L P S - 2

{ Design and Analysis of Algorithms }

Rachit Bhalla
21BAI1869

1.Given n 2-dimensional points (x1, y1), (x2, y2), ..., (xn, yn), Design an algorithm that follows the 'Divide-Conquer-Combine' strategy to
(a) arrange the n−points in an increasing order of the x coordinates.
(b) arrange the n−points in an increasing order of the y coordinates.
(c) arrange the n−points in decreasing order of the value (x−coordinate + y−coordinate)/2) coordinates.
For each of the above algorithms, execute the algorithm with an appropriate code. Compute t(n) for each of the algorithm.

Problem Statement:

To use merge sort to sort the given pairs of coordinates in increasing order of x coordinates, y coordinates and in decreasing order of (x + y)/2 coordinates.

Logic:

Merge sort uses divide and conquer approach , it divides the main array into left and right subarrays and then recursively calls the merge sort function till we have divided the given sub array into atomic values (or in other words left index is equal to right index).and then we use the merge function to merge the sorted sub arrays at each level.

(A)Algorithm:

```
1. Import the necessary libraries
2.  Void merge(coordinates,left,mid,right)
        Make 2 subarrays, left_arr & right_arr
        For fa_idx left to right(iterate)
                If (left_arr[left_idx]<right_arr[right_idx])
                Push elements of left_arr in coordinates
        (copy elements from left subarray intomain vector array coordinates)
                Else
                        Push elements of right_arr in coordinates
(copy elements from right subarray into main vector array coordintes)

3. Void mergesort(coordinates, left, right)
mid = (left+right)/2
mergesort(coordinates, left, mid)
mergesort(coordinates, mid+1, right)
merge(coordinates, left, mid, right)
4.
int main()
read the coordinates given by user
mergesort(coordinates, 0, coordinates.lenght -1)
print run time
print elements of the sorted array
```

(A)Pseudocode:

```
#include<iostream>
#include<vector>
#include<ctime>
#include<limits.h>
using namespace std;

void merge(vector<pair<int, int>>& coordinates, int left, int mid, int
right) {
    vector<pair<int, int>> left_arr, right_arr;
    int left_ele, right_ele, i, fa_idx, left_idx, right_idx;
    left_ele = mid - left + 1;
    right_ele = right - mid;

    for (i = 0; i < left_ele; i++)
        left_arr.push_back(coordinates[left + i]);
    left_arr.push_back({INT_MAX, INT_MAX});

    for (i = 0; i < right_ele; i++)
        right_arr.push_back(coordinates[mid + 1 + i]);
    right_arr.push_back({INT_MAX, INT_MAX});

    left_idx = 0;
    right_idx = 0;
    for (fa_idx = left; fa_idx <= right; fa_idx++) {
        if (left_arr[left_idx].first < right_arr[right_idx].first) {
            coordinates[fa_idx] = left_arr[left_idx];
            left_idx++;
        }
        else {
            coordinates[fa_idx] = right_arr[right_idx];
            right_idx++;
        }
    }
}

void mergeSort(vector<pair<int, int>>& coordinates, int left, int right) {
    int mid;
    if (left == right) {
        return;
    }
    mid = (left + right) / 2;
    mergeSort(coordinates, left, mid);
    mergeSort(coordinates, mid + 1, right);
    merge(coordinates, left, mid, right);
}

int main() {
    int n, i, x, y;
    vector<pair<int, int>> coordinates;
    cout << "Enter the number of coordinates: ";
    cin >> n;
    srand(time(0));
    for (i = 0; i < n; i++) {
        int x = rand() % n;
        int y = rand() % n;
        coordinates.push_back({x, y});
    }

    clock_t tStart = clock();
    mergeSort(coordinates, 0, n - 1);

    double time = (double)(clock() - tStart) / CLOCKS_PER_SEC;
    cout << "Time: " << time << endl;
    cout << "Coordinates sorted by x coordinates:\n";
    for (i = 0; i < n; i++) {
        cout << coordinates[i].first << " " << coordinates[i].second << "\n";
    }
    return 0;
}
```

(B)Algorithm:

```
1. Import the necessary libraries
2.  Void merge(coordinates,left,mid,right)
        Make 2 subarrays, left_arr & right_arr
        For fa_idx left to right(iterate)
                If (left_arr[left_idx]<right_arr[right_idx])
                Push elements of left_arr in coordinates
        (copy elements from left subarray intomain vector array coordinates)
                Else
                        Push elements of right_arr in coordinates
(copy elements from right subarray into main vector array coordintes)

3. Void mergesort(coordinates, left, right)
mid = (left+right)/2
mergesort(coordinates, left, mid)
mergesort(coordinates, mid+1, right)
merge(coordinates, left, mid, right)
4.
int main()
```

```
read the coordinates given by user
mergesort(coordinates, 0, coordinates.lenght -1)
print run time
print elements of the sorted array
```

(B)Pseudocode:

```
#include<iostream>
#include<vector>
#include<ctime>
#include<limits.h>
using namespace std;

void merge(vector<pair<int, int>>& coordinates, int left, int mid, int right) {
    vector<pair<int, int>> left_arr, right_arr;
    int left_ele, right_ele, i, fa_idx, left_idx, right_idx;
    left_ele = mid - left + 1;
    right_ele = right - mid;

    for (i = 0; i < left_ele; i++)
        left_arr.push_back(coordinates[left + i]);
    left_arr.push_back({INT_MAX, INT_MAX});

    for (i = 0; i < right_ele; i++)
        right_arr.push_back(coordinates[mid + 1 + i]);
    right_arr.push_back({INT_MAX, INT_MAX});

    left_idx = 0;
    right_idx = 0;
    for (fa_idx = left; fa_idx <= right; fa_idx++) {
        if (left_arr[left_idx].second < right_arr[right_idx].second) {
            coordinates[fa_idx] = left_arr[left_idx];
            left_idx++;
        }
        else {
            coordinates[fa_idx] = right_arr[right_idx];
            right_idx++;
        }
    }
}
```

```
void mergeSort(vector<pair<int, int>>& coordinates, int left, int right) {
    int mid;
    if (left == right) {
        return;
    }
    mid = (left + right) / 2;
    mergeSort(coordinates, left, mid);
    mergeSort(coordinates, mid + 1, right);
    merge(coordinates, left, mid, right);
}

int main() {
    int n, i, x, y;
    vector<pair<int, int>> coordinates;
    cout << "Enter the number of coordinates: ";
    cin >> n;
    srand(time(0));
    for (i = 0; i < n; i++) {
        int x = rand() % n;
        int y = rand() % n;
        coordinates.push_back({x, y});
    }

    clock_t tStart = clock();
    mergeSort(coordinates, 0, n - 1);

    double time = (double)(clock() - tStart) / CLOCKS_PER_SEC;
    cout << "Time: " << time << endl;
    cout << "Coordinates sorted by y coordinates:\n";
    for (i = 0; i < n; i++) {
        cout << coordinates[i].first << " " << coordinates[i].second << "\n";
    }
    return 0;
}
```

(C)Algorithm:

1. Import the necessary libraries

```
        calculateAvg(pair& point)
```

```
 {
return (double)(point.first + point.second) / 2;
}
2.  Void merge(coordinates,left,mid,right)
        Make 2 subarrays, left_arr & right_arr
                left_avg = calculateAvg(left_subarr[left_idx]);
                 right_avg = calculateAvg(right_subarr[right_idx]);
                If (left_avg>=right_avg)
                Push elements of left_arr in coordinates
        (copy elements from left subarray intomain vector array coordinates)
                Else
                        Push elements of right_arr in coordinates
(copy elements from right subarray into main vector array coordintes)

3. Void mergesort(coordinates, left, right)
mid = (left+right)/2
mergesort(coordinates, left, mid)
mergesort(coordinates, mid+1, right)
merge(coordinates, left, mid, right)
4.
int main()
read the coordinates given by user
mergesort(coordinates, 0, coordinates.lenght -1)
print run time
print elements of the sorted array
```

(C)Pseudocode:

```
#include <iostream>
#include <vector>
#include <limits.h>
#include<ctime>
using namespace std;

double calculateAvg(pair<int, int>& point) {
   return (double)(point.first + point.second) / 2;
}

void merge(vector<pair<int, int>>& coordinates, int left, int mid, int right) {
   vector<pair<int, int>> left_arr, right_arr;
   int left_ele, right_ele, i, fa_idx, left_idx, right_idx;
   left_ele = mid - left + 1;
   right_ele = right - mid;

   for (i = 0; i < left_ele; i++)
       left_arr.push_back(coordinates[left + i]);
   left_arr.push_back({INT_MIN, INT_MIN});

   for (i = 0; i < right_ele; i++)
       right_arr.push_back(coordinates[mid + 1 + i]);
   right_arr.push_back({INT_MIN, INT_MIN});

   left_idx = 0;
   right_idx = 0;
   for (fa_idx = left; fa_idx <= right; fa_idx++) {
       int left_avg = calculateAvg(left_arr[left_idx]);
       int right_avg = calculateAvg(right_arr[right_idx]);
       if (left_avg >= right_avg) {
           coordinates[fa_idx] = left_arr[left_idx];
           left_idx++;
       } else {
           coordinates[fa_idx] = right_arr[right_idx];
           right_idx++;
       }
   }
}

void mergeSort(vector<pair<int, int>>& coordinates, int left, int right) {
   int mid;
   if (left == right) {
       return;
   }
   mid = (left + right) / 2;
   mergeSort(coordinates, left, mid);
   mergeSort(coordinates, mid + 1, right);
   merge(coordinates, left, mid, right);
}
```

```
int main() {
    int n, i, x, y;
    vector<pair<int, int>> coordinates;
    cout << "Enter the number of coordinates: ";
    cin >> n;
    srand(time(0));
    for (i = 0; i < n; i++) {
        int x = rand() % n;
        int y = rand() % n;
        coordinates.push_back({x, y});
    }

    clock_t tStart = clock();
    mergeSort(coordinates, 0, n - 1);

    double time = (double)(clock() - tStart) / CLOCKS_PER_SEC;
    cout << "Time: " << time << endl;

    for (i = n - 1; i >= 0; i--) {
        cout << coordinates[i].first << " " << coordinates[i].second << "\n";
    }
    return 0;
}
```

Time Complexity: O(N*Log(N))

~>As we divide our array into n levels such that n/2^k=1 then k =logn
~>as at every step we perform each step n number of times. Thus, The time complexity becomes n for each step.
~> Thus final time complexity for the whole code comes to nlogn as at each step we divide the given array into 2 and at every divided step it will take n units.

Proof of Correctness:

Initialization:
At the start, we have an array of elements that is not organized in any particular order. We establish the initial values for the left and right indices, which define the limits of the subarrays that need to be arranged in ascending or descending order.

Maintenance:
The merge sort algorithm operates by breaking down the original array into smaller subarrays repeatedly until each subarray consists of only one element. This process is known as the "divide" step.
Once the array is divided into single-element subarrays, the algorithm starts merging adjacent subarrays together to create larger sorted subarrays. This step is called the "merge" operation. During the merge operation, the algorithm compares and reorganizes the elements from the two subarrays being merged, ensuring that the resulting merged subarray is sorted.
By recursively applying the divide and merge steps, the merge sort algorithm ultimately sorts the original unsorted array by creating and merging progressively larger subarrays until the entire array is sorted in the desired order.
Termination:
The algorithm follows the process of dividing and merging until it reaches the smallest possible subarrays, which consist of just one element. As you mentioned, these single-element subarrays are already considered sorted.

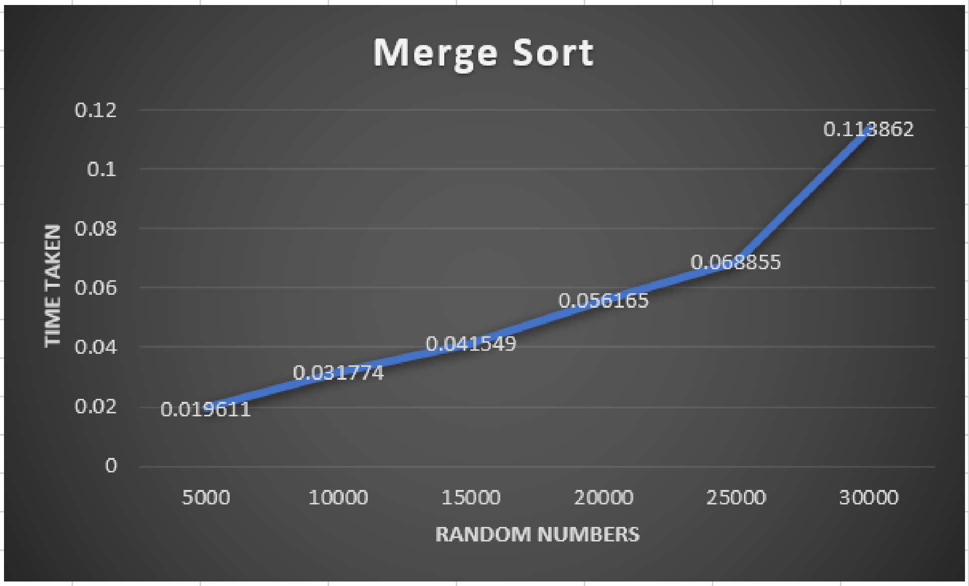
By merging adjacent subarrays, the algorithm progressively creates larger sorted subarrays. This process continues until all the subarrays are merged into a single sorted array. At this point, it can be concluded that the entire array is sorted.

Therefore, the algorithm terminates once all the subarrays are merged, and the array is sorted in the desired order.

Graph:

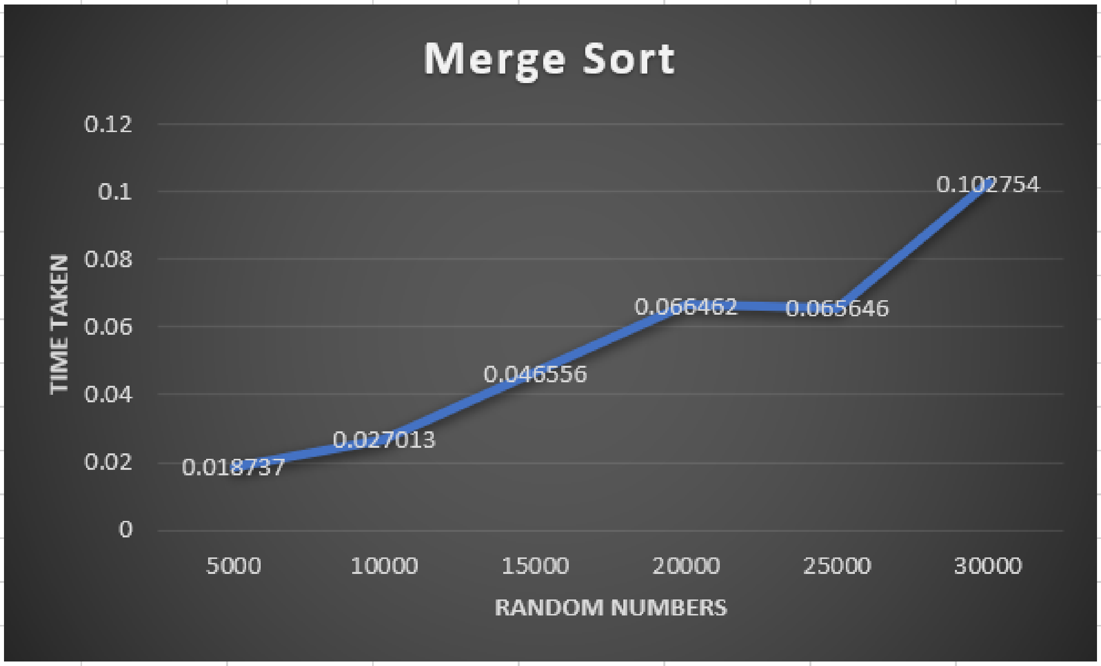
(A)

Random	Time Taken
5000	0.019611
10000	0.031774
15000	0.041549
20000	0.056165
25000	0.068855
30000	0.113862



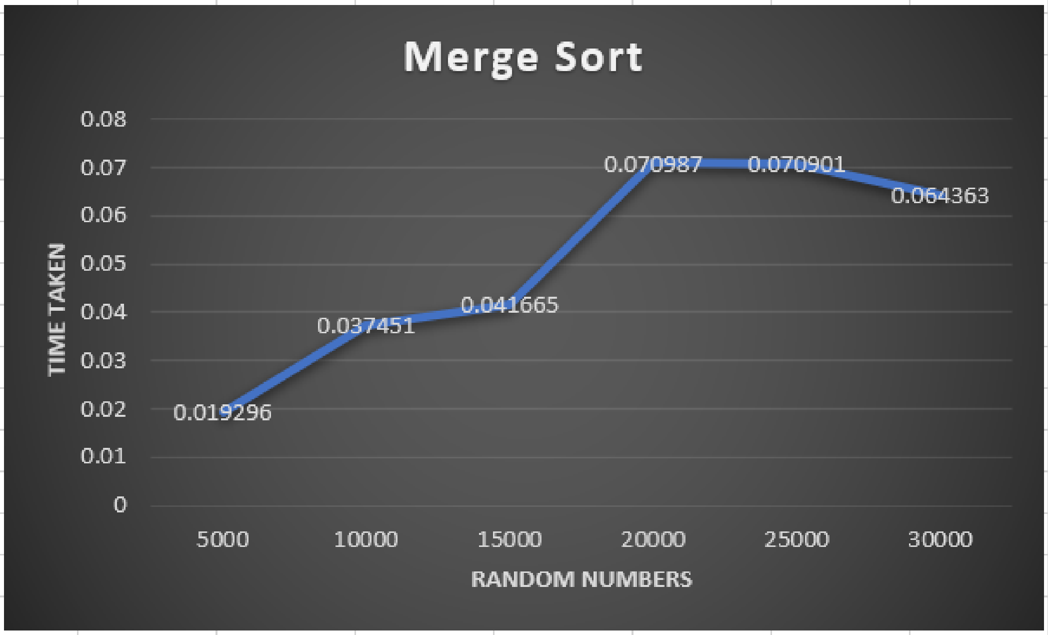
(B)

Random	Time Taken
5000	0.018737
10000	0.027013
15000	0.046556
20000	0.066462
25000	0.065646
30000	0.102754



(C)

Random	Time Taken
5000	0.019296
10000	0.037451
15000	0.041665
20000	0.070987
25000	0.070901
30000	0.064363



2. . Modify the Merge-sort algorithm (discussed in the class) such that the algorithm takes the input as words (of different lengths) and arrange them in an alphabetical order. Your words will have both lower-case letters as well as the upper-case letters. Compute the time-complexity t(n) of your algorithm in an experimental approach. Compare this algorithm with that of the algorithm which takes numbers as inputs and decide which consumes minimum time.

Problem Statement:

To apply the divide and conquer technique for sorting the alphabets in a word and arranging them in alphabetical order.
By repetitively dividing the word, sorting the individual parts, and then merging them back together, the divide and conquer technique enables the alphabets to be organized in the desired alphabetical order.

Logic:

Merge sort uses divide and conquer approach , it divides the main word into smaller words(or alphabets),By repetitively dividing the word, sorting the individual parts, and then merging them back together, the divide and conquer technique enables the alphabets to be organized in the desired alphabetical order.

Algorithm: 1.Import the necessary libraries

2. getCharValue(c) {

```
        If(c >= 'A' && c <= 'Z')
                return c - 'A' + 1
        return c - 'a' + 1
```

```
3.Void merge(word, left, mid, right)
create 2 subarrays, left and right
a1 = getCharValue(left_subarr[left_idx]);
a2 = getCharValue(right_subarr[right_idx]);
        if (a1 <= a2)
        push coordinates of left_subarr in words
        else
        push coordinates of right_subarr in words
```

```
4.
Void mergesort(words, left, right)
mid = (left+right)/2
mergesort(word, left, mid)
mergesort(word, mid+1, right)
merge(word, left, mid, right)
```

```
5. sortWord (word)
mergeSort(word, 0, word.length() - 1)
return word
```

Pseudocode:

```
#include <iostream>
#include <string>
#include <vector>
#include <limits.h>
#include <ctime>
using namespace std;

int getCharValue(char c) {
    if (c >= 'A' && c <= 'Z') {
        return c - 'A' + 1;
    }
    return c - 'a' + 1;
}
```

```
void merge(vector<string>& words, int left, int mid, int right) {
    int left_size = mid - left + 1;
    int right_size = right - mid;
    vector<string> left_arr(left_size);
    vector<string> right_arr(right_size);

    for (int i = 0; i < left_size; i++) {
        left_arr[i] = words[left + i];
    }

    for (int i = 0; i < right_size; i++) {
        right_arr[i] = words[mid + 1 + i];
    }

    int left_idx = 0, right_idx = 0, fa_idx = left;

    while (left_idx < left_size && right_idx < right_size) {
        if (left_arr[left_idx] <= right_arr[right_idx]) {
            words[fa_idx] = left_arr[left_idx];
            left_idx++;
        } else {
            words[fa_idx] = right_arr[right_idx];
            right_idx++;
        }
        fa_idx++;
    }

    while (left_idx < left_size) {
        words[fa_idx] = left_arr[left_idx];
        left_idx++;
        fa_idx++;
    }

    while (right_idx < right_size) {
        words[fa_idx] = right_arr[right_idx];
        right_idx++;
        fa_idx++;
    }
}

void mergeSort(vector<string>& words, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(words, left, mid);
        mergeSort(words, mid + 1, right);
        merge(words, left, mid, right);
    }
}

int main() {
    int n;
    cout << "Enter the number of words: ";
    cin >> n;
    vector<string> words(n);
    cout << "Enter " << n << " words:\n";
    for (int i = 0; i < n; i++) {
        cin >> words[i];
    }
    clock_t tStart=clock();
    mergeSort(words, 0, words.size() - 1);
    double time1= (double)(tStart-clock())/CLOCKS_PER_SEC;
    cout<< time1<<endl;
    cout << "Sorted words:\n";
    for (int i = 0; i < n; i++) {
        cout << words[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Time Complexity: O(N*Log(N))

~>As we divide our array(words) into n levels such that n/2^k=1 then k =logn

~>as at every step we perform each step n number of times. Thus, The time complexity becomes n for each step.

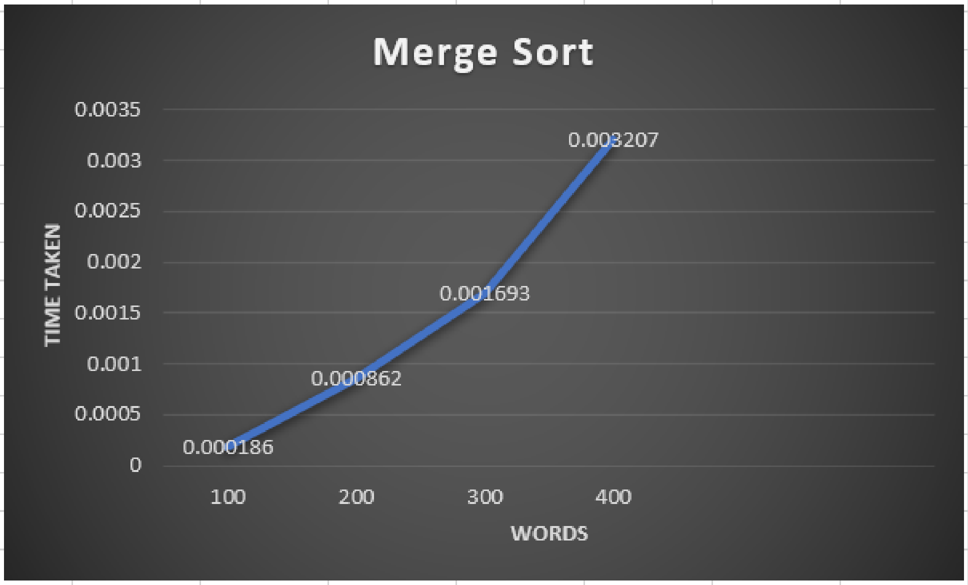
~> Thus final time complexity for the whole code comes to nlogn as at each step we divide the given words into 2 and at every divided step it will take n units for the code to sort through the given thing.

Proof of correctness:

Initialization: Initially, we have an unsorted word. We set the initial values for the left and right indices, representing the boundaries of the subarrays to be sorted.
Maintenance: The merge sort algorithm divides the word into alphabets.It then merges adjacent alphabets into sorted smaller words.It will keep on repeating the steps till it reaches the end for for loops without any discrepancy in the code.
Termination: The algorithm continues the process of merging till we get the sorted word.

Graph:

Words	Time Taken
100	0.000186
200	0.000862
300	0.001693
400	0.003207



3. Modify the merge-sort algorithm in such a way that the left subarray (got as a result of the first division) is divided further till we get a subarray of size 1 and the right-subarray (got as a result of the first division) is not subjected to any division. For sorting the right subarray, apply insertion-sort algorithm. Write the new algorithm.

Problem Statement:

To apply merge sort on the left subarray and to apply insertion sort on the right subarray of the main array and then to combine/merge the 2 sorted subarrays such that the result is a sorted final array.

Logic:

Merge sort uses divide and conquer approach , it will divide the main array into left and right subarrays. Then we will recursively call the merge sort function for the left subarray while we will just use insertion sort on the right subarray that we obtained after the first division.

Algorithm:

1.Import all the necessary libraries.

```
2. void merge(vector<int>& arr, int left, int mid, int right) {
    int left_idx = mid - left + 1;
    int right_idx = right - mid;
(Create temporary vectors for left and right subarrays)
  vector<int> left_arr(left_idx + 1);
  vector<int> right_arr(right_idx + 1);
// Copy data to temporary vectors
    for (int i = 0; i < left_idx; i++)
        left_arr[i] = arr[left + i];
    for (int j = 0; j < right_idx; j++)
        right_arr[j] = arr[mid + 1 + j];

    // Add sentinel values at the end of both subarrays
    left_arr[left_idx] = INT_MAX;
    right_arr[right_idx] = INT_MAX;

    int i = 0, j = 0, k = left;
    while (k <= right) {
        if (left_arr[i] <= right_arr[j]) {
            arr[k] = left_arr[i];         // put value in main array for both cases
            i++;
        } else {
            arr[k] = right_arr[j];
            j++;
        }
        k++;
    }
}
```

```
3. void insertionSort(vector<int>& arr, int left, int right) {
    for i from left+1 to right;
        int key = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
4.
void modifiedMergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        modifiedMergeSort(arr, left, mid);
        insertionSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
5.
int main()
read the vector array given by user.
modifiedMergeSort(arr, 0, arr.lenght -1)
print elements of the random array
print run time
print elements of the sorted array
```

Pseudocode:

```
#include <iostream>
#include <vector>
#include <limits.h>
# include<ctime>
using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {
    int left_idx = mid - left + 1;
    int right_idx = right - mid;
```

```
  vector<int> left_arr(left_idx + 1);
  vector<int> right_arr(right_idx + 1);
    for (int i = 0; i < left_idx; i++)
        left_arr[i] = arr[left + i];
    for (int j = 0; j < right_idx; j++)
        right_arr[j] = arr[mid + 1 + j];

    // Add sentinel values at the end of both subarrays
    left_arr[left_idx] = INT_MAX;
    right_arr[right_idx] = INT_MAX;

    int i = 0, j = 0, k = left;
    while (k <= right) {
        if (left_arr[i] <= right_arr[j]) {
            arr[k] = left_arr[i];
            i++;
        } else {
            arr[k] = right_arr[j];
            j++;
        }
        k++;
    }
}

void insertionSort(vector<int>& arr, int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int key = arr[i];
        int j = i - 1;

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

        arr[j + 1] = key;
    }
}
```

```
void modifiedMergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        modifiedMergeSort(arr, left, mid);
        insertionSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
int main() {
    vector<int> arr;
    int n,ele;
    cin>>n;
    srand(time(0));
    for(int i=0;i<n;i++){
        ele=rand()%n;
        arr.push_back(ele);
    }
    cout << "Original array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    clock_t tStart=clock();
    modifiedMergeSort(arr, 0, n - 1);
    double time=(double)(clock()-tStart)/CLOCKS_PER_SEC;
    cout<<time<<endl;
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Time Complexity:O(N*Log(N))

~> The modified merge sort divides the array into two halves recursively until the size becomes half. This division process has a time complexity of O(log n) since the array is divided in half at each recursive step.

such that n/2^k=1 then k =logn

~> The modified merge sort algorithm has two main parts:

1. Sorting the left subarray recursively: This part has a time complexity of O(n log n) since it uses the merge sort algorithm, which has a time complexity of O(n log n) for the average and worst cases.

2. Sorting the right subarray using insertion sort: The worst-case time complexity of the insertion sort algorithm is O(n^2). However, in this case, the right subarray is already partially sorted because it is the result of the first division. As a result, the actual time complexity for sorting the right subarray is closer to linear time O(n) in most practical cases.

Therefore, the overall time complexity of the modified merge sort algorithm is dominated by the recursive merge sort step, which is O(n log n). The insertion sort step for the right subarray has a lower impact on the overall time complexity.

In summary, the correct time complexity of the given code is O(n log n) for most practical cases, considering the average and worst-case scenarios.

Proof of correctness:

Initialization:

At the start of the algorithm, the input array is divided into subarrays, and each subarray of size 1 is considered sorted.

Maintenance:

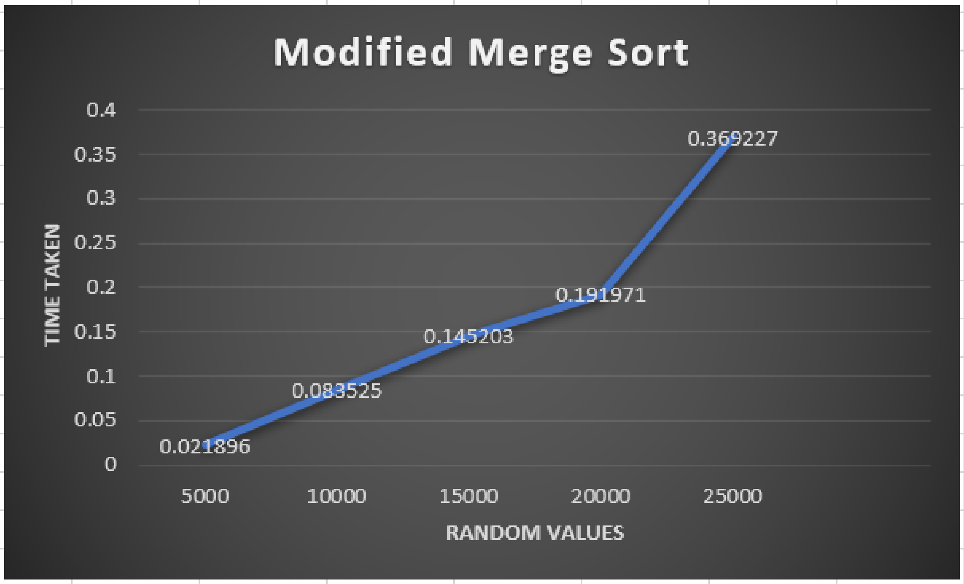
Assume that at each step of the algorithm, the subarrays being merged are sorted correctly. During the merge operation, the algorithm compares the elements from the left and right subarrays, selecting the smallest element and placing it in the merged array. Since the subarrays being merged are already sorted, the algorithm correctly merges them into a sorted subarray. Additionally, the insertion sort step for the right subarray ensures that the elements are sorted within that subarray. Thus, after each step, the algorithm maintains the property of having sorted subarrays.

Termination:

The algorithm recursively divides the array until the subarrays have a size of 1, at which point the merge sort process stops. The insertion sort is only applied to the right subarray, which is already partially sorted. Therefore, after the termination of the algorithm, the entire array is sorted correctly.

Graph:

Random	Time Taken
5000	0.021896
10000	0.083525
15000	0.145203
20000	0.191971
25000	0.369227


Modified Merge Sort
TIME TAKEN
RANDOM VALUES