Praktikum Algoritma dan Struktur Data

Quick and Merge Sort

Oleh :

Rayhan Elmo Athalah Saputra (5223600027)

Program Studi Sarjana Terapan Teknologi Game

Departemen Teknologi Multimedia Kreatif

Politeknik Elektronika Negeri Surabaya

2024

A. Percobaan

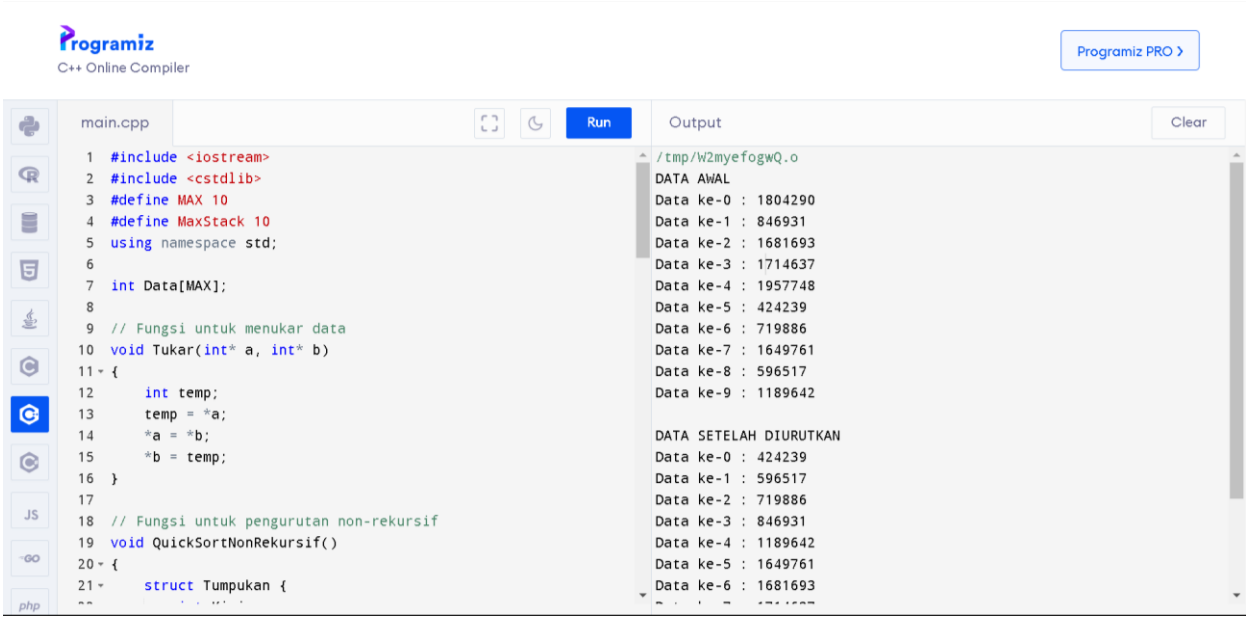
1. Pengurutan dengan metode Quick sort non-rekursif.

[Input]

```
#include <iostream>
#include <cstdlib>
#define MAX 10
#define MaxStack 10
using namespace std;

int Data[MAX];

// Fungsi untuk menukar data
void Tukar(int* a, int* b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

// Fungsi untuk pengurutan non-rekursif
void QuickSortNonRekursif()
{
    struct Tumpukan {
        int Kiri;
        int Kanan;
    } Tumpukan[MaxStack];

    int i, j, L, R, x, ujung = 1;

    // Inisialisasi tumpukan
    Tumpukan[1].Kiri = 0;
    Tumpukan[1].Kanan = MAX - 1;

    // Loop utama untuk pengurutan
    while (ujung != 0) {
        L = Tumpukan[ujung].Kiri;
        R = Tumpukan[ujung].Kanan;
        ujung--;

        // Pembagian data menjadi dua bagian
        while (R > L) {
```

```
            i = L;
            j = R;
            x = Data[(L + R) / 2];

            // Pindahkan elemen yang lebih kecil dari pivot ke kiri
            // dan elemen yang lebih besar dari pivot ke kanan
            while (i <= j) {
                while (Data[i] < x)
                    i++;
                while (x < Data[j])
                    j--;
                if (i <= j) {
                    Tukar(&Data[i], &Data[j]);
                    i++;
                    j--;
                }
            }

            // Periksa dan masukkan indeks baru ke tumpukan
            if (L < i) {
                ujung++;
                Tumpukan[ujung].Kiri = i;
                Tumpukan[ujung].Kanan = R;
            }
            R = j;
        }
    }
}

int main()
{
    srand(0);

    // Menghasilkan angka acak
    cout << "DATA AWAL" << endl;
    for (int i = 0; i < MAX; i++)
    {
        Data[i] = (int)rand() / 1000 + 1;
        cout << "Data ke-" << i << " : " << Data[i] << endl;
    }

    // Mengurutkan menggunakan QuickSort
    QuickSortNonRekursif();

    // Menampilkan data setelah diurutkan
    cout << "\nDATA SETELAH DIURUTKAN" << endl;
    for (int i = 0; i < MAX; i++)
```

```
    {
        cout << "Data ke-" << i << " : " << Data[i] << endl;
    }

    return 0;
}
```



Output

---

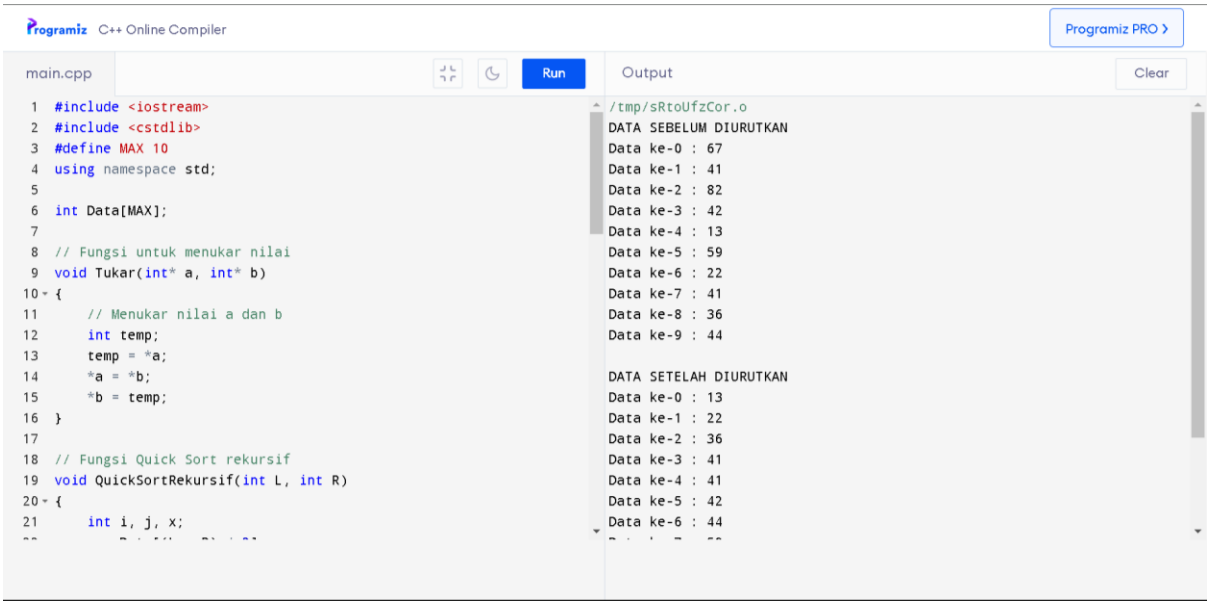
2. Pengurutan dengan metode Quick sort rekursif.

[Input]

```
#include <iostream>
#include <cstdlib>
#define MAX 10
using namespace std;

int Data[MAX];

// Fungsi untuk menukar nilai
void Tukar(int* a, int* b)
{
    // Menukar nilai a dan b
    int temp;
    temp = *a;
    *a = *b;
```

```
    *b = temp;
}

// Fungsi Quick Sort rekursif
void QuickSortRekursif(int L, int R)
{
    int i, j, x;
    x = Data[(L + R) / 2];
    i = L;
    j = R;
    while (i <= j) {
        while (Data[i] < x)
            i++;
        while (Data[j] > x)
            j--;
        if (i <= j) {
            // Menukar elemen yang ditemukan
            Tukar(&Data[i], &Data[j]);
            i++;
            j--;
        }
    }
    if (L < j)
        // Memanggil rekursif untuk bagian kiri
        QuickSortRekursif(L, j);
    if (i < R)
        // Memanggil rekursif untuk bagian kanan
        QuickSortRekursif(i, R);
}

int main()
{
    srand(42); // Mengatur seed agar hasil random tetap sama

    // Inisialisasi array dengan bilangan acak
    cout << "DATA SEBELUM DIURUTKAN" << endl;
    for (int i = 0; i < MAX; i++)
    {
        // Menggunakan operasi modulus untuk membatasi nilai random
        Data[i] = (int)rand() % 100 + 1;
        cout << "Data ke-" << i << " : " << Data[i] << endl;
    }

    // Panggil fungsi QuickSortRekursif untuk mengurutkan array
    QuickSortRekursif(0, MAX - 1);

    // Tampilkan array setelah diurutkan
```

```
    cout << "\nDATA SETELAH DIURUTKAN" << endl;
    for (int i = 0; i < MAX; i++)
    {
        cout << "Data ke-" << i << " : " << Data[i] << endl;
    }

    return 0;
}
```



Output

---

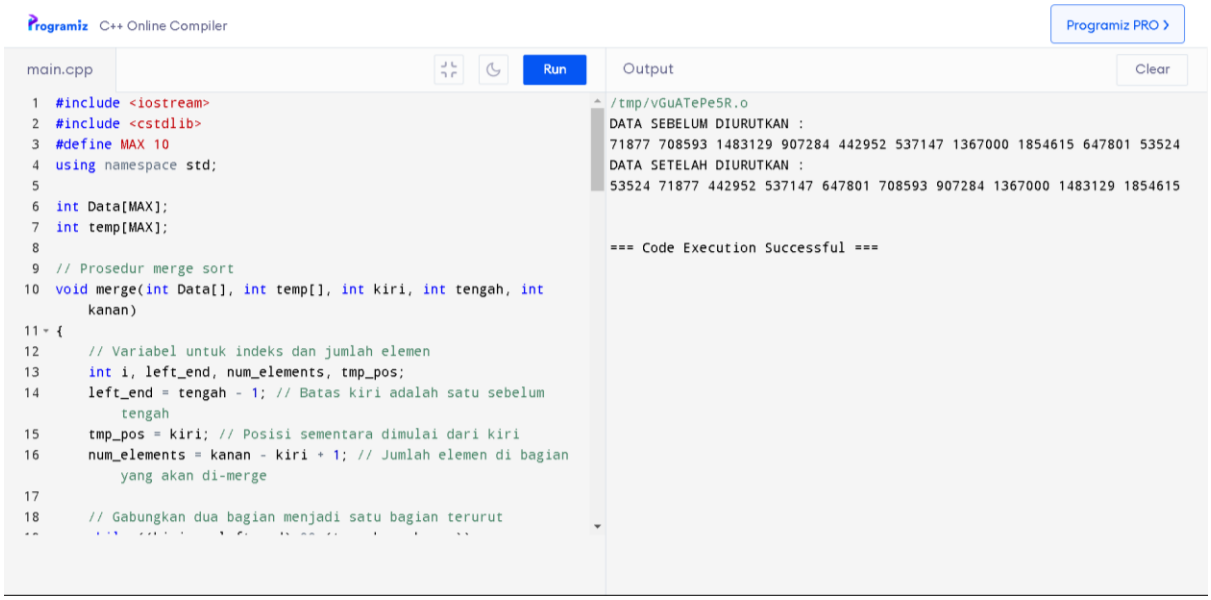
3. Pengurutan dengan metode Merge sort.

[Input]

```
#include <iostream>
#include <cstdlib>
#define MAX 10
using namespace std;

int Data[MAX];
int temp[MAX];

// Prosedur merge sort
void merge(int Data[], int temp[], int kiri, int tengah, int kanan)
{
    // Variabel untuk indeks dan jumlah elemen
    int i, left_end, num_elements, tmp_pos;
```

```
    left_end = tengah - 1; // Batas kiri adalah satu sebelum tengah
    tmp_pos = kiri; // Posisi sementara dimulai dari kiri
    num_elements = kanan - kiri + 1; // Jumlah elemen di bagian yang
akan di-merge

    // Gabungkan dua bagian menjadi satu bagian terurut
    while ((kiri <= left_end) && (tengah <= kanan))
    {
        if (Data[kiri] <= Data[tengah])
        {
            temp[tmp_pos] = Data[kiri];
            kiri++;
        }
        else
        {
            temp[tmp_pos] = Data[tengah];
            tengah++;
        }
        tmp_pos++;
    }

    // Salin sisa elemen dari bagian kiri (jika ada)
    while (kiri <= left_end)
    {
        temp[tmp_pos] = Data[kiri];
        kiri++;
        tmp_pos++;
    }

    // Salin sisa elemen dari bagian kanan (jika ada)
    while (tengah <= kanan)
    {
        temp[tmp_pos] = Data[tengah];
        tengah++;
        tmp_pos++;
    }

    // Salin elemen dari array temp ke array Data
    for (i = 0; i < num_elements; i++)
    {
        Data[kanan] = temp[kanan];
        kanan--;
    }
}

// Prosedur untuk membagi dan mengurutkan array secara rekursif
void m_sort(int Data[], int temp[], int kiri, int kanan)
```

```
{
    if (kanan > kiri)
    {
        int tengah = (kanan + kiri) / 2; // Tentukan titik tengah
        m_sort(Data, temp, kiri, tengah); // Panggil rekursif untuk
bagian kiri
        m_sort(Data, temp, tengah + 1, kanan); // Panggil rekursif
untuk bagian kanan
        merge(Data, temp, kiri, tengah + 1, kanan); // Gabungkan dua
bagian yang terurut
    }
}

// Fungsi merge sort utama
void mergeSort(int Data[], int temp[], int array_size)
{
    m_sort(Data, temp, 0, array_size - 1); // Panggil prosedur
rekursif dengan batas awal dan akhir
}

int main()
{
    srand(42); // Mengatur seed agar hasil random tetap sama

    // Inisialisasi array dengan bilangan acak
    cout << "DATA SEBELUM DIURUTKAN :\n";
    for (int i = 0; i < MAX; i++)
    {
        Data[i] = rand() / 1000 + 1; // Batasi rentang nilai random
        cout << Data[i] << " ";
    }

    mergeSort(Data, temp, MAX); // Panggil fungsi merge sort

    // Tampilkan array setelah diurutkan
    cout << "\nDATA SETELAH DIURUTKAN :\n";
    for (int i = 0; i < MAX; i++)
        cout << Data[i] << " ";
    cout << endl;

    return 0;
}
```



Output

B. Latihan

1. **Tambahkan kode program untuk menampilkan perubahan setiap iterasi dari proses pengurutan dengan Quick sort dan Merge sort.**

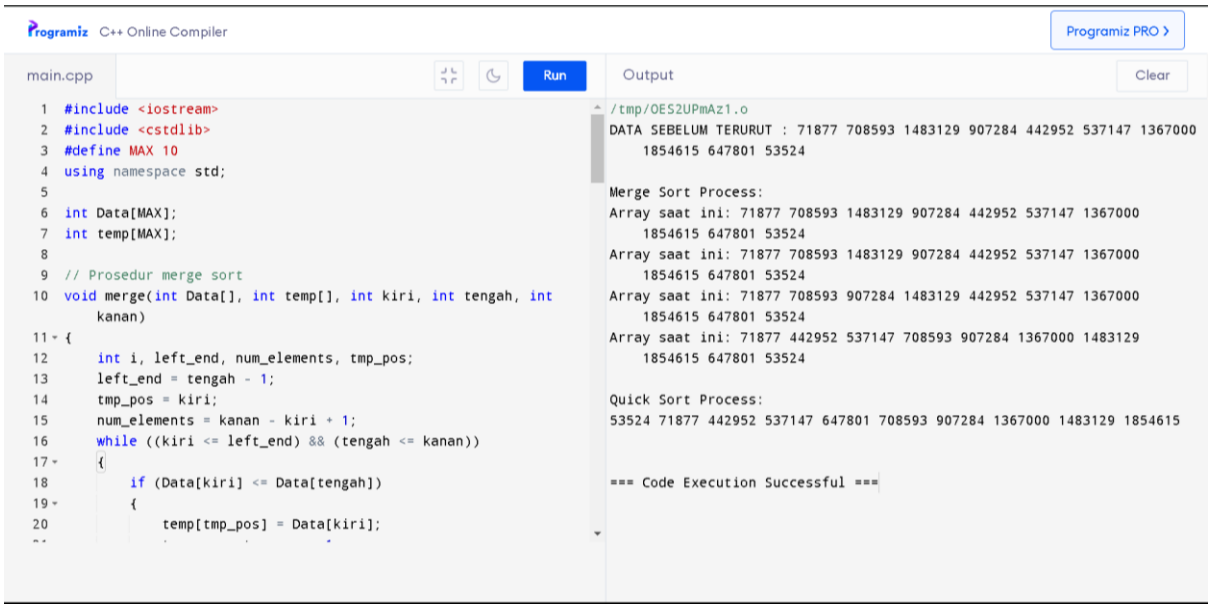
[Input]

```
#include <iostream>
#include <cstdlib>
#define MAX 10
using namespace std;

int Data[MAX];
int temp[MAX];

// Prosedur merge sort
void merge(int Data[], int temp[], int kiri, int tengah, int kanan)
{
    int i, left_end, num_elements, tmp_pos;
    left_end = tengah - 1;
    tmp_pos = kiri;
    num_elements = kanan - kiri + 1;
    while ((kiri <= left_end) && (tengah <= kanan))
    {
        if (Data[kiri] <= Data[tengah])
        {
```

```
            temp[tmp_pos] = Data[kiri];
            tmp_pos = tmp_pos + 1;
            kiri = kiri + 1;
        }
        else
        {
            temp[tmp_pos] = Data[tengah];
            tmp_pos = tmp_pos + 1;
            tengah = tengah + 1;
        }
    }
    while (kiri <= left_end)
    {
        temp[tmp_pos] = Data[kiri];
        kiri = kiri + 1;
        tmp_pos = tmp_pos + 1;
    }
    while (tengah <= kanan)
    {
        temp[tmp_pos] = Data[tengah];
        tengah = tengah + 1;
        tmp_pos = tmp_pos + 1;
    }
    for (i = 0; i < num_elements; i++)
    {
        Data[kanan] = temp[kanan];
        kanan = kanan - 1;
    }
}

// Prosedur membuat kumpulan data
void m_sort(int Data[], int temp[], int kiri, int kanan)
{
    int tengah;
    if (kanan > kiri)
    {
        tengah = (kanan + kiri) / 2;
        m_sort(Data, temp, kiri, tengah);
        m_sort(Data, temp, tengah + 1, kanan);
        merge(Data, temp, kiri, tengah + 1, kanan);
    }
}
void mergeSort(int Data[], int temp[], int array_size)
{
    m_sort(Data, temp, 0, array_size - 1);
}
```

```
// Prosedur pengurutan metode Quick Sort
void QuickSortRekursif(int L, int R)
{
    int i, j, x;
    x = Data[(L + R) / 2];
    i = L;
    j = R;
    while (i <= j) {
        while (Data[i] < x)
            i++;
        while (Data[j] > x)
            j--;
        if (i <= j) {
            swap(Data[i], Data[j]);
            i++;
            j--;
        }
    }
    if (L < j)
        QuickSortRekursif(L, j);
    if (i < R)
        QuickSortRekursif(i, R);
}

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

int main()
{
    srand(42); // Mengatur seed agar hasil random tetap sama

    // Membangkitkan bilangan acak
    cout << "DATA SEBELUM TERURUT : ";
    for (int i = 0; i < MAX; i++)
    {
        Data[i] = rand() / 1000 + 1;
        cout << Data[i] << " ";
    }
    cout << endl;

    // Merge Sort
    cout << "\nMerge Sort Process:" << endl;
    for (int size = 1; size <= MAX; size *= 2)
```

```
    {
        cout << "Array saat ini: ";
        mergeSort(Data, temp, size);
        displayArray(Data, MAX);
    }

    // Quick Sort
    cout << "\nQuick Sort Process:" << endl;
    QuickSortRekursif(0, MAX - 1);
    displayArray(Data, MAX);

    return 0;
}
```



Output

---

2. **Tambahkan kode program untuk menghitung banyaknya perbandingan dan pergeseran pada algoritma Quick sort dan Merge sort.**

[Input]

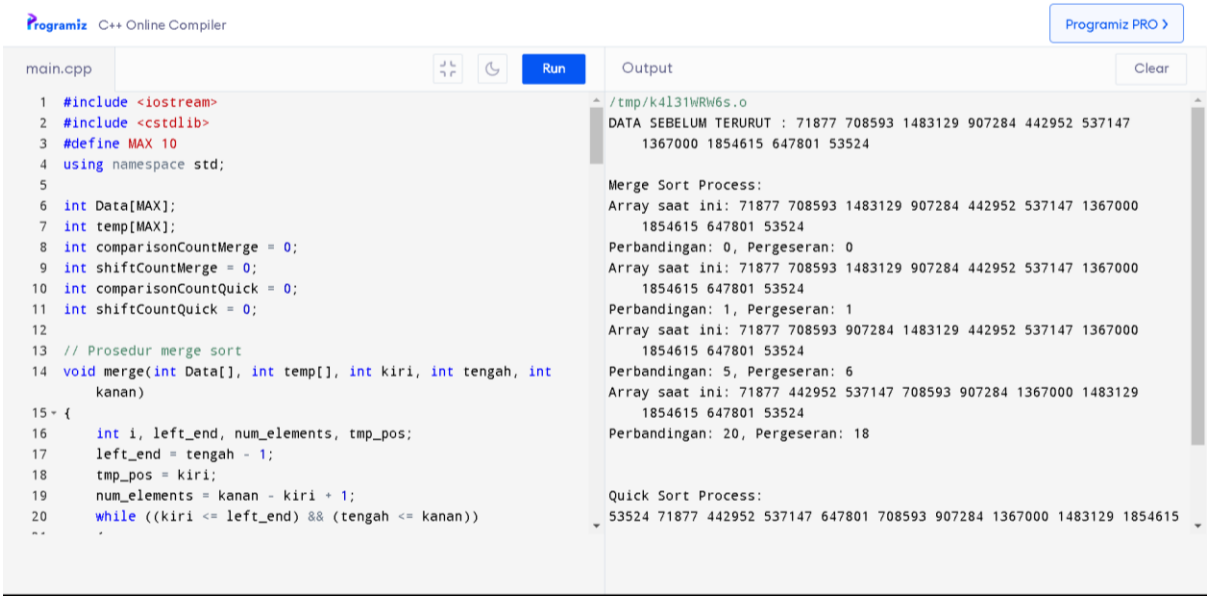
```
#include <iostream>
#include <cstdlib>
#define MAX 10
using namespace std;
```

```
int Data[MAX];
int temp[MAX];
int comparisonCountMerge = 0;
int shiftCountMerge = 0;
int comparisonCountQuick = 0;
int shiftCountQuick = 0;

// Prosedur merge sort
void merge(int Data[], int temp[], int kiri, int tengah, int kanan)
{
    int i, left_end, num_elements, tmp_pos;
    left_end = tengah - 1;
    tmp_pos = kiri;
    num_elements = kanan - kiri + 1;
    while ((kiri <= left_end) && (tengah <= kanan))
    {
        comparisonCountMerge++;
        if (Data[kiri] <= Data[tengah])
        {
            temp[tmp_pos] = Data[kiri];
            tmp_pos++;
            kiri++;
        }
        else
        {
            temp[tmp_pos] = Data[tengah];
            tmp_pos++;
            tengah++;
            shiftCountMerge++;
        }
    }
    while (kiri <= left_end)
    {
        temp[tmp_pos] = Data[kiri];
        tmp_pos++;
        kiri++;
        shiftCountMerge++;
    }
    while (tengah <= kanan)
    {
        temp[tmp_pos] = Data[tengah];
        tmp_pos++;
        tengah++;
        shiftCountMerge++;
    }
    for (i = 0; i < num_elements; i++)
    {
```

```
        Data[kanan] = temp[kanan];
        kanan--;
    }
}

// Prosedur membuat kumpulan data
void m_sort(int Data[], int temp[], int kiri, int kanan)
{
    int tengah;
    if (kanan > kiri)
    {
        tengah = (kanan + kiri) / 2;
        m_sort(Data, temp, kiri, tengah);
        m_sort(Data, temp, tengah + 1, kanan);
        merge(Data, temp, kiri, tengah + 1, kanan);
    }
}
void mergeSort(int Data[], int temp[], int array_size)
{
    m_sort(Data, temp, 0, array_size - 1);
}

// Prosedur pengurutan metode Quick Sort
void QuickSortRekursif(int L, int R)
{
    int i, j, x;
    x = Data[(L + R) / 2];
    i = L;
    j = R;
    while (i <= j) {
        comparisonCountQuick++;
        while (Data[i] < x)
        {
            i++;
            comparisonCountQuick++;
        }
        while (Data[j] > x)
        {
            j--;
            comparisonCountQuick++;
        }
        if (i <= j) {
            swap(Data[i], Data[j]);
            i++;
            j--;
            shiftCountQuick++;
        }
```

```
    }
    if (L < j)
        QuickSortRekursif(L, j);
    if (i < R)
        QuickSortRekursif(i, R);
}

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

int main()
{
    srand(42); // Seed random diatur agar hasil randomnya konsisten

    // Membangkitkan bilangan acak
    cout << "DATA SEBELUM TERURUT : ";
    for (int i = 0; i < MAX; i++)
    {
        Data[i] = rand() / 1000 + 1; // Mendapatkan bilangan acak
antara 1 dan 100
        cout << Data[i] << " ";
    }
    cout << endl;

    // Merge Sort
    cout << "\nMerge Sort Process:" << endl;
    for (int size = 1; size <= MAX; size *= 2)
    {
        mergeSort(Data, temp, size); // Panggil fungsi merge sort
        cout << "Array saat ini: ";
        displayArray(Data, MAX); // Tampilkan array saat ini
        cout << "Perbandingan: " << comparisonCountMerge << ",
Pergeseran: " << shiftCountMerge << endl; // Tampilkan jumlah
perbandingan dan pergeseran
    }
    cout << endl;

    // Quick Sort
    cout << "\nQuick Sort Process:" << endl;
    QuickSortRekursif(0, MAX - 1); // Panggil fungsi quick sort
    displayArray(Data, MAX); // Tampilkan array setelah diurutkan
```

```
    cout << "Perbandingan: " << comparisonCountQuick << ",
Pergeseran: " << shiftCountQuick << endl; // Tampilkan jumlah
perbandingan dan pergeseran

    return 0;
}
```



Output

---

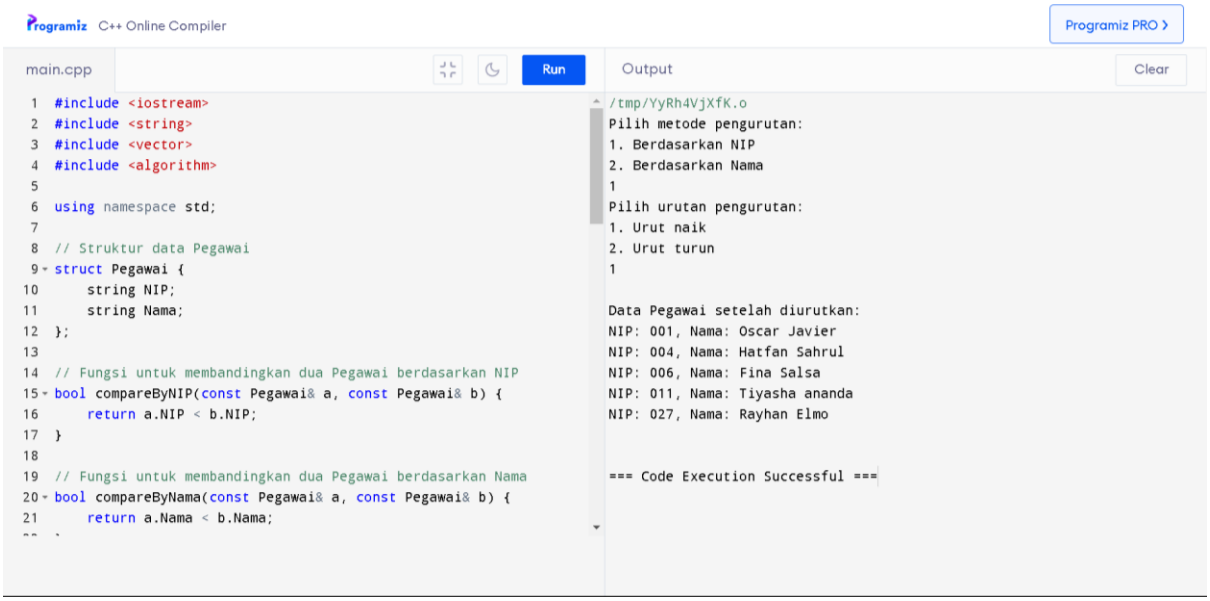
3. **Implementasikan pengurutan data pegawai pada tugas pendahuluan dengan ketentuan :**

[Input]

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

using namespace std;

// Struktur data Pegawai
struct Pegawai {
    string NIP;
    string Nama;
};

// Fungsi untuk membandingkan dua Pegawai berdasarkan NIP
```

```
bool compareByNIP(const Pegawai& a, const Pegawai& b) {
    return a.NIP < b.NIP;
}

// Fungsi untuk membandingkan dua Pegawai berdasarkan Nama
bool compareByNama(const Pegawai& a, const Pegawai& b) {
    return a.Nama < b.Nama;
}

// Fungsi untuk menampilkan data Pegawai
void displayPegawai(const vector<Pegawai>& pegawai) {
    for (const auto& p : pegawai) {
        cout << "NIP: " << p.NIP << ", Nama: " << p.Nama << endl;
    }
}

int main() {
    // Membuat data Pegawai
    vector<Pegawai> dataPegawai = {
        {"027", "Rayhan Elmo"},
        {"006", "Fina Salsa"},
        {"001", "Oscar Javier"},
        {"004", "Hatfan Sahrul"},
        {"011", "Tiyasha ananda"}
    };

    // Memilih metode pengurutan
    int choice;
    cout << "Pilih metode pengurutan:" << endl
         << "1. Berdasarkan NIP" << endl
         << "2. Berdasarkan Nama" << endl;
    cin >> choice;

    // Memilih urutan pengurutan
    int order;
    cout << "Pilih urutan pengurutan:" << endl
         << "1. Urut naik" << endl
         << "2. Urut turun" << endl;
    cin >> order;

    // Menggunakan fungsi pengurutan sesuai dengan pilihan
    if (choice == 1) {
        sort(dataPegawai.begin(), dataPegawai.end(), order == 1 ?
compareByNIP : [](const Pegawai& a, const Pegawai& b) {
            return a.NIP > b.NIP;
        });
    } else {
```

```
        sort(dataPegawai.begin(), dataPegawai.end(), order == 1 ?
compareByNama : [](const Pegawai& a, const Pegawai& b) {
            return a.Nama > b.Nama;
        });
    }

    // Menampilkan data Pegawai setelah diurutkan
    cout << "\nData Pegawai setelah diurutkan:" << endl;
    displayPegawai(dataPegawai);

    return 0;
}
```



Output

Kesimpulan

Dalam contoh-contoh di atas, kita telah memperkenalkan dan menerapkan dua algoritma pengurutan yang efektif: Quick Sort dan Merge Sort. Quick Sort memilih pivot untuk membagi array menjadi dua bagian, yang kemudian diurutkan secara rekursif. Ini dikenal karena kecepatan eksekusi yang tinggi, meskipun dapat memiliki performa buruk pada kasus tertentu. Sementara itu, Merge Sort membagi array menjadi sub-array, mengurutkan masing-masing, lalu menggabungkannya. Meskipun membutuhkan lebih banyak ruang, Merge Sort menawarkan performa yang konsisten.

Dalam implementasi praktisnya, kedua algoritma ini digunakan untuk mengurutkan data pegawai berdasarkan NIP atau nama, serta secara naik atau turun. Ini menunjukkan fleksibilitas algoritma dalam menangani berbagai kebutuhan pengurutan. Pilihan antara Quick Sort dan Merge Sort tergantung pada kebutuhan spesifik aplikasi, seperti ukuran data

dan kestabilan yang diinginkan, menekankan pentingnya memahami karakteristik dan trade-off dari setiap algoritma pengurutan.