LAB-1

Problem 1 Write a program that asks the user for a letter from the alphabet and then prints out the next character in the alphabet. Take account of both upper and lower case characters, and if the user enters 'z' or 'Z' then return 'a' or 'A', respectively. The program will loop continuously, until the user indicates the program should terminate. A sample run of the program follows.

```
#include <iostream>
using namespace std;

void nextletter(char c){
        if(c == 'z' || c == 'Z'){
        switch(c){
        case 'z':
                cout << "The next letter is: a" << endl;
                break;
        case 'Z':
                cout << "The next letter is: A" << endl;
                break;
        }
        }else{
        int val = (int)c;
        cout << "The next letter is: " << (char)(val+1) << endl;
        }
}

int main (int argc, char *argv[]) {
        bool flag = true;
        cout << "Hi! I'm a clever computer program that knows the alphabet."<< endl;
        do{
        char input;
        cout << "Please enter a letter:";
        cin >> input;
        if ((input > 'a' && input <= 'z') || (input > 'A' && input <= 'Z') ){
        nextletter(input);
        cout << "Do you want to enter another letter (y = yes)?";
        cin >> input;
        if(input != 'y'){
                flag = false;
```

```
            cout << "Goodbye" << endl;
        }
        }else{
        cout << "Enter a char type value!!" << endl;
        }
        }while(flag == true);
        return 0;
}
```

Problem 2. A computer program is required that reads positive integers, one per line, from the keyboard into a list. The user indicates that they are finished by entering any negative integer (the negative integer itself is not stored). The computer program then checks if any of the numbers in the list is the square root of any other number in the list. The program terminates after finding the first such number. Write this program using fixed-size arrays, the new and delete commands, and a loop to copy the elements from one array to another.

```
#include "iostream"
#include <algorithm>
#include <cmath>
#include <cwchar>
#include <functional>
using namespace std;

int* input(){ //recursive imp. of dynamic input through stack concept (stack is formed in
recursive function call)
        static int size;
        static int* arr;
        int val;
        cin >> val;
        if(val > 0){
        size++;
        input();
        arr[size--] = val;
        }else{
        arr = new int[size+1];
        arr[0] = size;
        }
        return arr;
```

```
}

void find_sqroot(int* arr, int size){

        int tmparr[size];
        for(int i = 0; i<size; i++){
        for(int j = 0; j<size; j++){
        if(arr[i] == arr[j]*arr[j]){
                cout << "Square root is: " << arr[i] << endl;
                return;
        }
        }
        }
        cout << "NO square root are found in the list." << endl;
        return;
}

int main (int argc, char *argv[]) {
        cout << "Enter the array: " << endl;
        int* tmplist = input(); //Declared function to take dynamic input from  user
        cout << endl;
        find_sqroot(tmplist+1,tmplist[0]); // actual function which finds the sqrt in the array
        delete []tmplist;
        return 0;
}
```

Problem 3 A computer program is required that reads single letters, one per line, from the keyboard into a list. The user indicates they are finished by entering a '%' character (the '%' itself is not stored). The computer program then checks if any of the letters, anywhere in the list, appear next to each other alphabetically. The program terminates after finding the first such pair of letters. Write this program using fixed-size arrays, the new and delete commands, and a loop to copy the elements from one array to another.

```
#include <iostream>
#include <pthread.h>
using namespace std;
int arr_len;
char* input(){
```

```
        static int size;
        static char* arr;
        char val;
        cin >> val;
        if(val != '%'){
        size++;
        input();
        arr[size--] = val;
        }else{
        arr = new char[size];
        arr_len = size;
        --size;
        }
        return arr;

}

void solve(char* arr){
        for(int i = 0; i<arr_len; i++){
        for(int j = 0; j<arr_len; j++){
        if(((int)arr[i])+1 == (int)arr[j]){
                cout << "Letters " << arr[i] << " and " << arr[j] << " apper next to each other
alphabetically." << endl;
                return;
        }
        }
        }
        cout << "No letters appear next to each other alphabetically." << endl;
}
int main (int argc, char *argv[]) {
        cout << "Enter the elements of the array: ";
        char* arr = input();
        solve(arr);
        delete []arr;
        return 0;
}
```

LAB - 2

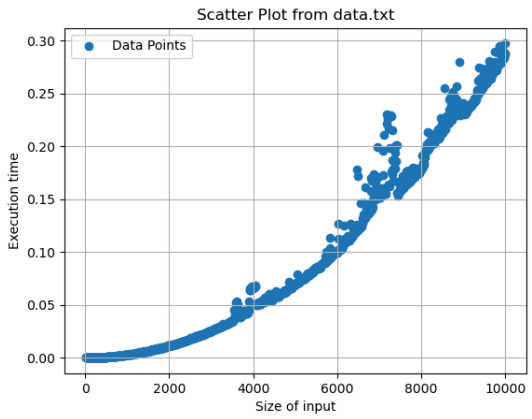
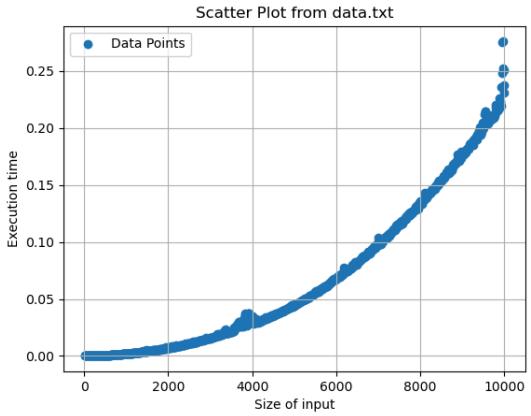
Algorithm Bubble sort

Code (Bubble Sort) :-

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>


void saveMatrixToFile(const char *filename, double matrix[][2], int rows) {
        FILE *file = fopen(filename, "w");
        if (file == NULL) {
        perror("Error opening file");
        return;
        }

        for (int i = 0; i < rows; i++)
        fprintf(file, "%f %f\n", matrix[i][0], matrix[i][1]);
        fclose(file);
}

int main(){
        double matrix[1000][2] = {};
        int counter = 0;
        for(int val = 10; val <= 10000; val+=10){
        clock_t start,end;
        int i,n,j,temp,k;
        n = val;
        int *p=(int*) malloc(n*sizeof(int));
        for(i=0;i<n;i++) *(p+i)=rand();
        start = clock();
        for(i=0;i<n;i++){
        for(j=0;j<n-1;j++){ //optimised version uses j<n-i-1 insted of j<n-1
                        if(*(p+j)>*(p+j+1)){
                                temp=*(p+j);
                                *(p+j)=*(p+j+1);
                                *(p+j+1)=temp;
                }
                }
        }
        end = clock();
        double time = (double)(end-start)/CLOCKS_PER_SEC;
```

```
        printf("The time for the event was: %f \n",time);
        matrix[counter][0] = time;
        matrix[counter++][1] = val;
        free(p);
        }
        int rows  = sizeof(matrix)/sizeof(matrix[0]);
        saveMatrixToFile("data.txt", matrix,rows);
        return 0;
}
```

Running Time Analysis :-

Unoptimised Bubblesort:	Optimised Bubblesort:
	

Plot Code:-

```
import matplotlib.pyplot as plt
import sys
# Read data from file
file = sys.argv[1]
with open(file, 'r') as file:
        lines = file.readlines()

# Extract x and y values
x_values = [float(line.split()[0]) for line in lines]
```

```
y_values = [float(line.split()[1]) for line in lines]

# Plot the data
plt.scatter(y_values, x_values, label='Data Points')
plt.title('Scatter Plot from data.txt')
plt.xlabel('Size of input')
plt.ylabel('Execution time')
plt.legend()
plt.grid(True)
plt.show()
```

LAB - 3

Algorithm Mergesort

Code (Merge Sort) :-

```
#include <iostream>
using namespace std;

void displayArray(int arr[], int size){
  for(int i = 0; i<size; i++){
        cout << arr[i] << " ";
  }
  cout << endl;

}
void merge(int array[], int start, int mid, int end){
        int arrayOneSize = mid - start+1;
        int arrayTwoSize = end - mid;

        int leftArray[arrayOneSize];
        int rightArray[arrayTwoSize];

        for(int i = 0; i<arrayOneSize; i++)
        leftArray[i] = array[start+i];
        //displayArray(leftArray,arrayOneSize);
        for(int i = 0; i<arrayTwoSize; i++)
        rightArray[i] = array[mid+1+i];
        //displayArray(rightArray,arrayTwoSize);
        int tmpOne =0;int tmpTwo = 0; int tmp = start;
        while(tmpOne < arrayOneSize && tmpTwo < arrayTwoSize){
        if(leftArray[tmpOne] <= rightArray[tmpTwo]){
        array[tmp++] = leftArray[tmpOne++];
        }
        else{
        array[tmp++] = rightArray[tmpTwo++];
        }
        }

        while(tmpOne < arrayOneSize){
        array[tmp++] = leftArray[tmpOne++];
        }
```

```
        while(tmpTwo < arrayTwoSize){
        array[tmp++] = rightArray[tmpTwo++];
        }


}


void mergesort(int array[], int start, int end){
        if(start < end){
        //cout << "Hello World" << endl;
        int mid = start+(end-start)/2;
        //cout << start << mid << end << endl;
        mergesort(array, start,mid);
        mergesort(array, mid+1,end);
        merge(array,start, mid, end);
        }
        return;
}

int main(){
        int size;
        cout << "Enter the size of the array" << endl;
        cin >> size;
        int randArray[size];
        for(int i=0;i<size;i++)
        randArray[i]=rand()%100;
        cout << "Array before sorting: " << endl;
        displayArray(randArray,size);
        mergesort(randArray,0,size-1);
        displayArray(randArray,size);
}
```

LAB - 4

Problem 4.1:

Objective:-Given an array S of unsorted elements. Design an algorithm and implement that to find a pair x, y such that x≠y from S that minimizes |x-y|.The worst case running time of the algorithm should be O (nlgn).

```
#include <iostream>
#include <iterator>
using namespace std;

void merge(int* array, int beg, int mid, int end){
        //this initial condition is very imp. for creation of array
        int arr_size_l = mid - beg + 1;
        int arr_size_r = end - mid;

        int tmp_arr_l[arr_size_l];
        int tmp_arr_r[arr_size_r];
        int tmp = 0;

        for(int i = beg; i<=mid; i++){
        tmp_arr_l[tmp++] = array[i];
        }

        tmp = 0;

        for(int i = mid+1; i<=end; i++){
        tmp_arr_r[tmp++] = array[i];
        }

        int tmp_l = 0, tmp_r = 0;
        tmp = beg;

        while(tmp_l < arr_size_l && tmp_r < arr_size_r){
        if(tmp_arr_l[tmp_l] <= tmp_arr_r[tmp_r]){
        array[tmp++] = tmp_arr_l[tmp_l++];
        }
        else{
        array[tmp++] = tmp_arr_r[tmp_r++];
        }
        }

        while(tmp_l < arr_size_l){
        array[tmp++] = tmp_arr_l[tmp_l++];
```

```
        }
        while(tmp_r < arr_size_r){
        array[tmp++]  = tmp_arr_r[tmp_r++];
        }
}
void merge_sort(int* array, int beg, int end){
        if(beg >= end){
        return;
        }
        int mid = (end + beg)/2;
        merge_sort(array,beg,mid);
        merge_sort(array,mid+1, end);
        merge(array,beg,mid,end);
}

void solve(int array[], int size){
        int min = array[size-1];
        int a, b;
        for(int i =0; i<size-1; i++){
        if(array[i+1] - array[i] < min){
        min = array[i+1] - array[i];
        a = array[i];
        b = array[i+1];
        }
        }
        cout << a << endl << b <<endl;
}

int main (int argc, char *argv[]) {
        int array[] = {4,15,8,1,19,0,12};
        merge_sort(array,0,size(array)-1); //O(n*logn)
        solve(array,size(array));
        return 0;
}
```


Problem 4.2:

Objective:-Given two arrays A1 , A2 of size n and a number x, design an algorithm to find whether there exists a pair of elements one from A1 and other from A2 whose sum is equal to x. Also find the indices of those elements

```
#include <iostream>
using namespace std;
```

```
void solution(int* arr1, int* arr2, int x, int size){
	int a,b;
	for(int i = 0; i<size; i++){
	for(int j = 0; j< size; j++){
	if(arr1[i] + arr2[j] == x){
		cout << i << " " << j << endl;
		return;
		}
	}
	}
	cout << "no" << endl;
}

int main (int argc, char *argv[]) {
	int a1[] = {4,5,8,1,3,9,0,2};
	int a2[] = {2,3,35,32,12,9,2};
	int size = sizeof(a1)/ sizeof(a1[0]);
	int x = 12;
	solution(a1,a2,x,size);
	return 0;
}
```

Problem 4.3:

Objective:-Develop an algorithm to solve the maximum segment sum problem.

```
#include <iostream>
using namespace std;

void solution(float* arr, int size){
	int greatest = arr[0];
	int sol, a, b;
	for(int i = 0; i<size; i++){
	sol = arr[i];
	for(int j = i+1; j<size; j++){
	sol += arr[j];
	if(greatest < sol){
		greatest = sol;
		a = i;
		b = j;
```

```
            }
            }
            }
            cout << sol << endl << a << " " << b << endl;
}

int main (int argc, char *argv[]) {
            float a[] = {4,-5, 8,-1, 3, -4.2, 0, 2, 12, 13};
            int size = sizeof(a)/sizeof(a[0]);
            solution(a,size);
            return 0;
}
```

LAB - 5

Objective:- Build a Max-Heap Tree for any array. Analyze the operation in terms of running time and find the time complexity of the algorithm.

```
#include <iostream>
using namespace std;

void swap(int *a, int *b){
        int tmp = *a;
        *a = *b;
        *b = tmp;
}

void maxHeapify(int array[], int n, int i){
        int largest = i;
        int left = 2*i+1;
        int right = 2*i+2;

        if(left < n && array[left] > array[largest]){
           largest = left;
        }
        if(right<n && array[right]>array[largest]){
           largest = right;
        }
        if(largest != i){
           swap(&array[i], &array[largest]);
           maxHeapify(array,n, largest);
        }
}
void buildMaxHeap(int array[], int n){
        for(int i = n/2-1; i>= 0; i--){
           maxHeapify(array,n,i);
        }
}

void display(int array[], int n){
        for(int i = 0; i<n ; i++){
           cout << array[i] << " ";
        }
        cout << endl;
}
```

```
int main(){
        int size;
        cout << "Enter the size of the array" << endl;
        cin >> size;
        int randArray[size];
        for(int i=0;i<size;i++)
           randArray[i]=rand()%100;
        cout << "Array before buildMaxHeap: ";
        display(randArray,size);
        buildMaxHeap(randArray,size);
        cout << "After buildMaxHeap: " ;
        display(randArray,size);
}
```

LAB - 6

Objective: - Implement an algorithm for Heap sort (for increasing order of elements) for any array. Analyze the operation in terms of running time and find the time complexity of the algorithm.

```
#include <iostream>
using namespace std;

void swap(int *a, int *b){
        int tmp = *a;
        *a = *b;
        *b = tmp;
}

void maxHeapify(int array[], int n, int i){
        int largest = i;
        int left = 2*i+1;
        int right = 2*i+2;

        if(left < n && array[left] > array[largest]){
           largest = left;
        }
        if(right<n && array[right]>array[largest]){
           largest = right;
        }
        if(largest != i){
           swap(&array[i], &array[largest]);
           maxHeapify(array,n, largest);
        }
}
void buildMaxHeap(int array[], int n){
        for(int i = n/2-1; i>= 0; i--){
           maxHeapify(array,n,i);
        }
}

void heapsort(int arr[], int n){
        buildMaxHeap(arr,n);
        for(int i = n-1; i>0; i--){
           swap(&arr[0], &arr[i]);
        maxHeapify(arr,i,0);
```

```
        }
}

void display(int array[], int n){
        for(int i = 0; i<n ; i++){
            cout << array[i] << " ";
        }
        cout << endl;
}

int main(){
        int size;
        cout << "Enter the size of the array" << endl;
        cin >> size;
        int randArray[size];
        for(int i=0;i<size;i++)
            randArray[i]=rand()%100;
        cout << "Array before Heapsort: ";
        display(randArray,size);
        heapsort(randArray,size);
        cout << "After Heapsort: " ;
        display(randArray,size);
}
```

LAB - 7

Objective: - Implement an algorithm of **Max-Priority-Queue** for any array of elements having **INSERT, EXTRACT-MAX, INCREASE-KEY** operations. Analyze its operations in terms of running time and find the time complexity of the algorithm.

```
#include <iostream>
#include <climits>

using namespace std;
const int MAX_SIZE = 100;

void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

void maxHeapifyUp(int A[], int n) {
  int i = n;
  while (i > 1 && A[i] > A[i / 2]) {
        swap(&A[i], &A[i / 2]);
        i /= 2;
  }
}

void maxHeapifyDown(int A[], int n, int i) {
  int largest = i;
  int left = 2 * i;
  int right = 2 * i + 1;

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

  if (largest != i) {
        swap(&A[i], &A[largest]);
        maxHeapifyDown(A, n, largest);
  }
}
```

```
void insert(int A[], int &n, int key) {
  if (n >= MAX_SIZE) {
        cout << "Overflow error: Queue is full\n";
        return;
  }
  n++;
  A[n] = INT_MIN;
  maxHeapifyUp(A, n);
  A[n] = key;
}


int extractMax(int A[], int &n) {
  if (n == 0) {
        cout << "Underflow error: Queue is empty\n";
        return -1;
  }
  int max = A[1];
  swap(&A[1], &A[n]);
  n--;
  maxHeapifyDown(A, n, 1);
  return max;
}


void increaseKey(int A[], int n, int i, int key) {
  if (i < 1 || i > n) {
        cout << "Invalid index\n";
        return;
  }
  if (key < A[i]) {
        cout << "Error: New key is smaller than current key\n";
        return;
  }
  A[i] = key;
  maxHeapifyUp(A, i);
}


void printQueue(int A[], int n) {
  for (int i = 1; i <= n; i++)
        cout << A[i] << " ";
  cout << endl;
}

int main() {
```

```
  int A[MAX_SIZE];
  int n = 0;
  insert(A, n, 3);
  insert(A, n, 2);
  insert(A, n, 15);
  insert(A, n, 5);

  cout << "Max-Heap: ";
  printQueue(A, n);
  cout << "Extracted max: " << extractMax(A, n) << endl;
  increaseKey(A, n, 2, 10);
  cout << "Max-Heap after increase: ";
  printQueue(A, n);
  return 0;
}
```