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

Experiment 2

Aim: Implement **Merge Sort** and **External Merge Sort**, and analyze their performance on different input sizes (10,000, 50,000, and 100,000 elements) with different input patterns (ascending, descending, random). Also, to compare execution times using tables and graphs, and conclude their efficiency.

Overview

Sorting is a fundamental operation in data structures and algorithms. While algorithms like Bubble Sort, Insertion Sort, and Selection Sort are simple, they become inefficient for large datasets due to their **O(n^2)** complexity.

To overcome this, **Merge Sort** and **External Merge Sort** are widely used:

- **Merge Sort**: A divide-and-conquer sorting algorithm with O(nlogn) complexity, suitable for datasets that fit in main memory.

- **External Merge Sort**: A file-based extension of Merge Sort designed for **very large datasets** that cannot fit into memory. It sorts data in chunks and merges them using disk storage.

1. Merge Sort

Main Logic (Code Snippet)

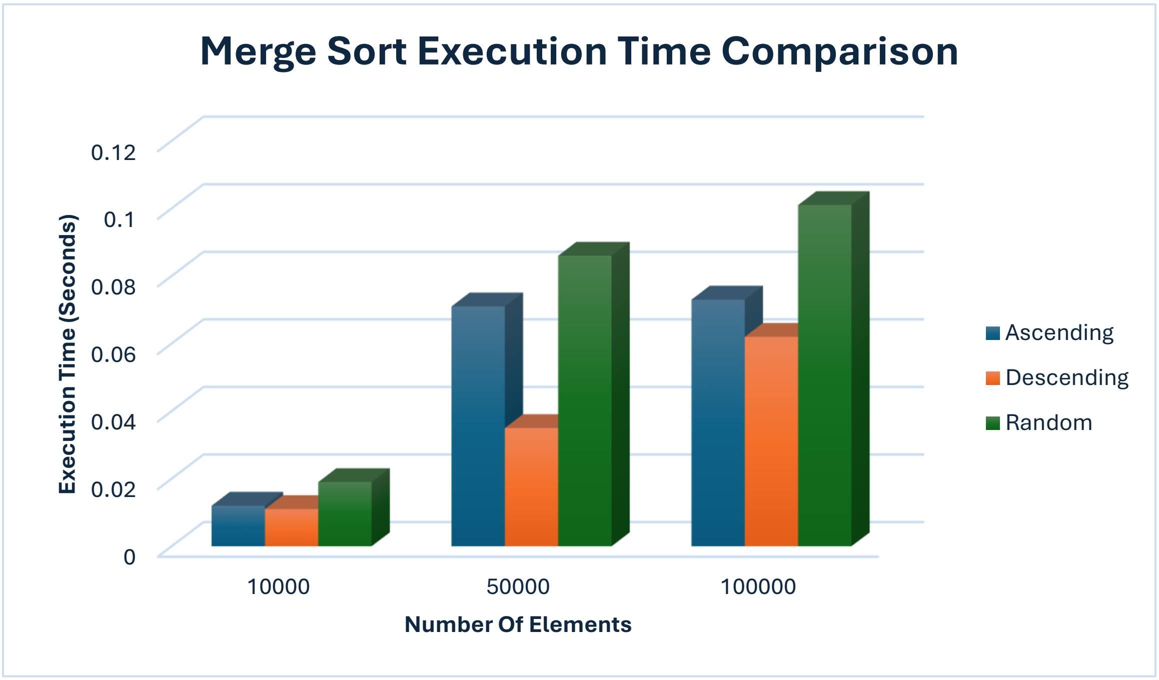
```
void merge(vector<int> &arr, int left, int mid, int right) {
   int n1 = mid - left + 1, n2 = right - mid;
   vector<int> L(n1), R(n2);
   for (int i = 0; i < n1; i++) L[i] = arr[left + i];
   for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];
   int i = 0, j = 0, k = left;
   while (i < n1 && j < n2) arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
   while (i < n1) arr[k++] = L[i++];
   while (j < n2) arr[k++] = R[j++];
}

void mergeSort(vector<int> &arr, int left, int right) {
   if (left < right) {
      int mid = left + (right - left) / 2;
```

```
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

Comparison Table

Elements	Ascending	Descending	Random
10000	0.012 Seconds	0.011 Seconds	0.019 Seconds
50000	0.071 Seconds	0.035 seconds	0.086 seconds
100000	0.073 seconds	0.062 seconds	0.101 seconds

Comparison Graph



Conclusion

- Merge Sort provides **fast and consistent performance** across all input types, unlike Bubble, Selection, or Insertion Sort.
- For 100,000 elements, execution is below 0.101 seconds, proving its efficiency.
- Minor variations (descending sometimes faster than ascending) are due to **CPU caching and memory access patterns**, not algorithmic differences.
- Merge Sort is suitable for **large datasets** that fit in memory, offering predictable **O(nlogn)** performance.

2. External Merge Sort

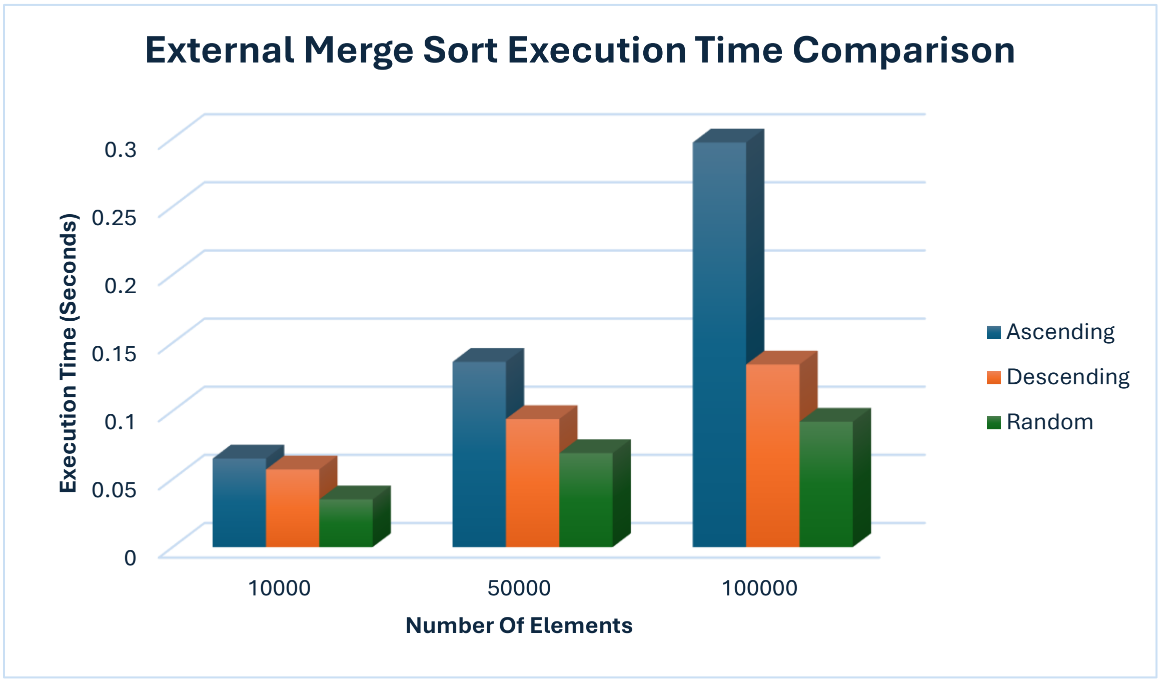
Main Logic (Code Snippet)

```
// Phase 1: Create sorted chunks
void createSortedChunks(string inputFile, int chunkSize, vector<string> &chunks) {
    ifstream fin(inputFile);
    vector<int> buffer; buffer.reserve(chunkSize);
    int value, chunkCount = 0;
    while (fin >> value) {
        buffer.push_back(value);
        if (buffer.size() == chunkSize) {
            sort(buffer.begin(), buffer.end());
            string chunkName = "chunk_" + to_string(chunkCount++) + ".txt";
            ofstream fout(chunkName);
            for (int x : buffer) fout << x << " ";
            fout.close();
            chunks.push_back(chunkName);
            buffer.clear();
        }
    }
    if (!buffer.empty()) { /* handle leftovers same way */ }
    fin.close();
}

// Phase 2: Merge sorted chunks
void mergeChunks(vector<string> &chunks, string outputFile) {
    vector<ifstream> fins(chunks.size());
    for (int i = 0; i < chunks.size(); i++) fins[i].open(chunks[i]);
    ofstream fout(outputFile);
    vector<int> current(chunks.size());
    vector<char> available(chunks.size(), 1);
    for (int i = 0; i < chunks.size(); i++) if (!(fins[i] >> current[i])) available[i] = 0;
    while (1) {
        int minIndex = -1, minValue = INT_MAX;
        for (int i = 0; i < chunks.size(); i++) if (available[i] && current[i] < minValue) { minValue = cu
rrent[i]; minIndex = i; }
        if (minIndex == -1) break;
        fout << minValue << " ";
        if (!(fins[minIndex] >> current[minIndex])) available[minIndex] = 0;
    }
    fout.close();
    for (auto &f : fins) f.close();
}
```

Comparison Table

Elements	Ascending	Descending	Random
10000	0.065 Seconds	0.057 Seconds	0.035 Seconds
50000	0.136 Seconds	0.094 seconds	0.069 seconds
100000	0.297 seconds	0.134 seconds	0.092 seconds

Comparison Graph



Conclusion

- External Merge Sort is essential when data **cannot fit into main memory**.
- It breaks input into **sorted chunks** and merges them efficiently from disk.
- Performance is slower than normal Merge Sort due to **disk I/O overhead**, but it can handle **huge datasets** beyond RAM size.
- Execution times scale predictably with dataset size.
- Used in **databases, search engines, and big data frameworks**.

Final Analysis and Conclusion

From the experimental results, we observe the following:

- **Merge Sort** consistently delivers excellent performance with execution times under 0.101 seconds for 100,000 elements. It is **input independent** and guarantees O(nlogn) time complexity, making it highly reliable for large datasets that fit into main memory.

- **External Merge Sort**, while slightly slower due to disk I/O overhead, allows sorting of datasets **much larger than available RAM**. Its performance scales predictably with input size and remains efficient for massive files.

Key Takeaways:

- For **moderately large datasets that fit in RAM**, **Merge Sort** is the preferred choice because of its speed and simplicity.

- For **very large datasets exceeding memory limits**, **External Merge Sort** is essential as it efficiently uses disk storage to handle the data.

- Both algorithms showcase the power of the **divide-and-conquer approach**, with Merge Sort being practical for in-memory tasks and External Merge Sort being crucial for real-world, large-scale applications such as databases, search engines, and big data frameworks.