Merge k sorted arrays | Set 1

Given **K** sorted arrays of size **N** each, merge them and print the sorted output.

Examples:

Input: *K = 3, N = 4, arr = { {1, 3, 5, 7}, {2, 4, 6, 8}, {0, 9, 10, 11}}*

Output: *0 1 2 3 4 5 6 7 8 9 10 11*

Explanation: *The output array is a sorted array that contains all the elements of the input matrix.*

Input: *k = 4, n = 4, arr = { {1, 5, 6, 8}, {2, 4, 10, 12}, {3, 7, 9, 11}, {13, 14, 15, 16}}*

Output: *1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16*

Explanation: *The output array is a sorted array that contains all the elements of the input matrix.*

Naive Approach for Merging k sorted arrays:

*Create an output array of size (N * K) and then copy all the elements into the output array followed by sorting.*

Follow the given steps to solve the problem:

- Create an output array of size N * K.
- Traverse the matrix from start to end and insert all the elements in the output array.
- Sort and print the output array.

Below is the implementation of the above approach:

C++14

```
// C++ program to merge K sorted arrays of size n each.


#include <bits/stdc++.h>

using namespace std;

#define N 4


// A utility function to print array elements

void printArray(int arr[], int size)
```

```
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
}

// This function takes an array of arrays as an argument and
// All arrays are assumed to be sorted. It merges them
// together and prints the final sorted output.
void mergeKArrays(int arr[][N], int a, int output[])
{
    int c = 0;

    // traverse the matrix
    for (int i = 0; i < a; i++) {
        for (int j = 0; j < N; j++)
            output = arr[i][j];
    }

    // sort the array
    sort(output, output + N * a);
}

// Driver's code
int main()
{
    // Change N at the top to change number of elements
    // in an array
```

```
    int arr[][N] = { { 2, 6, 12, 34 },
                     { 1, 9, 20, 1000 },
                     { 23, 34, 90, 2000 } };
    int K = sizeof(arr) / sizeof(arr[0]);

    int output[N * K];

    // Function call
    mergeKArrays(arr, 3, output);

    cout << "Merged array is " << endl;
    printArray(output, N * K);

    return 0;
}
```

Output

```
Merged array is
1 2 6 9 12 20 23 34 34 90 1000 2000
```

Time Complexity: O(N * K * log (N*K)), Since the resulting array is of size N*K.
Space Complexity: O(N * K), The output array is of size N * K.

Merge K sorted arrays using merging:

The process begins with merging arrays into groups of two. After the first merge, there will be K/2 arrays remaining. Again merge arrays in groups, now K/4 arrays will be remaining. This is similar to merge sort. Divide K arrays into two halves containing an equal number of arrays until there are two arrays in a group. This is followed by merging the arrays in a bottom-up manner.

Follow the given steps to solve the problem:

- Create a recursive function that takes K arrays and returns the output array.

- In the recursive function, if the value of K is 1 then return the array else if the value of K is 2 then merge the two arrays in linear time and return the array.
- If the value of K is greater than 2 then divide the group of k elements into two equal halves and recursively call the function, i.e 0 to K/2 array in one recursive function and K/2 to K array in another recursive function.
- Print the output array.

Below is the implementation of the above approach:

C++14

```
// C++ program to merge K sorted arrays of size n each.


#include <bits/stdc++.h>
using namespace std;
#define N 4


// Merge arr1[0..N1-1] and arr2[0..N2-1] into
// arr3[0..N1+N2-1]
void mergeArrays(int arr1[], int arr2[], int N1, int N2,
                 int arr3[])
{
    int i = 0, j = 0, k = 0;


    // Traverse both array
    while (i < N1 && j < N2) {
        // Check if current element of first
        // array is smaller than current element
        // of second array. If yes, store first
        // array element and increment first array
```

```
        // index. Otherwise do same with second array
        if (arr1[i] < arr2[j])
            arr3[k++] = arr1[i++];
        else
            arr3[k++] = arr2[j++];
    }

    // Store remaining elements of first array
    while (i < N1)
        arr3[k++] = arr1[i++];

    // Store remaining elements of second array
    while (j < N2)
        arr3[k++] = arr2[j++];
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
}

// This function takes an array of arrays as an argument and
// All arrays are assumed to be sorted. It merges them
// together and prints the final sorted output.
void mergeKArrays(int arr[][N], int i, int j, int output[])
```

```
{
    // If one array is in range
    if (i == j) {
        for (int p = 0; p < N; p++)
            output[p] = arr[i][p];
        return;
    }

    // if only two arrays are left them merge them
    if (j - i == 1) {
        mergeArrays(arr[i], arr[j], N, N, output);
        return;
    }

    // Output arrays
    int out1[N * (((i + j) / 2) - i + 1)],
        out2[N * (j - ((i + j) / 2))];

    // Divide the array into halves
    mergeKArrays(arr, i, (i + j) / 2, out1);
    mergeKArrays(arr, (i + j) / 2 + 1, j, out2);

    // Merge the output array
    mergeArrays(out1, out2, N * (((i + j) / 2) - i + 1),
                N * (j - ((i + j) / 2)), output);
}
```

```
// Driver's code
int main()
{
    // Change N at the top to change number of elements
    // in an array
    int arr[][N] = { { 2, 6, 12, 34 },
                     { 1, 9, 20, 1000 },
                     { 23, 34, 90, 2000 } };
    int K = sizeof(arr) / sizeof(arr[0]);
    int output[N * K];
    mergeKArrays(arr, 0, 2, output);

    // Function call
    cout << "Merged array is " << endl;
    printArray(output, N * K);

    return 0;
}
```

Output

```
Merged array is
1 2 6 9 12 20 23 34 34 90 1000 2000
```

Time Complexity: O(N * K * log K). There are log K levels as in each level the K arrays are divided in half and at each level, the K arrays are traversed.
Auxiliary Space: O(N * K * log K). In each level O(N * K) space is required.

Merge K sorted arrays using Min-Heap:

The idea is to use Min Heap. This MinHeap based solution has the same time complexity which is O(NK log K). But for a different and particular sized array, this solution works much better. The process must start with creating a

MinHeap and inserting the first element of all the k arrays. Remove the root element of Minheap and put it in the output array and insert the next element from the array of removed element. To get the result the step must continue until there is no element left in the MinHeap.

Follow the given steps to solve the problem:

- Create a min Heap and insert the first element of all the K arrays.
- Run a loop until the size of MinHeap is greater than zero.
    - Remove the top element of the MinHeap and print the element.
    - Now insert the next element from the same array in which the removed element belonged.
    - If the array doesn't have any more elements, then replace root with infinite. After replacing the root, heapify the tree.
- Return the output array

Below is the implementation of the above approach:

C++

```
// C++ program to merge K sorted

// arrays of size N each.

#include <bits/stdc++.h>

using namespace std;


#define N 4


// A min-heap node

struct MinHeapNode {

    // The element to be stored

    int element;


    // index of the array from which the element is taken

    int i;
```

```
    // index of the next element to be picked from the array
    int j;
};

// Prototype of a utility function to swap two min-heap
// nodes
void swap(MinHeapNode* x, MinHeapNode* y);

// A class for Min Heap
class MinHeap {

    // pointer to array of elements in heap
    MinHeapNode* harr;

    // size of min heap
    int heap_size;

public:
    // Constructor: creates a min heap of given size
    MinHeap(MinHeapNode a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int);

    // to get index of left child of node at index i
    int left(int i) { return (2 * i + 1); }
```

```
    // to get index of right child of node at index i
    int right(int i) { return (2 * i + 2); }

    // to get the root
    MinHeapNode getMin() { return harr[0]; }

    // to replace root with new node x and heapify() new
    // root
    void replaceMin(MinHeapNode x)
    {
        harr[0] = x;
        MinHeapify(0);
    }
};

// This function takes an array of arrays as an argument and
// All arrays are assumed to be sorted. It merges them
// together and prints the final sorted output.
int* mergeKArrays(int arr[][N], int K)
{

    // To store output array
    int* output = new int[N * K];

    // Create a min heap with k heap nodes.
    // Every heap node has first element of an array
```

```
MinHeapNode* harr = new MinHeapNode[K];
for (int i = 0; i < K; i++) {

    // Store the first element
    harr[i].element = arr[i][0];

    // index of array
    harr[i].i = i;

    // Index of next element to be stored from the array
    harr[i].j = 1;
}

// Create the heap
MinHeap hp(harr, K);

// Now one by one get the minimum element from min
// heap and replace it with next element of its array
for (int count = 0; count < N * K; count++) {
    // Get the minimum element and store it in output
    MinHeapNode root = hp.getMin();
    output[count] = root.element;

    // Find the next element that will replace current
    // root of heap. The next element belongs to same
    // array as the current root.
    if (root.j < N) {
```

```
            root.element = arr[root.i][root.j];
            root.j += 1;
        }
        // If root was the last element of its array
        // INT_MAX is for infinite
        else
            root.element = INT_MAX;

        // Replace root with next element of array
        hp.replaceMin(root);
    }

    return output;
}

// FOLLOWING ARE IMPLEMENTATIONS OF
// STANDARD MIN HEAP METHODS FROM CORMEN BOOK
// Constructor: Builds a heap from a given
// array a[] of given size
MinHeap::MinHeap(MinHeapNode a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int i = (heap_size - 1) / 2;
    while (i >= 0) {
        MinHeapify(i);
        i--;
```

```
    }
}

// A recursive method to heapify a
// subtree with root at given index.
// This method assumes that the subtrees
// are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;

    if (l < heap_size && harr[l].element < harr[i].element)
        smallest = l;

    if (r < heap_size
        && harr[r].element < harr[smallest].element)
        smallest = r;

    if (smallest != i) {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
```

```
void swap(MinHeapNode* x, MinHeapNode* y)
{
    MinHeapNode temp = *x;
    *x = *y;
    *y = temp;
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
}

// Driver's code
int main()
{
    // Change N at the top to change number of elements
    // in an array
    int arr[][N] = { { 2, 6, 12, 34 },
                     { 1, 9, 20, 1000 },
                     { 23, 34, 90, 2000 } };
    int K = sizeof(arr) / sizeof(arr[0]);

    // Function call
    int* output = mergeKArrays(arr, K);
```

```
    cout << "Merged array is " << endl;
    printArray(output, N * K);

    return 0;
}
```

Output

```
Merged array is
1 2 6 9 12 20 23 34 34 90 1000 2000
```

Time Complexity: O(N * K * log K), Insertion and deletion in a Min Heap requires log K time.
Auxiliary Space: O(K), If Output is not stored then the only space required is the Min-Heap of K elements.