Complexity Theory and Algorithms(3CS1109)

Practical:-2

Aim:Implement following sorting algorithms.

a) 2 Way Merge sort

b) External Merge Sort

Evaluate the time complexity of each algorithm on already sorted (ascending and descending) and non-sorted input values with varying size of input values. Visualize the same using graphical representation.

❖ Implementation and analysis of 2 way Merge Sort algorithm

```
//Merge Sort
#include<stdio.h>
#include<time.h>
double totaltime=0;
void merge(int a[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;
   /* create temp arrays */
   int L[n1], R[n2];
   /* Copy data to temp arrays L[] and R[] */
   for (i = 0; i < n1; i++)
```

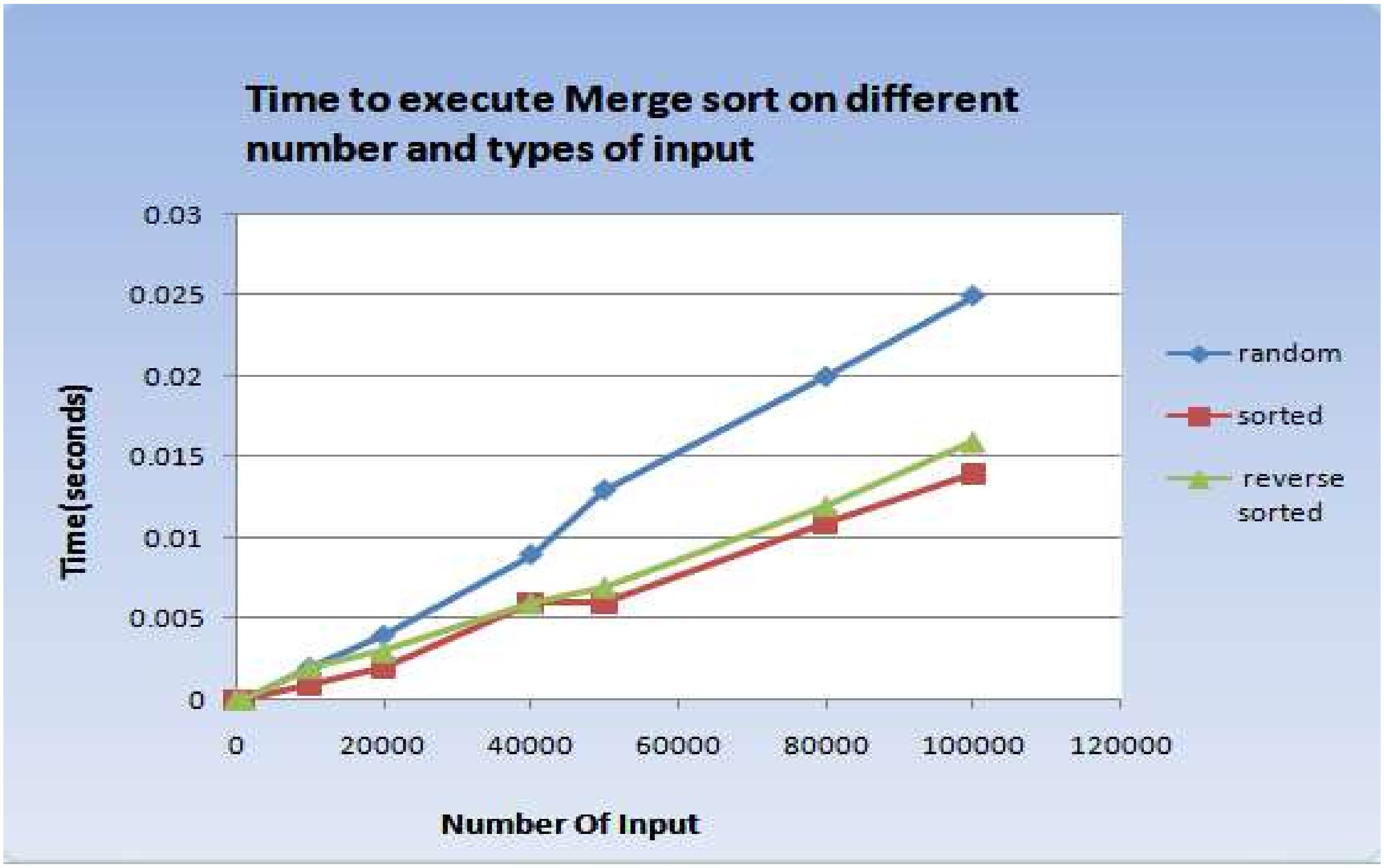
```
    L[i] = a[l + i];
for (j = 0; j < n2; j++)
    R[j] = a[m + 1 + j];


/* Merge the temp arrays back into arr[l..r]*/
i = 0; // Initial index of first subarray
j = 0; // Initial index of second subarray
k = l; // Initial index of merged subarray
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        a[k] = L[i];
        i++;
    }
    else {
        a[k] = R[j];
        j++;
    }
    k++;
}


/* Copy the remaining elements of L[], if there
are any */
while (i < n1) {
        a[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
    are any */
    while (j < n2) {
        a[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int *a, int l, int r)
{
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;
        // Sort first and second halves
        mergeSort(a, l, m);

        mergeSort(a, m + 1, r);

        merge(a, l, m, r);

    }

}


void print(int *a,int n)

{

    long i,p;

    printf("\nSorted array is:\n ");

    for (i=0; i<10; i++)

    {

        printf(" %d",a[i]);

    }

    if(n>10)

    {

        printf(".............");

        p=n-10;

        for (i=p; i<n; i++)

        {

            printf(" %d",a[i]);

        }

    }
```

```
}

void main()

{

   clock_t start, end;

  long  int *a,*b,*c;

   long int n,i,j;

   char ch;

   printf("----------------------------------------------------Merge  Sort--------------------------------------
-----------------------\n");

   do{

   printf("\nEnter the range of  elements in array :");

   scanf("%d",&n);

   a= (long int*)malloc(n * sizeof(long int));

   b= (long int*)malloc(n * sizeof(long int));

   c= (long int*)malloc(n * sizeof(long int));


   printf("\n---------------------------------Sorting of Random numbers---------------------------\n");

   for (i = 0; i < n; i++)

   {

      a[i] = rand();

   }

   start = clock();

   mergeSort(a, 0, n- 1);
```

```
    end = clock();
    //printf("\n%d",etime);
    totaltime=((double) (end - start)) / CLOCKS_PER_SEC;
    print(a,n);
    printf("\n total time in sorting: %f",totaltime);
    printf(" sec\n");


    printf("\n------------------Sorting of numbers which are sorted in ascending order-----------
\n");
    for (i = 0; i < n; i++)
    {
        b[i] = a[i];
    }
     start = clock();
    mergeSort(b, 0,n- 1);
    end = clock();
    //printf("\n%d",etime);
    totaltime=((double) (end - start)) / CLOCKS_PER_SEC;
    print(b,n);
    printf("\n total time in sorting: %f",totaltime);
    printf(" sec\n");


    printf("\n------------------Sorting of numbers which are sorted in descending order----------
\n");
       for (i=0,j=n-1; i < n; i++,j--)
    {
        c[i] = b[j];
    }
    start = clock();
    mergeSort(c,0,n- 1);
    end = clock();
    //printf("\n%d",etime);
    totaltime=((double) (end - start)) / CLOCKS_PER_SEC;
    print(c,n);
    printf("\n total time in sorting: %f",totaltime);
    printf(" sec\n");


    printf("\n Do you want to continue ? Press 'y' to continue:");
    fflush(stdin);
    scanf("%c",&ch);
     free(a);
    }
while((ch=='y') || (ch=='Y'));
}
```

Outcome:

Input Size	random	sorted	reverese
10	0	0	0
100	0	0	0
1000	0	0	0
10000	0.002	0.001	0.002
20000	0.004	0.002	0.003
40000	0.009	0.006	0.006
50000	0.013	0.006	0.007
80000	0.02	0.011	0.012
100000	0.025	0.014	0.016



Observation:

- For random input, Merge sort algorithm takes largest amount of time for sorting.
- For reversed sorted input, Merge sort algorithm takes moderate amount of time for sorting.
- For sorted input, Merge sort algorithm takes lowest amount of time for sorting.

Time Complexity:

- Best case: O(nlg(n))
- Average case: Θ (nlg(n))
- Worst case: Ω(nlg(n))

❖ Implementation and analysis of External Merge Sort algorithm

```
//External Merge Sort
#include <bits/stdc++.h>
using namespace std;

struct MinHeapNode {
        int element;
        int i;
};
void swap(MinHeapNode* x, MinHeapNode* y);
class MinHeap {
        MinHeapNode* harr;
        int heap_size;
public:
        MinHeap(MinHeapNode a[], int size);
        void MinHeapify(int);
        int left(int i) { return (2 * i + 1); }
        int right(int i) { return (2 * i + 2); }
        // to get the root
        MinHeapNode getMin() { return harr[0]; }
        void replaceMin(MinHeapNode x)
        {
```

```
        harr[0] = x;

        MinHeapify(0);

    }

};


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--;

    }

}


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);

    }

}


void swap(MinHeapNode* x, MinHeapNode* y)

{

    MinHeapNode temp = *x;

    *x = *y;

    *y = temp;

}


// Merges two subarrays of arr[].

// First subarray is arr[l to m]

// Second subarray is arr[m+1 to r]

void merge(int arr[], int l, int m, int r)

{

    int i, j, k;

    int n1 = m - l + 1;

    int n2 = r - m;
    int L[n1], R[n2];

    for (i = 0; i < n1; i++)

        L[i] = arr[l + i];

    for (j = 0; j < n2; j++)

        R[j] = arr[m + 1 + j];

    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(int arr[], int l, int r)

{

    if (l < r) {
        // Same as (l+r)/2
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

FILE* openFile(char* fileName, char* mode)
{
    FILE* fp = fopen(fileName, mode);
    if (fp == NULL) {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE);
    }
    return fp;
}

void mergeFiles(char* output_file, int n, int k)
{
    FILE* in[k];
    for (int i = 0; i < k; i++) {
        char fileName[2];
        // convert i to string
        snprintf(fileName, sizeof(fileName),
                "%d", i);
        in[i] = openFile(fileName, (char*)"r");
    }
    FILE* out = openFile(output_file, (char*)"w");
    // Create a min heap with k heap nodes.
    // Every heap node has first element of scratch
    // output file
    MinHeapNode* harr = new MinHeapNode[k];
    int i;
    for (i = 0; i < k; i++) {
        // break if no output file is empty and
        // index i will be no. of input files
        if (fscanf(in[i], "%d ", &harr[i].element) != 1)
            break;
        // Index of scratch output file
        harr[i].i = i;
    }
    // Create the heap
    MinHeap hp(harr, i);
    int count = 0;
    while (count != i) {
        // Get the minimum element
        // and store it in output file
        MinHeapNode root = hp.getMin();
        fprintf(out, "%d ", root.element);
        // Find the next element that
        // will replace current
        // root of heap. The next element
        // belongs to same
        // input file as the current min element.
        if (fscanf(in[root.i], "%d ",
                &root.element)
            != 1) {
            root.element = INT_MAX;
            count++;
        }
        // Replace root with next
        // element of input file
        hp.replaceMin(root);
    }
    for (i = 0; i < k; i++)
        fclose(in[i]);
    fclose(out);
}
```

```
// create the initial runs and divide them evenly among
// the output files
void createInitialRuns(char* input_file, int run_size,int num_ways)
{
    // For big input file
    FILE* in = openFile(input_file, (char*)"r");
    FILE* out[num_ways];
    char fileName[2];
    for (int i = 0; i < num_ways; i++) {
        // convert i to string
        snprintf(fileName, sizeof(fileName),
                "%d", i);
        out[i] = openFile(fileName, (char*)"w");
    }
    int* arr = (int*)malloc(run_size * sizeof(int));
    bool more_input = true;
    int next_output_file = 0;
    int j;
    while (more_input) {
        // write elements
        // into arr from input file
        for (j = 0; j < run_size; j++) {
            if (fscanf(in, "%d ", &arr[j]) != 1) {
                more_input = false;

                break;

            }

        }

        // sort array using merge sort

        mergeSort(arr, 0, j - 1);

        for (int k = 0; k < j; k++)

            fprintf(out[next_output_file],

                    "%d ", arr[k]);

        next_output_file++;

    }

    for (int k = 0; k < num_ways; k++)

        fclose(out[k]);

    fclose(in);

}


int main()

{

   printf("-------------------------------------------------External Merge Sort---------------------------
--------------------------------\n");

   int num_ways,run_size,ch;

   printf("Enter the number of way which you want to use for merge sort:");

   scanf("%d",&num_ways);
```

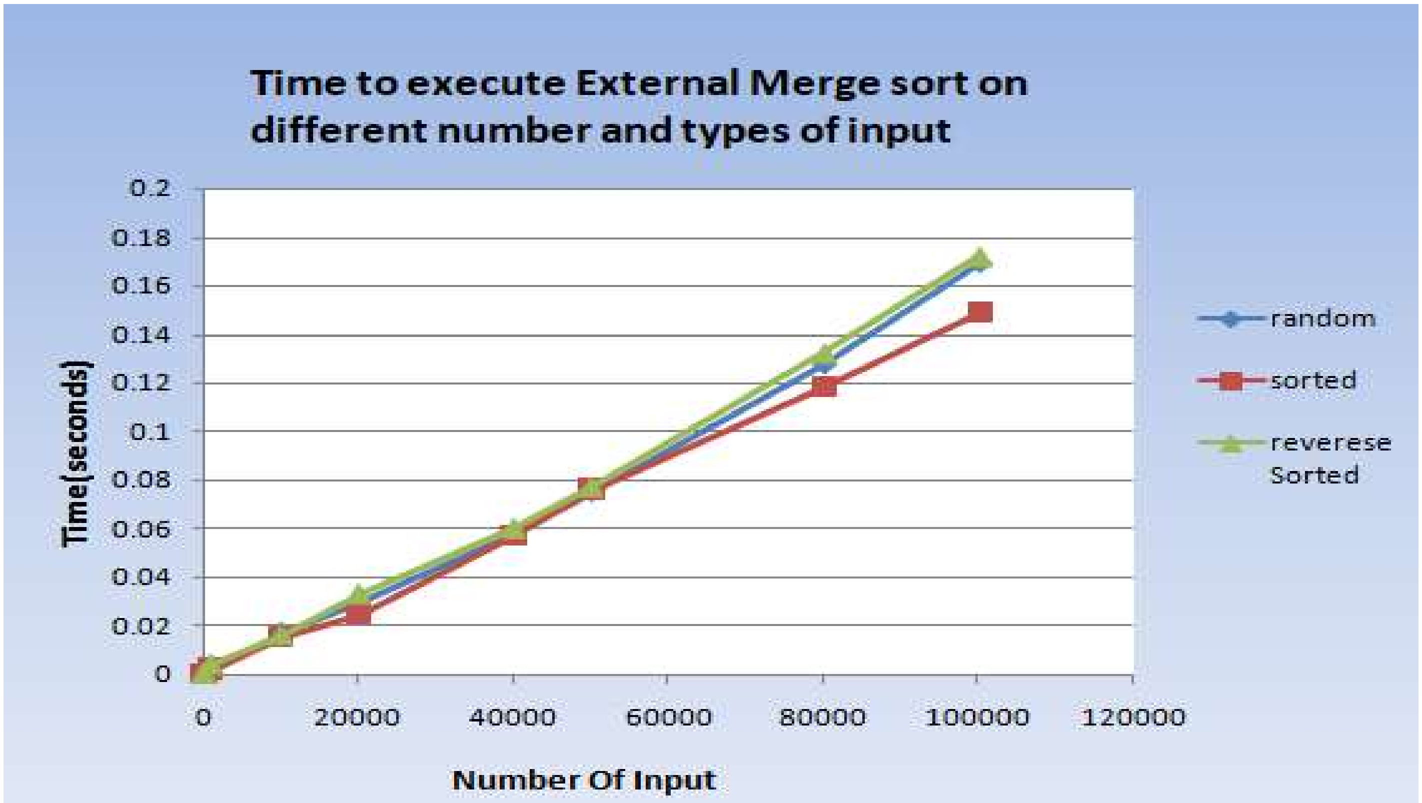
```
    printf("\nEnter the run size:");

    scanf("%d",&run_size);

    clock_t start_t, end_t;

    double total_t;

    char input_file[] = "input.txt";

    char output_file[] = "output.txt";

    FILE* in = openFile(input_file, (char*)"w");

    printf("\n################Input type#######################");

    printf("\n1.Random input");

    printf("\n2.Sorted input");

    printf("\n3.reversed input");

    printf("\nEnter your choice:");

    scanf("%d",&ch);


    switch (ch)

    {

    case 1:

        printf("\n------------------------------------Sorting of Random numbers-----------------------------
\n");

        srand(time(NULL));

        for (int i = 0; i < num_ways*run_size; i++)

                fprintf(in, "%d ", rand());

        break;
```

```
   case 2:

       printf("\n------------------Sorting of numbers which are sorted in ascending order------------
\n");

       for (int i = 0; i < num_ways*run_size; i++)

                 fprintf(in, "%d ", i*999);

       break;


   case 3:

       printf("\n------------------Sorting of numbers which are sorted in descending order----------
--\n");

       for (int i = num_ways*run_size -1; i >= 0; i--)

                 fprintf(in, "%d ", i*999);

       break;


   default:

       printf("Invalid Choice");

       exit(0);

   }

         fclose(in);

         start_t = clock();

         createInitialRuns(input_file, run_size, num_ways);

         mergeFiles(output_file,run_size, num_ways);

         end_t = clock();
```

```
        total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC;

        printf("Total time taken by CPU: %f",total_t);

        return 0;

}
```

Outcome:

Input Size	random	sorted	reverese
10	0	0	0
100	0	0	0
1000	0.0021	0.002	0.004
10000	0.017	0.015	0.016
20000	0.03	0.025	0.033
40000	0.058	0.057	0.06
50000	0.075	0.076	0.077
80000	0.1277	0.1179	0.132
100000	0.1694	0.148681	0.17203



Observation:

- External Merge sort algorithm, which sorts chunks that each fit in RAM, then merges the sorted chunks together.
- The file into runs such that the size of a run is small enough to fit into main memory. Then sort each run in main memory using Merge sort sorting algorithm. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.
- So, random, sorted and reversed sorted input, External Merge sort algorithm takes nearly equal amount of time for sorting.

Time Complexity:

- Best case: O(n + run_size lg run_size)
- Average case: Θ (n + run_size lg run_size)
- Worst case: Ω (n + run_size lg run_size)