Merge K Sorted List

Presented by :

- Deep Pathak
- Prince Patel

Overview

- Introduction of Problem
- Challanges in Problem
- Brute Force
- Divide and Conquer
- Optimal Approach
- Dry Run
- Comaparison of Approaches
- Application

Problem

We are given an array of K sorted linked lists, where each linked list is individually sorted in ascending order. The task is to merge all these lists into a single sorted linked list and return the final result. This problem is important because merging multiple sorted sequences is a common requirement in real-world applications like databases, search engines, and external sorting.

For example, consider lists = [[1,4,5], [1,3,4], [2,6]].

Each represents a sorted linked list:

- List 1 : 1 → 4 → 5
- List 2 : 1 → 3 → 4
- List 3 : 2 → 6

After merging them into one sorted linked list, the final output becomes:

1 → 1 → 2 → 3 → 4 → 4 → 5 → 6

Challanges

- Handling multiple linked lists at once instead of just two.
- Maintaining the sorted order while merging K lists.
- Avoiding excessive time complexity when K and total nodes are large.
- Efficiently traversing all nodes without missing or repeating elements.
- Managing different lengths of linked lists (some may be empty).
- Preventing high memory usage when storing intermediate results
- Choosing the right data structure (simple merge, divide & conquer, or heap).
- Ensuring the algorithm works for extreme constraints (10,000 lists, 50,000+ nodes).

(Solution) → Brute Force

The brute force approach works by first traversing all K linked lists and collecting every element into an array. Then, we sort this array to ensure the elements are in ascending order.

- Finally, we build a new linked list from the sorted elements. This method is simple and easy to implement but not very efficient.
- Its main drawback is the sorting step, which increases time complexity.
- Thus, it is useful for small inputs but not optimal for large datasets.

```
for(int i=0;i<list.size();i++)
{
  Node temp=list[i];
  while(temp!=NULL)
  {
    arr.add(temp->data);
    temp=temp->next;
  }
  sort(arr.begin(),arr.end());
  Node newHead=convert(arr);
  return newHead;
}
```

(Solution) → Divide & Conqure

In the divide and conquer approach, we merge the K lists in pairs until only one final sorted list remains. This is similar to the merge step in merge sort. At each step, we reduce the number of lists approximately by half. This significantly improves efficiency compared to brute force. The process continues until all lists are merged into one. The overall time complexity is O(N log K), where N is the total number of nodes.

```
ListNode*mergeTwoLists(ListNode*l1,ListNode* l2)
ListNode temp(0);
    ListNode* curr = &temp;
    while (l1 && l2) {
        if (l1->val < l2->val) {
            curr->next = l1;
            l1 = l1->next;
        }
        else {   curr->next = l2;
            l2 = l2->next;    }
        curr = curr->next;    }
    curr->next = l1 ? l1 : l2;
    return temp.next;
}

ListNode* mergeKLists(vector<ListNode*>& lists) {
    if (lists.empty()) return nullptr;
    while (lists.size() > 1) {
        vector<ListNode*> merged;
        for (int i = 0; i < lists.size(); i += 2) {
            if (i + 1 < lists.size()){
 merged.push_back(mergeTwoLists(lists[i], lists[i+1]);}
            else{   merged.push_back(lists[i]);    }
        lists = merged;
    }
    return lists[0];
}
```

(Solution) →Optimal Approach

In the optimal approach, we use a min heap (priority queue) to always extract the smallest node among the K lists. Initially, the first node of each list is inserted into the heap. Then, repeatedly remove the smallest node from the heap and add its next node (if present). This ensures that we always maintain the sorted order while merging. The process continues until the heap becomes empty. The time complexity is O(N log K), which is optimal for this problem.

```
struct Compare {
    bool operator()(ListNode* a, ListNode* b) {
        return a->val > b->val;     }             };

ListNode* mergeKLists(vector<ListNode*>& lists) {
priority_queue<ListNode*,vector<ListNode*>,Compare> minHeap;
  for (auto l : lists) {
    if (l) minHeap.push(l);   }
ListNode temp(0);
  ListNode* curr = &temp;

  while (!minHeap.empty()) {
    ListNode* node = minHeap.top();
    minHeap.pop();

    curr->next = node;
    curr = curr->next;

    if (node->next) minHeap.push(node->next);
  }

  return temp.next;
}
```

Dry Run

Step 1: Insert the first node of each list into the min heap → {1, 1, 2}

Step 2: Extract the smallest (1), add it to result. Push its next node (4) → Heap = {1, 2, 4}
Result = 1

Step 3: Extract smallest (1), add to result. Push its next node (3) → Heap = {2, 3, 4}
Result = 1 → 1

Step 4: Extract (2), add to result. Push its next node (6) → Heap = {3, 4, 6}
Result = 1 → 1 → 2

Step 5: Extract (3), add to result. Push next node (4) → Heap = {4, 4, 6}
Result = 1 → 1 → 2 → 3

Step 6: Extract (4), add to result. Push next node (5) → Heap = {4, 5, 6}
Result = 1 → 1 → 2 → 3 → 4

Step 7: Extract (4), add to result. Push next node (NULL) → Heap = {5, 6}
Result = 1 → 1 → 2 → 3 → 4 → 4

Step 8: Extract (5), add to result. Push next node (NULL) → Heap = {6}
Result = 1 → 1 → 2 → 3 → 4 → 4 → 5

Step 9: Extract (6), add to result. Heap empty → Stop.
Final Result = 1 → 1 → 2 → 3 → 4 → 4 → 5 → 6

Factor	Brute Force	Divide & Conquer	Min Heap (Optimal)
Implementation	Very simple, easy to write	Moderate, recursive/iterative merging	Slightly complex (heap required)
Time Complexity	O(N log N) (due to sorting)	O(N log K)	O(N log K)
Space Complexity	O(N) (array for storing values)	O(1) extra (in-place merging)	O(K) (heap stores one node per list)
Efficiency	Poor for large K	Good for large K	Best and most practical
When to Use	Small inputs, learning purpose	Balanced performance & clarity	Large inputs, real-world systems
Drawbacks	Sorting overhead, not scalable	Extra merging steps required	Slightly harder to implement

Application

1. Database Systems – Efficiently merging results from multiple sorted queries.
2. Search Engines – Combining sorted results from different servers or indexes.
3. Big Data & External Sorting – Handling large datasets that don’t fit in memory.
4. File Systems / Logs – Merging sorted log files from distributed systems.
5. Job Scheduling & Task Management – Merging sorted task queues in cloud or OS.

Thank You

For your attention