Name	Surti Keyur
Roll No.	25MCE026
Branch	M.Tech (CSE)

Experiment 3

Aim: Implement a Quick Sort algorithm with different ways to select the pivot element and analyze their performance on different input sizes (10,000, 50,000, and 100,000 elements) and input patterns (ascending, descending, random). Compare execution times using tables and graphs, and conclude their efficiency.

Overview

Quick Sort is a divide-and-conquer sorting algorithm that partitions the input array around a pivot element. Its average-case complexity is **O(nlogn)**, but it can degrade to **O(n²)** in the worst case if pivot selection is poor.

The performance of Quick Sort heavily depends on **how the pivot is chosen**. In this experiment, we evaluate the following pivot selection strategies:

1. **First Element as Pivot**
2. **Last Element as Pivot**
3. **Random Element as Pivot**
4. **Median-of-Three as Pivot**

We test these strategies on datasets of sizes **10,000, 50,000, and 100,000**, under three input conditions:

- Ascending (already sorted)
- Descending (reverse sorted)
- Random

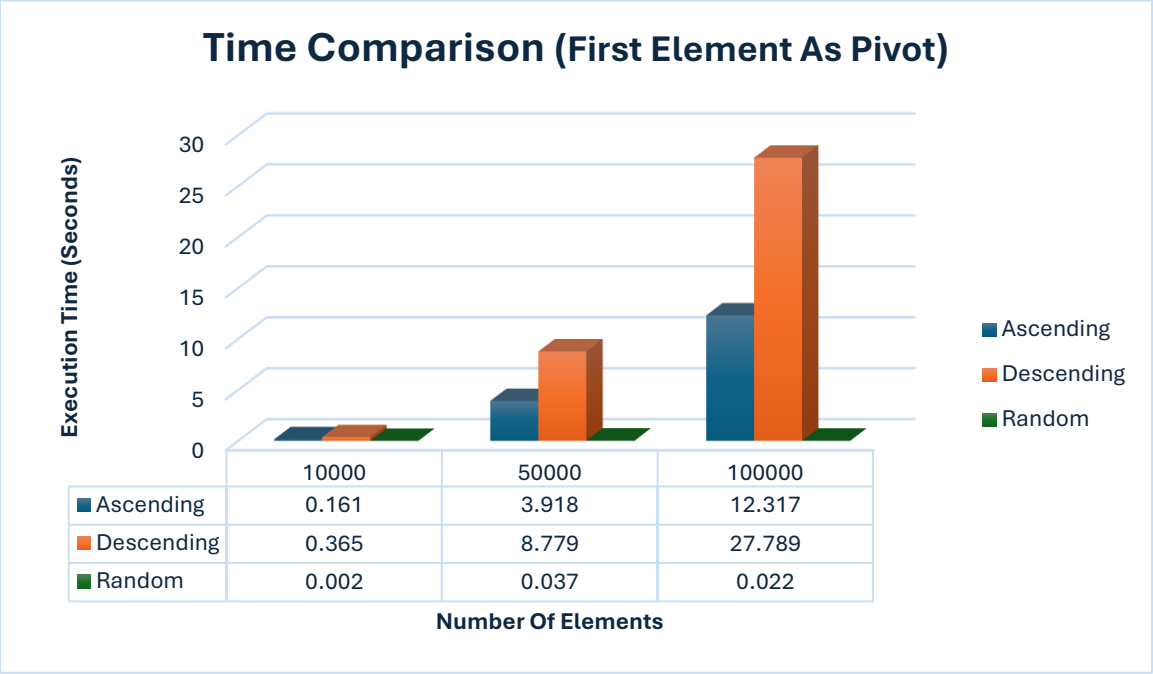
Pivot Selection Strategies

1. First Element as Pivot

Main Logic (Code Snippet)

```
int partitionFirst(vector<int> &arr, int low, int high) {
    int pivot = arr[low];  // Choose first element as pivot
    int i = low + 1;

    for (int j = low + 1; j <= high; j++) {
        if (arr[j] < pivot) {
            swap(arr[i], arr[j]);
            i++;
        }
    }
    swap(arr[low], arr[i - 1]);
    return i - 1;
}

void quickSortFirst(vector<int> &arr, int low, int high) {
    if (low < high) {
        int pi = partitionFirst(arr, low, high);
        quickSortFirst(arr, low, pi - 1);
        quickSortFirst(arr, pi + 1, high);
    }
}
```

Comparison Graph & Table



	10000	50000	100000
Ascending	0.161	3.918	12.317
Descending	0.365	8.779	27.789
Random	0.002	0.037	0.022

Conclusion

- For **10k elements**: Performs moderately (0.161s on ascending, 0.365s on descending), but is already slower compared to random/median pivot. Random input is still very fast (0.002s).

- For **50k elements**: The performance drastically drops (3.9s for ascending, 8.7s for descending), while random input remains efficient at just 0.037s.

- For **100k elements**: The time jumps extremely high (12.3s ascending, 27.7s descending), clearly showing the **O(n²) worst-case nature**. Random input remains the only scenario where this pivot is efficient (0.022s).

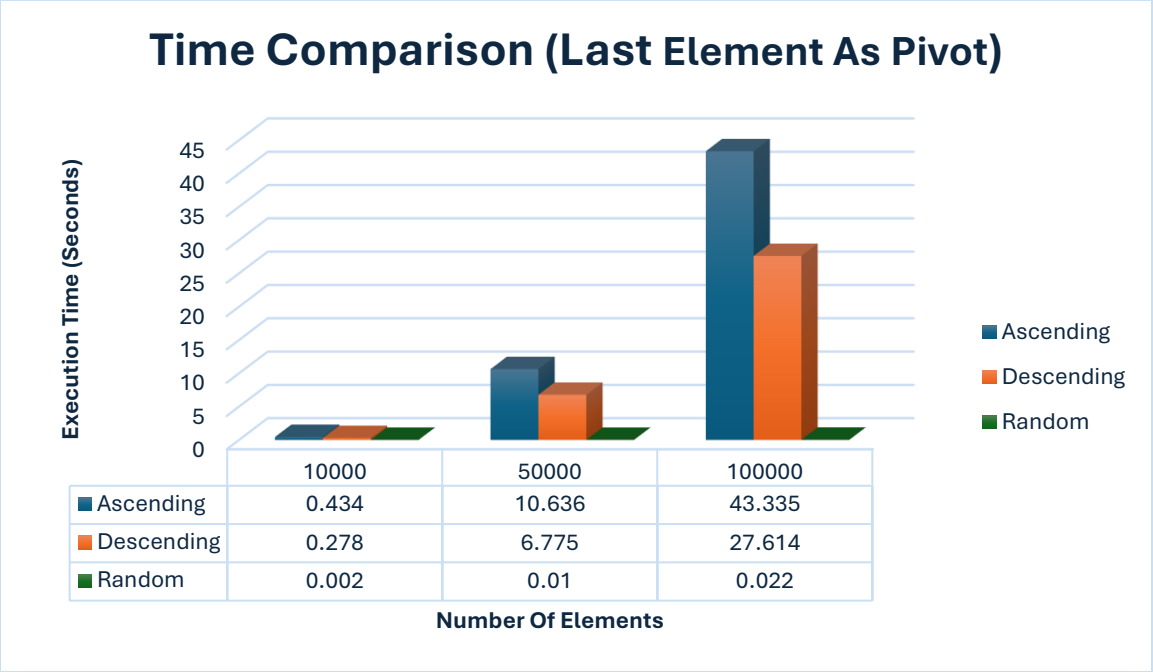
  Good only for random input patterns.
  Highly inefficient for sorted and reverse-sorted data, especially as input size increases.

2. Last Element as Pivot

Main Logic (Code Snippet)

```
int partitionLast(vector<int> &arr, int low, int high) {
    int pivot = arr[high];  // Choose last element as pivot
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSortLast(vector<int> &arr, int low, int high) {
    if (low < high) {
        int pi = partitionLast(arr, low, high);
        quickSortLast(arr, low, pi - 1);
        quickSortLast(arr, pi + 1, high);
    }
}
```

Comparison Graph & Table



	10000	50000	100000
Ascending	0.434	10.636	43.335
Descending	0.278	6.775	27.614
Random	0.002	0.01	0.022

Conclusion

- For **10k elements**: Performs worse than first pivot on ascending input (0.434s), while descending is a bit better (0.278s). Random input is extremely fast (0.002s).

- For **50k elements**: The algorithm shows significant slowdown (10.6s ascending, 6.7s descending), with random input still very fast (0.01s).

- For **100k elements**: Ascending input becomes extremely costly (43.3s), and descending also takes a long time (27.6s). Random input stays efficient (0.022s).

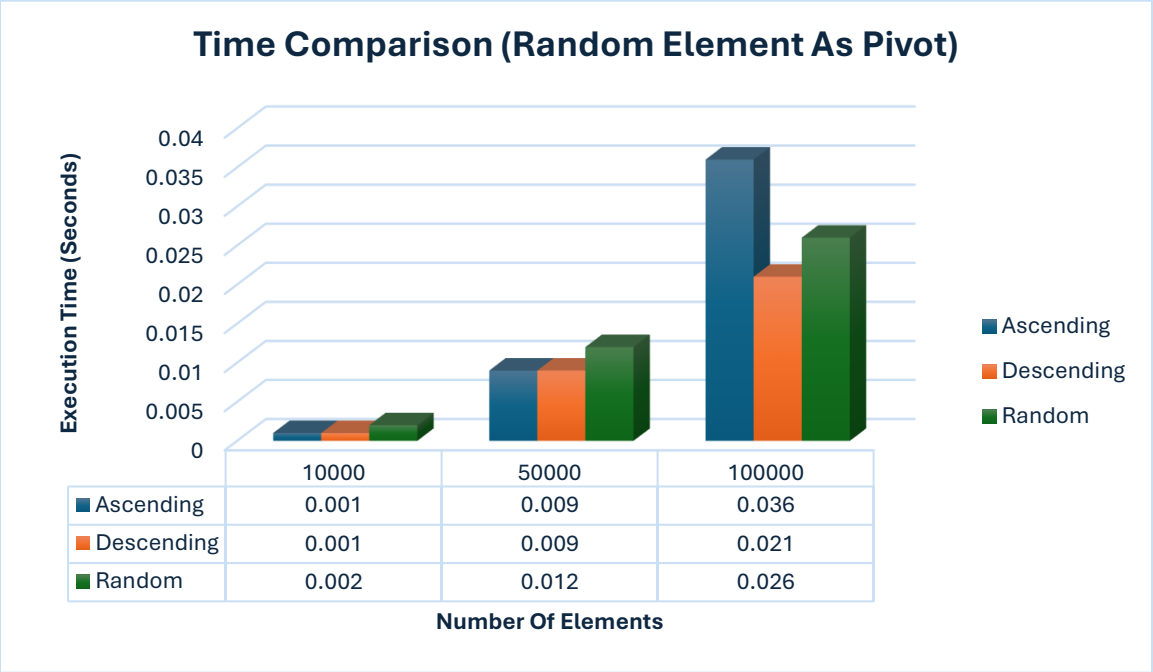
  Works well on random inputs (almost identical to first pivot).
  Performs poorly on ordered inputs, and ascending order in particular triggers worst-case behavior.

3. Random Element as Pivot

Main Logic (Code Snippet)

```
int partitionRandom(vector<int> &arr, int low, int high) {
    srand(time(0));
    int randomIndex = low + rand() % (high - low + 1);
    swap(arr[randomIndex], arr[high]);  // Move random element to end

    return partitionLast(arr, low, high);  // Use last element partition
}

void quickSortRandom(vector<int> &arr, int low, int high) {
    if (low < high) {
        int pi = partitionRandom(arr, low, high);
        quickSortRandom(arr, low, pi - 1);
        quickSortRandom(arr, pi + 1, high);
    }
}
```

Comparison Graph & Table



	10000	50000	100000
Ascending	0.001	0.009	0.036
Descending	0.001	0.009	0.021
Random	0.002	0.012	0.026

Conclusion

- For **10k elements**: Lightning-fast across all input types (0.001–0.002s), showing immunity to input order.
- For **50k elements**: Still consistent (0.009–0.012s), scaling well with input size.
- For **100k elements**: Sorting time remains small (0.021–0.036s), proving that random pivot avoids worst-case consistently.

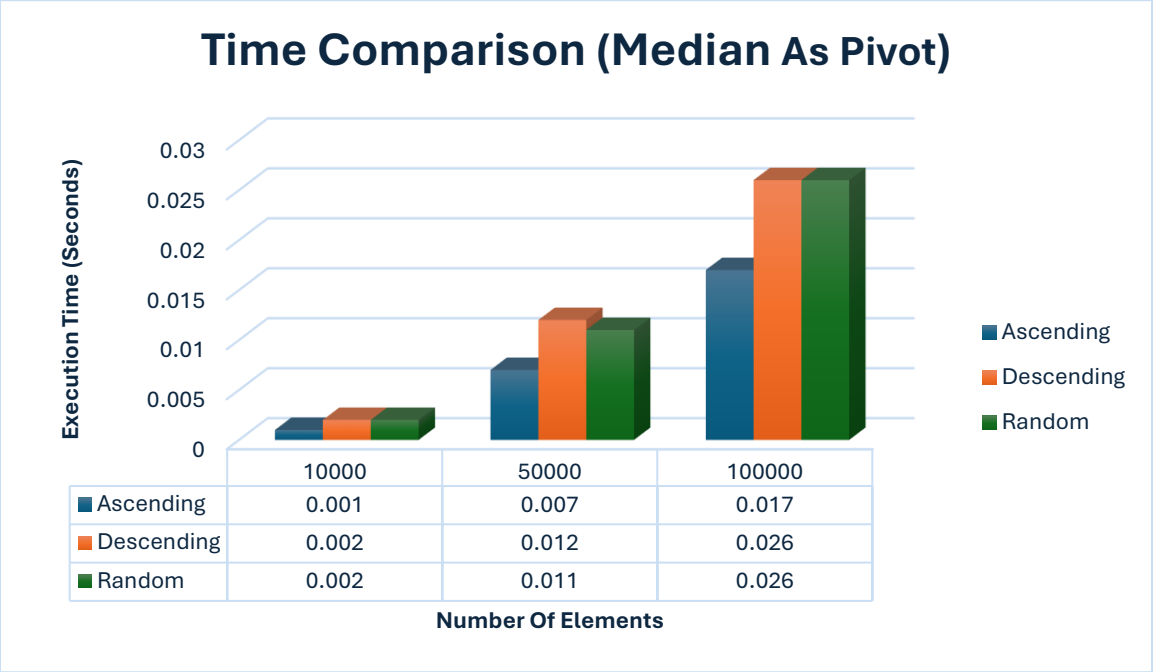
  Best balance of speed and stability.
  Works efficiently for ascending, descending, and random inputs alike.

4. Median as Pivot (Median-of-Three)

Main Logic (Code Snippet)

```
int medianOfThree(vector<int> &arr, int low, int high) {
    int mid = low + (high - low) / 2;

    if (arr[mid] < arr[low]) swap(arr[mid], arr[low]);
    if (arr[high] < arr[low]) swap(arr[high], arr[low]);
    if (arr[high] < arr[mid]) swap(arr[high], arr[mid]);

    return mid;
}

int partitionMedian(vector<int> &arr, int low, int high) {
    int medianIndex = medianOfThree(arr, low, high);
    swap(arr[medianIndex], arr[high]);  // Move median to end

    return partitionLast(arr, low, high);
}

void quickSortMedian(vector<int> &arr, int low, int high) {
    if (low < high) {
        int pi = partitionMedian(arr, low, high);
        quickSortMedian(arr, low, pi - 1);
        quickSortMedian(arr, pi + 1, high);
    }
}
```

Comparison Graph & Table



	10000	50000	100000
Ascending	0.001	0.007	0.017
Descending	0.002	0.012	0.026
Random	0.002	0.011	0.026

Conclusion

- For **10k elements**: Performs very fast (0.001–0.002s) on all input types, comparable to random pivot.
- For **50k elements**: Also, extremely efficient (0.0011–0.007s), unaffected by input order.
- For **100k elements**: Remains stable and consistent (0.017–0.026s), with negligible differences across ascending, descending, or random inputs.

Consistently delivers near-optimal performance.
Best suited for large inputs where balanced partitioning is crucial.

Final Analysis and Conclusion

From the analysis, we observe that **pivot selection strongly influences Quick Sort performance**:

- **First and Last Element pivots** work well only for random inputs but **perform very poorly on sorted and reverse-sorted data**, especially as input size grows ($O(n^2)$ behavior).

- **Random Pivot** consistently delivers fast results for all input sizes and patterns, avoiding worst cases with high probability.

- **Median-of-Three Pivot** provides the most **balanced and reliable performance**, matching random pivot while ensuring stable partitions for large datasets.

Overall, **Random and Median-of-Three pivot strategies are the most efficient and practical choices**, while **First and Last element pivots should be avoided when input may be ordered**.