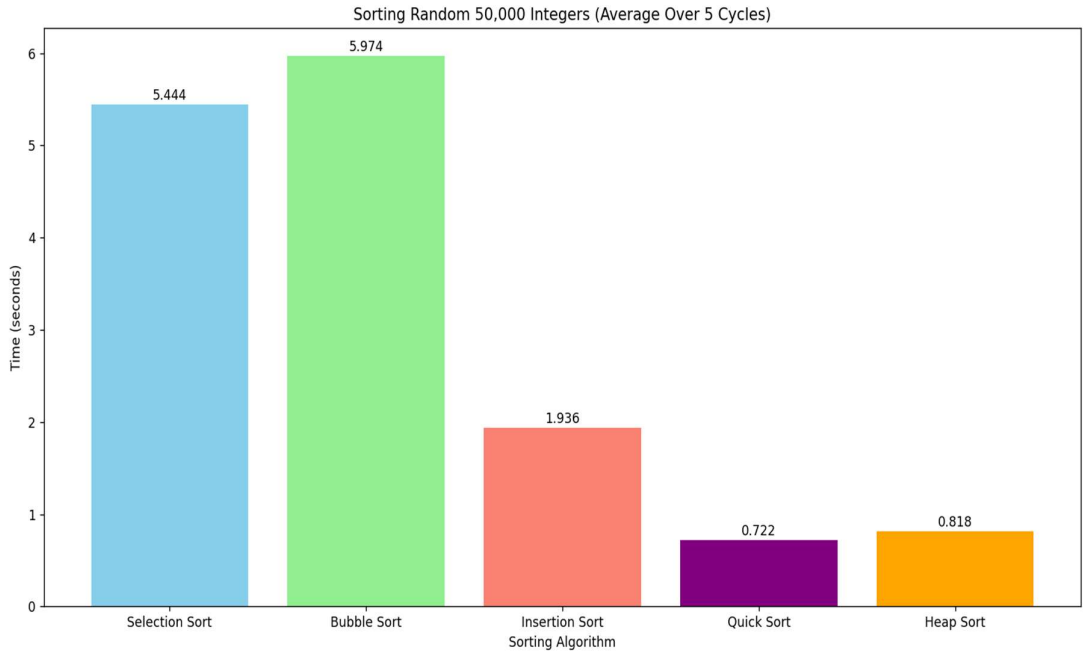
Selection Sort: 5.444s

Bubble Sort: 5.974s

Insertion Sort: 1.936s

Heap Sort: 0.818s

Quick Sort: 0.722s



Observations:

- **Quick Sort outperformed all the sorting algorithms, with a speed of 0.722s**
- **Bubble Sort was the slowest algorithm for this dataset**
- **All the Algorithms followed the Abstract Time Complexity Analysis, with a clear difference in O(n^2) and O(n logn) procedures.**

DATASET-3:

Particulars: 50000 Integers, Sorted in Ascending Order

Mean: Over 5 Cycles

Performance:

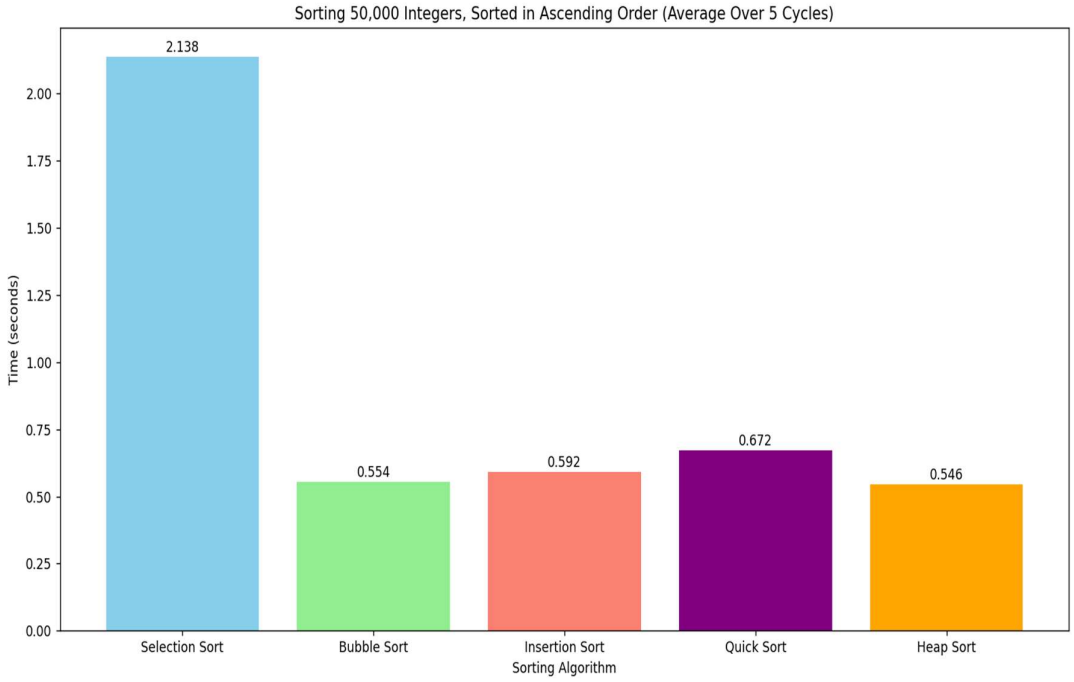
Selection Sort: 2.138s

Bubble Sort: 0.544s

Insertion Sort: 0.592s

Heap Sort: 0.546s

Quick Sort: 0.672s



Observations:

- **Heap Sort outperformed all other algorithms, with a speed of 0.546s.**
- **Selection Sort was the slowest algorithm for this dataset due to the nested minimum checking condition for every iteration.**
- **All the Algorithms followed the Abstract Time Complexity Analysis with a clear difference seen in O(n) and O(n^2) procedures.**

DATASET-4:

Particulars: 100000 Integers, Sorted in Descending Order

Mean: Over 3 Cycles

Performance:

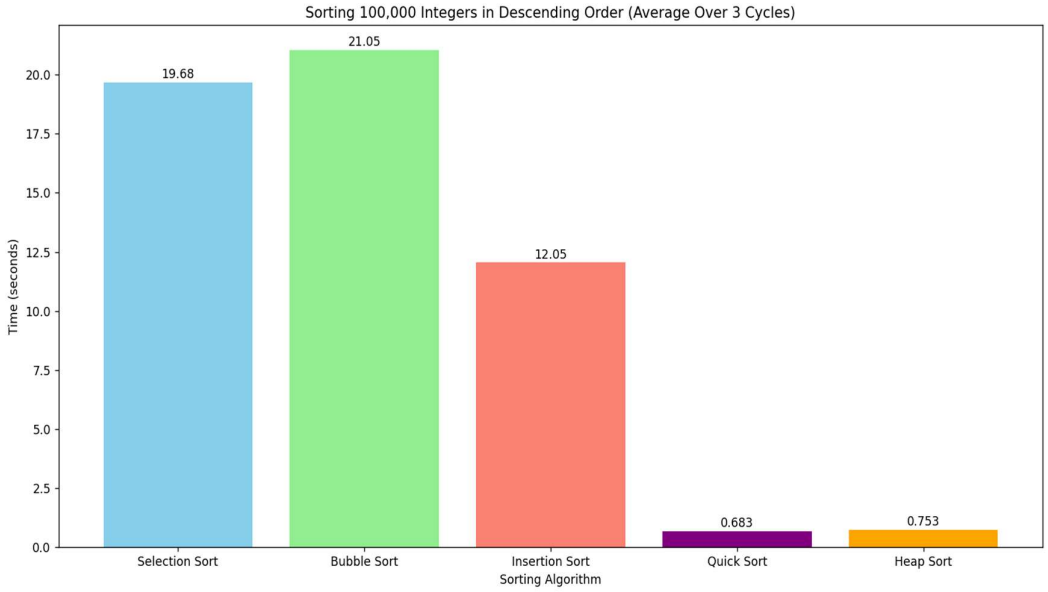
Selection Sort: 19.68s

Bubble Sort: 21.05s

Insertion Sort: 12.05s

Heap Sort: 0.753s

Quick Sort: 0.683s



Observations:

- **Quick Sort outperformed the other algorithms, with a speed of 0.683s**
- **Strange pattern in timings observed, due to worst case scenario analysis**
- **All the Algorithms followed the Abstract Time Complexity Analysis**

FINDINGS:

When the dataset is already sorted or nearly sorted (as seen in **Dataset-3**, where data was sorted in ascending order), algorithms like **Insertion Sort** and **Bubble Sort** perform much better because they require fewer comparisons. In particular, Bubble Sort's best-case time complexity of O(n) is evident when no swaps are needed, leading to improved timing.

Conversely, in **Dataset-4**, where the data was sorted in descending order (the worst-case scenario for some algorithms), performance deteriorated for those that depend on ordered

traversal. For example, Insertion Sort's worst-case time complexity ($O(n^2)$) was highlighted in this scenario due to repeated shifting of elements.

In **QuickSort**, the choice of pivot can significantly affect performance. A poor pivot choice (eg, always selecting the first element in an already sorted dataset) can degrade QuickSort to $O(n^2)$. Also the recursive nature can also affect timing if the dataset size triggers deeper recursive calls, impacting stack usage and causing fluctuations.

HeapSort builds a heap initially, which has a consistent O(n log n) time complexity. In sorted or nearly sorted datasets, the heap construction phase can sometimes benefit from the initial order, leading to better performance. This explains why HeapSort outperformed other algorithms consistently across datasets, with minor fluctuations due to data order influencing the heap's adjustments.

The number of swaps in **Bubble Sort** greatly impacts performance. In already sorted datasets, Bubble Sort can stop early due to a lack of swaps, resulting in its best-case time of O(n), as seen in **Dataset-3**. However, for randomly ordered or reverse datasets, the number of swaps and comparisons significantly increases, causing performance degradation.

CONCLUSIONS:

For **Sorted data,** favor algorithms like Insertion Sort and Bubble Sort.

Random large datasets are better suited for algorithms with logarithmic scaling (**Heap Sort**, **Quick Sort**), reinforcing their practical utility for general use cases.

For **Reverse Sorted data**, inefficient algorithms with quadratic behavior should be avoided.

QUESTIONS:

Q) Why is Binary search more efficient than Linear search on sorted data?

- **Binary Search** operates by dividing a sorted dataset in half repeatedly, which allows it to eliminate half of the remaining elements with each comparison. This gives it a logarithmic time complexity, O(log n), making it very efficient for large datasets.
- In contrast, **Linear Search** checks each element one by one, resulting in a linear time complexity, O(n). This means its efficiency decreases as the dataset size increases.

Q) Why does Quick Sort have a better average-case time complexity compared to Bubble Sort, Selection Sort, and Insertion Sort?

- **Quick Sort** is based on the divide-and-conquer approach. By selecting a pivot and partitioning the array into sub-arrays (elements smaller and larger than the pivot), **Quick Sort** effectively reduces the problem size logarithmically.

- Its average time complexity is O(n log n), due to the efficient partitioning, while **Bubble Sort**, **Selection Sort**, and **Insertion Sort** all have $O(n^2)$ average-case complexity because they involve nested comparisons or swaps.

Q) Under what conditions would insertion sort outperform Quick Sort?

- **Insertion Sort** is particularly effective for small datasets or when the dataset is nearly sorted. In such cases, it has a best-case time complexity of O(n), making it faster than QuickSort's average O(n logn).

Q) Why is Heap Sort often preferred in systems where memory allocation is a concern?

- **Heap Sort** has a space complexity of O(1), meaning it is an in-place sorting algorithm that requires a fixed amount of extra memory regardless of the dataset size. It uses a binary heap structure without the need for additional storage, making it ideal for environments with tight memory constraints.

----------***END***----------