```
#include <algorithm>
#include <iostream>
#include <chrono>
#include <ctime>
#include <iomanip>
#include <cstdlib>

using namespace std;
using namespace chrono;

// Function prototypes
void generateRandomNumber(int arr[], int size);
void sortInsertion(int arr[], int size);
void sortMerge(int arr[], int bottom, int top);
void sortHeap(int arr[], int size);

int main()
{ // Array arraySize for testing
    const int arraySize[] = {1000, 10000, 25000, 50000, 150000,
250000};
    const int sizeNums = sizeof(arraySize) /
sizeof(arraySize[0]);

    // Seed the random number generator
    srand(time(0));
```

```cpp
    // Displaying table headers
    cout << left << setw(10) << "Input" << setw(15) << "Heap Sort" << setw(20) << "Insertion Sort" << setw(15) <<"Merge Sort" << setw(15) <<"Best Time" <<endl;
    cout << "======================================================================" << endl;

    // Loop through different arrays
    for (int i = 0; i < sizeNums; ++i)
    {
        int size = arraySize[i];
        int* mergeArray = new int[size];
        int* insertionArray = new int[size];
        int* arrayInitial = new int[size];

        // Generate a random arr and copy it for different sorting algorithms
        generateRandomNumber(arrayInitial, size);
        copy(arrayInitial, arrayInitial + size, mergeArray);
        copy(arrayInitial, arrayInitial + size, insertionArray);

        // Measure execution time for Heap Sort
        auto heapSort = high_resolution_clock::now();
        sortHeap(arrayInitial, size);
        auto heapStop = high_resolution_clock::now();
        auto heapDuration =
duration_cast<duration<double>>(heapStop - heapSort);

        // Measure execution time for Insertion Sort
        auto insertionStart = high_resolution_clock::now();
        sortInsertion(insertionArray, size);
        auto insertionStop = high_resolution_clock::now();
        auto insertionDuration =
duration_cast<duration<double>>(insertionStop - insertionStart);

        // Measure execution time for Merge Sort
        auto mergeStart = high_resolution_clock::now();
        sortMerge(mergeArray, 0, size - 1);
        auto mergeStop = high_resolution_clock::now();
        auto mergeDuration =
duration_cast<duration<double>>(mergeStop - mergeStart);

        // Determine the best sorting algorithm based on
execution time
        string timeBest;
        if (heapDuration <= insertionDuration && heapDuration <=
mergeDuration)
        {
            timeBest = "Heap";
        }
        else if (insertionDuration <= heapDuration &&
insertionDuration <= mergeDuration)
        {
            timeBest = "Insertion";
        }
        else
        {
            timeBest = "Merge";
        }

        // Displaying the results in a table format
        cout << left << setw(12) << size << setw(18) << fixed <<
setprecision(3) << heapDuration.count() << setw(18) << fixed <<
setprecision(3) << insertionDuration.count() << setw(15) <<
fixed << setprecision(3) << mergeDuration.count() << setw(5)  <<
timeBest << endl;

        // Free allocated memory
        delete[] mergeArray;
        delete[] insertionArray;
        delete[] arrayInitial;
    }
    return 0;
}

    // Function to generate random integers in an array
    void generateRandomNumber(int arr[], int size)
    {
        for (int c = 0; c < size; ++c)
        {
            arr[c] = rand() % 1000;
        }
    }
```

```
    // Function implementing Heap Sort algorithm
    void sortHeap(int arr[], int size)
    {
        for(int k = size / 2 - 1; k >= 0; --k){
            int ancestor = k;
```

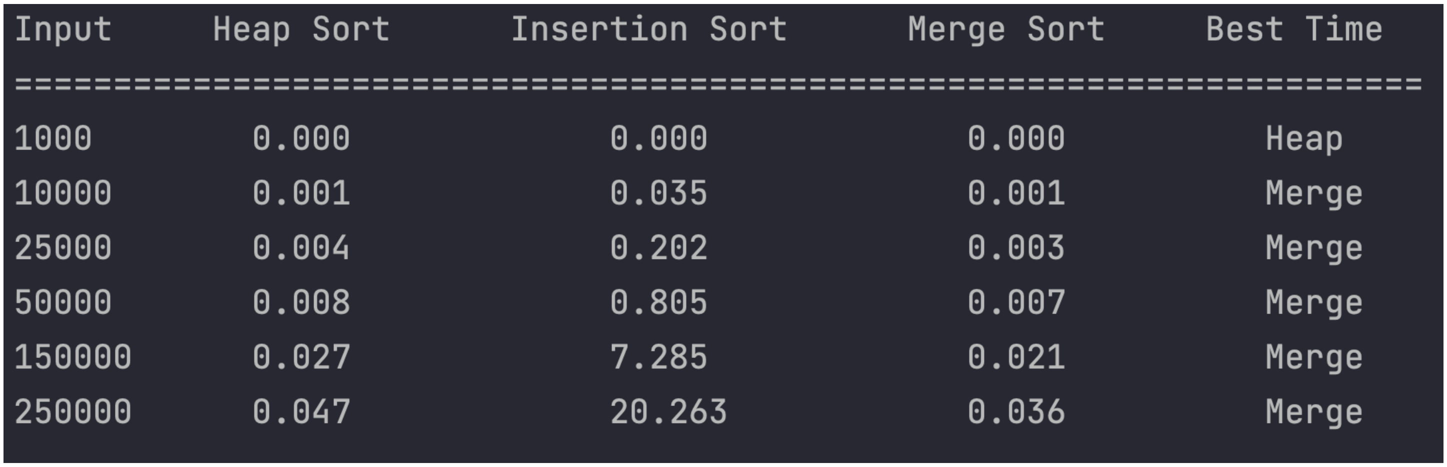
```
            while(ancestor < size / 2){
                int childLeft = 2 * ancestor + 1;
                int childRight = childLeft + 1;
                int childMax = (childRight < size && arr[childRight] > arr[childLeft]) ? childRight : childLeft;
```

```
                if(arr[ancestor] >= arr[childMax]){
                    break;
                }
                swap(arr[ancestor], arr[childMax]);
                ancestor = childMax;
```

```
            }
        }
        for(int i = size - 1; i > 0; --i){
            swap(arr[0], arr[i]);
            int ancestor = 0;

            while(ancestor < i / 2){
                int childLeft = 2 * ancestor + 1;
                int childRight = childLeft + 1;
                int childMax = (childRight < i &&
arr[childRight] > arr[childLeft]) ? childRight : childLeft;

                if(arr[ancestor] >= arr[childMax])
                {
                    break;
                }
                swap(arr[ancestor], arr[childMax]);
                ancestor = childMax;
            }
        }
    }

    // Function implementing Merge Sort algorithm
    void sortMerge(int arr[], int bottom, int top)
    {
        if(bottom < top)
        {
            int middle = bottom + (top - bottom) / 2;

            sortMerge(arr, bottom, middle);
            sortMerge(arr, middle + 1, top);

            int bottomToMiddle = middle - bottom + 1;
            int topToMiddle = top - middle;
            int* arrayLeft = new int[bottomToMiddle];
            int* arrayRight = new int[topToMiddle];

            for(int i = 0; i < bottomToMiddle; ++i)
                arrayLeft[i] = arr[bottom + i];
            for(int j = 0; j < topToMiddle; ++j)
                arrayRight[j] = arr[middle + 1 + j];

            int z = 0;
            int p = 0;
            int b = bottom;

            while(z < bottomToMiddle && p < topToMiddle)
            {
                if(arrayLeft[z] <= arrayRight[p])
                {
                    arr[b] = arrayLeft[z];
                    ++z;
                }
                else
                {
                    arr[b] = arrayRight[p];
                    ++p;
                }
                ++b;
            }

            while(z < bottomToMiddle)
            {
                arr[b] = arrayLeft[z];
                ++z;
                ++b;
            }
            while(p < topToMiddle)
            {
                arr[b] = arrayRight[p];
                ++p;
                ++b;
            }
            delete[] arrayLeft;
            delete[] arrayRight;
        }
    }
```

```
    // Function implementing Insertion Sort algorithm
    void sortInsertion(int arr[], int size){
        for (int t = 1; t < size; ++t){
            int key = arr[t];
            int j = t - 1;
```

```
            while (j >= 0 && arr[j] > key){
                arr[j + 1] = arr[j];
                --j;
            }
            arr[j + 1] = key;
        }
```



```
Input     Heap Sort      Insertion Sort      Merge Sort     Best Time
=======================================================================
1000        0.000             0.000             0.000          Heap
10000       0.001             0.035             0.001          Merge
25000       0.004             0.202             0.003          Merge
50000       0.008             0.805             0.007          Merge
150000      0.027             7.285             0.021          Merge
250000      0.047             20.263            0.036          Merge
```

```
        }
```