**PROJEK UJIAN AKHIR SEMESTER**

**ALGORITMA DAN PEMROGRAMAN DASAR**



**Tim Penyusun :**

**GILBERTH VITTHO - 422024003  
WILLIAMS - 422024010  
STEVEN CREDENTIA IVANEMAGA ZEGA - 422024020**

**PROGRAM STUDI SISTEM INFORMASI**

**FAKULTAS TEKNIK DAN ILMU KOMPUTER**

**UNIVERSITAS KRISTEN KRIDA WACANA**

**JAKARTA**

**2024**

**Insertion Sort Algorithm**

1. Insertion sort adalah algoritma yang mengurutkan dengan cara menyisipkan elemen satuper satu ke posisi yang benar di dalam daftar yang sudah terurut. Dimulai dari elemen kedua, elemen tersebut dibandingkan dengan elemen sebelumnya dan disisipkan pada tempat yang tepat. Proses ini diulang sampai seluruh elemen terurut.

**Aplikasi :**

* **Penyusunan** data aplikasi spreadsheet.
* **Penyusunan** item dalam toko online.
* **Pengurutan** skor real-time dalam game.
* **Sorting** kartu dalam permainan.

1. **Langkah**-**Langkah**:

* **Inisialisasi Loop:**

Periksa apakah i < len(arr). Jika ya, lanjut ke langkah berikutnya. Jika tidak, keluar dari loop.

* **Simpan** elemen arr[i] ke variabel key.
* **Tetapkan** j = i – 1 dan periksa apakah j >= 0 **dan** arr[j] > key.
* **Ulangi** hingga salah satu kondisi tidak terpenuhi.
* **Letakkan** key di posisi yang benar: arr[j + 1] = key.
* Jika **(i >= len(arr)),** maka algoritma selesai dan array diurutkan.

Mulai

Inisialisasi Loop

Simpan Key

Inisialisasi Loop

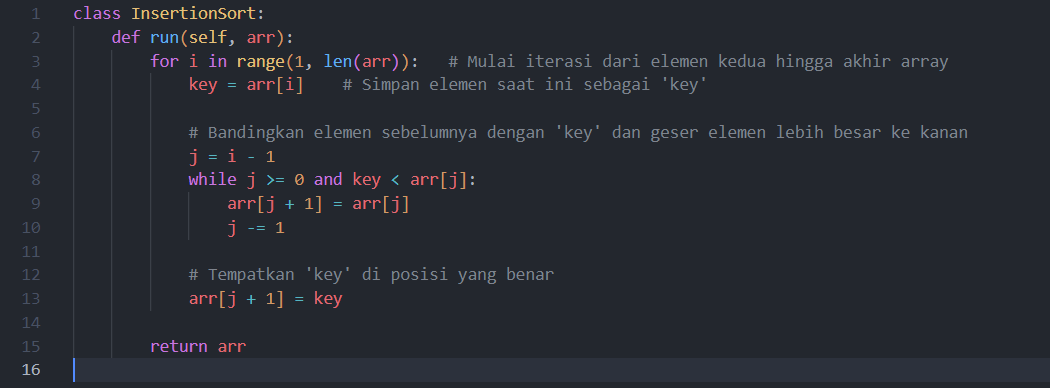
Tempatkan Key

Selesai

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space Complexity** |
| **Best** | **Average** | **Worst** |
| Insertion Sort | O(n) | O(n2) | O(n2) | O(1) |

1. **Implementasi Python**



**Quick Sort Algorithm**

1. Quick sort adalah algoritma yang menggunakan teknik **"**divide and conquer**"** yaitu dengan memilih satu elemen sebagai pivot dan memisahkan elemen-elemen lainnya ke kelompok yang lebih kecil dari pivot dan kelompok yang lebih besar. Kelompok tersebut diurutkan secara rekursif.

**Aplikasi** :

* **Pencarian** dalam basis data besar
* **Penyusunan** **data** dalam aplikasi
* **Pengurutan** nama atau ID dalam sistem HR.

1. **Langkah-Langkah**:

* **Periksa panjang** array, jika len(arr) <= 1 berarti array sudah terurut.
* **Pilih elemen pertama** dari array sebagai pivot.
* **Buat dua sub-array**, *less* dan *greater*.
* **Panggil *QuickSort*** secara rekursif untuk less dan greater.
* **Gabungkan** hasil self.run(less), [pivot], dan self.run(greater) menjadi satu array yang terurut.
* **Kembalikan** array.

Mulai

Periksa Array

Pilih Pivot

Rekursi

Pisahkan Elemen

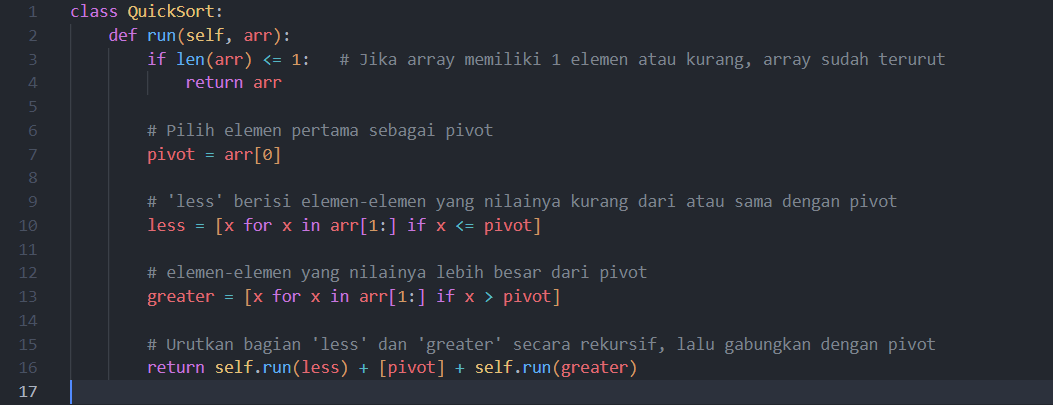
Gabungkan Hasil

Selesai

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space Complexity** |
| **Best** | **Average** | **Worst** |
| Quick Sort | O(nlogn) | O(nlogn) | O(n2) | O(logn) |

1. **Implementasi Python**



**Merge Sort Algorithm**

1. Merge sort menggunakan prinsip **"**divide and conquer**"**. Algoritma ini membagi daftar menjadi dua bagian, mengurutkan masing-masing bagian, kemudian menggabungkan dua bagian yang sudah terurut menjadi satu daftar yang terurut.

**Aplikasi :**

* **Pengolahan file** besar atau data dalam penyimpanan terdistribusi.
* **Pengolahan** data statistik.
* **Pengurutan** data besar yang tidak dapat dimuat sepenuhnya ke dalam memori.

1. **Langkah-Langkah:**

* **Periksa panjang array**, jika len(arr) <= 1 berarti array sudah terurut.
* **Tentukan titik tengah** dan pisahkan array menjadi *left* dan *right.*
* **Urutkan left dan right** secara rekursif dengan memanggil MergeSort**.**
* **Panggil fungsi merge**(left, right) untuk menggabungkan kedua array yang sudah terurut.
* **Kembalikan** hasil.

Mulai

Panjang Array

Bagi Array

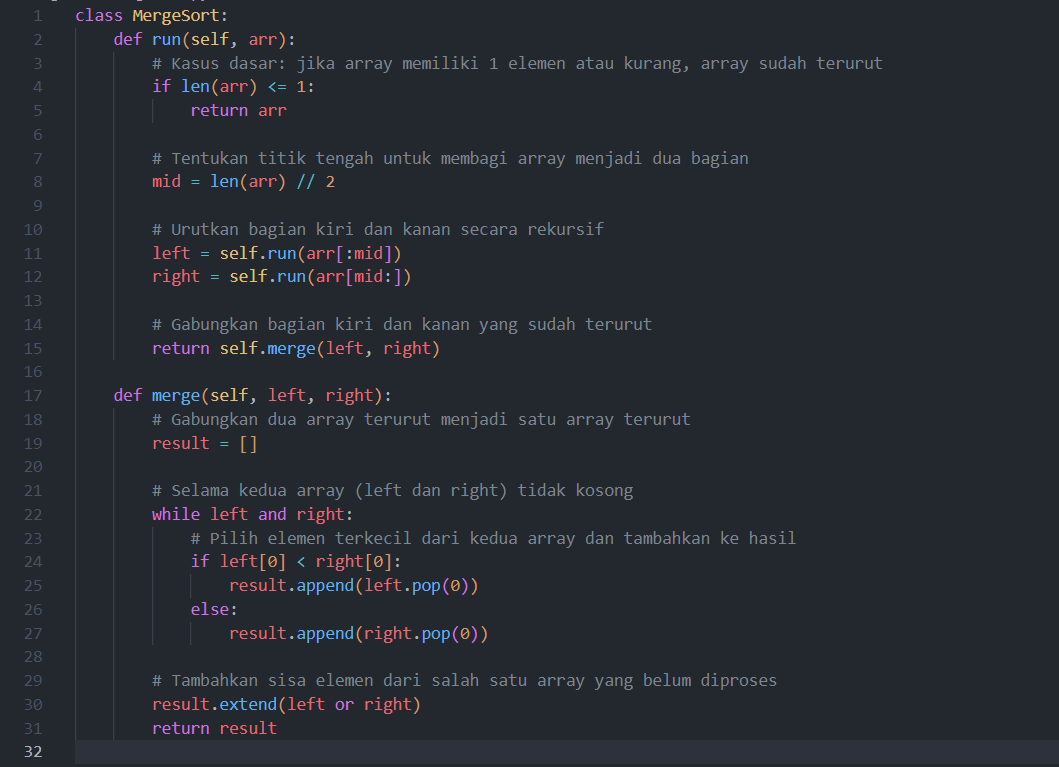
Gabungkan Bagian

Urutkan dengan Rekursi

Selesai

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space Complexity** |
| **Best** | **Average** | **Worst** |
| Merge Sort | O(nlogn) | O(nlogn) | O(nlogn) | O(n) |

1. **Implementasi Python**

**Heap Sort Algorithm**

1. Heap Sort Algorithm menggunakan struktur data heap untuk mengurutkan elemen. Heap adalah pohon biner khusus di mana setiap elemen lebih besar (untuk heap maksimum) atau lebih kecil (untuk heap minimum) daripada elemen anaknya. Dalam heap sort, dibuat heap dari daftar yang belum terurut, lalu elemen terbesar dipindahkan ke akhir daftar dan heap disesuaikan ulang. Proses ini diulang sampai semua elemen terurut.

**Aplikasi** :

* **Algoritma** dalam game.
* **Penyusunan** **prioritas** dalam sistem manajemen antrian.
* **Pengurutan** dalam event manegement berdasarkan waktu mulai/berakhir.
* **Pengurutan** data kompetisi atau game berbasis skor dengan efisien.

1. **Langkah-Langkah:**

* **Ubah** array menjadi Heap dengan dungsi fungsi heapq.heapify(arr).
* **Ambil** elemen terkecil dari heap lalu tambahkan ke aray hasil.
* **Kembalikan** array hasil yang berisi elemen terurut.

Mulai

Ubah Array

Ekstraksi Elemen

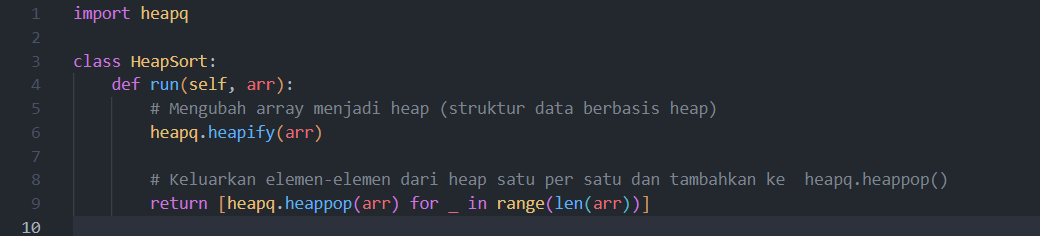
Kembalikan Array

Selesai

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space Complexity** |
| **Best** | **Average** | **Worst** |
| Heap Sort | O(nlogn) | O(nlogn) | O(nlogn) | O(1) |

1. **Implementasi Python**



**Selection Sort Algorithm**

1. Selection Sort Algorithm adalah algoritma yang mengurutkan dengan cara memilih elementerkecil (atau terbesar) dari daftar dan menukarnya dengan elemen pertama. Setelah itu, elemen terkecil berikutnya dipilih dan ditukar dengan elemen kedua, dan seterusnya sampai seluruh daftar terurut.

**Aplikasi** :

* **Sistem pemesanan** sederhana di restoran.
* **Mengurutkan** nilai siswa dalam daftar kecil secara manual.
* **Mengurutkan** array kecil dalam sistem embedded atau IoT.

1. **Langkah-Langkah**:

* **Mulai iterasi** dari i = 0 hingga i < len(arr). Jika i >= len(arr), maka algoritma selesai.
* **Tetapkan** min\_idx = I sebagai indeks minimum.
* **Cari** elemen terkecil.
* **Tukar** elemen pada indeks i dengan elemen pada min\_idx.
* **Tambahkan** I dan kembalikan array.

Indeks Minimum

Inisialisasi Loop

Mulai

Elemen Terkecil

Tukar Elemen

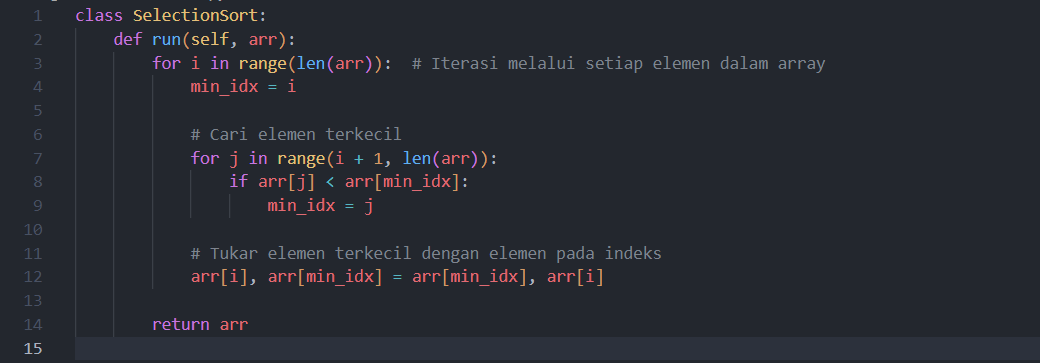
Tingkatkan Indeks

Selesai

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space Complexity** |
| **Best** | **Average** | **Worst** |
| Selection Sort | O(n2) | O(n2) | O(n2) | O(1) |

1. **Implementasi Python**



**Counting Sort Algorithm**

1. Counting Sort Algorithm adalah algoritma pengurutan yang bekerja dengan cara menghitung frekuensi kemunculan setiap elemen dalam daftar. Kemudian, elemen-elemen disusun berdasarkan frekuensinya. Algoritma ini sangat cepat jika rentang nilai elemen tidak terlalu besar dan nilai-nilai elemen dalam daftar adalah angka bulat.

**Aplikasi** :

* **Pengolahan data numerik** dalam aplikasi statistik atau pengolahan gambar.
* **Mengurutkan** nilai piksel untuk memproses gambar.
* **Mengurutkan** data kategori, seperti usia, kota, atau kelas sosial.

1. **Langkah-Langkah**:

* **Cari nilai maksimum** dalam array menggunakan max(arr) dan simpan ke max\_val.
* **Buat** array count dengan ukuran max\_val + 1 yang diisi dengan nilai 0.
* **Tambahkan** 1 pada indeks num di array count (count[num] += 1).
* **Buat** array kosong sorted\_arr dan Iterasi setiap indeks i dan nilainya c dalam count.
* **Tambahkan** elemen i sebanyak c kali ke sorted\_arr.

Mulai

Nilai Maksimum

Inisialisasi Array

Bangun Array

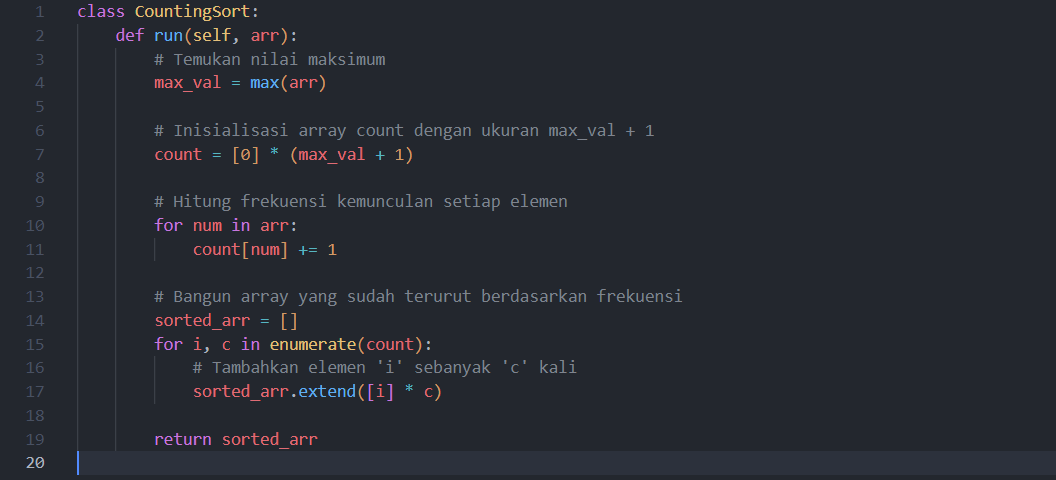
Hitung Frekuensi

Selesai

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space Complexity** |
| **Best** | **Average** | **Worst** |
| Counting Sort | O(n+k) | O(n+k) | O(n+k) | O(k) |

1. **Implementasi Python**



**Binary Searching**

1. Binary search adalah algoritma untuk mencari elemen dalam array atau list yang sudah terurut. Ide dasarnya adalah memanfaatkan fakta bahwa elemen dalam array terurut, sehingga kita dapat mengabaikan sebagian besar data pada setiap langkah pencarian. Hal ini membuat binary search sangat efisien dibandingkan dengan pencarian linear (sequential search) yang memeriksa setiap elemen satu per satu.

**Aplikasi :**

* **Pencarian dalam Daftar Terurut**: mencari angka nilai dalam daftar data numerik.
* **Pencarian dalam database** : mempercepat pencarian data dalam tabel yang sudah terurut berdasarkan suatu kolom.
* **Pemrograman Komputer dan Algoritma** : menemukan elemen dengan cepat, misalnya dalam algoritma pencarian jarak terpendek atau algoritma pemrograman dinamis.
* **Pencarian dalam Sistem File atau Direktori** : mencari file atau folder tertentu dalam struktur direktori yang sudah terurut.

1. **Langkah – langkah :**

* Tentukan **posisi kiri dan kanan** dari Array
* Cari **elemen tengah**
* **Bandingkan elemen tengah dengan target** :
* Jika arr[mid] == target, **berhenti**, target ditemukan
* Jika arr[mid] < target, cari di sisi kanan dengan mengganti left = mid + 1
* Jika arr[mid] > target, cari di sisi kiri dengan mengganti right = mid – 1
* **Ulangi langkah 2 dan 3** hingga target ditemukan atau tidak ada elemen yang tersisa untuk diperiksa.
* **Jika target tidak ditemukan**, kembalikan pesan "Target tidak ditemukan."

right = mid-1

Target ditemukan

Selesai

Kembali ke Hitung mid

left = mid+1

Apakah arr[mid] < target?

Hitung mid

Apakah arr[mid] == target?

Tentukan left = 0, right = len(arr) - 1

Mulai

**YA**

**TIDAK**

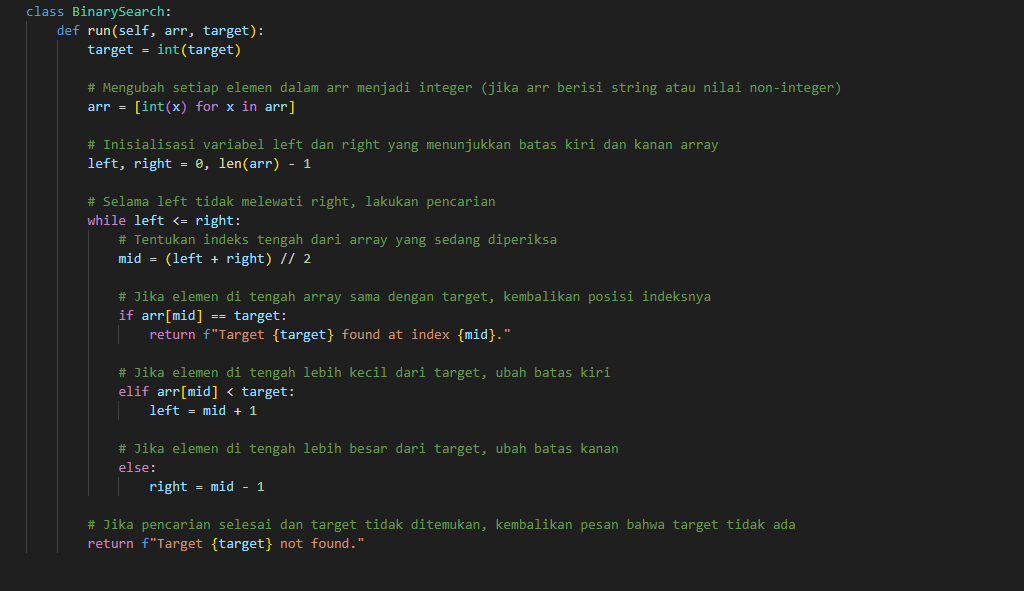
**YA**

**TIDAK**

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Binary | O(1) | O(log n) | O(log n) | O(1) |

1. **Implementasi Python**



**Linear Search Algorithm**

1. Linear adalah algoritma pencarian yang memeriksa elemen yang dicari satu per satu mulai dari elemen pertama hingga elemen terakhir dalam suatu array atau list.

**Aplikasi** :

* **Pencarian produk** di toko online dengan database yang tidak terurut.
* **Pencarian** **sederhana** dalam aplikasi mobile atau game.
* **Pencarian** nama di buku telepon.

1. **Langkah-Langkah**:

* **Iterasi** setiap elemen dalam array arr dengan indeks i dan nilai val.
* Jika val == target, **kembalikan** indeks i dan selesai. Jika tidak, lanjutkan iterasi.
* Jika sudah mencapai akhir array tanpa menemukan target, **kembalikan** -1.

Mulai

Periksa Elemen

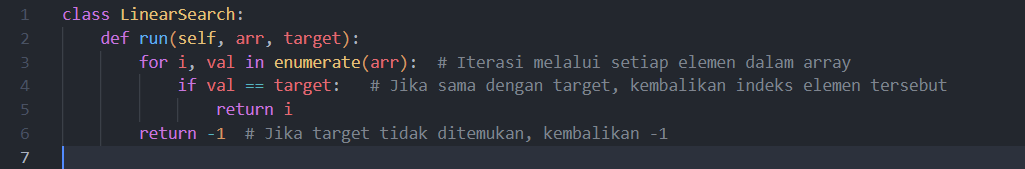
Inisialisasi Loop

Selesai

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space Complexity** |
| **Best** | **Average** | **Worst** |
| Linear Sort | O(1) | O(n) | O(n) | O(1) |

1. **Implementasi Python**



**Depth First Search Algorithm**

1. Depth\_First adalah algoritma pencarian dalam graf atau pohon yang menjelajahi simpul secara mendalam terlebih dahulu sebelum beralih ke simpul lain. Algoritma ini menjelajahi setiap jalur sampai kedalaman tertentu.

**Aplikasi :**

* **Pemetaan** jaringan listrik.
* **Penyusunan** jalur distribusi yang lebih rumit.
* **Mencari** rute atau jalan di peta.
* **Pemrosesan** Parse Tree.

1. **Langkah-Langkah:**

* **Inisialisasi**:

Buat *set visited* untuk menyimpan node yang sudah dikunjungi.

Buat *stack* dengan node awal (start).

Buat *result* sebagai list kosong untuk menyimpan urutan traversal.

* **Periksa** *stack*, Jika *stack* kosong, traversal selesai. Jika tidak, lanjutkan ke langkah berikutnya.
* **Ambil** elemen terakhir dari stack sebagai vertex.
* **Tambahkan** vertex ke visited dan result.
* **Tambahkan** semua tetangga vertex ke dalam stack.
* **Kembalikan** result yang berisi urutan traversal DFS.

Mulai

Inisialisasi

Periksa Stack

Ambil Node

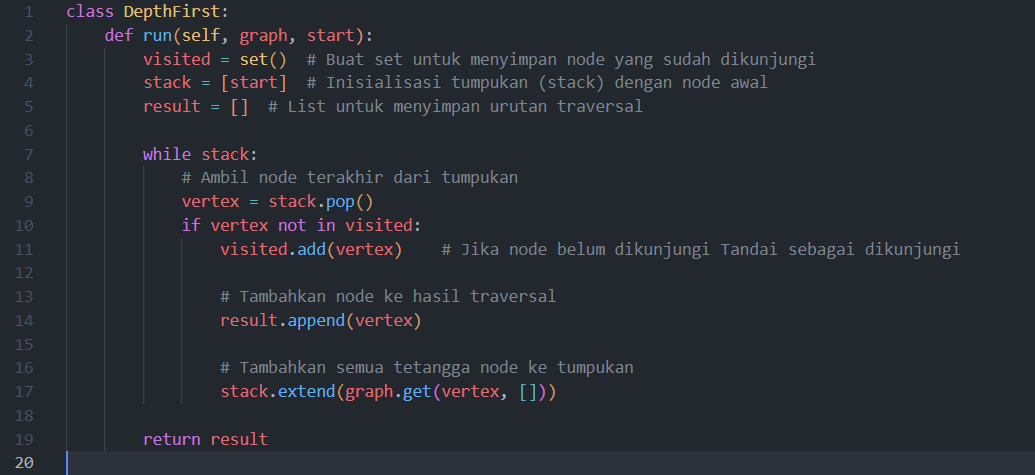
Tambahkan Vertex

Selesai

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space Complexity** |
| **Best** | **Average** | **Worst** |
| Depth\_First Sort | O(n+k) | O(n+k) | O(n+k) | O(n) |

1. **Implementasi Python**



**Breadth First Search Algorithm**

1. Breadth First adalah algoritma pencarian dalam graf atau pohon yang menjelajahi simpul (vertex) secara level per level (atau lapisan per lapisan). Mulai dari simpul awal, BFS mengunjungi semua simpul tetangga yang langsung terhubung (level pertama), kemudian melanjutkan ke level berikutnya (simpul yang terhubung ke simpul-simpul level pertama), dan seterusnya.

**Aplkasi** :

* **Penelusuran** jaringan komputer untuk menemukan koneksi terdekat.
* **Penyusunan** jalur distribusi barang dalam jaringan logistik.
* **Mencari** hubungan antar pengguna di jejaring sosial.

1. **Langkah-Langkah:**

* **Buat *set visited*** untuk menyimpan node yang sudah dikunjungi.
* **Buat *queue*** dengan node awal (start).
* **Buat *result*** sebagai list kosong untuk menyimpan urutan traversal.
* Jika queue kosong, traversal selesai. Jika tidak, lanjutkan ke langkah berikutnya.
* **Ambil elemen pertama** dari queue sebagai vertex (menggunakan queue.pop(0)).
* Jika vertex belum ada di visited:

Tambahkan vertex ke visited.

Tambahkan vertex ke result.

Tambahkan semua tetangga vertex (menggunakan graph.get(vertex, [])) ke dalam queue.

* **Ulangi** langkah hingga queue kosong.

Mulai

Inisialisasi

Ambil Node

Periksa Queue

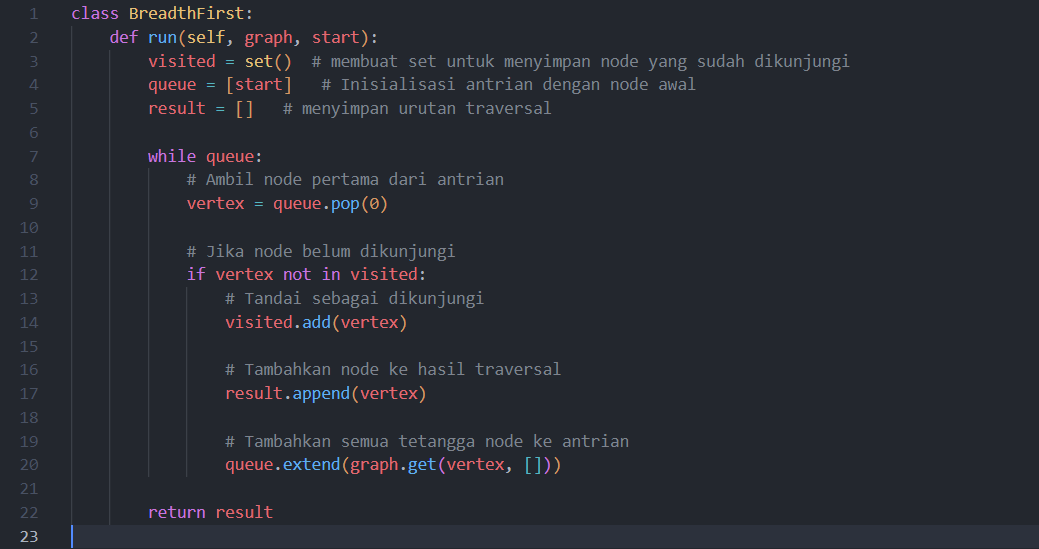
Periksa Node

Selesai

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space Complexity** |
| **Best** | **Average** | **Worst** |
| Breadth\_First Sort | O(n+k) | O(n+k) | O(n+k) | O(k) |

1. **Implementasi Python**



**Kruskal’s Algorithm**

1. Kruskal digunakan untuk mencari Minimum Spanning Tree (MST), yaitu pohon yang menghubungkan semua simpul dengan total bobot sisi paling kecil. Algoritma ini bekerja dengan memilih sisi-sisi graf yang memiliki bobot terkecil dan menghubungkan simpul-simpulnya tanpa membentuk siklus, hingga membentuk pohon yang mencakup semua simpul. Misalnya, di jaringan listrik atau jaringan komputer untuk menghubungkan titik-titik dengan biaya minimal.

**Aplikasi :**

* Desain jaringan (misalnya jaringan komputer, jaringan listrik).
* Penyusunan jalur distribusi.
* Rangkaian pembangkit listrik.

1. **Langkah-langkah:**
   * **Inisialisasi Union-Find**

Buat struktur data UnionFind untuk mengelola komponen yang terhubung dalam graf. Setiap simpul awalnya berada di set sendiri, dan peringkat (rank) awalnya 0.

* + **Urutkan semua sisi** berdasarkan bobotnya dari terkecil hingga terbesar
  + **Proses setiap sisi** :
* Ambil sisi satu per satu
* Cek kedua simpul terhubung atau tidak dalam komponen yang sama. Bila tidak, gabungkan danm tambahkan sisi tersebut ke MST. Bila terbuhung, lewati sisi tersebut
  + **Kembalikan MST**

Setelah proses selesai, kita mendapatkan **Minimum Spanning Tree** yang berisi semua sisi dengan bobot minimum yang menghubungkan semua simpul.

Pilih sisi terkecil

Mulai

Urutkan sisi

Inisialisasi UnionFind – Set Parent, Rank

Gabungkan U dan V dan tambah sisi ke MST

Proses sisi berikutnya (Jika ada)

Cek U dan V di komponen yang sama

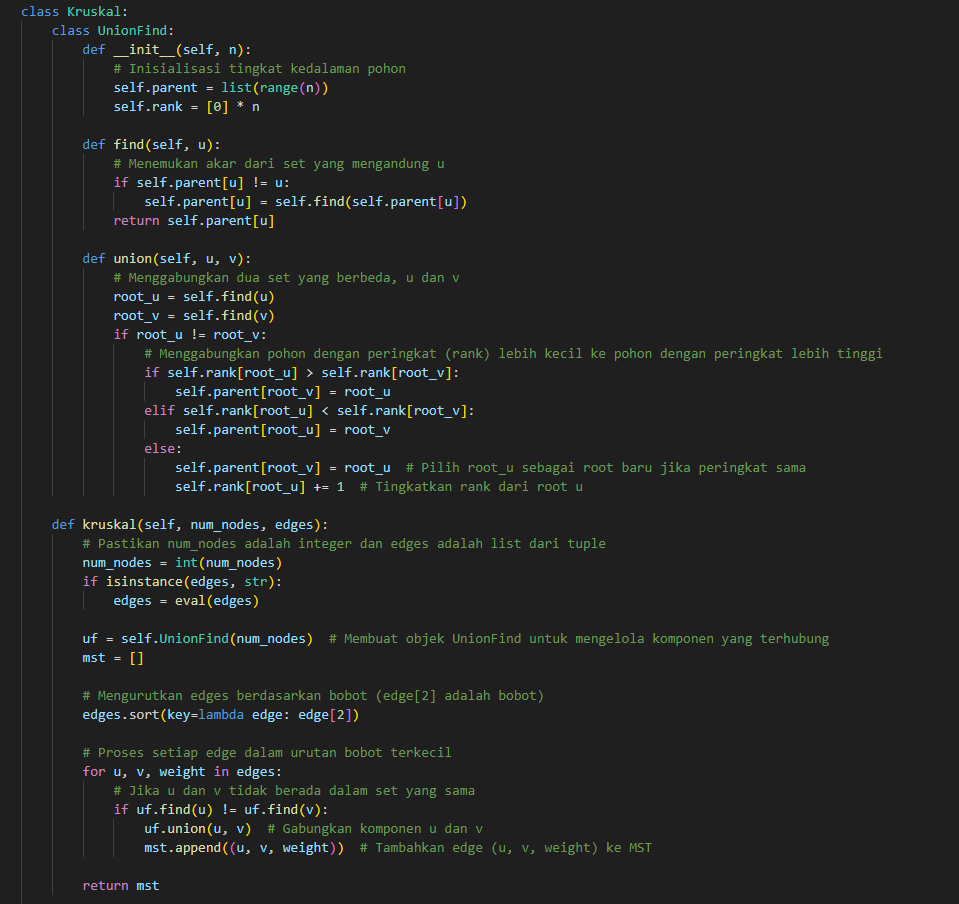
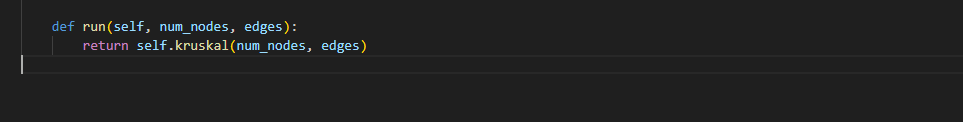
SELESAI

Kembalikan MST (Hasil)

1. **Time complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Kruskal | O(E log E) | O(E log E) | O(E log E) | O(V + E) |

1. **Implementasi Python**



**Dijkstra's Algorithm**

1. Dijkstra digunakan untuk menyelesaikan masalah pencarian jalur terpendek dalam sebuah graf berbobot. Masalah ini bertujuan untuk menemukan jalur terpendek dari titik awal ke titik tujuan dalam sebuah graf, yang bisa berupa peta jalan, jaringan komputer, atau masalah lain yang melibatkan hubungan antar titik.

**Aplikasi :**

* **Navigasi GPS** : mencari jalur terpendek antara dua lokasi
* **Jaringan komputer** : menemukan jalur terpendek antara dua komputer di jaringan.
* **Perencanaan dan optimasi rute** (contoh : sistem transportasi dan distribusi)

1. **Langkah – langkah :**
   * **Terima** input graph dan start\_node.
   * **Inisialisasi**: Set jarak awal ke start\_node = 0, node lainnya = ∞.
   * **Ambil Node**: Ambil node dengan jarak terkecil dari priority queue.
   * **Perbarui Tetangga**: Jika jarak lebih pendek ditemukan untuk tetangga, update jarak dan masukkan ke priority queue.
   * **Ulangi**: Ulangi hingga queue kosong.
   * **Kembalikan** jarak terpendek ke semua node.

Inisialisasi – Set jarak awal ke start node

Teirma input graf dan start\_node

Mulai

Selesai

Ulangi hingga queue kosong

Kembalikan jarak terpendek ke semua node

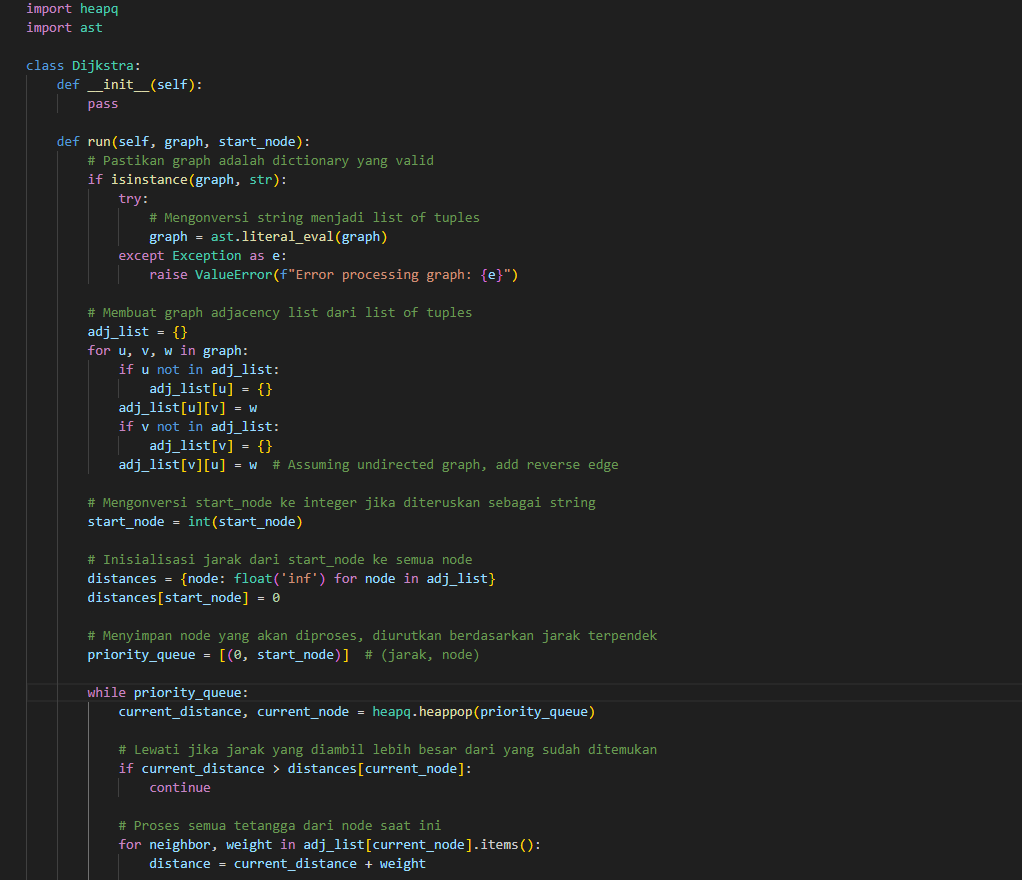
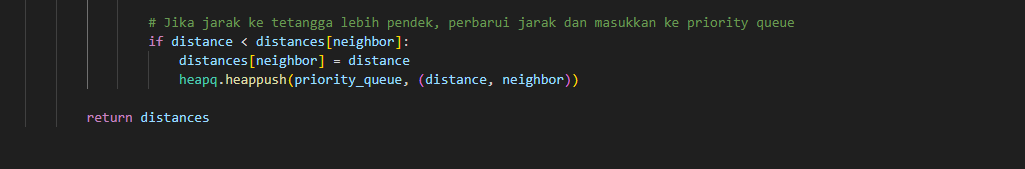
Perbarui Tetangga

Ambil node dengan jarak terkecil

1. **Time complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Dijkstra | |  | | --- | | O(E + V log V) |  |  | | --- | |  | | |  | | --- | | O(E + V log V) |  |  | | --- | |  | | O(E + V log V) | O(V + E) |

1. **Implementasi Python**



**Bellman - Ford Algorithm**

1. Bellman-Ford Algorithm adalah algoritma yang digunakan untuk mencari jalur terpendek dari satu titik (node) ke semua titik lainnya dalam graf berbobot. Berbeda dengan Dijkstra, Bellman-Ford dapat menangani graf dengan berat sisi negatif dan dapat mendeteksi siklus negatif (negative cycle), yang merupakan salah satu keunggulan utamanya.

**Aplikasi :**

* **Jaringan Komputer**: Menghitung rute terpendek dalam jaringan yang memiliki biaya yang bervariasi atau bahkan negatif.
* **Optimasi Transportasi dan Logistik**: Menemukan jalur tercepat dengan bobot biaya yang berbeda, termasuk biaya negatif (misalnya, diskon atau insentif).
* **Analisis Keuangan**: Menghitung jalur dengan biaya atau keuntungan negatif dalam model optimasi keuangan.
* **Deteksi Siklus Negatif**: Memeriksa apakah ada siklus negatif dalam graf, yang penting dalam banyak aplikasi seperti routing atau analisis graf.

1. **Langkah – langkah :**
   * **Inisialisasi** : Set jarak semua node ke ∞, kecuali node 0 yang diset ke 0.
   * **Relaksasi** :
     + Lakukan iterasi num\_nodes - 1 kali.
     + Perbarui jarak untuk setiap sisi (u, v) jika jarak baru lebih kecil.
   * **Cek siklus negatif** :
     + Lakukan satu iterasi lagi untuk memeriksa apakah masih ada pembaruan jarak.
     + Jika ada pembaruan, berarti ada siklus negatif.
   * **Kembalikan** jarak terpendek dari node 0 ke semua node, jika tidak ada siklus negatif.

Selesai

Untuk setiap sisi (u, v), periksa apakah dist[u] + weight < dist[v], jika ya, update dist[v]

cek siklus negatif

Jika ada pembaruan, ada siklus negatifJika ada pembaruan, ada siklus negatif

Kembalikan jarak terpendek

Lakukan iterasi (num – nodes – 1)

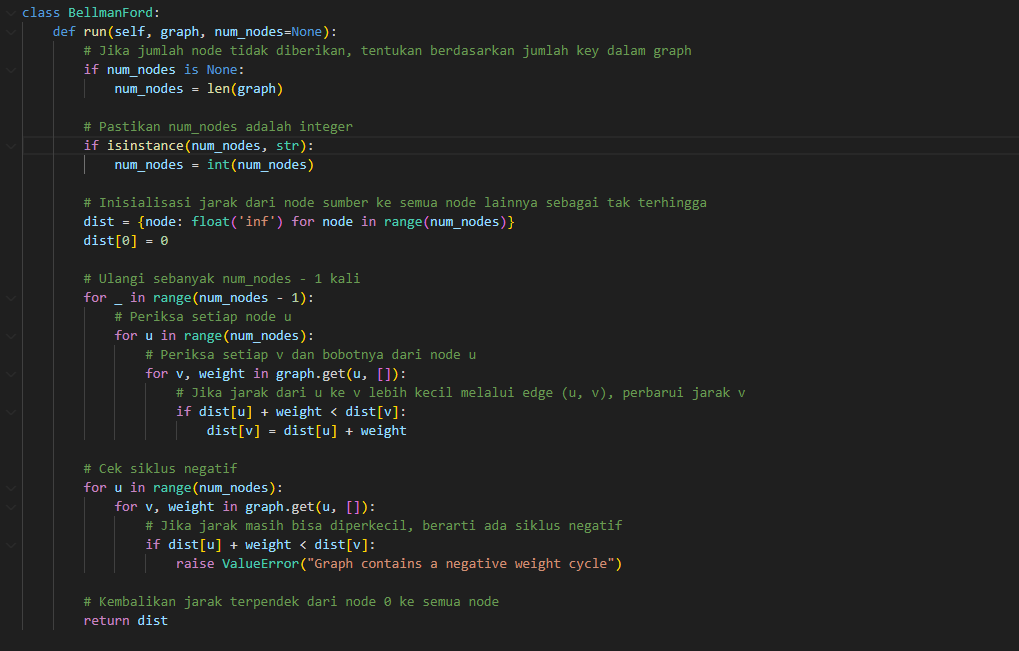
Mulai

Inisialisasi jarak ke smua node sebagai infinity , kecuali node 0

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Bellman-Ford | O(VE) | O(VE) | O(VE) | O(V) |

1. **Implementasi Python**

****

**Floyd Warshal Algorithm**

1. Floyd-Warshall adalah algoritma yang digunakan untuk menemukan jalur terpendek antara semua pasangan simpul (all pairs shortest paths) dalam graf berbobot, baik yang berbobot positif maupun negatif. Algoritma ini sangat efisien untuk graf yang memiliki banyak node dan sisi, karena dapat menghitung jarak terpendek antara setiap pasangan node dalam graf secara simultan.

**Aplikasi** :

* **Jaringan Komunikasi**: Menemukan jalur terpendek antara semua pasangan node dalam jaringan komputer atau telekomunikasi.
* **Sistem Transportasi**: Untuk menghitung rute tercepat atau paling efisien antara semua kota dalam suatu jaringan transportasi.
* **Optimasi Jaringan**: Dapat digunakan dalam jaringan untuk menemukan jalur terpendek antar titik
* **Game dan Graf dalam Komputasi**: Berguna untuk menentukan jalur terpendek antar titik dalam game atau aplikasi yang melibatkan graf besar.

1. **Langkah – langkah :**

* **Inisialisasi Matriks Jarak** : Set jarak antar node ke **∞**, kecuali jarak dari node ke dirinya sendiri yang diset ke **0**.
* **Isi jarak langsung** : Jika ada edge langsung antara dua node, set jaraknya sesuai dengan bobot edge tersebut
* **Perbarui jarak** : Cek semua pasangan node (i, j). Jika ada jalur yang lebih pendek melalui node k, perbarui jaraknya.
* **Kembalikan** matriks jarak terpendek antara semua pasangan node.

Mulai

Kembalikan hasil

Perbarui jarak untuk setiap pasangan node melalui node perantara (k)

Isi jarak untuk semua node langsung

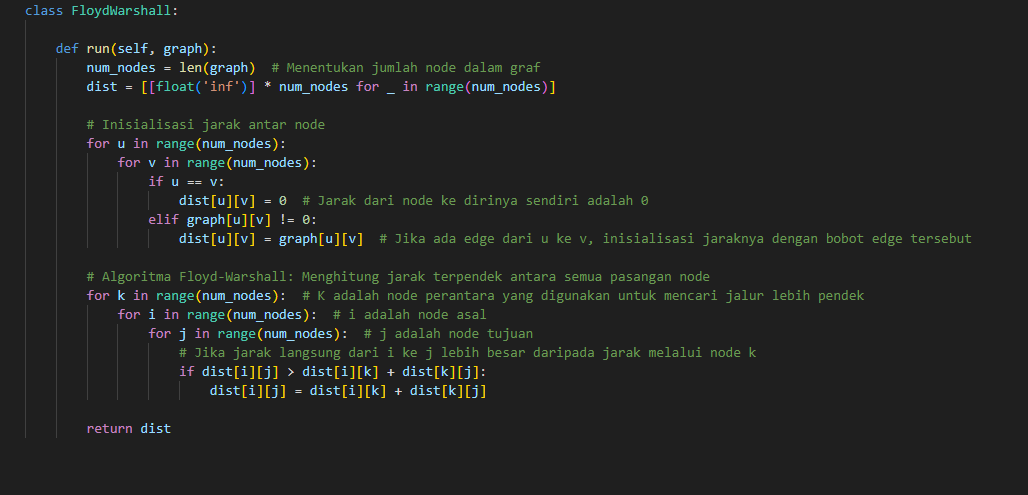
Inisialisasi matriks jarak

Selesai

1. **Time Complexity dan Sapce Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Floyd Warshall | O(V³) | O(V³) | O(V³) | O(V²) |

1. **Implementasi Python**



**Topological Sort Algorithm**

1. Topological Sort adalah algoritma yang digunakan untuk mengurutkan elemen-elemen dalam graf terarah yang tidak mengandung siklus (DAG - Directed Acyclic Graph). Dalam urutan ini, untuk setiap sisi u → v, u harus muncul sebelum v. Artinya, node u harus diproses sebelum node v jika ada hubungan ketergantungan di antara keduanya.

**Aplikasi :**

* **Penjadwalan Tugas**: Untuk mengurutkan tugas-tugas yang memiliki ketergantungan, seperti tugas A harus diselesaikan sebelum tugas B.
* **Kompilasi Kode**: Dalam kompilasi kode sumber, modul yang bergantung pada modul lain harus diproses terlebih dahulu.
* Pengelolaan Kursus: Dalam pendidikan, menentukan urutan kursus yang harus diambil berdasarkan kursus yang menjadi prasyarat.
* **Sistem Manajemen Proyek**: Menyusun urutan tugas dalam proyek agar tugas-tugas yang lebih penting diselesaikan terlebih dahulu

1. **Langkah – langkah :**

* **Periksa format graf** , pastikan berupa dictionary
* **Inisialisasi** :
* Buat set visited untuk melacak node yang sudah dikunjungi.
* Buat list result untuk menyimpan hasil urutan topologis.
* **Fungsi DFS** :
* Untuk setiap node yang belum dikunjungi, jalankan DFS untuk mengunjungi node tersebut dan semua tetangganya.
* Setelah mengunjungi semua tetangga, tambahkan node ke dalam result.
* **Kembalikan** urutan node dalam result untuk mendapatkan urutan topologis yang benar.

Loop untuk setiap node dalam graf

Selesai

**YA**

Lewati ke node berikutnya

Apakah node sudah dikunjungi ?

JALANKAN DFS

Kunjungi node dan tetangga

Tambahkan node ke result

Balikan hasil result

Periksa format graf

Inisialisasi visited dan result

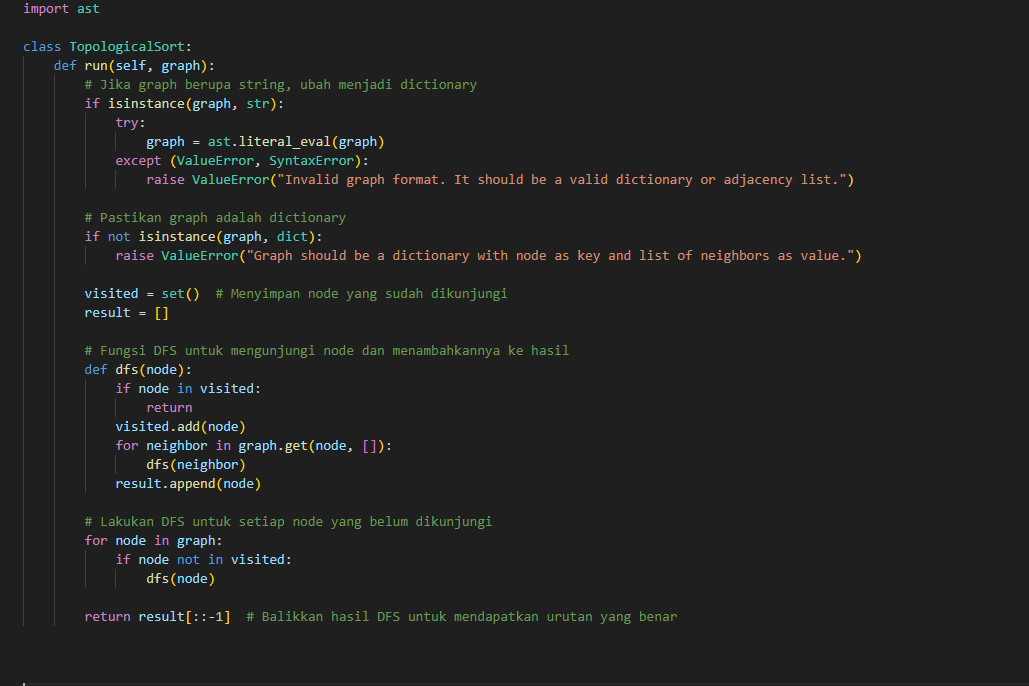
Mulai

**TIDAK**

1. **Time Complexity dan Sapce Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Topological | O(V + E) | O(V + E) | O(V + E) | O(V + E) |

1. **Implementasi Python**



**Flood Fill Algorithm**

1. Flood Fill adalah algoritma yang digunakan untuk mengisi area yang terhubung dalam sebuah grid atau gambar dengan warna atau nilai tertentu. Ini seperti alat "bucket fill" yang ada di aplikasi pengeditan gambar seperti MS Paint, yang digunakan untuk mewarnai area yang terhubung dengan warna yang dipilih.

**Aplikasi :**

* **Pengeditan Gambar**: Digunakan dalam alat pengisian "bucket fill" pada editor gambar seperti Photoshop dan lannya.
* **Pencarian dalam Graf**: Digunakan untuk menemukan area yang terhubung dalam graf atau matriks
* **Peta dan Game**: Digunakan untuk menandai area tertentu yang terhubung dalam game atau peta.
* **Simulasi**: Digunakan dalam aplikasi untuk mengisi area tertentu dalam sebuah grid (misalnya mengisi ruang kosong dalam game atau aplikasi grafis).

1. **Langkah – langkah :**

* **Terima Parameter: Algoritma menerima 4 parameter (grid, x, y, warna baru)**
* **Validasi Parameter**: Memeriksa apakah parameter valid (grid harus 2D, x dan y dalam batas grid).
* **Periksa Warna**: Jika warna yang ingin diganti sama dengan warna yang ada, algoritma akan selesai lebih awal.
* **Mulai DFS**: Jika warna berbeda, DFS akan dimulai untuk mengganti warna area yang terhubung.
* **Proses DFS**: DFS akan memeriksa dan mengganti warna titik-titik yang terhubung.
* **Kembalikan Grid** : Kembalikan grid yang sudah diperbarui.

**YA**

Selesai

Apakah warna di (x, y) sama dengan new\_color?

Terima Parameter: grid, x, y, new\_color

Mulai

**TIDAK**

Cek tetangga (atas, bawah, kiri, kanan)

Ulangi sampai selesai

Kembalikan grid yang sudah diubah

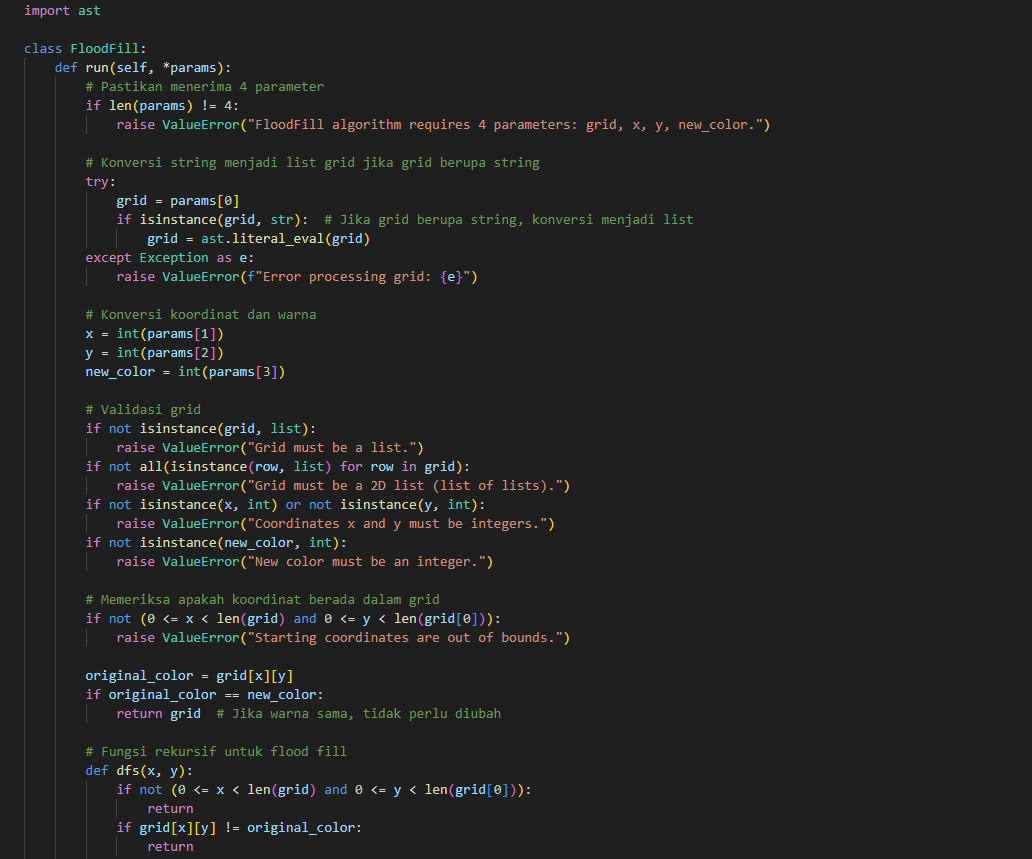
Mulai DFS: Ganti warna

Selesai

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Flood Fill | O(V + E) | O(V + E) | O(V + E) | O(V + E) |

1. **Implementasi Python**



**Lee Algorithm**

1. Lee Algorithm adalah algoritma pencarian jalur terpendek yang digunakan untuk menemukan rute terbaik antara dua titik pada grid (peta 2D), biasanya dalam masalah pergerakan di ruang yang memiliki beberapa hambatan atau penghalang. Biasanya digunakan dalam masalah navigasi robot atau pencarian jalur dalam game berbasis grid.

**Aplikasi :**

* Navigasi robot atau kendaraan otonom dalam lingkungan yang terhalang.
* Pencarian jalur pada peta 2D dengan beberapa penghalang.
* Aplikasi di sistem peta, misalnya dalam permainan (game) berbasis grid.
* Penyelesaian masalah "shortest path" di jaringan yang dapat berbentuk grid (peta 2D) dengan hambatan.

1. **Langkah – langkah** :

* Tentukan Grid dan Titik Awal/Tujuan
* **Inisialisasi** : Masukkan titik awal ke antrian dan tandai jarak titik awal sebagai 0
* **Proses pencarian** :
* Ambil titik dari antrian dan cek tetangga (atas, bawah, kiri, kanan).
* Jika tetangga jalan dan belum dikunjungi, masukkan ke antrian dan tandai jaraknya.
* **Cek Tujuan** : Jika tujuan ditemukan, hentikan pencarian.
* **Rekonstruksi Jalur** : Jika tujuan tercapai, mundur dari tujuan ke awal untuk mendapatkan jalurnya.
* **Kembalikan** jalur yang ditemukan dari titik awal ke tujuan.

**YA**

**TIDAK**

Selesai

Masukkan titik awal keantrian

Kembalikan Jalur

Periksa Tetangga

Rekonstruksi Jalur dari Tujuan ke Titik Awal

Cek Apakah Titik Tujuan Ditemukan?

Inisialisasi Grid, Titik Awal, dan Tujuan

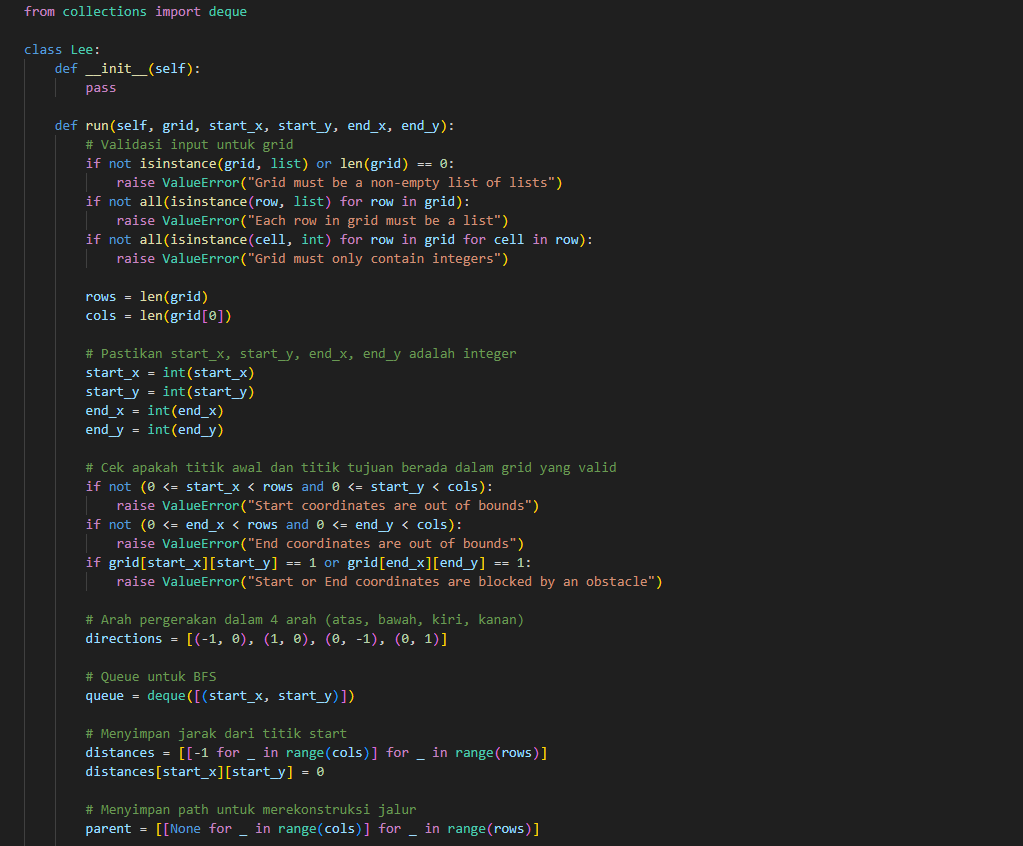
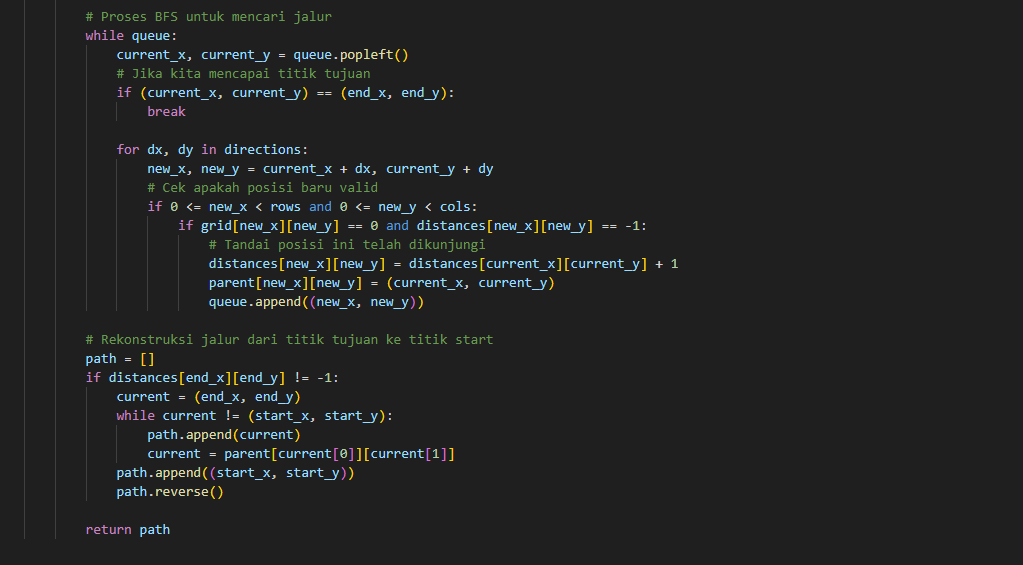
Mulai

Rekonstruksi Jalur dari Tujuan ke Titik Awal

1. **Time Complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Lee | O(V \* E) | O(V \* E) | O(V \* E) | O(V + E) |

1. **Implementasi Python**

****

**Huffman Coding (Compression Algorithm)**

1. **Huffman Coding** adalah algoritma kompresi data tanpa kehilangan (lossless compression) yang digunakan untuk mengurangi ukuran data dengan cara menggantikan karakter-karakter yang sering muncul dengan kode yang lebih pendek, dan karakter-karakter yang jarang muncul dengan kode yang lebih panjang. Algoritma ini ditemukan oleh David A. Huffman pada tahun 1952 dan menjadi dasar banyak metode kompresi yang efisien, seperti di format file ZIP, JPEG, dan MP3.

**Aplikasi:**

* Format File Kompresi (ZIP, GZIP, 7z)
* Kompresi Gambar (JPEG), Kompresi audio(MP3), Kompresi video(H.264,H.265)
* Sistem Pengiriman Data
* Pengkodean dalam Protokol Komunikasi

**2**. **Langkah-langkah:**

* **Menyiapkan Class Huffman**
* Buat kelas Huffman untuk mengelompokkan semua fungsi yang berkaitan dengan algoritma Huffman.

- **Menambahkan Fungsi** run

* Fungsi ini bertugas untuk menghitung frekuensi karakter dalam sebuah data input.

- **Mengembalikan Hasil**

* Fungsi run akan mengembalikan dictionary freq yang berisi informasi frekuensi karakter-karakter yang terdapat dalam data yang diberikan sebagai input.

Mulai

Class Huffman

**3**. **Time complexity dan Space Complexity**

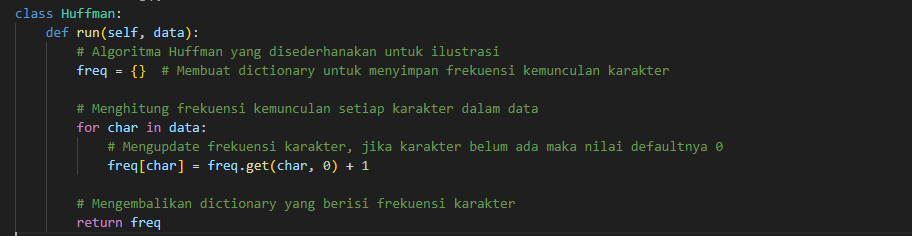
Selesai

Mengembalikan Hasil

Fungsi run

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Huffman | O(n log n) | O(n log n) | O(n log n) | O(n) |

**4**. **Implementasi Python**



**Euclid's Algorithm**

1. **Euclid's Algorithm** adalah algoritma yang digunakan untuk menghitung **Greatest Common Divisor (GCD)** atau **Faktor Persekutuan Terbesar (FPB)** dari dua bilangan. GCD adalah bilangan bulat positif terbesar yang dapat membagi dua bilangan tanpa menyisakan sisa.

**Aplikasi:**

* Kriptografi (RSA Encryption)
* Penyederhanaan Pecahan
* Perhitungan Modular Inverse
* Teori Bilangan

**2**. **Langkah-langkah:**

- **Menyiapkan Class Euclid:**

* Buat kelas Euclid untuk mengelompokkan fungsi yang berkaitan dengan algoritma Euclid.
* Buat kelas Euclid untuk mengelompokkan fungsi yang berkaitan dengan algoritma Euclid
* Kelas ini hanya memiliki satu fungsi utama yaitu run untuk menghitung GCD.

**- Menambahkan Fungsi run:**

* Fungsi ini bertugas untuk menghitung GCD dua bilangan menggunakan Algoritma Euclid.

- **Mengembalikan Hasil:**

* Setelah GCD ditemukan, kembalikan hasilnya dalam format string, misalnya f"GCD is {a}" di mana a adalah nilai GCD yang ditemukan.

- **Validasi input**

Memvalidasi input

Fungsi run

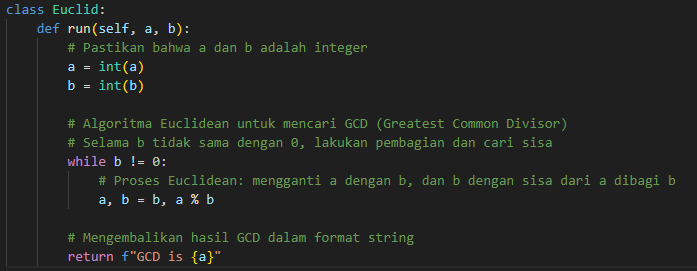
Buat class

Mulai

Selesai

**3**. **Time complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Euclid | |  | | --- | | O(log(min(a, b))) |  |  | | --- | |  | | |  | | --- | | O(log(min(a, b)) |  |  | | --- | |  | | O(log(min(a, b))) | O(1) |

**4**. **Implementasi Python** 

**Union Find Algorithm**

1. **Union Find Algorithm**, juga dikenal sebagai **Disjoint Set Union (DSU),** adalah sebuah struktur data yang digunakan untuk melacak elemen-elemen yang berada di dalam kumpulan atau set yang terpisah (disjoint). Algoritma ini sangat efisien dalam mengelola partisi dari elemen-elemen yang saling berhubungan.

**Aplikasi:**

* Kruskal's Algorithm
* Connected Components dalam Graf
* Periksa Relasi dan Kesetaraan
* Masalah Dinamis (Dynamic Connectivity)

**2.** **Langkah-langkah:**

- **Inisialisasi (**\_\_init\_\_**)**

- **Fungsi find(x):**

* Mencari induk dari elemen x.

- **Fungsi union(x, y):**

* Menggabungkan dua set yang berisi elemen x dan y.

- **Fungsi connected(x, y):**

* Memeriksa apakah elemen x dan y berada dalam set yang sama (terhubung).

- **Fungsi run(n, operations):**

* Menjalankan serangkaian operasi berdasarkan input

Menggabungkan x,y

Mencari induk x

Inisialisai “init”

Mulai

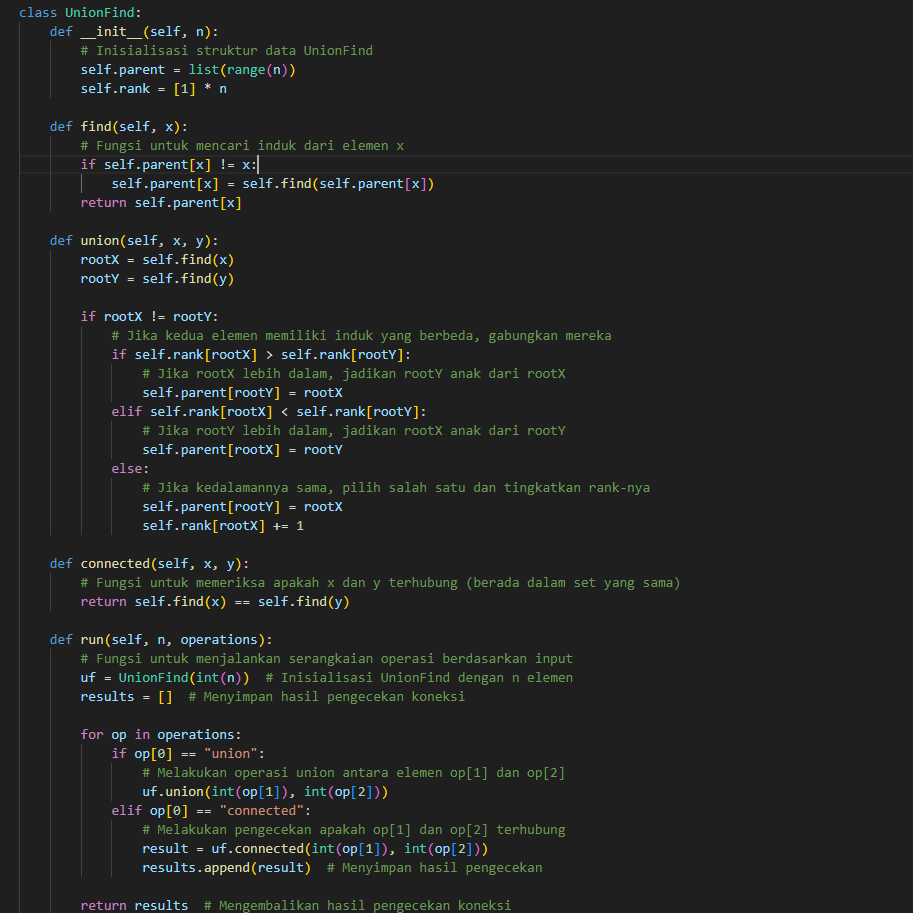
Selesai

Menjalankan input

**3**. **Time complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Union Find | O(α(n)) | O(α(n)) | O(α(n)) | O(n) |

**4.** **Implementasi Python**



**Kadane's Algorithm**

1. **Kadane's Algorithm** adalah algoritma yang digunakan untuk menemukan **subarray dengan jumlah elemen terbesar** dalam sebuah array bilangan. Subarray adalah urutan elemen yang berurutan dalam array. Algoritma ini sangat efisien karena hanya membutuhkan waktu **O(n),** di mana **n** adalah panjang array.

**Aplikasi:**

* Masalah Maksimum Subarray Sum
* Masalah Laba Maksimal dalam Perdagangan Saham
* Masalah Subarray dengan Jumlah Terendah
* Masalah Subarray Terpanjang dengan Jumlah Positif

**2. Langkah-langkah:**

**- Inisialisasi Variabel:**

○ Buat dua variabel:

* max\_current: Menyimpan jumlah subarray saat ini. Inisialisasi dengan elemen pertama array.
* max\_global: Menyimpan jumlah subarray maksimum yang ditemukan. Inisialisasi dengan elemen pertama array.

**- Iterasi Melalui Array:**

* Mulai iterasi dari elemen kedua (indeks 1) hingga akhir array.
* **Perbarui max\_global:**
* Jika max\_current lebih besar dari max\_global, perbarui nilai max\_global dengan nilai max\_current.
* **Kembalikan Hasil:**
* Setelah iterasi selesai, kembalikan nilai max\_global, yang berisi jumlah subarray maksimum.

Mulai

Tentukan max\_current

Iterasi Melalui Array

Inisialisasi Variabel

Selesai

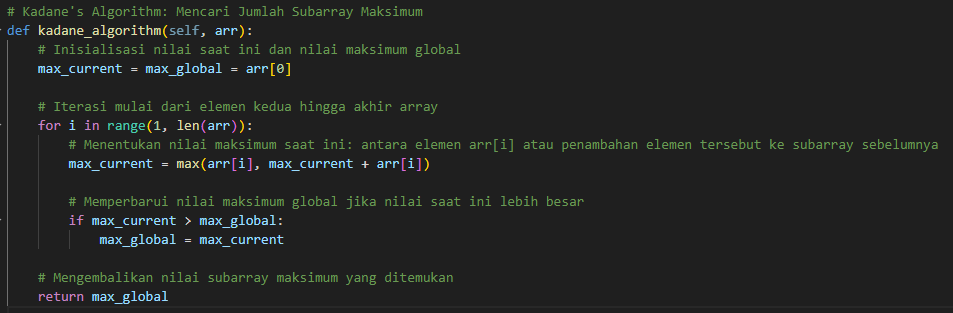
Kembalikan Hasil

Periksa max\_global

**3.** **Time complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Kadane | O(n) | O(n) | O(n) | O(1) |

**4.** **Implementasi Python**



**Floyd's Algorithm (Detection Algorithm)**

**1**. **Floyd's Algorithm**, juga dikenal sebagai **Floyd's Cycle Detection Algorithm** atau **Floyd's Tortoise and Hare Algorithm**, adalah algoritma yang digunakan untuk mendeteksi **siklus** (cycle) dalam sebuah struktur data yang berbentuk linked list atau dalam urutan elemen yang terhubung (seperti graf berarah). Algoritma ini sangat efisien, berjalan dalam **O(n)** waktu dengan **O(1)** ruang, dan tidak memerlukan penyimpanan tambahan untuk melacak elemen yang sudah dikunjungi.

**Aplikasi:**

* Deteksi Siklus dalam Linked List
* Deteksi Siklus dalam Graf Berarah
* Deteksi Siklus dalam Sistem Operasi (Deadlock)
* Deteksi Siklus dalam Algoritma Graf Traversal

2. **Langkah-langkah**:

**- Inisialisasi Pointer**: slow( bergerak 1 langkah), Fast(bergerak 2 langkah)

**- Perulangan untuk Mencari Siklus**: (lakukan perulangan selama pointer tidak mencapai akhhir)

**- Cek Pertemuan Pointer:**

Jika pada suatu titik pointer lambat dan pointer cepat bertemu, maka telah terdeteksi adanya siklus.

**- Pengecekan Jika Tidak Ada Siklus:**

Jika pointer fast mencapai None (artinya tidak ada siklus), keluar dari perulangan dan return False untuk menunjukkan bahwa tidak ada siklus dalam linked list.

Mulai

Pertemuan pointer

Mencari siklus

Inisialisasi Pointer

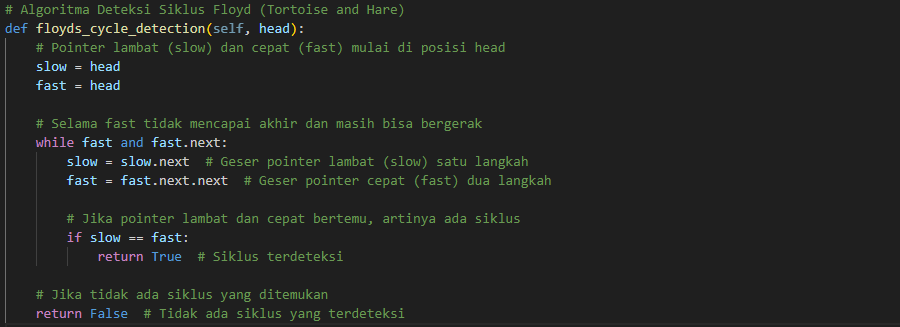
Cek tidak ada siklus

Selesai

**3**. **Time complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Floyd | O(V + E) | O(V + E) | O(V + E) | O(1) |

**4**. **Implementasi Python**



**KMP Algorithm**

1. **KMP Algorithm** (Knuth-Morris-Pratt Algorithm) adalah algoritma yang digunakan untuk **mencari substring** dalam sebuah string utama dengan cara yang efisien. Algoritma ini dikembangkan oleh **Donald Knuth**, \*\* Vaughan Pratt\*\*, dan **James H. Morris** pada tahun 1977, dan dikenal karena kemampuannya untuk mencari pola dalam waktu **O(n + m)**, di mana n adalah panjang string utama dan m adalah panjang pola yang dicari.

**Aplikasi:**

* Pencarian Substring dalam Teks
* Pencocokan Pola dalam Pencarian Teks Berbasis Web
* Pengolahan Data Genetik

1. **Langkah-langkah** :

* **Inisialisasi dan Penyiapan:**
* Definisikan fungsi utama kmp\_algorithm(self, text, pattern) yang menerima dua argumen: text (string utama) dan pattern (pola yang dicari dalam teks).
* Tentukan dua variabel: i dan j, yang akan berfungsi sebagai indeks untuk menelusuri text dan pattern secara berurutan.
* **Fungsi Compute LPS (Longest Prefix Suffix)**

menghitung array LPS (Longest Prefix Suffix) yang berisi panjang prefix yang juga merupakan suffix pada setiap posisi dalam pola.

* **Pencocokan Pola dengan Teks:**

Lakukan pencocokan karakter antara pola dan teks menggunakan informasi dari LPS.

* **Return Hasil**

Jika pencocokan tidak ditemukan setelah memeriksa seluruh teks, kembalikan -1 untuk menunjukkan bahwa pola tidak ditemukan dalam teks.

Compute LPS

Pencocokan pola

Inisialisasi

Mulai

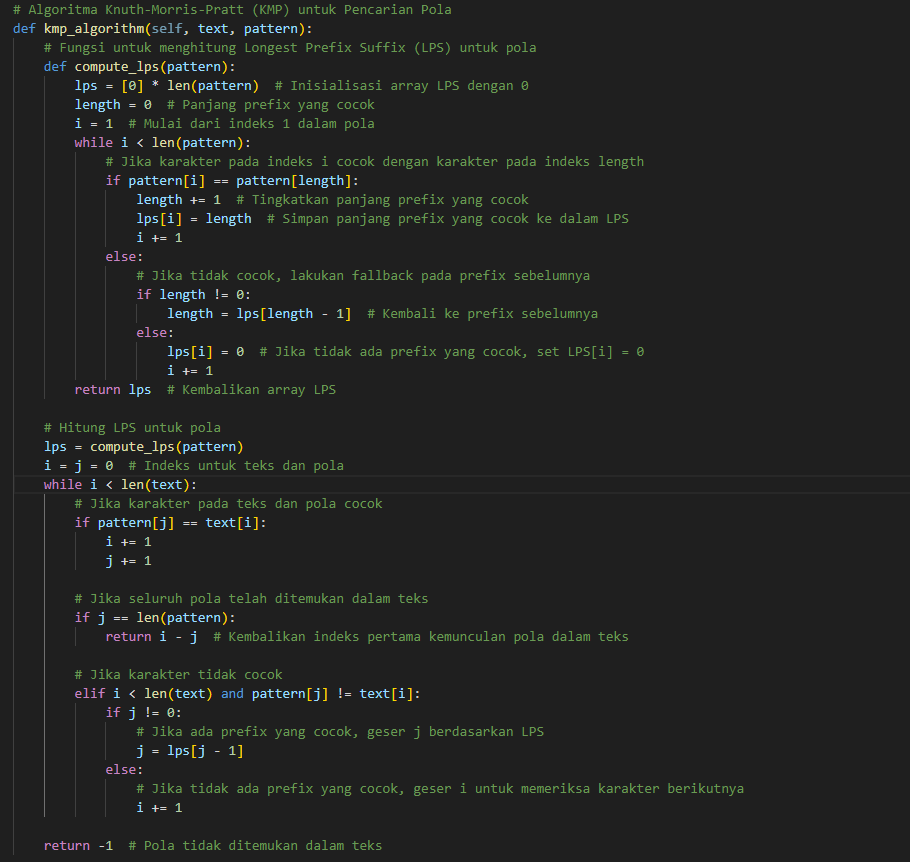
Selesai

Return

**3**. **Time complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| KMP Algorithm | O(n + m) | O(n + m) | O(n + m) | O(m) |

**4**. **Implementasi Python**



**Quick Select Algorithm**

1. **Quick Select Algorithm** adalah algoritma yang digunakan untuk menemukan elemen ke-k terkecil (atau terbesar) dalam sebuah array atau list yang tidak terurut.

**Aplikasi:**

* Mencari Elemen ke-k Terkecil atau Terbesar dalam Array
* Median
* Top-K Largest/Smallest Elements
* Pencarian Quick Select dalam Data Besar

1. **Langkah-langkah:**

- **Konversi Input:** Ubah elemen array dan k menjadi tipe integer.

- **Validasi:** Periksa apakah k valid (1 ≤ k ≤ panjang array).

- **Pilih Elemen :**

Jika indeks pivot = k, elemen ditemukan.

Jika k < pivot, cari di bagian kiri.

Jika k > pivot, cari di bagian kanan

- **Kembalikan** elemen ke-k terkecil.

Bangun table

Validasi input

Mulai

Menjalankan hasil

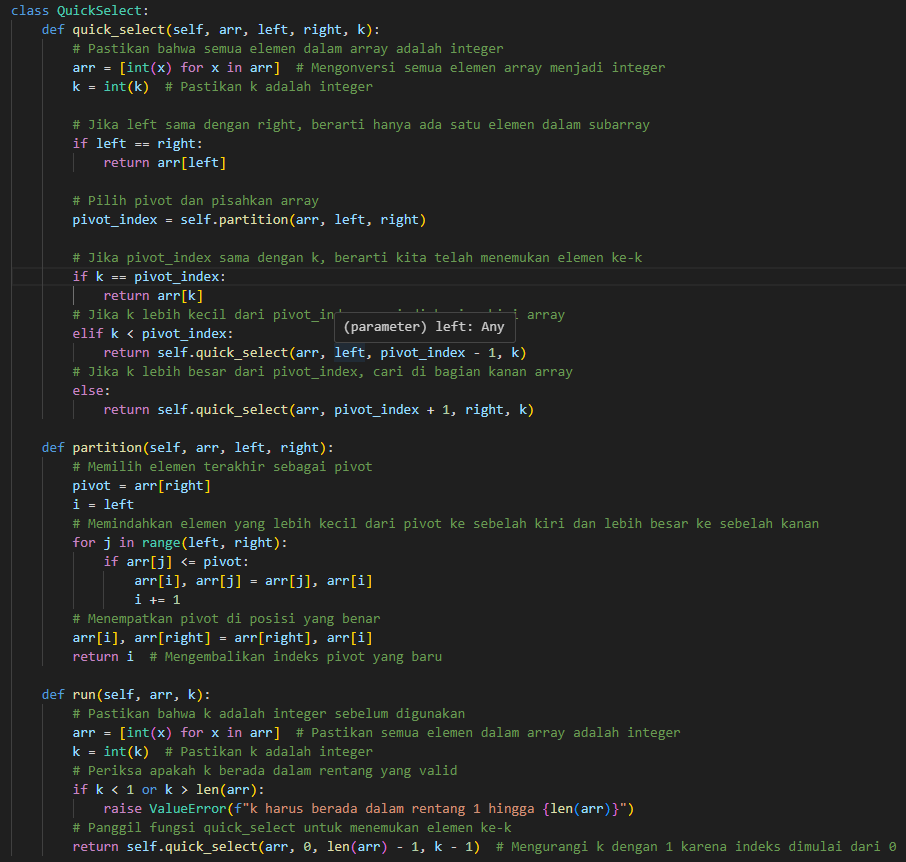
Pengembalian hasil

Fungsi pencarian

Selesai

3. **Time complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Quick Select | O(n) | O(n) | O(n^2) | O(1) |

**4.Implementasi Python** 

**Boyer-More Majority (Vote Algorithm)**

1. **Boyer-Moore Majority Vote Algorithm** adalah algoritma yang digunakan untuk **mencari elemen mayoritas** dalam sebuah array atau daftar. Elemen mayoritas adalah elemen yang muncul lebih dari setengah kali dalam array (dalam konteks array dengan jumlah elemen yang genap atau ganjil, mayoritas adalah elemen yang muncul lebih dari n/2 kali).

**Aplikasi:**

* Mencari Elemen Mayoritas dalam Array
* Pencarian Elemen Paling Sering dalam Data Besar
* Pemrosesan Teks (Character Frequency Analysis)
* Identifikasi Mayoritas dalam Voting

**2. Langkah-langkah:**

**- Menyiapkan Class QuickSelect**:

* Buat kelas QuickSelect untuk mengelompokkan fungsi-fungsi terkait.
* Kelas ini akan memiliki dua fungsi utama: quick\_select dan partition.

- **Menambahkan Fungsi quick\_select**

* Fungsi ini akan mencari elemen ke-k terkecil dalam array menggunakan metode Quick Select.

- **Menambahkan Fungsi partition**

* Fungsi ini digunakan untuk melakukan partisi array dengan memilih elemen terakhir sebagai pivot.

- **Menambahkan Fungsi run**

* Fungsi ini bertugas untuk memvalidasi input dan memanggil fungsi quick\_select.

Fungsi pertition

Fungsi quickselect

Class quickselect

Mulai

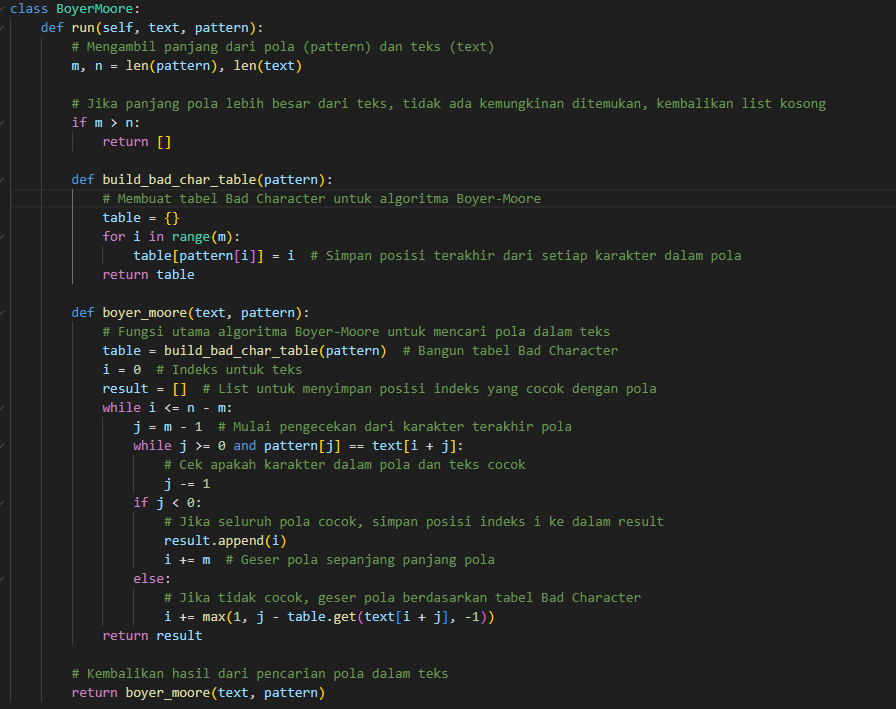
Selesai

Fungsi run

3. **Time complexity dan Space Complexity**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | | **Space complexity** |
| **Best** | **Average** | **Worst** |
| Boyer-Moore Majority Algorithm | O(n) | O(n) | O(n) | O(1) |

4. **Implementasi Python**



Github Link : [*https://github.com/StevenZega/UAS\_Algoritma*](https://github.com/StevenZega/UAS_Algoritma)

**Komentar untuk rekan tim :**

Menurut saya, kami bertiga bekerja sama dengan sangat baik dalam pembuatan project ini. Kami menyepakati untuk membagi tugas berdasarkan kategori setiap algoritma juga berdiskusi bersama dalam menjalankan terminal yang mana menurut kami seikit tricky.

Dimulai dari coding, pembuatan dokumentasi juga pembahasan repository kami lakukan semua secara bersamaan dengan pembagiannya masing-masing

Oleh karena itu, di dalam tim ini tidak ada yang merasa siapa paling dominan pengaruhnya. Setiap dari kami punya kelebihan masing-masing, contohnya Steven yang kami andalkan dalam mengartikan fungsi beberapa algoritma, Gilbert yang teliti dalam sintaks coding, dan juga saya yang mengecek kerapian kode dan keberhasilannya. Jadi, saya rasa kami membawa dampak yang sama bagi kelancaran UAS ini.