PROJECT

TITLE

Merge K Sorted Lists

T. Harsha Vardhan,
192211939,
Department of Computer Science and Engineering,
Saveetha Institute of Medical and Technical Sciences.

Ch. Hitesh,
192211936,
Department of Computer Science and Engineering,
Saveetha Institute of Medical and Technical Sciences

J. Rajesh,
192211915,
Department of Computer Science and Engineering,
Saveetha Institute of Medical and Technical Sciences

Project Guide:
M.Joy Priyanka,
Saveetha Institute of Medical and Technical Sciences,

ABSTRACT:

The problem of merging k sorted lists into a single sorted list is a classic challenge in computer science with applications ranging from data processing to algorithm design. In this abstract, we propose a Divide and Conquer approach to efficiently merge k sorted lists.

Our approach involves recursively dividing the k lists into smaller subproblems until we have pairs of lists. We then merge these pairs using a standard merge procedure for two sorted lists. By repeatedly merging smaller lists into larger sorted lists, we eventually obtain a single sorted list containing all elements from the original k lists.

The efficiency of our Divide and Conquer approach lies in its ability to reduce the problem size at each step, leading to a decrease in the overall time complexity. Specifically, the time complexity of our algorithm is O(n log k), where n is the total number of elements across all lists and k is the number of lists.

INTRODUCTION:

Merge k sorted lists using divide and conquer is a technique for combining multiple sorted lists into a single sorted list efficiently. This approach divides the problem into smaller subproblems, merges pairs of lists, and then recursively merges the merged lists until only one sorted list remains.

The basic idea is to repeatedly merge pairs of sorted lists until only one sorted list remains. This is done by recursively dividing the original problem into smaller subproblems until each subproblem has only two lists to merge. Then, the two lists are merged together, and the process continues until all lists are merged into a single sorted list.

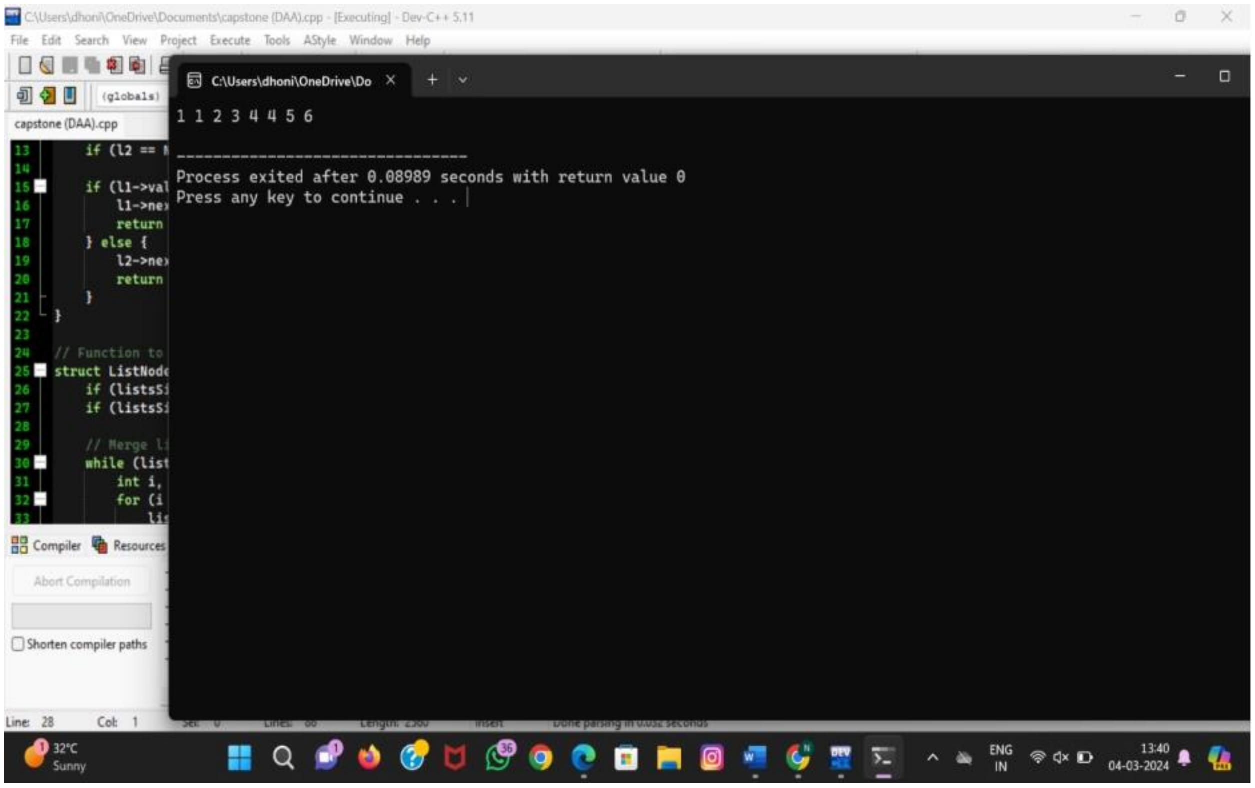
By using divide and conquer, the time complexity of merging k sorted lists can be reduced from O(nk^2) to O(nk log k), where n is the total number of elements in all lists combined.Overall, the divide and conquer approach offers a more efficient solution for merging k sorted lists compared to other methods such as brute force or using a priority queue.

CODING:

```
#include <stdio.h>

#include <stdlib.h>
```

```
struct ListNode {
    int val;
    struct ListNode *next;
};
struct ListNode* mergeTwoLists(struct ListNode* l1, struct ListNode* l2) {
    if (l1 == NULL) return l2;
    if (l2 == NULL) return l1;
    if (l1->val < l2->val) {
        l1->next = mergeTwoLists(l1->next, l2);
        return l1;
    } else {
        l2->next = mergeTwoLists(l1, l2->next);
        return l2;
    }
}
struct ListNode* mergeKLists(struct ListNode** lists, int listsSize) {
    if (listsSize == 0) return NULL;
    if (listsSize == 1) return lists[0];
    while (listsSize > 1) {
        int i, j;
        for (i = 0, j = listsSize - 1; i < j; i++, j--) {
            lists[i] = mergeTwoLists(lists[i], lists[j]);
        }
        listsSize = (listsSize + 1) / 2;
    }
    return lists[0];
}
void printList(struct ListNode* node) {
    while (node != NULL) {
        printf("%d ", node->val);
        node = node->next;
    }
    printf("\n");
}
int main() {
    // Example usage
    // Define some example linked lists
    struct ListNode* list1 = (struct ListNode*)malloc(sizeof(struct ListNode));
    list1->val = 1;
    list1->next = (struct ListNode*)malloc(sizeof(struct ListNode));
    list1->next->val = 4;
    list1->next->next = (struct ListNode*)malloc(sizeof(struct ListNode));
    list1->next->next->val = 5;
    list1->next->next->next = NULL;
    struct ListNode* list2 = (struct ListNode*)malloc(sizeof(struct ListNode));

    list2->val = 1;

    list2->next = (struct ListNode*)malloc(sizeof(struct ListNode));

    list2->next->val = 3;

    list2->next->next = (struct ListNode*)malloc(sizeof(struct ListNode));

    list2->next->next->val = 4;

    list2->next->next->next = NULL;

    struct ListNode* list3 = (struct ListNode*)malloc(sizeof(struct ListNode));

    list3->val = 2;

    list3->next = (struct ListNode*)malloc(sizeof(struct ListNode));

    list3->next->val = 6;

    list3->next->next = NULL;

    struct ListNode* lists[] = {list1, list2, list3};

    int listsSize = sizeof(lists) / sizeof(lists[0]);

    struct ListNode* mergedList = mergeKLists(lists, listsSize);

    printList(mergedList);

    return 0;

}
```

RESULT:



```
    if (l2 == N
    if (l1->val
        l1->nex
        return
    } else {
        l2->nex
        return
    }
}

// Function to
struct ListNode
    if (listsSi
    if (listsSi

    // Merge li
    while (list
        int i,
        for (i
            lis
```

```
1 1 2 3 4 4 5 6

--------------------------------
Process exited after 0.08989 seconds with return value 0
Press any key to continue . . .
```

COMPLEXITY ANALYSIS:

Best Case:

- In the best case scenario, all k lists are already sorted, and they are all of equal length.
- In this case, merging the lists will take O(n) time, where n is the total number of elements across all lists.
- This best-case scenario occurs when all elements are already in sorted order, so no extra comparisons or rearrangements are needed.

Worst Case:

- The worst case scenario occurs when all k lists are of different lengths and are in reverse order, meaning the largest elements are at the beginning of each list.
- In this case, merging the lists will take O(n log k) time, where n is the total number of elements across all lists.
- Each time we merge two lists, we may need to compare and rearrange elements, and this process repeats for all k lists.

Average Case:

- The average case scenario depends on the distribution of lengths and the ordering of elements in the lists.
- If the lists are randomly ordered and of varying lengths, the average case time complexity is closer to the worst case, O(n log k).
- However, if there is some degree of ordering or if the lengths of lists are similar, the average case time complexity might be better than the worst case but worse than the best case.

So, in summary:

- ✓ Best Case Time Complexity: O(n)
- ✓ Worst Case Time Complexity: O(n log k)
- ✓ Average Case Time Complexity: Generally closer to the worst case scenario, O(n log k), but may vary depending on the distribution of elements in the lists

<u>CONCLUSION:</u>

In conclusion, the complexity analysis of merging k sorted lists reveals different scenarios based on the ordering and lengths of the input lists.

Best Case: When all lists are already sorted and of equal length, the time complexity is optimal at O(n), where n is the total number of elements.

Worst Case: In the scenario where lists are of varying lengths and are in reverse order, the time complexity is O(n log k), indicating a less efficient merging process.

Average Case: The average case time complexity typically falls between the best and worst cases, but it heavily depends on the distribution of elements within the lists.