## Merge K sorted linked lists

Given K sorted linked lists of size N each, merge them and print the sorted output.

Example:

```
Input: k = 3, n =  4
list1 = 1->3->5->7->NULL
list2 = 2->4->6->8->NULL
list3 = 0->9->10->11

Output:
0->1->2->3->4->5->6->7->8->9->10->11
```

**Method 1 (Simple)**

A Simple Solution is to initialize result as first list. Now traverse all lists starting from second list. Insert every node of currently traversed list into result in a sorted way. Time complexity of this solution is $O(N^2)$ where N is total number of nodes, i.e., N = kn.

**Method 2 (Using Min Heap)**

A **Better solution** is to use Min Heap based solution which is discussed here for arrays. Time complexity of this solution would be O(nk Log k)

**Method 3 (Using Divide and Conquer))**

In this post, **Divide and Conquer** approach is discussed. This approach doesn't require extra space for heap and works in O(nk Log k)

We already know that merging of two linked lists can be done in O(n) time and O(1) space (For arrays O(n) space is required). The idea is to pair up K lists and merge each pair in linear time using O(1) space. After first cycle, K/2 lists are left each of size 2*N. After second cycle, K/4 lists are left each of size 4*N and so on. We repeat the procedure until we have only one list left.

Below is C++ implementation of the above idea.

```
// C++ program to merge k sorted arrays of size n each
#include <bits/stdc++.h>
using namespace std;

// A Linked List node
struct Node
{
    int data;
    Node* next;
};

/* Function to print nodes in a given linked list */
void printList(Node* node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Takes two lists sorted in increasing order, and merge
   their nodes together to make one big sorted list. Below
   function takes O(Log n) extra space for recursive calls,
   but it can be easily modified to work with same time and
   O(1) extra space  */
```

```
Node* SortedMerge(Node* a, Node* b)
{
    Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return (b);
    else if(b == NULL)
        return (a);

    /* Pick either a or b, and recur */
    if(a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }

    return result;
}

// The main function that takes an array of lists
// arr[0..last] and generates the sorted output
Node* mergeKLists(Node* arr[], int last)
{
    // repeat until only one list is left
    while (last != 0)
    {
        int i = 0, j = last;

        // (i, j) forms a pair
        while (i < j)
        {
            // merge List i with List j and store
            // merged list in List i
            arr[i] = SortedMerge(arr[i], arr[j]);

            // consider next pair
            i++, j--;

            // If all pairs are merged, update last
            if (i >= j)
                last = j;
        }
    }

    return arr[0];
}

// Utility function to create a new node.
Node *newNode(int data)
{
    struct Node *temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    int k = 3; // Number of linked lists
    int n = 4; // Number of elements in each list

    // an array of pointers storing the head nodes
    // of the linked lists
    Node* arr[k];
```

```
    arr[0] = newNode(1);
    arr[0]->next = newNode(3);
    arr[0]->next->next = newNode(5);
    arr[0]->next->next->next = newNode(7);

    arr[1] = newNode(2);
    arr[1]->next = newNode(4);
    arr[1]->next->next = newNode(6);
    arr[1]->next->next->next = newNode(8);

    arr[2] = newNode(0);
    arr[2]->next = newNode(9);
    arr[2]->next->next = newNode(10);
    arr[2]->next->next->next = newNode(11);

    // Merge all lists
    Node* head = mergeKLists(arr, k - 1);

    printList(head);

    return 0;
}
```

Output :

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Time Complexity of above algorithm is O(nk logk) as outer while loop in function mergeKLists() runs log k times and every time we are processing nk elements.