Nama: Farid Mardan Aziz

Nim: 231011403214

Matkul: Algoritma

No 1

a) Cara Menentukan Algoritma Sorting yang Paling Tepat

Pemilihan algoritma sorting tergantung pada karakteristik data dan kebutuhan aplikasi. Langkah-langkah untuk menentukan algoritma yang paling tepat:

Analisis Ukuran Data:

Untuk data kecil, algoritma sederhana seperti Insertion Sort atau Bubble Sort dapat digunakan karena overhead kecil.
Untuk data besar, algoritma efisien seperti Merge Sort, Quick Sort, atau Heap Sort lebih disukai.
Tingkat Keterurutan Data:

Jika data hampir terurut, algoritma seperti Insertion Sort bekerja sangat baik.
Untuk data acak atau hampir terbalik, Quick Sort atau Merge Sort lebih baik.
Kebutuhan Stabilitas:

Jika stabilitas diperlukan (elemen dengan nilai sama mempertahankan urutan awalnya), gunakan Merge Sort atau Bubble Sort.
Ketersediaan Memori:

Jika memori terbatas, gunakan algoritma in-place seperti Quick Sort atau Heap Sort.

b) Faktor-Faktor yang Perlu Dipertimbangkan

Ukuran Data:

Data kecil: $O(n^2)$ algoritma sederhana seperti Insertion atau Bubble cukup.
Data besar: Gunakan algoritma O(n log n) seperti Merge, Quick, atau Heap Sort.
Tingkat Keterurutan Awal:

Data hampir terurut: Insertion Sort efisien dengan kompleksitas O(n).
Data sangat acak: Quick Sort lebih cepat.
Stabilitas Algoritma:

Jika diperlukan stabilitas (misalnya saat data memiliki kunci tambahan), pilih Merge Sort atau Insertion Sort.
Keterbatasan Memori:

Quick Sort bekerja in-place, memerlukan ruang tambahan kecil dibanding Merge Sort yang membutuhkan O(n) ruang.
Kecepatan Rata-Rata vs Terburuk:

Quick Sort cepat dalam rata-rata kasus, tapi dapat melambat hingga $O(n^2)$ di kasus buruk.
Merge Sort memiliki waktu konsisten O(n log n), tetapi membutuhkan lebih banyak memori.

c) Perbandingan Binary Search dan Linear Search
Linear Search:
Metode: Mencari elemen dengan memeriksa satu per satu dari awal hingga akhir.
Kompleksitas Waktu: O(n).
Keuntungan: Tidak memerlukan data terurut.
Kelemahan: Lambat untuk dataset besar.
Binary Search:
Metode: Membagi data menjadi dua bagian berulang kali, mencari di bagian yang relevan.
Kompleksitas Waktu: O(log n).
Keuntungan: Sangat cepat untuk dataset besar.
Kelemahan: Membutuhkan data terurut.
Situasi Binary Search Lebih Disukai:
Saat dataset besar dan telah diurutkan.
Contoh: Mencari elemen dalam daftar harga yang sudah diurutkan.

No 2
a) Cara Menggabungkan Dua Array yang Sudah Terurut
Untuk menggabungkan dua array yang sudah terurut menjadi satu array yang tetap terurut, algoritma Merge digunakan. Algoritma ini bekerja dengan membandingkan elemen-elemen dari kedua array secara berurutan dan menambahkannya ke array hasil.

Kompleksitas waktu untuk algoritma ini adalah O(n + m), di mana n dan m adalah ukuran kedua array, karena kita hanya melakukan satu kali iterasi melalui setiap elemen array.

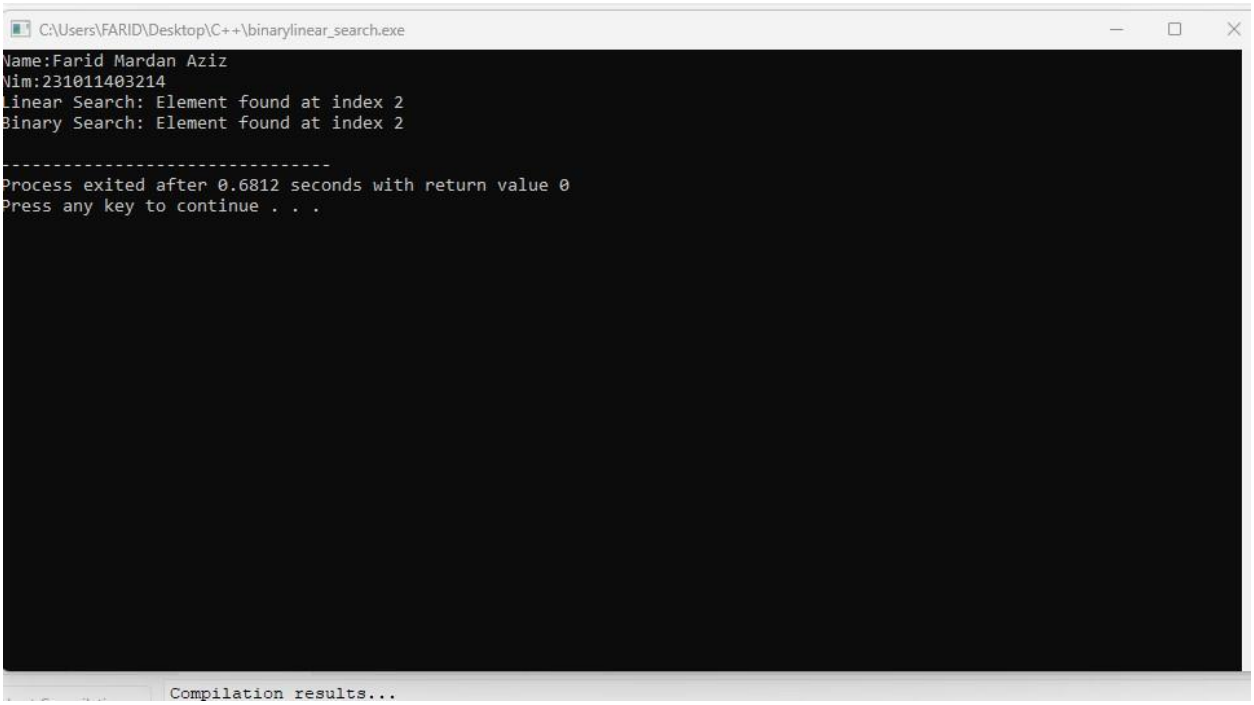
b) Algoritma Penggabungan yang Dipilih
Algoritma Merge dipilih karena:

Efisien: Algoritma ini hanya memerlukan iterasi linear melalui elemen-elemen dari kedua array.
Sederhana: Implementasinya langsung dengan satu loop melalui kedua array.
In-Place (Opsional): Dapat dilakukan tanpa alokasi memori tambahan jika dimodifikasi.

No 1 bagian c

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Linear Search Function
int linearSearch(const vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == target)
            return i; // Return index
    }
    return -1; // Not found
}

// Binary Search Function
int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2; // Avoid overflow
        if (arr[mid] == target)
            return mid; // Return index
        else if (arr[mid] < target)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1; // Not found
}

int main() {
            cout<<"Name:Farid Mardan Aziz"<<endl;
    cout<<"Nim:231011403214"<<endl;
    vector<int> data = {10, 20, 30, 40, 50};
    int target = 30;

    // Linear Search
    int linearResult = linearSearch(data, target);
    if (linearResult != -1)
        cout << "Linear Search: Element found at index " << linearResult << endl;
    else
        cout << "Linear Search: Element not found" << endl;

    // Binary Search (requires sorted data)
    sort(data.begin(), data.end()); // Ensure data is sorted
```

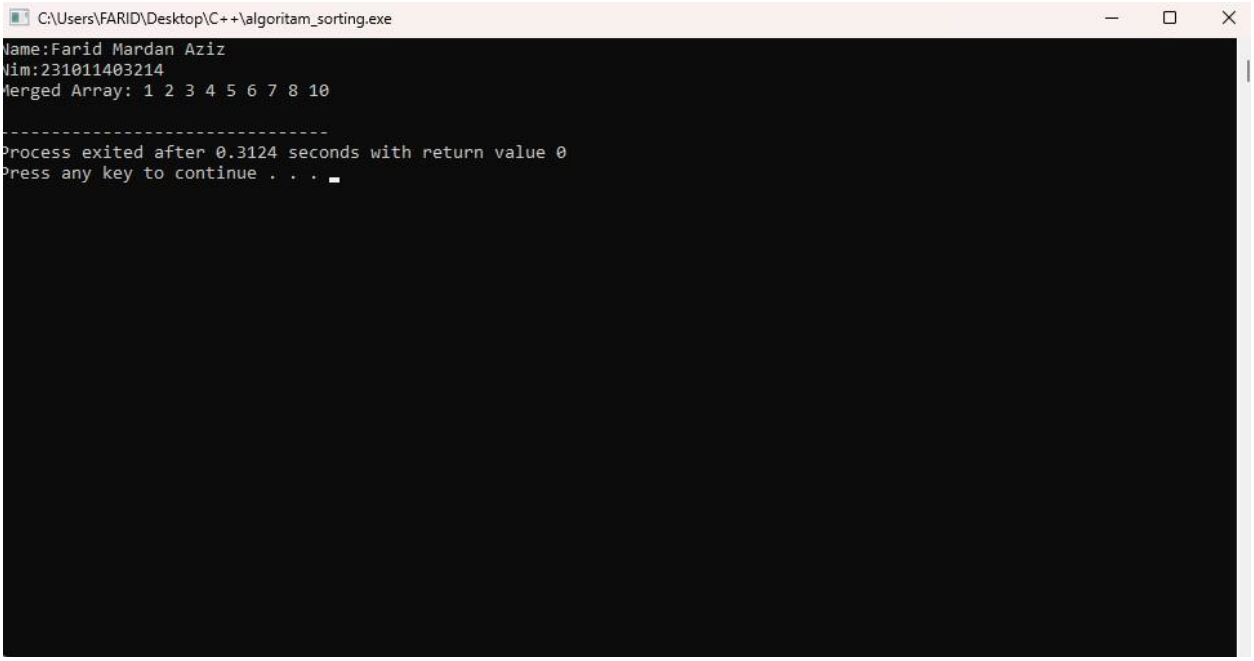
```
    int binaryResult = binarySearch(data, target);
    if (binaryResult != -1)
        cout << "Binary Search: Element found at index " << binaryResult << endl;
    else
        cout << "Binary Search: Element not found" << endl;

    return 0;
}
```



No 2bagian c

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

// Function to merge two sorted arrays
vector<int> mergeArrays(const vector<int>& arr1, const vector<int>& arr2) {
    vector<int> merged; // Resultant merged array
    int i = 0, j = 0;

    // Merge both arrays
    while (i < arr1.size() && j < arr2.size()) {
        if (arr1[i] <= arr2[j]) {
            merged.push_back(arr1[i]);
            i++;
        } else {
            merged.push_back(arr2[j]);
```

```
            j++;
        }
    }

    // Add remaining elements of arr1
    while (i < arr1.size()) {
        merged.push_back(arr1[i]);
        i++;
    }

    // Add remaining elements of arr2
    while (j < arr2.size()) {
        merged.push_back(arr2[j]);
        j++;
    }

    return merged;
}

int main() {
    // Example input
    cout<<"Name:Farid Mardan Aziz"<<endl;
    cout<<"Nim:231011403214"<<endl;
    vector<int> array1 = {1, 3, 5, 7};
    vector<int> array2 = {2, 4, 6, 8, 10};

    // Merge arrays
    vector<int> result = mergeArrays(array1, array2);

    // Print merged array
    cout << "Merged Array: ";
    for (int num : result) {
        cout << num << " ";
    }
    cout << endl;

    return 0;
}
```


C:\Users\FARID\Desktop\C++\algoritam_sorting.exe
Name:Farid Mardan Aziz
Nim:231011403214
Merged Array: 1 2 3 4 5 6 7 8 10

Process exited after 0.3124 seconds with return value 0
Press any key to continue . . .