ROLL NO: 20MIT3308

NAME: JAYNIL PATEL

BRANCH: DATA SCIENCE

COURSE: COMPLEXITY THEORY AND ALGORITHMS (3CS1109)

TOPIC: PRACTICAL 2

AIM

Implement following sorting algorithms.

a) 2 Way - Merge Sort

b) External Merge Sort

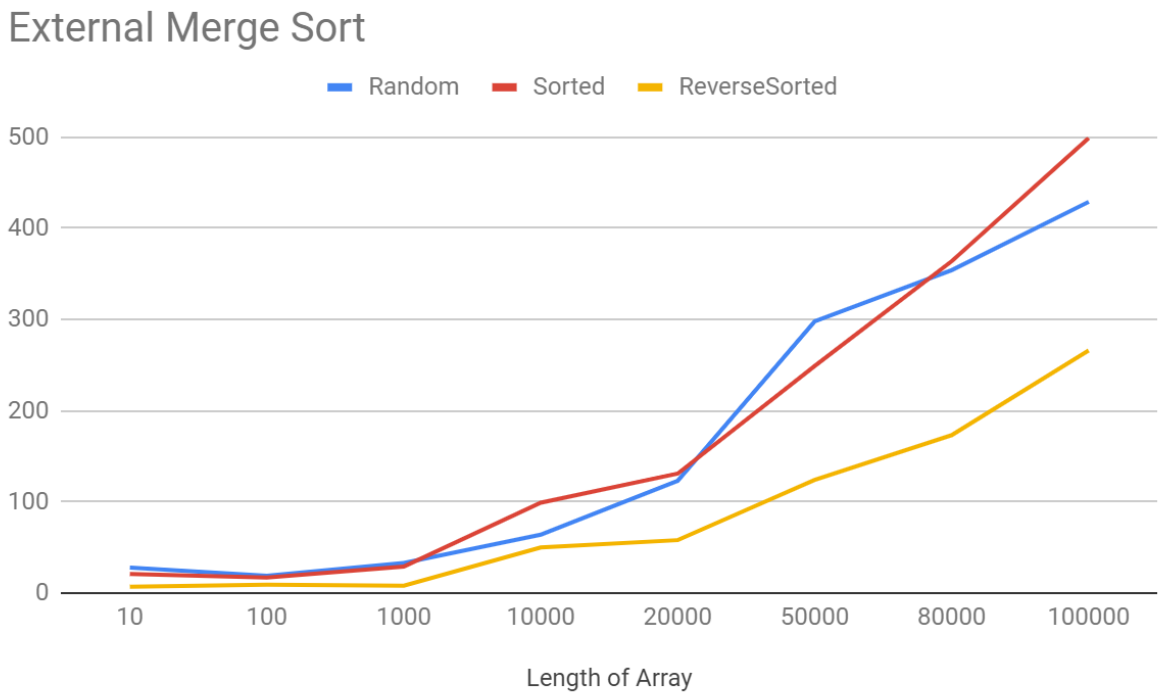
- Evaluate the time complexity of algorithm on an already sorted (ascending and descending) and non-sorted input values with varying size of input values.
- Visualize the same using graphical representation.

2- WAY MERGE SORT

Length	Random	Sorted	Reverse Sorted
10	0	0	0
100	0	0	0
1000	0	0	0
10000	1	1	2
20000	6	2	4
50000	11	8	5
80000	17	11	13
100000	21	12	13

EXTERNAL MERGE SORT

Length	Random	Sorted	Reverse Sorted
10	28	21	7
100	19	17	9
1000	33	29	8
10000	64	99	50
20000	123	131	58
50000	298	249	124
80000	354	364	173
100000	429	499	266



CODE (C++)

```cpp
#include <bits/stdc++.h>
#include<cstdlib>
#include<ctime>
#include<algorithm>
#include <fstream>
#include <sstream>
#include <iostream>
#include<time.h>
using namespace std;

struct MinHeapNode {
    // The element to be stored
    int element;

    // index of the array from which
    // the element is taken
    int i;
};

// Prototype of a utility function
// to swap two min heap nodes
void swap(MinHeapNode* x, MinHeapNode* y);

// A class for Min Heap
class MinHeap {
    // pointer to array of elements in heap
    MinHeapNode* harr;

    // size of min heap
    int heap_size;

public:
    // Constructor: creates a min
    // heap of given size
    MinHeap(MinHeapNode a[], int size);

    // to heapify a subtree with
    // root at given index
    void MinHeapify(int);

    // to get index of left child
    // of node at index i
    int left(int i) { return (2 * i + 1); }
```

```
    // to get index of right child
    // of node at index i
    int right(int i) { return (2 * i + 2); }

    // to get the root
    MinHeapNode getMin() { return harr[0]; }

    // to replace root with new node
    // x and heapify() new root
    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);
    }
}

// A utility function to swap two elements
void swap(MinHeapNode* x, MinHeapNode* y)
{
    MinHeapNode temp = *x;
    *x = *y;
    *y = temp;
}

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], 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] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    // Initial index of first subarray
    i = 0;

    // Initial index of second subarray
    j = 0;

    // Initial index of merged subarray
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }

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

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

/* l is for left index and r is right index of the
   sub-array of arr to be sorted */
void mergeSort(int arr[], 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(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

20MIT3308

```
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;
}

// Merges k sorted files. Names of files are assumed
// to be 1, 2, 3, ... k
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);

        // Open output files in read mode.
        in[i] = openFile(fileName, "r");
    }

    // FINAL OUTPUT FILE
    FILE* out = openFile(output_file, "w");


    MinHeapNode* harr = new MinHeapNode[k];
    int i;
    for (i = 0; i < k; i++) {

        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) {

        MinHeapNode root = hp.getMin();
        fprintf(out, "%d \n", root.element);

        if (fscanf(in[root.i], "%d ",
                   &root.element)
            != 1) {
            root.element = INT_MAX;
```

```
            count++;
        }


        hp.replaceMin(root);
    }

    // close input and output files
    for (int i = 0; i < k; i++)
        fclose(in[i]);

    fclose(out);
}



void printArray(int A[], int size)
{
    for(int i = 0; i < size; i++)
        cout << A[i] << " ";
}

void reverseArray(int arr[], int n){
   for (int low = 0, high = n - 1; low < high; low++, high--){
      swap(arr[low], arr[high]);
   }
}
void createInitialRuns(
    char* input_file, int run_size,
    int num_ways)
{
    // For big input file
    FILE* in = openFile(input_file, "r");

    // output scratch files

    FILE* out[num_ways];
    char fileName[run_size];
    for (int i = 0; i < num_ways; i++) {
        // convert i to string
        snprintf(fileName, sizeof(fileName),
                 "%d", i);


        out[i] = openFile(fileName, "w");
    }


    int* arr = (int*)malloc(
        run_size * sizeof(int));

    bool more_input = true;
    int next_output_file = 0;

    int i;
    while (more_input) {
```

```
        for (i = 0; i < run_size; i++) {
            if (fscanf(in, "%d ", &arr[i]) != 1) {
                more_input = false;
                break;
            }
        }


        mergeSort(arr, 0, i - 1);


        for (int j = 0; j < i; j++)
            fprintf(out[next_output_file],
                    "%d ", arr[j]);

        next_output_file++;
    }


    for (int i = 0; i < num_ways; i++)
        fclose(out[i]);

    fclose(in);
}


void externalSort(
    char* input_file, char* output_file,
    int num_ways, int run_size)
{

    createInitialRuns(input_file,
                      run_size, num_ways);


    mergeFiles(output_file, run_size, num_ways);
}


int main()
{
    ofstream extmerges;
    extmerges.open("extmerge.csv");
    extmerges << "Length,Random,Sorted,ReverseSorted\n" ;
    int len_arr[] = {10,100,1000,10000,20000,50000,80000,100000};


    int debug_len = 8;
    for (int i = 0; i < debug_len; i++)
     {
         int run_size = len_arr[i];

         int num_ways = 5 ;
         extmerges << run_size << ",";
```

```
    char input_file[] = "input_random.txt";
    char output_file[] = "output_random.txt";

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

    srand(time(NULL));

    // generate input
    for (int i = 0; i < num_ways * run_size; i++)
        fprintf(in, "%d \n", rand());

    fclose(in);

    cout << "\nLength of  random numbers = " << run_size << "\n";
    cout<< "Seq Type = Random\n";

    clock_t t;
    t=clock();
    externalSort(input_file, output_file, num_ways,run_size);
    t=clock()-t;
    double time_taken_1= ((double)t);
    cout << fixed  << time_taken_1 << setprecision(5) << "sec\t\n";
    extmerges << fixed << time_taken_1 << setprecision(5) << ",";


     char input_file_sorted[] = "output_random.txt";
     char output_file_sorted[] = "output_sorted.txt";

    cout<< "Seq Type = Sorted\n";
    t=clock();
    externalSort(input_file_sorted, output_file_sorted, num_ways,run_size);
    t=clock()-t;
    double time_taken_2= ((double)t);
    cout << fixed  << time_taken_2 << setprecision(5) << "sec\t\n";
    extmerges << fixed << time_taken_2 << setprecision(5) << ",";




    ifstream ifs("output_sorted.txt");
    int x;


   int temp_arr[run_size];
   int p = 0;
   while (ifs >> x)
   {
       //cout<<x << "\n";
       temp_arr[p] = x;
       if(p < run_size)
       {
```

```
            p++;
        }

    }
    ifs.close();
    reverseArray(temp_arr,run_size);
    //printArray(temp_arr,run_size);

    ofstream revsorted;
    revsorted.open("input_rev_sorted.txt");

    for (int i = 0; i < run_size; i++)
    {
          revsorted <<  temp_arr[i] << "\n";
    }
     revsorted.close();
     char input_file_rev_sorted[] = "input_rev_sorted.txt";
     char output_file_rev_sorted[] = "output_rev_sorted.txt";

     cout<< "Seq Type = ReverseSorted\n";
     t=clock();
     externalSort(input_file_rev_sorted, output_file_rev_sorted, num_ways,run_size);
     t=clock()-t;
     double time_taken_3= ((double)t);
     cout << fixed  << time_taken_3 << setprecision(5) << "sec\t\n";
     extmerges << fixed << time_taken_3 << setprecision(5) << ",\n";



      }
extmerges.close();
    return 0;
}
```

OBSERVATIONS

1. 2 - Way Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one..

- Time complexity of Merge Sort is θ(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
- For Rnadom inputs it takes maximum amount of time to run and for sorted and reverse sorted it take almost similar time to run.

2. External Merge Sort

External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead, they must reside in the slower external memory (usually a hard drive).

- Time taken for merge sort is O(nlogn), but there are at most run_size elements. So the time complexity is O(run_size log run_size) and then to merge the sorted arrays the time complexity is O(n). Therefore, the overall time complexity is O(n + run_size log run_size). Here the run_size = 2 .
- For the Random inputs it takes maximum amont of time and for reverse sorted it takes minimum amount of time.