```cpp
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <math.h>
#include <cstdlib>
#include "mpi.h"

// Functions
void mergesort(int *, int, int);
void smerge(int *, int, int, int, int, int * = NULL, int = 0, int = 0);
int rank(int *, int, int, int);
void pmerge(int *, int, int, int);
void print(int *, int, int);
bool uniqueEntry(int *, int, int);
void setA(int *, int, int, int = 0, int * = NULL);
bool sortedCheck(int *, int);
void setRank(int *, int *, int, int, int, int, int, int);
void setSubproblem(int *, int *, int, int);

// Global Varibales
int my_rank; // my CPU number for this process
int p;       // number of CPUs that we have

int main(int argc, char *argv[])
{
    int source;        // rank of the sender
    int dest;          // rank of destination
    int tag = 0;       // message number
    MPI_Status status; // return status for receive

    // Start MPI
    MPI_Init(&argc, &argv);

    // Find out my rank!
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Find out the number of processes!
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    // Getting the max number size in the array and the array size
    int max = atoi(argv[1]);
    int power = atoi(argv[2]);

    // Determining if the problem is possible
    int n = pow(2, power);
    if (n > max)
    {
        if (my_rank == 0)
            std::cout << "\nAn array of size " << n << " with unique entries between 0 - " << max
            << " isn't possible\n.";

        MPI_Finalize();
        return 0;
    }

    // Seed Random Number Generator
    srand(0);

    // a is an array defined dynamically based on a user-inputted size
    int *a = new int[n];

    // Process 0 will set the array a with numbers between 0 and a user inputted max
    if (my_rank == 0)
    {
        for (int i = 0; i < n; i++)
        {
            int num = rand() % (max + 1);
            while (uniqueEntry(a, i, num) != true)
                num = rand() % (max + 1);
        }
        std::cout << "\nUnsorted: " << std::endl;
        print(a, 0, n - 1);
    }

    // Broadcasting the entire array a to all processors
    MPI_Bcast(&a[0], n, MPI_INT, 0, MPI_COMM_WORLD);

    // Begin Merge Sorting
    mergesort(a, 0, n - 1);

    // Print Sorted Array
    if (my_rank == 0)
    {
        std::cout << "\nSorted: " << std::endl;
        print(a, 0, n - 1);
        std::cout << std::endl;
    }

    if (!sortedCheck(a, n))
        std::cout << "Error, array is not sorted." << std::endl;

    // Garbage Collection
    delete[] a;

    MPI_Finalize();

    return 0;
} // end main

void mergesort(int *a, int first, int last)
{
    if (last - first == 1)
        smerge(a, first, first, last, last);
    else
    {
        int mid = (first + last) / 2;
        mergesort(a, first, mid);
        mergesort(a, mid + 1, last);
        pmerge(a, first, last, mid);
    }
    return;
} // end mergesort

void smerge(int *a, int first1, int last1, int first2, int last2, int *out, int inc, int size)
{
    // Counters
    int i = first1;
    int j = first2;
    int k = 0;

    // An array to store the sorting of the partitions
    int *b = new int[(last2 - first2 + 1) + (last1 - first1 + 1)];
    setA(b, 0, (last2 - first2) + (last1 - first1) + 1, 0);

    // Sort each partition into an array b
    while (i <= last1 && j <= last2)
        if (a[i] < a[j])
            b[k++] = a[i++];
        else
            b[k++] = a[j++];

    // Get any remaining entries in the partitions
    for (int x = i; x <= last1; x++)
        b[k++] = a[x];
    for (int x = j; x <= last2; x++)
        b[k++] = a[x];

    // Copy back b into a or b into out
    if (out == NULL)
```

```cpp
        for (int x = first1; x <= last2; x++)
            a[x] = b[x - first1];
    else
        for (int x = 0; x < (last2 - first2 + 1) + (last1 - first1 + 1); x++)
            out[x + first1 + first2 - inc] = b[x];

    // Garbage collection
    delete[] b;

    return;
} // end smerge

int rank(int *a, int first, int last, int valToFind)
{
    if (first == last)
    {
        if (valToFind < a[first])
            return 0;
        else
            return 1;
    }

    int mid = (first + last) / 2;
    if (valToFind < a[mid + 1])
        return rank(a, first, mid, valToFind);
    else
        return ((last - first + 1) / 2) + rank(a, mid + 1, last, valToFind);
} // end rank

void pmerge(int *a, int first, int last, int mid)
{
    // Stage 1: Partitioning
    // Calculate sRank array size and segment sizes
    int localStart = my_rank;
    int segSize = (last - first + 1) / 2;
    int logSegSize = log2(segSize);
    int localSize = ceil(((double)segSize / (double)logSegSize));

    // Initialize the sRank arrays
    int *sRankA = new int[localSize];
    setA(sRankA, 0, localSize - 1, 0);
    int *sRankB = new int[localSize];
    setA(sRankB, 0, localSize - 1, 0);
    int *totalSRankA = new int[localSize];
    int *totalSRankB = new int[localSize];

    // Calculate the arrays via striping
    setRank(sRankA, a, mid + 1, last, first, localStart, localSize, logSegSize);
    setRank(sRankB, a, first, mid, mid + 1, localStart, localSize, logSegSize);

    // Sharing all the srank arrays
    MPI_Allreduce(&sRankA[0], &totalSRankA[0], localSize, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&sRankB[0], &totalSRankB[0], localSize, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    // Stage 2: Actual Rankking for the Smaller Problems of A and B
    // Initialize the subproblem (shape) arrays
    int *subproblemA = new int[2 * localSize + 1];
    int *subproblemB = new int[2 * localSize + 1];

    // Seting up each subproblem (shape) with segment sizes and ranks
    setSubproblem(subproblemA, totalSRankB, localSize, logSegSize);
    setSubproblem(subproblemB, totalSRankA, localSize, logSegSize);
    subproblemA[2 * localSize] = (last - first + 1) / 2;
    subproblemB[2 * localSize] = (last - first + 1) / 2;

    // Using smerge to sort our ranks
    smerge(subproblemA, 0, localSize - 1, localSize, 2 * localSize);
    smerge(subproblemB, 0, localSize - 1, localSize, 2 * localSize);

    // Creating the win array
    int *win = new int[last + 1];
    int *localWin = new int[last + 1];
    setA(localWin, 0, last, 0);

    // Smerge Shapes
    for (int i = localStart; i < 2 * localSize; i += p)
        smerge(a, first + subproblemA[i], first + subproblemA[i + 1] - 1, mid + 1 +
        subproblemB[i], mid + 1 + subproblemB[i + 1] - 1, localWin, mid + 1, last);

    MPI_Allreduce(&localWin[0], &win[0], last + 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    // Copy win array back over to a because win is the sorted a array
    setA(a, first, last, 0, win);

    // Garbage Collection
    delete[] sRankA;
    delete[] sRankB;
    delete[] totalSRankA;
    delete[] totalSRankB;
    delete[] subproblemA;
    delete[] subproblemB;
    delete[] win;
    delete[] localWin;

    return;
} // end pmerge

void print(int *a, int first, int last)
{
    // Print the array in lengths of 16
    for (int i = first; i <= last; i++)
        if (i != 0 && i % 16 == 0)
            std::cout << "\n"
                      << std::setw(4) << a[i];
        else
            std::cout << std::setw(4) << a[i];
    std::cout << std::endl;
    return;
} // end of print

bool uniqueEntry(int *a, int b, int num)
{
    // This entry already exists
    for (int i = 0; i < b; i++)
        if (a[i] == num)
            return false;
    // The entry did not exist
    a[b] = num;
    return true;
} // end uniqueEntry

void setA(int *a, int first, int last, int val, int *b)
{
    // Set every element in a to val or copy array b
    if (b == NULL)
        for (int i = first; i <= last; i++)
            a[i] = val;
    else
        for (int i = first; i <= last; i++)
            a[i] = b[i];
    return;
} // end setA

bool sortedCheck(int *a, int last)
{
    // Array has size 1 or there are no elements left to approve
    if (last == 1 || last == 0)
        return 1;
    // Unsorted pair is found in the array
    if (a[last - 1] < a[last - 2])
```

```
        return 0;
    // We had a sorted pair, check the next
    return sortedCheck(a, last - 1);
} // end sortedCheck

void setRank(int *sRank, int *a, int first, int last, int pos, int localStart, int localSize, int
logSegSize)
{
    // Set the ranks for our sample
    for (int i = localStart; i < localSize; i += p)
        sRank[i] = rank(a, first, last, a[pos + (i * logSegSize)]);
}

void setSubproblem(int *subproblem, int *sRank, int localSize, int logSegSize)
{
    // Set up the subproblem with segment sizes and ranks
    for (int i = 0; i < localSize; i++)
    {
        subproblem[i] = i * logSegSize;
        subproblem[i + localSize] = sRank[i];
    }
} // end sortedCheck
```