DAA Assignment No 4

Name:Najeebullah
Sap id:45824
Git hub link:https://github.com/knajee/Assignment-No-4-DAA.git

Designing a Hybrid Algorithm

Problem Selection:

Let's select **sorting** as our problem, which can be solved using various algorithms.

Algorithm A: Quick Sort

Algorithm B: Merge Sort

Analysis:

Quick Sort:

Strengths: Fast average-case performance (O(n log n)), good cache performance.

Weaknesses: Poor worst-case performance (O(n^2)) with highly unbalanced partitions.

Merge Sort:

Strengths: Guarantees O(n log n) performance, stable, and good for linked lists.

Weaknesses: Requires additional memory (O(n)).

Hybrid Design:

To leverage the strengths of both algorithms, our hybrid algorithm will use Quick Sort for its average-case speed and switch to Merge Sort when it detects that partitions are becoming unbalanced.

- **Initial Partitioning**: Use Quick Sort for partitioning until the partitions reach a small threshold size.

- **Balanced Sorting**: Switch to Merge Sort for these smaller sublists to ensure consistent O(n log n) performance.

pseudocode for a hybrid sorting algorithm combining Quick Sort and Merge Sort in C++

```
#include <iostream>
```

```
#include <vector>
using namespace std;

// Function to partition the array (Quick Sort logic)
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    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;
}

// Merge function (Merge Sort logic)
void merge(vector<int>& arr, int low, int mid, int high) {
    int n1 = mid - low + 1;
    int n2 = high - mid;

    vector<int> left(n1);
```

```
vector<int> right(n2);

for (int i = 0; i < n1; ++i)
    left[i] = arr[low + i];
for (int j = 0; j < n2; ++j)
    right[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = low;
while (i < n1 && j < n2) {
    if (left[i] <= right[j]) {
        arr[k] = left[i];
        ++i;
    } else {
        arr[k] = right[j];
        ++j;
    }
    ++k;
}

while (i < n1) {
    arr[k] = left[i];
    ++i;
    ++k;
}
```

```
    while (j < n2) {
        arr[k] = right[j];
        ++j;
        ++k;
    }
}

// Merge Sort function
void mergeSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int mid = low + (high - low) / 2;

        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);

        merge(arr, low, mid, high);
    }
}

// Hybrid Sort function
void hybridSort(vector<int>& arr, int low, int high, int threshold)
{
    if (high - low + 1 <= threshold) {
```

```
            mergeSort(arr, low, high);
        } else {
            int pivot = partition(arr, low, high);
            hybridSort(arr, low, pivot - 1, threshold);
            hybridSort(arr, pivot + 1, high, threshold);
        }
    }

    // Main function to test the hybrid sort
    int main() {
        vector<int> arr = {38, 27, 43, 3, 9, 82, 10};
        int threshold = 10; // You can adjust this threshold

        hybridSort(arr, 0, arr.size() - 1, threshold);

        cout << "Sorted array: ";
        for (int val : arr) {
            cout << val << " ";
        }

        return 0;
    }
```

Example Array:

```
[38, 27, 43, 3, 9, 82, 10]
```

Steps:

1.

Initial Array: `[38, 27, 43, 3, 9, 82, 10]`

2.
3.

Threshold: `10`

4.
    1.

    Since the length of the array (7) is less than the threshold (10), the algorithm will use `mergeSort` directly instead of `quickSort`.

    2.

Merge Sort Process:

Divide:

Split the array into two halves: `[38, 27, 43]` and `[3, 9, 82, 10]`

- **Divide Further**:

  For `[38, 27, 43]`: Split into `[38]` and `[27, 43]`

  For `[27, 43]`: Split into `[27]` and `[43]`

  For `[3, 9, 82, 10]`: Split into `[3, 9]` and `[82, 10]`

  For `[3, 9]`: Split into `[3]` and `[9]`

  For `[82, 10]`: Split into `[82]` and `[10]`

- **Merge**:

  Merge `[38]` and `[27, 43]`

  `[27, 43]` is sorted first into `[27, 43]`

  Then merged with `[38]` to form `[27, 38, 43]`

  Merge `[3, 9]` and `[82, 10]`

  `[3]` and `[9]` merge to `[3, 9]`

`[82]` and `[10]` merge to `[10, 82]`

`[3, 9]` and `[10, 82]` merge to form `[3, 9, 10, 82]`

Finally, merge `[27, 38, 43]` and `[3, 9, 10, 82]` to get `[3, 9, 10, 27, 38, 43, 82]`

Output:

Sorted array: 3 9 10 27 38 43 82

Explanation:

Divide and Conquer: The array is recursively split into smaller subarrays until they can be easily sorted and then merged back together in order.

Merge Sort: Ensures the array is sorted with guaranteed O(n log n) performance.

Hybrid Logic: The algorithm leverages merge sort's consistency and stability by using it when the array size is below the threshold.

The output is a sorted version of the initial array, showcasing how the hybrid algorithm efficiently handles the sorting process by combining the strengths of Quick Sort and Merge Sort. Let me know if you have more questions!

Performance Analysis:

Theoretical:

Quick Sort: Efficient initial partitioning.

Merge Sort: Stable and consistent performance on smaller partitions.

Experimental:

Test and compare the hybrid algorithm's performance against pure Quick Sort and Merge Sort using various datasets.

Key Considerations for Hybridization:

Problem Characteristics: Sorting, which benefits from both algorithms.

Algorithm Complementarity: Quick Sort's speed complements Merge Sort's stability.

Hybrid Strategy: Switching based on input size to optimize performance.

Parameter Tuning: Adjusting the threshold for optimal results.

Experimental Validation: Rigorous testing on different datasets to validate performance improvements.

This hybrid approach combines the speed of Quick Sort with the stability of Merge Sort, optimizing sorting performance across various scenarios. What do you think of this approach?