Merge K Sorted Lists (medium)

We'll cover the following

Problem Statement

Given an array of 'K' sorted LinkedLists, merge them into one sorted list.

Example 1:

```
Input: L1=[2, 6, 8], L2=[3, 6, 7], L3=[1, 3, 4]
Output: [1, 2, 3, 3, 4, 6, 6, 7, 8]
```

Example 2:

```
Input: L1=[5, 8, 9], L2=[1, 7]
Output: [1, 5, 7, 8, 9]
```

Try it yourself

Try solving this question here:

C++

```cpp
using namespace std;

#include <iostream>
#include <queue>
#include <vector>

class ListNode {
 public:
  int value = 0;
  ListNode *next;

  ListNode(int value) {
    this->value = value;
    this->next = nullptr;
  }
};

class MergeKSortedLists {
 public:
  static ListNode *merge(const vector<ListNode *> &lists) {
    ListNode *resultHead = nullptr;
    // TODO: Write your code here
    return resultHead;
  }
};

int main(int argc, char *argv[]) {
  ListNode *l1 = new ListNode(2);
  l1->next = new ListNode(6);
  l1->next->next = new ListNode(8);

```

Solution#

A brute force solution could be to add all elements of the given ‘K’ lists to one list and sort it. If there are a total of ‘N’ elements in all the input lists, then the brute force solution will have a time complexity of $O(N * logN)$ as we will need to sort the merged list. Can we do better than this? How can we utilize the fact that the input lists are individually sorted?

If we have to find the smallest element of all the input lists, we have to compare only the smallest (i.e. the first) element of all the lists. Once we have the smallest element, we can put it in the merged list. Following a similar pattern, we can then find the next smallest element of all the lists to add it to the merged list.

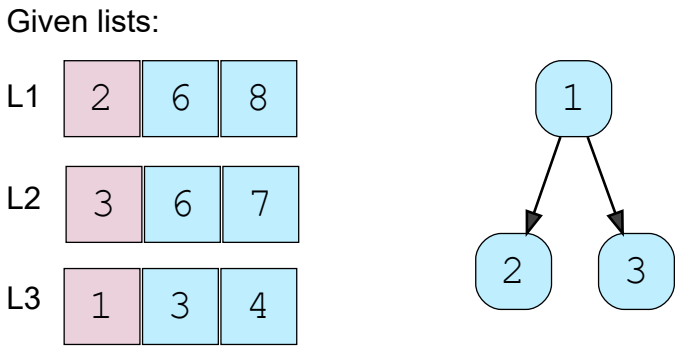
The best data structure that comes to mind to find the smallest number among a set of 'K' numbers is a **Heap**. Let's see how can we use a heap to find a better algorithm.

1. We can insert the first element of each array in a **Min Heap**.
2. After this, we can take out the smallest (top) element from the heap and add it to the merged list.
3. After removing the smallest element from the heap, we can insert the next element of the same list into the heap.
4. We can repeat steps 2 and 3 to populate the merged list in sorted order.

Let's take the Example-1 mentioned above to go through each step of our algorithm:

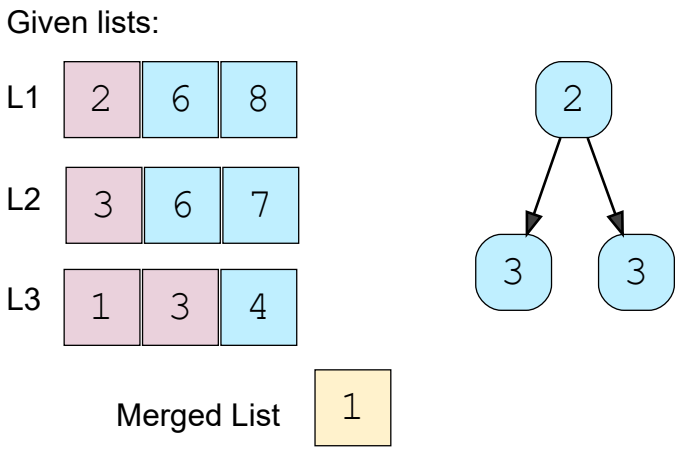
Given lists: `L1=[2, 6, 8]`, `L2=[3, 6, 7]`, `L3=[1, 3, 4]`

1. After inserting the 1st element of each list, the heap will have the following elements:

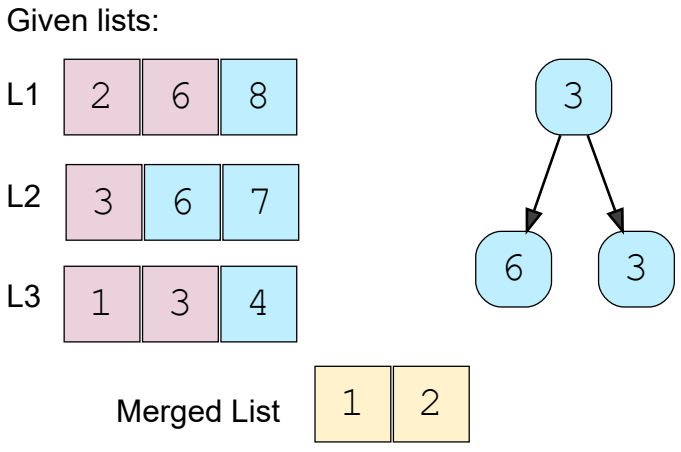


Insert the first number from each array in the heap

2. We'll take the top number from the heap, insert it into the merged list and add the next number in the heap.

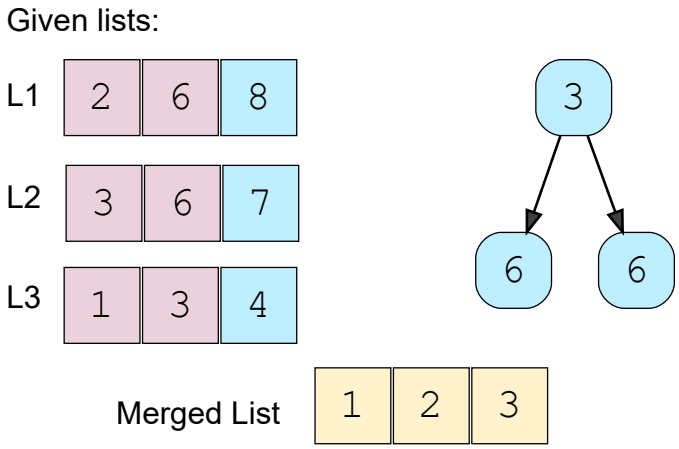


3. Again, we'll take the top element of the heap, insert it into the merged list and add the next number to the heap.



4. Repeating the above step, take the top element of the heap, insert it into the merged list and add the next number to the heap. As there are two

3s in the heap, we can pick anyone but we need to take the next element from the corresponding list to insert in the heap.



We'll repeat the above step to populate our merged array.

Code

Here is what our algorithm will look like:

C++

```
};

class MergeKSortedLists {
 public:
  struct valueCompare {
    bool operator()(const ListNode *x, const ListNode *y) { return x->value > y->val
  };

    priority_queue<ListNode *, vector<ListNode *>, valueCompare> minHeap;

    // put the root of each list in the min heap
    for (auto root : lists) {
      if (root != nullptr) {
        minHeap.push(root);
      }
    }

```

```
34      // take the smallest (top) element form the min-heap and t
35      // if the top element has a next element add it to the h
36      ListNode *resultHead = nullptr, *resultTail = nullptr;
37      while (!minHeap.empty()) {
38        ListNode *node = minHeap.top();
39        minHeap.pop();
40        if (resultHead == nullptr) {
41          resultHead = resultTail = node;
42        } else {
43          resultTail->next = node;
44          resultTail = resultTail->next;
45        }
46        if (node->next != nullptr) {
```

Time complexity#

Since we'll be going through all the elements of all arrays and will be removing/adding one element to the heap in each step, the time complexity of the above algorithm will be $O(N * logK)$, where 'N' is the total number of elements in all the 'K' input arrays.

Space complexity#

The space complexity will be $O(K)$ because, at any time, our min-heap will be storing one number from all the 'K' input arrays.

← Back

Introduction

Next →

Kth Smallest Number in M Sorted List...

Mark as Completed

Report an Issue