ALGORITHM LABORATORY

ASSIGNMENT-3

PROBLEM STATEMENT: Implement Merge sort, Heap sort and Quick sort taking first element as the pivot.

1. *Merge Sort:*

ALGORITHM (MERGE SORT):

a. Divide
   i. If the array has one or zero elements, it is already sorted
   ii. Otherwise, split the array into two halves

b. Conquer
   i. Recursively apply merge sort to both halves

c. Merge
   i. Merge the two sorted halves back together into a single sorted array
   ii. Compare elements from both halves and place them in the correct order in the merged array

d. Repeat
   i. Continue the process until the entire array is sorted.

PROGRAM CODE:

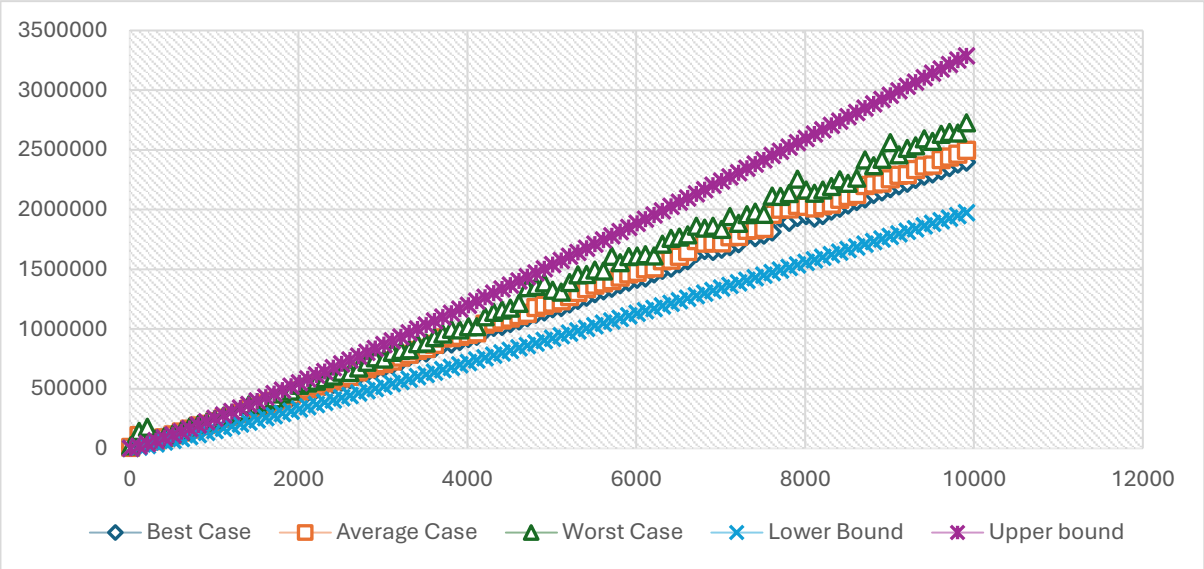
```
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

void merge(vector<int>& arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int i = 0; i < n2; i++)
        R[i] = arr[m + 1 + i];

    int i = 0, j = 0, k = l;
```

```
    while (i < n1 && j < n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }
    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];
}

void mergeSort(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
int main() {
    srand(time(NULL));
    for (int k = 10;k<=10000;k+=100) {
        double best_case_average_time = INT_MAX;
        double worst_case_average_time = INT_MIN;
        double random_case_average_time = 0;
        int num_trials = 100;
        if (k == 1000000) num_trials = 10;

        for (int i = 0; i < num_trials; i++) {
            vector<int> arr(k);
            for (int j = 0; j < k; j++)
                arr[j] = rand() % (5 * k);

            auto start = high_resolution_clock::now();
            mergeSort(arr, 0, k - 1);
            auto end = high_resolution_clock::now();
            random_case_average_time += duration_cast<nanoseconds>(end -
start).count();
            best_case_average_time = min((double)duration_cast<nanoseconds>(end -
start).count(),best_case_average_time);
            worst_case_average_time = max((double)duration_cast<nanoseconds>(end -
start).count(),worst_case_average_time);
        }

        random_case_average_time /= num_trials;

        cout <<k<<","<< best_case_average_time << ","<< random_case_average_time
<< "," << worst_case_average_time << endl;
    }
    return 0;
}
```

TIME COMPLEXITY:

- Best Case: nlog(n)
- Average Case: nlog(n)
- Worst Case: nlog(n)

OUTPUT and PLOT:



2. *Heap Sort:*

ALGORITHM (HEAP SORT):

a. Build Max Heap
   i. Convert the given array into a **max heap** (a complete binary tree where the root is the largest element).
   ii. Start from the last non-leaf node and apply **heapify** in a **bottom-up** manner.

b. Heap Sort Process
   i. Swap the root (largest element) with the last element in the heap.
   ii. Reduce the heap size (ignore the last element, which is now sorted).
   iii. Apply **heapify** on the root to restore the max heap property

c. Repeat
    i. Continue swapping and heapifying until only one element remains in the heap
    ii. The array is now sorted in **ascending order**

PROGRAM CODE:

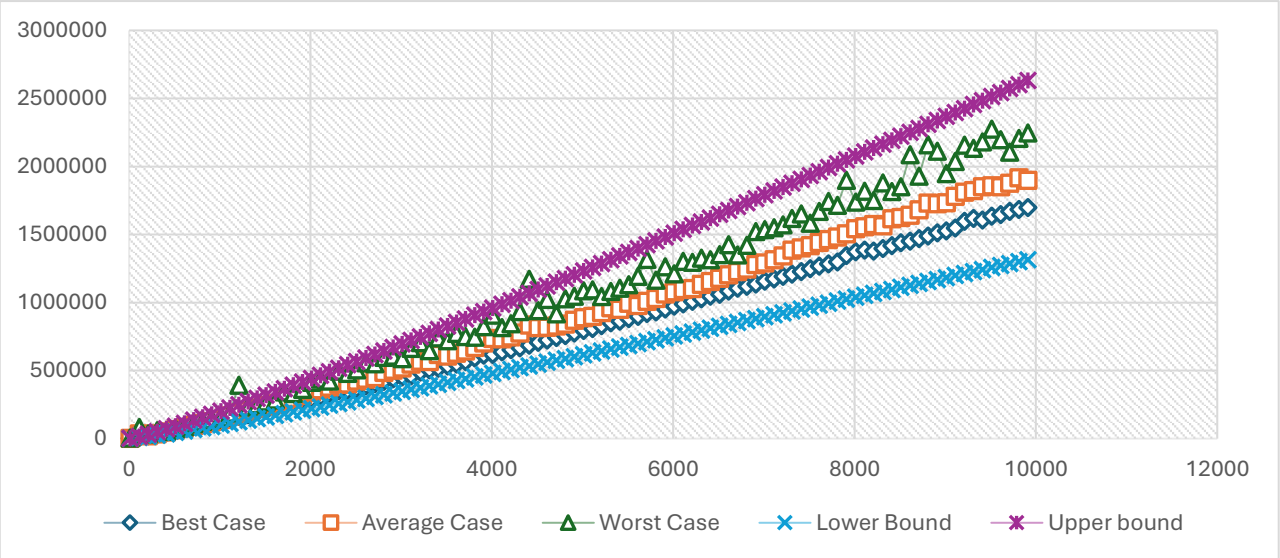
```
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < n && arr[left] > arr[largest])
        largest = left;

    if (right < n && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

int main() {
    srand(time(NULL));
    for (int k = 10;k<=10000;k+=100) {
        double best_case_average_time = INT_MAX;
        double worst_case_average_time = INT_MIN;
        double random_case_average_time = 0;
        int num_trials = 100;
        if (k == 1000000) num_trials = 10;

        for (int i = 0; i < num_trials; i++) {
            vector<int> arr(k);
```

```
            for (int j = 0; j < k; j++)
                arr[j] = rand() % (5 * k);

            auto start = high_resolution_clock::now();
            heapSort(arr);
            auto end = high_resolution_clock::now();
            random_case_average_time += duration_cast<nanoseconds>(end -
start).count();
            best_case_average_time = min((double)duration_cast<nanoseconds>(end -
start).count(),best_case_average_time);
            worst_case_average_time = max((double)duration_cast<nanoseconds>(end -
start).count(),worst_case_average_time);
        }

        random_case_average_time /= num_trials;

        cout <<k<<","<< best_case_average_time << ","<< random_case_average_time
<< "," << worst_case_average_time << endl;
    }
    return 0;
}
```

TIME COMPLEXITY:

- Best Case: nlog(n)
- Average Case: nlog(n)
- Worst Case: nlog(n)

OUTPUT and PLOT:



3. *Quick Sort (first element as pivot):*

ALGORITHM (QUICK SORT (first element as pivot)):

a. Choose a Pivot
    i. Select the first element of the array as the pivot
b. Partition the Array
    i. Rearrange elements such that:
        1. Elements smaller than the pivot go to the left
        2. Elements larger than the pivot go to the right
    ii. The pivot is now in its correct sorted position
c. Recursively Apply Quick Sort
    i. Apply Quick Sort to the left and right subarrays (excluding the pivot)
d. Repeat Until Sorted
    i. Continue the process until all subarrays are of size 1 or empty

PROGRAM CODE:

```
#include <bits/stdc++.h>
using namespace std;
using namespace std::chrono;

int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[low];
    int i = low + 1;

    for (int j = low + 1; j <= high; j++) {
        if (arr[j] < pivot) {
            swap(arr[i], arr[j]);
            i++;
        }
    }
    swap(arr[low], arr[i - 1]);
    return i - 1;
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```
int main() {
    srand(time(NULL));
    for (int k = 10;k<=10000;k+=100) {
        double best_case_average_time = INT_MAX;
        double worst_case_average_time = INT_MIN;
        double random_case_average_time = 0;
        int num_trials = 100;
        if (k == 1000000) num_trials = 10;

        for (int i = 0; i < num_trials; i++) {
            vector<int> arr(k);
            for (int j = 0; j < k; j++)
                arr[j] = rand() % (5 * k);

            auto start = high_resolution_clock::now();
            quickSort(arr, 0, k - 1);
            auto end = high_resolution_clock::now();
            random_case_average_time += duration_cast<nanoseconds>(end -
start).count();
            best_case_average_time = min((double)duration_cast<nanoseconds>(end -
start).count(),best_case_average_time);

            start = high_resolution_clock::now();
            quickSort(arr,0,k-1);
            end = high_resolution_clock::now();
            worst_case_average_time = max((double)duration_cast<nanoseconds>(end -
start).count(),worst_case_average_time);
        }

        random_case_average_time /= num_trials;

        cout <<k<<","<< best_case_average_time << ","<< random_case_average_time
<< "," << worst_case_average_time << endl;
    }
    return 0;
}
```

TIME COMPLEXITY:

- Best Case: nlog(n)
- Average Case: nlog(n)
- Worst Case: n^2

OUTPUT and PLOT: