NAME: MUHAMMAD AKBAR

SAPID: 44901

SECTION: CYBER SECURITY -5

SUBJECT: ANALYSIS OF ALGORITHMS

ASSIGNMENT: NO (4)

Assignment Title: Designing a Hybrid Sorting Algorithm Using Merge Sort and Quick Sort

Objective:

To design, implement, and analyze a hybrid sorting algorithm that combines the strengths of Merge Sort and Quick Sort to optimize performance across different input cases.

1. *Problem Selection*

The **sorting problem** is chosen for this assignment due to its fundamental role in computer science and its applicability across various data-intensive applications. Sorting is a widely studied problem with numerous algorithms, each offering unique trade-offs in terms of time and space efficiency. We focus on **Merge Sort** and **Quick Sort**, two powerful sorting algorithms that are widely used and exhibit complementary strengths.

Algorithms Selected:

- **Quick Sort**: An efficient, in-place, comparison-based sort that has an average-case time complexity of O(nlog⁡n)O(n \log n)O(nlogn) but suffers from a worst-case time complexity of O(n2)O(n^2)O(n2).
- **Merge Sort**: A reliable, stable sorting algorithm with consistent O(nlog⁡n)O(n \log n)O(nlogn) time complexity in all cases. However, it requires additional memory space, which can be a disadvantage in space-limited environments.

2. Analysis of Algorithms

Detailed Strengths and Weaknesses:

- **Quick Sort:**

  - **Strengths:**
    - **Efficient for small datasets**: Quick Sort is often faster than Merge Sort for smaller arrays due to fewer memory allocations.
    - **In-place sorting:** Quick Sort sorts without additional memory allocation, making it more space-efficient.
    - **Cache efficiency**: Quick Sort's partitioning step makes better use of CPU cache, speeding up the sorting process on small arrays.
  - **Weaknesses:**
    - **Worst-case performance**: When the pivot is consistently chosen poorly (e.g., the smallest or largest element), Quick Sort degenerates to O(n2)O(n^2)O(n2), which can be catastrophic for performance.
    - **Instability:** Quick Sort is not a stable sort, meaning equal elements might not retain their relative positions.

- **Merge Sort:**

  - **Strengths:**
    - **Consistent performance:** Merge Sort guarantees O(nlog⁡n)O(n \log n)O(nlogn) performance in all cases, including worst-case scenarios.
    - **Stability**: It's a stable sort, which is beneficial when sorting data with multiple attributes.
    - **Well-suited for large datasets**: Merge Sort performs well on larger datasets and lists that are already mostly sorted.

- **Weaknesses**:
    - **Space complexity**: Merge Sort requires additional memory for merging, which can be limiting in memory-constrained environments.
    - **Memory allocation overhead**: The allocation and copying involved in Merge Sort can slow down execution, especially on smaller arrays.

Optimal Scenarios:

- **Quick Sort** is preferable for smaller, random datasets.
- **Merge Sort** is advantageous for larger datasets or datasets where the worst-case time complexity of Quick Sort is a concern.

3. Hybrid Design

The hybrid algorithm, **HybridSort**, will blend Quick Sort and Merge Sort to capitalize on their strengths. We'll define a **threshold** to switch between Quick Sort and Merge Sort depending on the subarray size.

- **Algorithm Switching**:
    - **Threshold-Based Switching**: When sorting subarrays, we use Quick Sort if the size of the subarray is below a certain threshold (e.g., 10 elements), as Quick Sort generally performs faster for small inputs.
    - **Fallback to Merge Sort**: For larger subarrays, Merge Sort's predictable O(nlog⁡n)O(n \log n)O(nlogn) behavior and stability make it more efficient.
- **Hybrid Algorithm Design**:
    - Partition the array using Quick Sort's in-place partitioning.
    - If a partitioned subarray's size is below the threshold, apply Quick Sort on it.
    - If the partitioned subarray's size is above the threshold, apply Merge Sort on it for stable performance.

Hybrid Algorithm Implementation

```
#include <iostream>

#include <vector>

// Quick Sort Partition

int partition(std::vector<int>& arr, int low, int high) {

    int pivot = arr[high];

    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {

        if (arr[j] < pivot) {

            i++;

            std::swap(arr[i], arr[j]);

        }

    }

    std::swap(arr[i + 1], arr[high]);

    return (i + 1);

}

// Quick Sort Function

void quickSort(std::vector<int>& arr, int low, int high) {

    if (low < high) {

        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
// Merge Sort Merge Step
void merge(std::vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    std::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) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
// Merge Sort Function
void mergeSort(std::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);
    }
}
// Hybrid Sorting Function
void hybridSort(std::vector<int>& arr, int left, int right) {
    const int threshold = 10; // Threshold for switching
    if ((right - left) <= threshold) {
        quickSort(arr, left, right); // Use Quick Sort for small segments
    } else {
        mergeSort(arr, left, right); // Use Merge Sort for larger segments
    }
}
int main() {
    std::vector<int> arr = {34, 7, 23, 32, 5, 62, 32, 1, 12};
    int n = arr.size();
    hybridSort(arr, 0, n - 1);
    std::cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Performance Analysis

Theoretical Analysis:

- **Time Complexity**: The hybrid approach generally maintains O(nlog⁡n)O(n \log n)O(nlogn) time complexity by switching to Merge Sort for large subarrays and using Quick Sort for small subarrays.
- **Space Complexity**: The hybrid algorithm balances Quick Sort's in-place advantage with Merge Sort's space requirement, achieving a middle ground.

Experimental Analysis:

1. **Execution Time Comparison**: Measure execution time for different input sizes (e.g., 1000, 10,000, and 100,000 elements).
2. **Edge Cases**:
   - Pre-sorted arrays to test worst-case scenarios for Quick Sort.
   - Randomized and reverse-sorted arrays.
3. **Memory Usage**: Observe memory usage through profiling tools, especially for large datasets.

This hybrid algorithm combines the best aspects of both Quick Sort and Merge Sort, achieving balanced performance and efficiency across various input scenarios.

GITHUB LINK : https://github.com/harry44901/AnalysisOfAlgorithms

======================== THE END ===========================