UAS

ALGORITMA DAN PEMOGRAMAN II

Dosen pengajar :

Fajar Agung Nugroho

Disusun Oleh:

Nama : Muhammad Faiz Rabbani

Nim : 231011402539

Kelas : 03TPLP029

Link github : [BrongSkuy/Ujian-Akhir-Semester-Algo-II]

TEKNIK INFORMATIKA

FKKULTAS ILMU KOMPUTER

Jl. Puspitek Raya No 10, Serpong, Tangerang Selatan

NOMOR 1

Penjelasan Algoritma Huffman

1. **Frekuensi Karakter**:
    - Hitung frekuensi masing-masing karakter dalam string input.
2. **Pembuatan Pohon Huffman**:
    - Gunakan frekuensi karakter untuk membuat pohon Huffman. Karakter dengan frekuensi lebih rendah akan berada di tingkat yang lebih tinggi (lebih dekat ke akar).
    - Setiap karakter adalah daun, dan node internal menyimpan jumlah frekuensi anak-anaknya.
3. **Proses Encoding**:
    - Dari pohon Huffman, buat tabel encoding dengan cara traversal (DFS atau BFS) untuk menetapkan bitstream unik ke setiap karakter.
4. **Proses Decoding**:
    - Decode string terkompresi dengan traversal pohon Huffman dari akar berdasarkan bitstream.

Seourch Code

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

void profile(){
cout <<"Nama: Muhammad Faiz Rabbani"<< endl;
cout <<"NIM : 231011402539"<< endl;
cout <<"Kelas: 03TPLP029"<< endl<< endl;
```

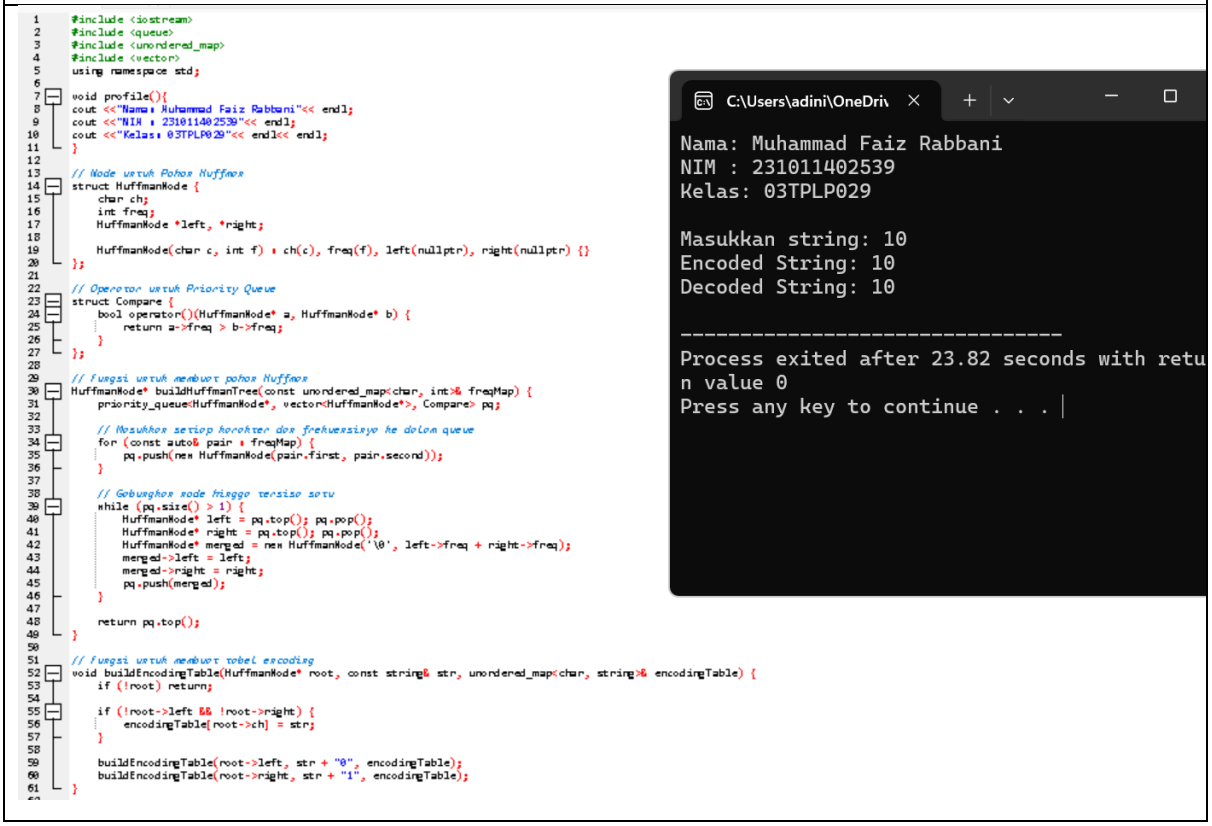
```
}

// Node untuk Pohon Huffman
struct HuffmanNode {
    char ch;
    int freq;
    HuffmanNode *left, *right;

    HuffmanNode(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}
};

// Operator untuk Priority Queue
struct Compare {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        return a->freq > b->freq;
    }
};

// Fungsi untuk membuat pohon Huffman
HuffmanNode* buildHuffmanTree(const unordered_map<char, int>& freqMap) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> pq;

    // Masukkan setiap karakter dan frekuensinya ke dalam queue
    for (const auto& pair : freqMap) {
        pq.push(new HuffmanNode(pair.first, pair.second));
    }

    // Gabungkan node hingga tersisa satu
    while (pq.size() > 1) {
        HuffmanNode* left = pq.top(); pq.pop();
        HuffmanNode* right = pq.top(); pq.pop();
        HuffmanNode* merged = new HuffmanNode('\0', left->freq + right->freq);
        merged->left = left;
```

```
        merged->right = right;
        pq.push(merged);
    }

    return pq.top();
}

// Fungsi untuk membuat tabel encoding
void buildEncodingTable(HuffmanNode* root, const string& str, unordered_map<char, string>& encodingTable) {
    if (!root) return;

    if (!root->left && !root->right) {
        encodingTable[root->ch] = str;
    }

    buildEncodingTable(root->left, str + "0", encodingTable);
    buildEncodingTable(root->right, str + "1", encodingTable);
}

// Fungsi untuk encoding string
string encode(const string& text, const unordered_map<char, string>& encodingTable) {
    string encodedStr;
    for (char ch : text) {
        encodedStr += encodingTable.at(ch);
    }
    return encodedStr;
}

// Fungsi untuk decoding string
string decode(const string& encodedStr, HuffmanNode* root) {
    string decodedStr;
    HuffmanNode* current = root;
```

```
    for (char bit : encodedStr) {
        current = (bit == '0') ? current->left : current->right;

        if (!current->left && !current->right) {
            decodedStr += current->ch;
            current = root;
        }
    }

    return decodedStr;
}

int main() {
            profile();
    string text;
    cout << "Masukkan string: ";
    cin >> text;

    // Hitung frekuensi karakter
    unordered_map<char, int> freqMap;
    for (char ch : text) {
        freqMap[ch]++;
    }

    // Bangun pohon Huffman
    HuffmanNode* root = buildHuffmanTree(freqMap);

    // Buat tabel encoding
    unordered_map<char, string> encodingTable;
    buildEncodingTable(root, "", encodingTable);

    // Encoding string
```

```
    string encodedStr = encode(text, encodingTable);
    cout << "Encoded String: " << encodedStr << endl;

    // Decoding string
    string decodedStr = decode(encodedStr, root);
    cout << "Decoded String: " << decodedStr << endl;

    return 0;
}
```

Screenshot



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

void profile(){
cout <<"Nama: Muhammad Faiz Rabbani"<< endl;
cout <<"NIM : 231011402539"<< endl;
cout <<"Kelas: 03TPLP029"<< endl<< endl;
}

// Node untuk Pohon Huffman
struct HuffmanNode {
    char ch;
    int freq;
    HuffmanNode *left, *right;

    HuffmanNode(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}
};

// Operator untuk Priority Queue
struct Compare {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        return a->freq > b->freq;
    }
};

// Fungsi untuk membuat pohon Huffman
HuffmanNode* buildHuffmanTree(const unordered_map<char, int>& freqMap) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> pq;

    // Masukkan setiap karakter dan frekuensinya ke dalam queue
    for (const auto& pair : freqMap) {
        pq.push(new HuffmanNode(pair.first, pair.second));
    }

    // Gabungkan node hingga tersisa satu
    while (pq.size() > 1) {
        HuffmanNode* left = pq.top(); pq.pop();
        HuffmanNode* right = pq.top(); pq.pop();
        HuffmanNode* merged = new HuffmanNode('\0', left->freq + right->freq);
        merged->left = left;
        merged->right = right;
        pq.push(merged);
    }

    return pq.top();
}

// Fungsi untuk membuat tabel encoding
void buildEncodingTable(HuffmanNode* root, const string& str, unordered_map<char, string>& encodingTable) {
    if (!root) return;

    if (!root->left && !root->right) {
        encodingTable[root->ch] = str;
    }

    buildEncodingTable(root->left, str + "0", encodingTable);
    buildEncodingTable(root->right, str + "1", encodingTable);
}
```

```
C:\Users\adini\OneDriv
Nama: Muhammad Faiz Rabbani
NIM : 231011402539
Kelas: 03TPLP029

Masukkan string: 10
Encoded String: 10
Decoded String: 10

--------------------------------
Process exited after 23.82 seconds with retu
n value 0
Press any key to continue . . .
```

NOMOR2

Algoritma

1. Simpan semua elemen dari array pertama (arr1) dalam struktur data hash (seperti unordered_set) agar pencarian menjadi O(1)O(1)O(1).
2. Iterasikan setiap elemen dalam array kedua (arr2).

- Untuk setiap elemen yyy dari arr2, hitung nilai yang diperlukan dari arr1 sebagai K−yK - yK−y.
- Periksa apakah K−yK - yK−y ada dalam hash set. Jika ada, tambahkan pasangan tersebut ke dalam daftar hasil.

3. Kembalikan daftar pasangan hasil.

Kompleksitas

- **Waktu**:
  - Membuat hash dari elemen arr1: O(n)O(n)O(n), dengan nnn adalah panjang arr1.
  - Iterasi elemen arr2 dan pencarian dalam hash: O(m)O(m)O(m), dengan mmm adalah panjang arr2.
  - Total: O(n+m)O(n + m)O(n+m).
- **Ruang**:
  - Ruang untuk hash: O(n)O(n)O(n).
  - Total ruang: O(n)O(n)O(n).

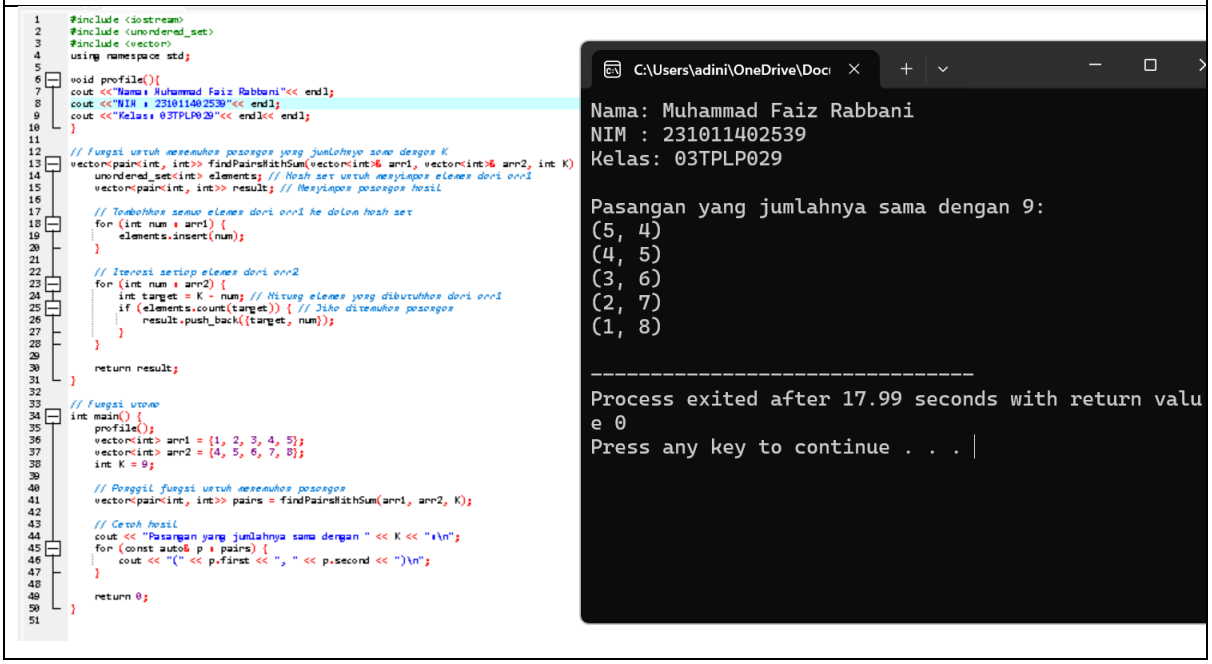
Seourch Code

#include <iostream>
#include <unordered_set>
#include <vector>
using namespace std;
 	
void profile(){
cout <<"Nama: Muhammad Faiz Rabbani"<< endl;
cout <<"NIM : 231011402539"<< endl;
cout <<"Kelas: 03TPLP029"<< endl<< endl;
}
 	
// Fungsi untuk menemukan pasangan yang jumlahnya sama dengan K

```
vector<pair<int, int>> findPairsWithSum(vector<int>& arr1, vector<int>& arr2, int K) {
    unordered_set<int> elements; // Hash set untuk menyimpan elemen dari arr1
    vector<pair<int, int>> result; // Menyimpan pasangan hasil

    // Tambahkan semua elemen dari arr1 ke dalam hash set
    for (int num : arr1) {
        elements.insert(num);
    }

    // Iterasi setiap elemen dari arr2
    for (int num : arr2) {
        int target = K - num; // Hitung elemen yang dibutuhkan dari arr1
        if (elements.count(target)) { // Jika ditemukan pasangan
            result.push_back({target, num});
        }
    }

    return result;
}

// Fungsi utama
int main() {
            profile();
    vector<int> arr1 = {1, 2, 3, 4, 5};
    vector<int> arr2 = {4, 5, 6, 7, 8};
    int K = 9;

    // Panggil fungsi untuk menemukan pasangan
    vector<pair<int, int>> pairs = findPairsWithSum(arr1, arr2, K);

    // Cetak hasil
    cout << "Pasangan yang jumlahnya sama dengan " << K << ":\n";
    for (const auto& p : pairs) {
```

```
        cout << "(" << p.first << ", " << p.second << ")\n";
    }


    return 0;
}
```

Screen Shot



NOMOR 3

Berikut adalah implementasi algoritma Quick Sort dalam gaya fungsional menggunakan bahasa C++. Pendekatan ini menghindari loop eksplisit dan mutasi data dengan memanfaatkan rekursi dan fungsi-fungsi berbasis koleksi, seperti pemfilteran (filter) dan penggabungan daftar.

Implementasi Quick Sort Fungsional di C++

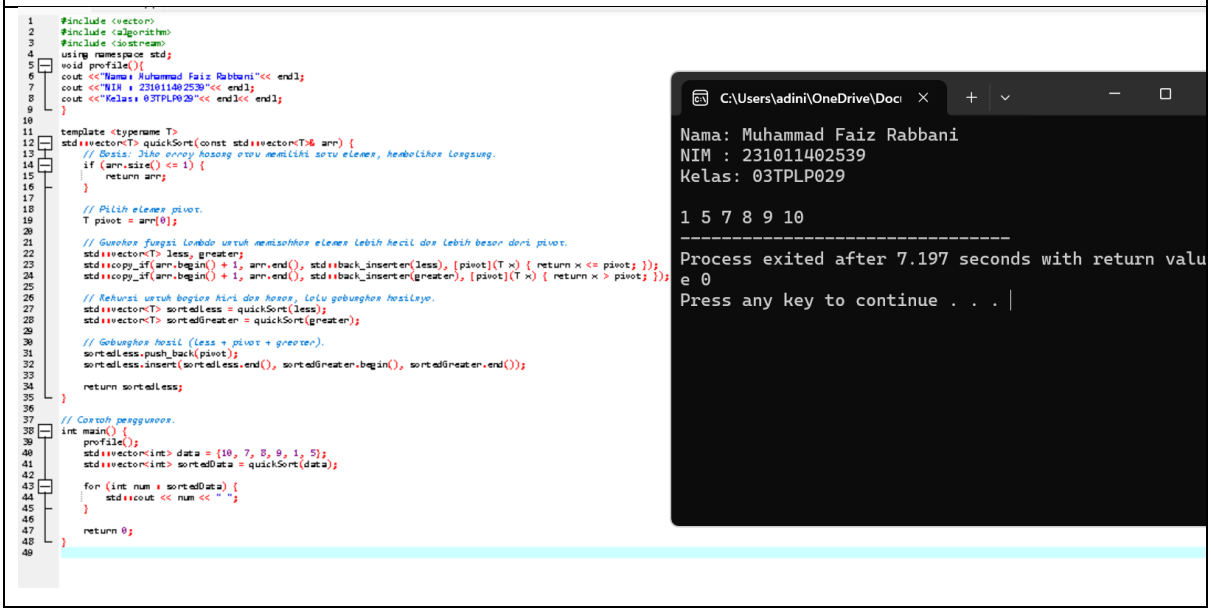
Seourch Code

```
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;
void profile(){
cout <<"Nama: Muhammad Faiz Rabbani"<< endl;
```

```
cout <<"NIM : 231011402539"<< endl;
cout <<"Kelas: 03TPLP029"<< endl<< endl;
}

template <typename T>
std::vector<T> quickSort(const std::vector<T>& arr) {
    // Basis: Jika array kosong atau memiliki satu elemen, kembalikan langsung.
    if (arr.size() <= 1) {
        return arr;
    }

    // Pilih elemen pivot.
    T pivot = arr[0];

    // Gunakan fungsi lambda untuk memisahkan elemen lebih kecil dan lebih besar dari pivot.
    std::vector<T> less, greater;
    std::copy_if(arr.begin() + 1, arr.end(), std::back_inserter(less), [pivot](T x) { return x <=
pivot; });
    std::copy_if(arr.begin() + 1, arr.end(), std::back_inserter(greater), [pivot](T x) { return x
> pivot; });

    // Rekursi untuk bagian kiri dan kanan, lalu gabungkan hasilnya.
    std::vector<T> sortedLess = quickSort(less);
    std::vector<T> sortedGreater = quickSort(greater);

    // Gabungkan hasil (less + pivot + greater).
    sortedLess.push_back(pivot);
    sortedLess.insert(sortedLess.end(), sortedGreater.begin(), sortedGreater.end());

    return sortedLess;
}

// Contoh penggunaan.
```

```
int main() {
        profile();
    std::vector<int> data = {10, 7, 8, 9, 1, 5};
    std::vector<int> sortedData = quickSort(data);

    for (int num : sortedData) {
        std::cout << num << " ";
    }

    return 0;
}
```

Screenshot



Penjelasan Implementasi

1. **Rekursi**: Fungsi quickSort memanggil dirinya sendiri untuk mengurutkan bagian yang lebih kecil dari pivot dan bagian yang lebih besar. Hal ini menggantikan loop eksplisit.
2. **Pemfilteran**: Menggunakan std::copy_if untuk memisahkan elemen berdasarkan pivot.
3. **Immutable Data**: Alih-alih memodifikasi array input, algoritma ini membuat array baru untuk setiap langkah.

Kinerja Versi Fungsional vs Imperatif

1. **Kompleksitas Teoretis**: Kompleksitas tetap O(nlog⁡n)O(n \log n)O(nlogn) dalam kasus rata-rata, dan O(n2)O(n^2)O(n2) dalam kasus terburuk.
2. **Overhead Memory**: Versi fungsional cenderung menggunakan lebih banyak memori karena membuat salinan baru dari array untuk setiap langkah rekursi. Dalam versi imperatif, array diubah di tempat, menghemat memori.
3. **Efisiensi**:
    - **Imperatif**: Lebih cepat secara umum karena mutasi langsung dan overhead memori lebih kecil.
    - **Fungsional**: Lebih elegan dan lebih mudah dibaca untuk dataset kecil, tetapi dapat menjadi lambat untuk dataset besar karena alokasi memori berulang.
4. **Stack Overflow**: Karena rekursi, implementasi fungsional lebih rentan terhadap stack overflow jika dataset sangat besar, dibandingkan versi imperatif yang dapat dioptimalkan dengan loop eksplisit.

Versi fungsional sangat cocok untuk pendekatan yang mementingkan keterbacaan kode dan menekankan konsep immutability, tetapi kurang efisien dibandingkan versi imperatif untuk kebutuhan performa tinggi.

NOMOR 4

Analisis Kompleksitas

1. Radix Sort

- **Kompleksitas Waktu**:
    - Untuk nnn elemen dan ddd digit, kompleksitas waktu adalah O(d⋅(n+k))O(d \cdot (n + k))O(d⋅(n+k)), di mana kkk adalah jumlah nilai dalam setiap digit (misalnya, 10 untuk angka desimal).
    - Jika ddd dianggap konstan, maka kompleksitasnya mendekati O(n)O(n)O(n).
- **Kompleksitas Ruang**:
    - Radix Sort menggunakan memori tambahan untuk array sementara, sehingga kompleksitas ruang adalah O(n+k)O(n + k)O(n+k).

2. Quick Sort

- **Kompleksitas Waktu**:
  - Kasus rata-rata: O(nlog⁡n)O(n \log n)O(nlogn).
  - Kasus terburuk: O(n2)O(n^2)O(n2), jika partisi tidak seimbang (dapat dioptimalkan dengan memilih pivot secara acak atau median).
- **Kompleksitas Ruang**:
  - O(log⁡n)O(\log n)O(logn) untuk stack rekursi.

3. Merge Sort

- **Kompleksitas Waktu**:
  - Selalu O(nlog⁡n)O(n \log n)O(nlogn), karena membagi dan menggabungkan array secara teratur.
- **Kompleksitas Ruang**:
  - O(n)O(n)O(n), karena membutuhkan array tambahan untuk penggabungan.

Algoritma	Kompleksitas Waktu	Kompleksitas Ruang	Keunggulan
Radix Sort	$O(d \cdot (n + k))$	$O(n + k)$	Cepat untuk data numerik besar dengan panjang digit kecil.
Quick Sort	$O(n \log n)$ (rata-rata), $O(n^2)$ (terburuk)	$O(\log n)$	Cepat untuk array kecil hingga sedang.
Merge Sort	$O(n \log n)$	$O(n)$	Stabil, cocok untuk dataset besar.

Kondisi Radix Sort Unggul

1. Data berupa angka dengan panjang digit kecil.
2. Dataset besar yang tidak memerlukan stabilitas dalam perbandingan kunci.
3. Ketika kompleksitas linear (O(n)O(n)O(n)) lebih diinginkan dibandingkan logaritmik (O(nlog⁡n)O(n \log n)O(nlogn)).

Sourcecode
#include <iostream>

```
#include <vector>
#include <algorithm>
#include <cmath>
using namespace std;
void profile(){
cout <<"Nama: Muhammad Faiz Rabbani"<< endl;
cout <<"NIM : 231011402539"<< endl;
cout <<"Kelas: 03TPLP029"<< endl<< endl;
}


// Radix Sort
void countingSort(std::vector<int>& arr, int exp) {
    int n = arr.size();
    std::vector<int> output(n);
    int count[10] = {0};

    // Hitung frekuensi digit
    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    // Akumulasi count
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // Bangun array output
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
```

```
    // Salin output ke array asli
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

void radixSort(std::vector<int>& arr) {
    int maxVal = *std::max_element(arr.begin(), arr.end());
    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        countingSort(arr, exp);
    }
}

// Quick Sort
template <typename T>
std::vector<T> quickSort(const std::vector<T>& arr) {
    if (arr.size() <= 1) return arr;

    T pivot = arr[0];
    std::vector<T> less, greater;

    std::copy_if(arr.begin() + 1, arr.end(), std::back_inserter(less), [pivot](T x) { return x <=
pivot; });
    std::copy_if(arr.begin() + 1, arr.end(), std::back_inserter(greater), [pivot](T x) { return x
> pivot; });

    auto sortedLess = quickSort(less);
    auto sortedGreater = quickSort(greater);

    sortedLess.push_back(pivot);
    sortedLess.insert(sortedLess.end(), sortedGreater.begin(), sortedGreater.end());

    return sortedLess;
```

```
}

// Merge Sort
template <typename T>
void merge(std::vector<T>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    std::vector<T> L(n1), R(n2);
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;
    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++];
}

template <typename T>
void mergeSort(std::vector<T>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
```

```
}

// Main Function
int main() {
        profile();
   std::vector<int> data = {170, 45, 75, 90, 802, 24, 2, 66};

   std::vector<int> data1 = data;
   std::vector<int> data2 = data;
   std::vector<int> data3 = data;

   // Radix Sort
   radixSort(data1);
   std::cout << "Radix Sort: ";
   for (int num : data1) std::cout << num << " ";
   std::cout << "\n";

   // Quick Sort
   data2 = quickSort(data2);
   std::cout << "Quick Sort: ";
   for (int num : data2) std::cout << num << " ";
   std::cout << "\n";

   // Merge Sort
   mergeSort(data3, 0, data3.size() - 1);
   std::cout << "Merge Sort: ";
   for (int num : data3) std::cout << num << " ";
   std::cout << "\n";

   return 0;
}
```

Screenshot

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
using namespace std;
void profile(){
cout <<"Nama: Muhammad Faiz Rabbani"<< endl;
cout <<"NIM : 231011402539"<< endl;
cout <<"Kelas: 03TPLP029"<< endl<< endl;
}

// Radix Sort
void countingSort(std::vector<int>& arr, int exp) {
    int n = arr.size();
    std::vector<int> output(n);
    int count[10] = {0};

    // Hitung frekuensi digit
    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    // Akumulasi count
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // Bangun array output
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Salin output ke array asli
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

void radixSort(std::vector<int>& arr) {
    int maxVal = *std::max_element(arr.begin(), arr.end());
    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        countingSort(arr, exp);
    }
}

// Quick Sort
template <typename T>
std::vector<T> quickSort(const std::vector<T>& arr) {
    if (arr.size() <= 1) return arr;

    T pivot = arr[0];
    std::vector<T> less, greater;

    std::copy_if(arr.begin() + 1, arr.end(), std::back_inserter(less), [pivot](T x) { return x <= pivot; });
    std::copy_if(arr.begin() + 1, arr.end(), std::back_inserter(greater), [pivot](T x) { return x > pivot; });

    auto sortedLess = quickSort(less);
    auto sortedGreater = quickSort(greater);

    sortedLess.push_back(pivot);
    sortedLess.insert(sortedLess.end(), sortedGreater.begin(), sortedGreater.end());

    return sortedLess;
}
```

```
C:\Users\adini\OneDrive\Doc

Nama: Muhammad Faiz Rabbani
NIM : 231011402539
Kelas: 03TPLP029

Radix Sort: 2 24 45 66 75 90 170 802
Quick Sort: 2 24 45 66 75 90 170 802
Merge Sort: 2 24 45 66 75 90 170 802

--------------------------------
Process exited after 16.69 seconds with return value 0
Press any key to continue . . .
```

```cpp
// Merge Sort
template <typename T>
void merge(std::vector<T>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    std::vector<T> L(n1), R(n2);
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];

    int i = 0, j = 0, k = left;
    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++];
}

template <typename T>
void mergeSort(std::vector<T>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

// Main Function
int main() {
    profile();
    std::vector<int> data = {170, 45, 75, 90, 802, 24, 2, 66};

    std::vector<int> data1 = data;
    std::vector<int> data2 = data;
    std::vector<int> data3 = data;

    // Radix Sort
    radixSort(data1);
    std::cout << "Radix Sort: ";
    for (int num : data1) std::cout << num << " ";
    std::cout << "\n";

    // Quick Sort
    data2 = quickSort(data2);
    std::cout << "Quick Sort: ";
    for (int num : data2) std::cout << num << " ";
    std::cout << "\n";

    // Merge Sort
    mergeSort(data3, 0, data3.size() - 1);
    std::cout << "Merge Sort: ";
    for (int num : data3) std::cout << num << " ";
    std::cout << "\n";

    return 0;
}
```

Penjelasan

1. **Radix Sort** menggunakan pengurutan berdasarkan digit dan array tambahan untuk setiap langkah digit.
2. **Quick Sort** diimplementasikan dalam gaya fungsional menggunakan rekursi.
3. **Merge Sort** membagi array menjadi dua bagian, mengurutkan secara rekursif, dan menggabungkannya kembali.

Kesimpulan

- **Radix Sort unggul** untuk dataset besar dengan elemen numerik yang panjang digitnya kecil.
- **Quick Sort unggul** untuk array kecil hingga sedang karena overhead rendah.
- **Merge Sort stabil** untuk array besar tetapi memiliki overhead memori yang lebih tinggi.

NOMOR 5

Prinsip Rekursif dalam Pembentukan Fractal

Fractal adalah objek yang memiliki struktur yang mirip pada berbagai skala, yakni **self-similarity**. Dalam hal ini, kita menggambar pola yang serupa pada tingkat yang lebih kecil. Prinsip rekursif berarti kita mendefinisikan cara menggambar objek dengan cara memanggil kembali proses tersebut pada bagian yang lebih kecil.

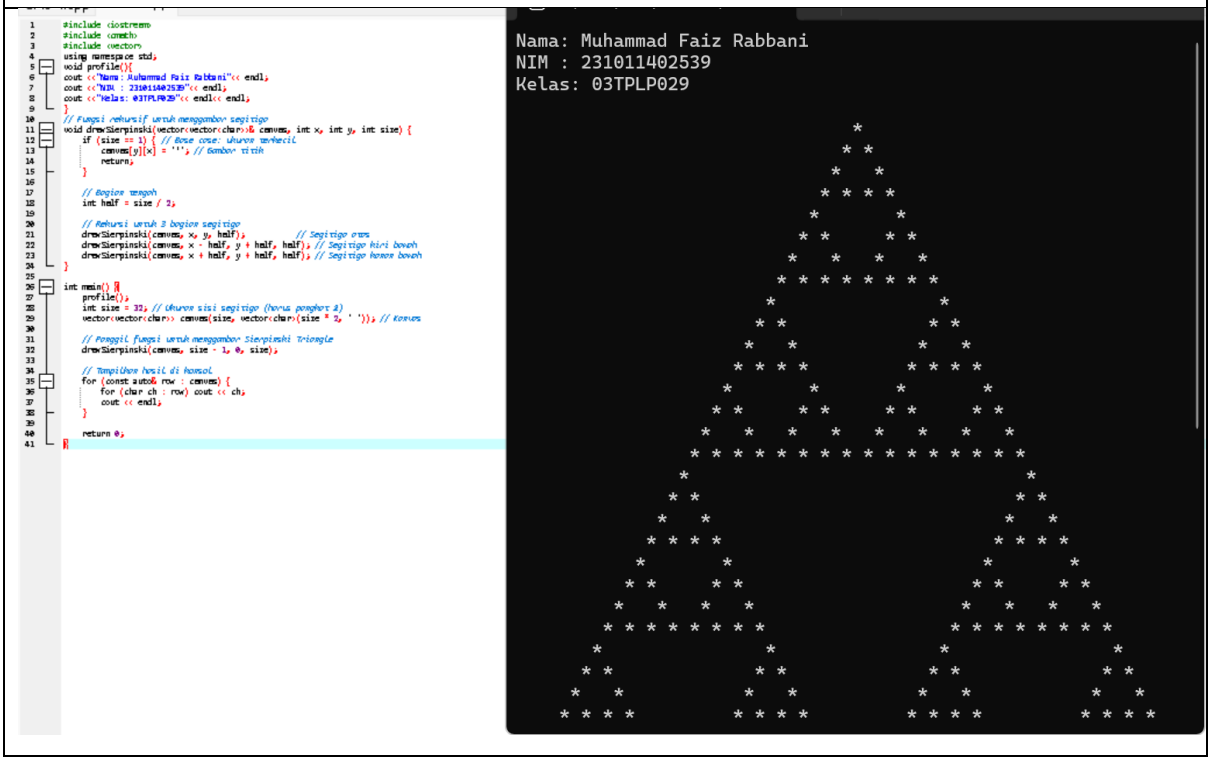
Pada **Sierpinski Triangle**, misalnya, kita mulai dengan menggambar segitiga besar. Kemudian, kita menggambar tiga segitiga kecil di dalamnya, dan untuk setiap segitiga kecil, kita ulangi proses tersebut. Proses ini berlanjut hingga mencapai kedalaman tertentu.

Seourcecode
#include <iostream> #include <cmath> #include <vector>

```
using namespace std;
void profile(){
cout <<"Nama: Muhammad Faiz Rabbani"<< endl;
cout <<"NIM : 231011402539"<< endl;
cout <<"Kelas: 03TPLP029"<< endl<< endl;
}
// Fungsi rekursif untuk menggambar segitiga
void drawSierpinski(vector<vector<char>>& canvas, int x, int y, int size) {
    if (size == 1) { // Base case: ukuran terkecil
        canvas[y][x] = '*'; // Gambar titik
        return;
    }

    // Bagian tengah
    int half = size / 2;

    // Rekursi untuk 3 bagian segitiga
    drawSierpinski(canvas, x, y, half);           // Segitiga atas
    drawSierpinski(canvas, x - half, y + half, half); // Segitiga kiri bawah
    drawSierpinski(canvas, x + half, y + half, half); // Segitiga kanan bawah
}

int main() {
            profile();
    int size = 32; // Ukuran sisi segitiga (harus pangkat 2)
    vector<vector<char>> canvas(size, vector<char>(size * 2, ' ')); // Kanvas

    // Panggil fungsi untuk menggambar Sierpinski Triangle
    drawSierpinski(canvas, size - 1, 0, size);

    // Tampilkan hasil di konsol
    for (const auto& row : canvas) {
        for (char ch : row) cout << ch;
```

```
    cout << endl;
  }

  return 0;
}
```

Screenchot



Penjelasan Kode:

1. **Fungsi profile**:
    - Fungsi ini hanya untuk menampilkan informasi profil pengguna: nama, NIM, dan kelas.
2. **Fungsi drawSierpinski**:
    - Fungsi rekursif ini menggambar segitiga Sierpinski pada kanvas yang diberikan.
    - **Parameter:**
        - canvas: Sebuah vektor dua dimensi (matriks) yang menyimpan karakter-karakter untuk menggambar segitiga.

- x, y: Koordinat titik yang menandakan posisi awal untuk menggambar segitiga.
- size: Ukuran sisi segitiga yang sedang digambar.

- **Algoritma:**
  - Jika ukuran segitiga mencapai 1 (size == 1), fungsi ini menggambar titik di posisi (x, y) pada kanvas.
  - Jika ukuran segitiga lebih besar dari 1, fungsi ini membagi segitiga menjadi 3 bagian yang lebih kecil:
    1. Segitiga bagian atas (dengan posisi awal (x, y) dan ukuran setengah).
    2. Segitiga bagian kiri bawah (dengan posisi (x - half, y + half) dan ukuran setengah).
    3. Segitiga bagian kanan bawah (dengan posisi (x + half, y + half) dan ukuran setengah).
  - Fungsi akan dipanggil secara rekursif untuk menggambar setiap bagian segitiga.

3. **Fungsi main**:
   - Memanggil fungsi profile untuk menampilkan informasi pengguna.
   - Mendeklarasikan ukuran segitiga (size) yang harus berupa angka pangkat dua (misalnya 32).
   - Membuat kanvas berbentuk matriks dengan ukuran size x size * 2 untuk menampung segitiga.
   - Memanggil fungsi drawSierpinski untuk menggambar segitiga pada kanvas.
   - Menampilkan kanvas yang berisi gambar segitiga Sierpinski pada layar konsol.

Algoritma:

1. Tampilkan profil pengguna.
2. Tentukan ukuran segitiga, misalnya 32.

3. Buat kanvas berbentuk matriks berukuran size x size * 2.
4. Panggil fungsi drawSierpinski untuk menggambar segitiga Sierpinski pada kanvas.
    - Jika ukuran segitiga adalah 1, gambar titik pada posisi (x, y).
    - Jika ukuran segitiga lebih besar, bagi segitiga menjadi 3 bagian dan panggil rekursi untuk setiap bagian.
5. Tampilkan kanvas yang berisi segitiga Sierpinski di konsol.