UAS ALgoritma Pemrograman
Group 4

Anggota:
- Taneshia Aurelia Hilton - 422024002
- Nikeisha Yosephin Palingu - 422024026
- Vanessa Maria Lumongga Pardede - 422024030

TASK 1
The description of each category and its algorithms:

Bacic Alghorithms

Pengertian:
Kumpulan langkah-langkah mendasar untuk menyelesaikan masalah secara sistematis.

Logic:

- Membagi masalah menjadi langkah sederhana.
- Memanfaatkan struktur kontrol seperti if-else, loop (for/while), dan fungsi.

Fungsinya:
Dasar untuk membangun algoritma yang lebih kompleks dalam pemrograman.

1. Huffman Coding

- **Pernyataan Masalah:** Huffman Coding adalah algoritma yang dirancang untuk mengurangi ukuran data dengan memberikan kode yang lebih pendek kepada karakter yang sering muncul dan kode lebih panjang kepada karakter yang jarang muncul.
- **Prinsip Kerja**: Mengompresi data dengan membuat pohon biner berdasarkan frekuensi karakter, menghasilkan kode pendek untuk karakter yang sering muncul dan kode panjang untuk karakter yang jarang.
- **Penerapan:**

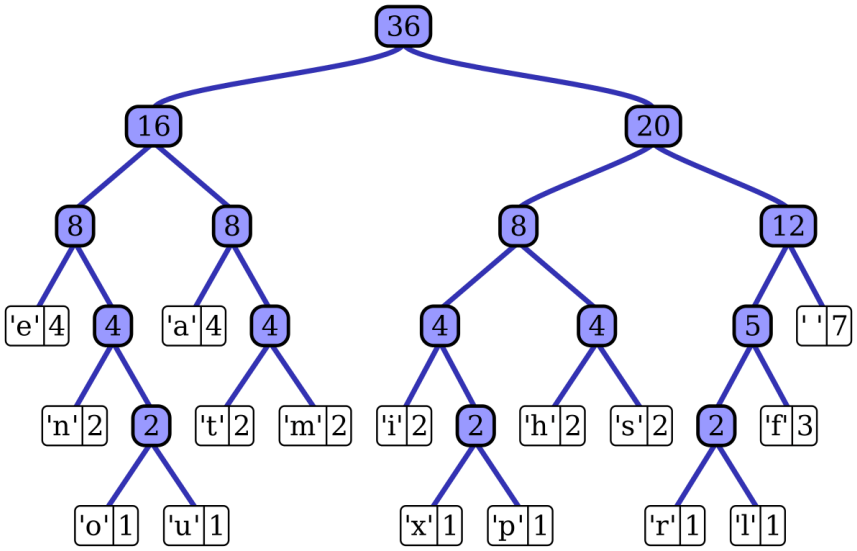
1. Digunakan untuk kompresi data (ZIP, RAR).
2. Format multimedia (JPEG, MP3).
3. Penyimpanan informasi
4. Kompresi file teks, misalnya pada email.

- **Langkah-langkah:**

1. Hitung seberapa sering setiap huruf muncul dalam teks.
2. Gabungkan dua huruf yang paling jarang muncul menjadi satu kelompok.

3. Ulangi penggabungan sampai semua huruf tergabung menjadi satu pohon besar.
4. Tentukan kode biner untuk setiap huruf berdasarkan posisi di pohon (kiri 0, kanan 1).
5. Ganti setiap huruf di teks dengan kode binernya.

- **Kelebihan**: Efisien untuk kompresi data, menghasilkan file yang lebih kecil.
- **Kekurangan**: Memerlukan waktu untuk membangun pohon dan pemrosesan lebih kompleks dibandingkan algoritma kompresi sederhana.



2. Algoritma Euclid

- **Pernyataan Masalah:** Mencari faktor persekutuan terbesar (FPB) dari dua bilangan bulat positif.
- **Prinsip Kerja**: Menggunakan pendekatan rekursif untuk menemukan **FPB** (Faktor Persekutuan Terbesar) antara dua angka dengan membagi angka yang lebih besar dengan angka yang lebih kecil, lalu menggantinya dengan sisa pembagian.
- **Penerapan:**

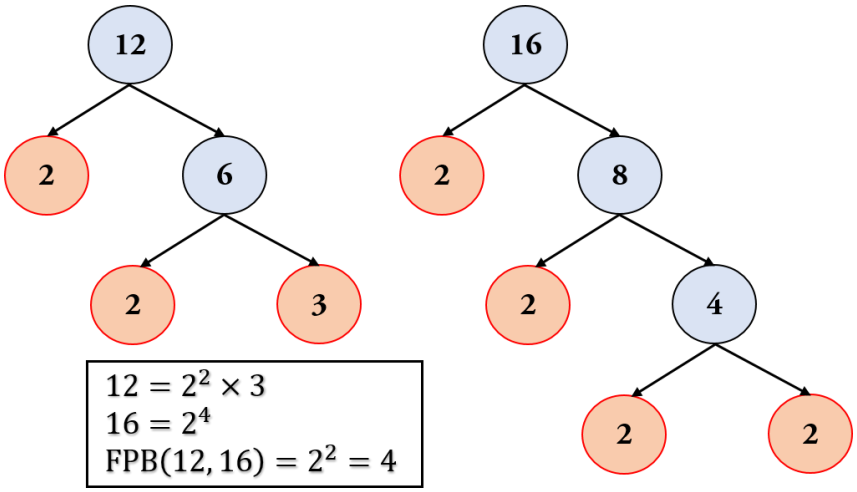
1. Menyederhanakan pecahan.
2. Digunakan dalam enkripsi RSA.
3. Menghitung LCM dalam tugas-tugas manajemen siklus.

- **Langkah-langkah:**

1. Ambil dua angka yang ingin dicari pembagiannya (misalnya A dan B).
2. Bagi angka yang lebih besar dengan yang lebih kecil, lalu catat sisanya.
3. Ganti angka yang lebih besar dengan angka yang lebih kecil, dan angka yang lebih kecil dengan sisa hasil pembagian.
4. Ulangi proses ini sampai sisanya menjadi nol.

5. Angka terakhir yang bukan nol adalah hasilnya (faktor persekutuan terbesar).

- **Kelebihan:** Sangat efisien dan cepat dalam menghitung FPB.
- **Kekurangan:** Tidak dapat digunakan untuk masalah yang lebih kompleks seperti kelipatan terkecil.



3. Algoritma Union-Find

- **Pernyataan Masalah:** Union-Find adalah struktur data yang digunakan untuk melacak kelompok yang terpisah dan menggabungkannya, digunakan dalam graf.
- **Prinsip Kerja**: Mengelola kumpulan elemen yang terbagi menjadi beberapa himpunan dengan dua operasi utama: **find** (untuk menemukan himpunan elemen) dan **union** (untuk menggabungkan dua himpunan).
- **Penerapan:**

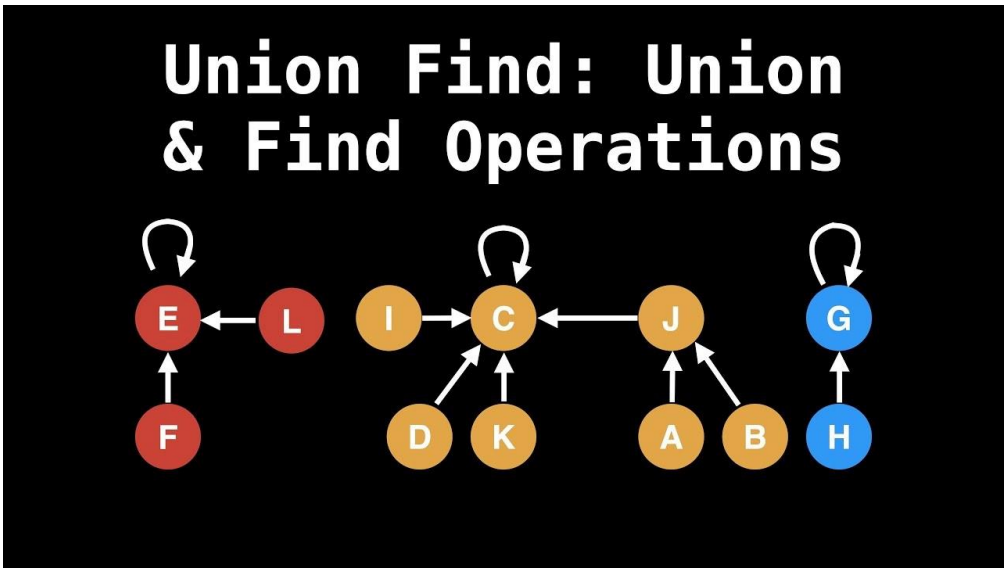
1. Digunakan oleh Boost Graph Library untuk mengimplementasikan fungsionalitas Incremental Connected Components
2. Mendeteksi siklus pada graf.
3. Memeriksa konektivitas jaringan.
4. Digunakan dalam algoritma Kruskal.

- **Langkah-langkah:**

1. Urutkan semua sisi berdasarkan bobot dari yang terkecil hingga terbesar.
2. Inisialisasi struktur data seperti union-find untuk melacak komponen terhubung dan mendeteksi siklus.
3. Pilih sisi dari daftar yang diurutkan: Tambahkan sisi ke MST jika tidak menyebabkan siklus.
4. Hentikan ketika MST memiliki tepat n−1n-1n−1 sisi (dengan nnn adalah jumlah simpul dalam graf).

- **Kelebihan**: Efisien untuk masalah yang melibatkan pengelompokan elemen, seperti dalam graf atau jaringan.

- **Kekurangan**: Penggunaan memori bisa menjadi lebih tinggi, terutama untuk struktur data yang besar.



Sorting Algorithms

Pengertian:

Algoritma untuk menyusun elemen dalam urutan tertentu, baik itu naik (ascending) atau turun (descending).

Fungsinya:

Digunakan dalam pengolahan data, misalnya mengurutkan daftar nama, angka, atau barang di e-commerce**.**

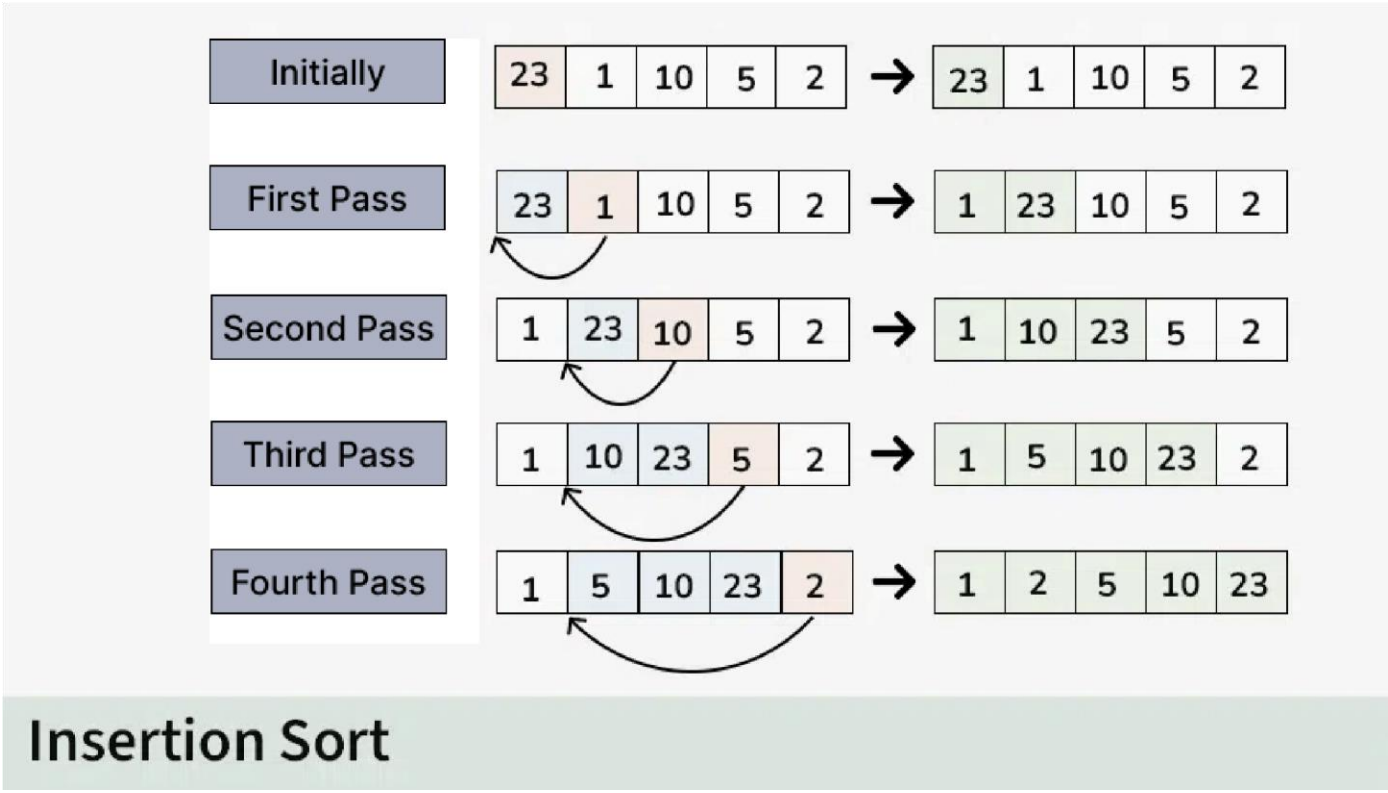
1. Insertion Sort

- **Pernyataan Masalah:** Insertion Sort adalah algoritma yang mengurutkan dengan menyisipkan elemen pada posisi yang tepat, satu per satu.
- **Prinsip Kerja**: Mengurutkan dengan cara mengambil elemen satu per satu dan menyisipkannya pada posisi yang tepat dalam bagian yang sudah terurut.
- **Penerapan:**

1. Cocok untuk dataset kecil atau yang sudah hampir terurut.
2. Digunakan dalam pengurutan kartu pada permainan kartu remi.
3. Digunakan dalam perangkat lunak sederhana untuk mengurutkan daftar nama siswa.

- **Langkah-langkah:**

1. Mulai dari elemen kedua dalam array, anggap elemen pertama sudah terurut.
2. Bandingkan elemen ini dengan elemen sebelumnya.
3. Jika lebih kecil, geser elemen sebelumnya ke kanan.



4. Ulangi hingga elemen berada di posisi yang benar.
5. Lanjutkan ke elemen berikutnya dan ulangi langkah 2-4.

- **Kelebihan**: Mudah diimplementasikan dan efisien untuk data yang sudah hampir terurut.
- **Kekurangan**: Kurang efisien untuk dataset besar.

2. Selection Sort

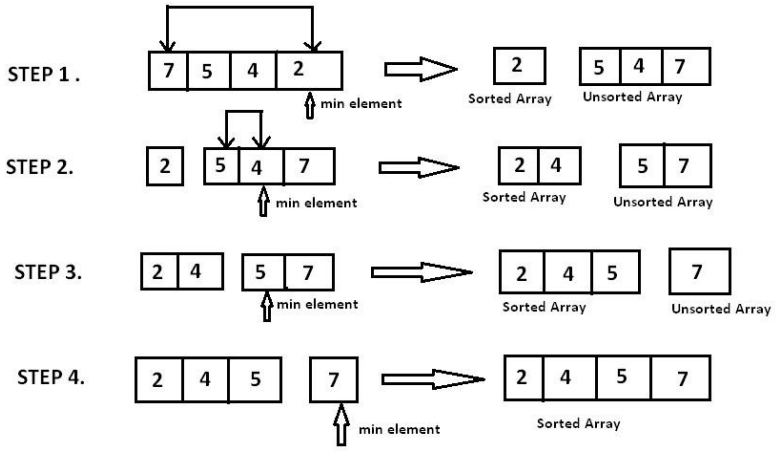
- **Pernyataan Masalah:** Selection Sort adalah algoritma yang berulang kali memilih elemen terkecil dari bagian yang belum urut, lalu memindahkannya ke posisi yang benar.
- **Prinsip Kerja**: Mengurutkan dengan memilih elemen terkecil dari bagian yang belum terurut dan menukarnya dengan elemen pertama, lalu mengulang proses untuk elemen berikutnya.
- **Penerapan:**

1. Pengurutan dataset kecil.
2. Mengurutkan daftar harga dalam penerapan kasir.
3. Digunakan pada sistem di mana operasi menulis lebih mahal daripada membaca, seperti EEPROM.

- **Langkah-langkah:**

1. Cari nilai terkecil dari daftar.
2. Tukar nilai terkecil itu dengan nilai di posisi pertama.
3. Ulangi langkah 1 dan 2 untuk sisa daftar, mulai dari posisi kedua, ketiga, dan seterusnya.
4. Lanjutkan hingga semua nilai dalam daftar sudah diurutkan.

- **Kelebihan**: Tidak membutuhkan banyak ruang tambahan.
- **Kekurangan**: Kurang efisien untuk dataset besar.



3. Heap Sort

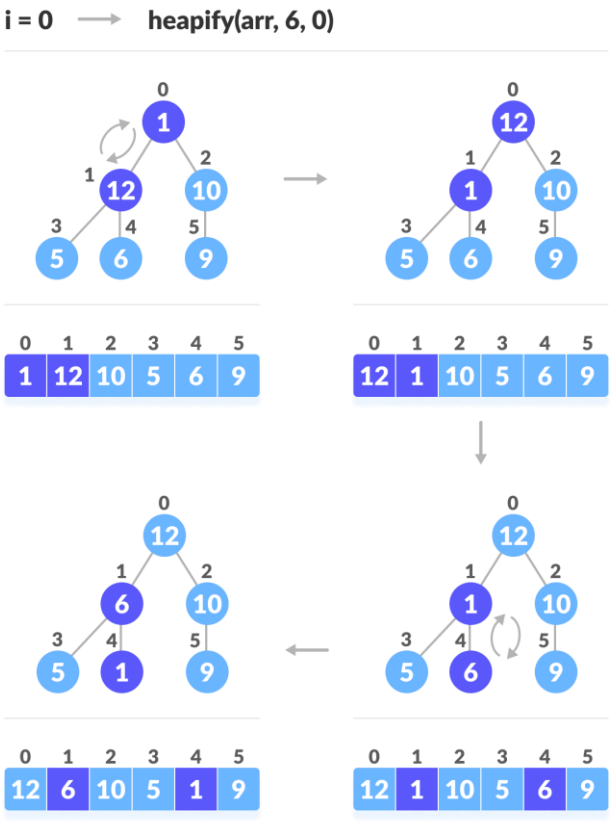
- **Pernyataan Masalah:** Heap Sort menggunakan heap untuk mengurutkan data dengan efisien.
- **Prinsip Kerja**: Menggunakan struktur data heap untuk mengurutkan elemen dengan cara membangun heap maksimum dan mengambil elemen terbesar satu per satu.
- **Penerapan:**

1. Digunakan dalam implementasi antrean prioritas.
2. Cocok untuk pengurutan skala besar.
3. Contoh nyata: Pengurutan tugas dalam sistem operasi berbasis antrean prioritas.

- **Langkah-langkah:**

1. Susun data menjadi struktur heap (pohon yang memenuhi sifat heap).
2. Ambil elemen terbesar (akar heap) dan pindahkan ke posisi akhir dalam array.
3. Perbaiki struktur heap untuk elemen yang tersisa agar tetap menjadi heap.
4. Ulangi proses hingga semua elemen tersusun dari kecil ke besar (atau besar ke kecil, sesuai kebutuhan).

- **Kelebihan**: Efisien dan tidak membutuhkan ruang tambahan.
- **Kekurangan**: Implementasi lebih kompleks dibandingkan algoritma lain.



4. Merge Sort :

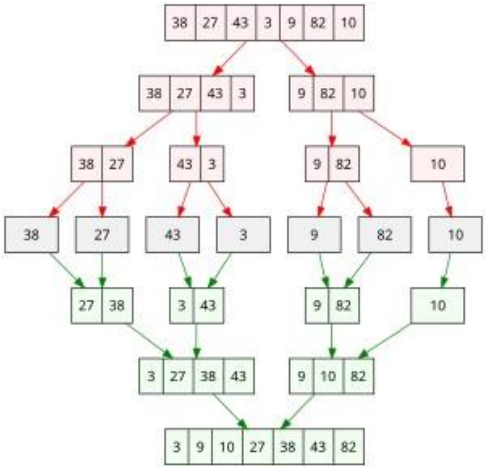
- **Pernyataan Masalah:**
  Mengurutkan data dengan membagi array menjadi dua bagian, mengurutkan tiap bagian, lalu menggabungkannya kembali.
- **Prinsip Kerja**: Membagi array menjadi dua bagian, mengurutkan kedua bagian, lalu menggabungkannya menjadi satu array yang terurut.
- **Penerapan:**

1. Pengurutan data besar.
2. Digunakan dalam penerapan seperti sistem basis data dan algoritma pencarian.

- **Langkah-langkah:**

1. Bagi array menjadi dua bagian sampai setiap bagian hanya berisi satu elemen.
2. Gabungkan dua bagian yang sudah terurut secara berurutan.
3. Ulangi proses penggabungan hingga semua elemen terurut.

- **Kelebihan**: Efisien untuk dataset besar.
- **Kekurangan**: Memerlukan ruang tambahan untuk penggabungan



5. Quick Sort:

- **Pernyataan Masalah:**
  Mengurutkan data dengan memilih elemen pivot, membagi data ke dua bagian, dan mengurutkan setiap bagian secara terpisah.

- **Prinsip Kerja**: Menggunakan pivot untuk membagi array menjadi dua bagian, kemudian mengurutkan bagian-bagian tersebut secara rekursif.

- **Penerapan:**

1. · Pengurutan data cepat untuk dataset besar.

2. · Banyak digunakan di penerapan seperti pengurutan file dan pencarian.

- **Langkah-langkah:**

1. Pilih elemen sebagai pivot

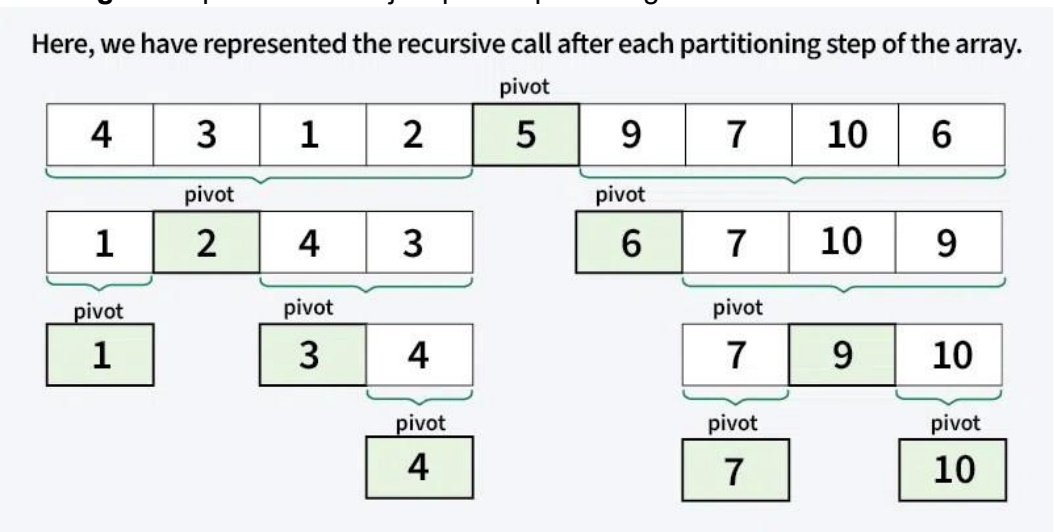
2. Pisahkan elemen yang lebih kecil dan lebih besar dari pivot ke dua bagian.

3. Ulangi proses tersebut untuk setiap bagian sampai array terurut.

4. Gabungkan bagian-bagian yang sudah terurut.

- **Kelebihan**: Cepat untuk dataset besar.

- **Kekurangan**: Dapat melambat jika pivot dipilih dengan buruk.



6. Counting Sort

- **Pernyataan Masalah:**
  Mengurutkan data dengan menghitung frekuensi kemunculan setiap elemen dan menyusunnya berdasarkan frekuensi tersebut.

- **Prinsip Kerja**: Mengurutkan dengan menghitung frekuensi setiap elemen, lalu menggunakan informasi frekuensi untuk menempatkan elemen dalam urutan yang benar.

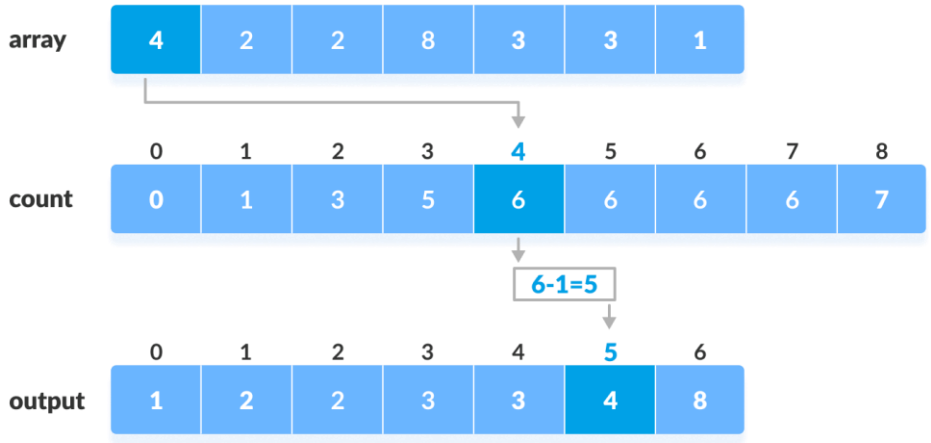
- **Penerapan:**

1. Pengurutan angka atau data dengan rentang nilai terbatas.
2. Sering digunakan dalam penerapan pengolahan gambar atau pengurutan dalam permainan.

- **Langkah-langkah:**

1. Temukan nilai maksimum dalam data.
2. Buat array untuk menghitung frekuensi setiap elemen.
3. Ubah penghitung frekuensi menjadi posisi elemen yang terurut.
4. Tempatkan elemen dalam urutan yang benar.
5. Salin hasilnya kembali ke array asli.
6. Data kini sudah terurut.

- **Kelebihan**: Sangat cepat untuk data dengan rentang nilai terbatas.
- **Kekurangan**: Kurang efisien untuk data dengan rentang nilai yang sangat besar.



Arrays Algorithms

Pengertian:
Algoritma yang bekerja dengan array (kumpulan data yang tersusun dalam satu dimensi atau lebih).

Fungsinya:
Mengelola data yang berurutan seperti daftar nilai, matriks, atau data yang mudah diakses berdasarkan indeks.

1. Kadane's Algorithm

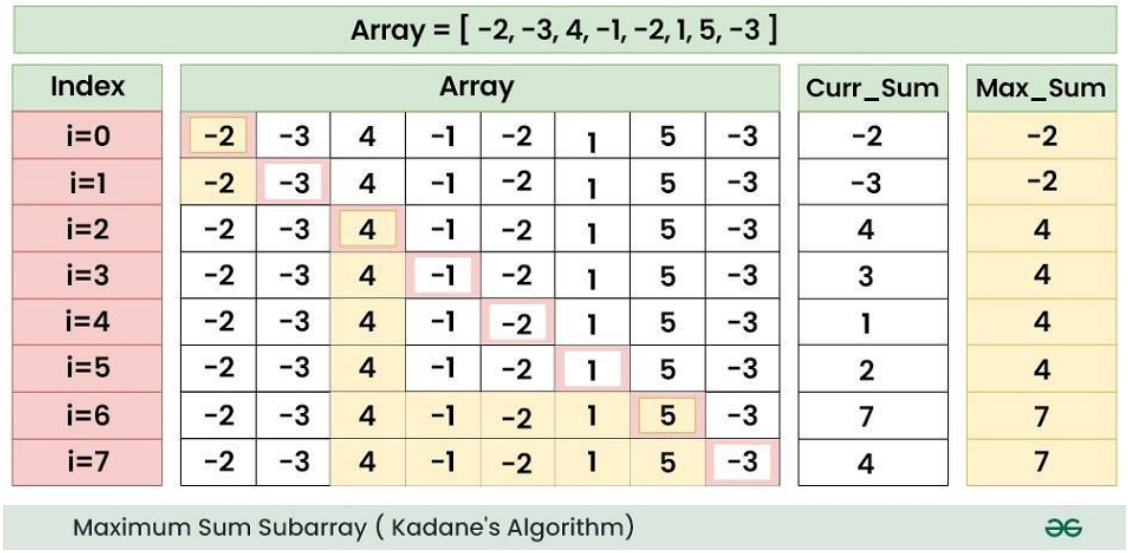
- **Pernyataan Masalah:** Algoritma Kadane digunakan untuk menemukan subarray dengan jumlah maksimum.
- **Prinsip Kerja**: Algoritma mencari subarray dengan jumlah terbesar dengan melacak jumlah sementara dan memperbarui hasil maksimal jika ditemukan jumlah yang lebih besar.
- **Penerapan:**

1. Digunakan dalam analisis keuangan untuk menemukan periode keuntungan maksimum.
2. Contoh nyata: Menentukan tren penjualan terbaik dalam periode tertentu.

- **Langkah-langkah:**

1. Mulai dengan dua variabel: max_current dan max_global, yang keduanya bernilai elemen pertama array.
2. Telusuri array mulai dari elemen kedua.
3. Perbarui max_current dengan nilai terbesar antara elemen saat ini atau elemen saat ini ditambah max_current.
4. Perbarui max_global jika max_current lebih besar dari max_global.
5. Ulangi langkah ini sampai seluruh array selesai ditelusuri.
6. Hasil akhirnya adalah max_global, yaitu jumlah maksimum subarray.

- **Kelebihan**: Sangat efisien untuk mencari subarray maksimal (O(n)).
- **Kekurangan**: Hanya dapat digunakan untuk kasus subarray yang jumlahnya positif atau nol.



2. Floyd's Algorithm (Cycle Detection)

- **Pernyataan Masalah:** Algoritma Floyd digunakan untuk mendeteksi siklus dalam linked list.

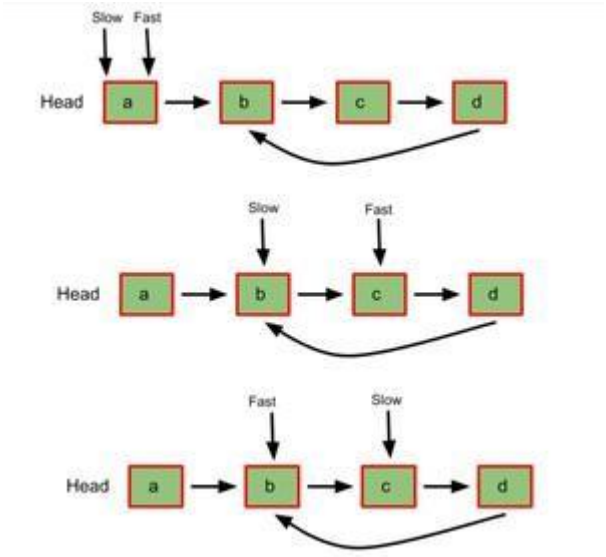
- **Prinsip Kerja**: Menggunakan dua pointer yang bergerak dengan kecepatan berbeda untuk mendeteksi siklus dalam struktur data seperti linked list.
- **Penerapan:**

1. Digunakan dalam debugging struktur data.
2. mendeteksi siklus pada urutan
3. Contoh nyata: Mendeteksi loop dalam sistem navigasi.

- **Langkah-langkah:**

1. Mulai dengan dua pointer, "lambat" dan "cepat," di posisi awal.
2. Gerakkan pointer lambat satu langkah, dan pointer cepat dua langkah.
3. Jika pointer lambat dan cepat bertemu, berarti ada siklus.
4. Jika pointer cepat mencapai akhir (null), berarti tidak ada siklus.
5. Untuk menemukan awal siklus, pindahkan pointer lambat ke awal, lalu gerakkan kedua pointer satu langkah per iterasi hingga bertemu.
6. Tempat pertemuan adalah awal siklus.

- **Kelebihan**: Memungkinkan deteksi siklus tanpa memerlukan banyak ruang memori.
- **Kekurangan**: Tidak dapat digunakan untuk graf berbobot atau struktur data yang lebih kompleks.



3. KMP Algorithm (Knuth-Morris-Pratt)

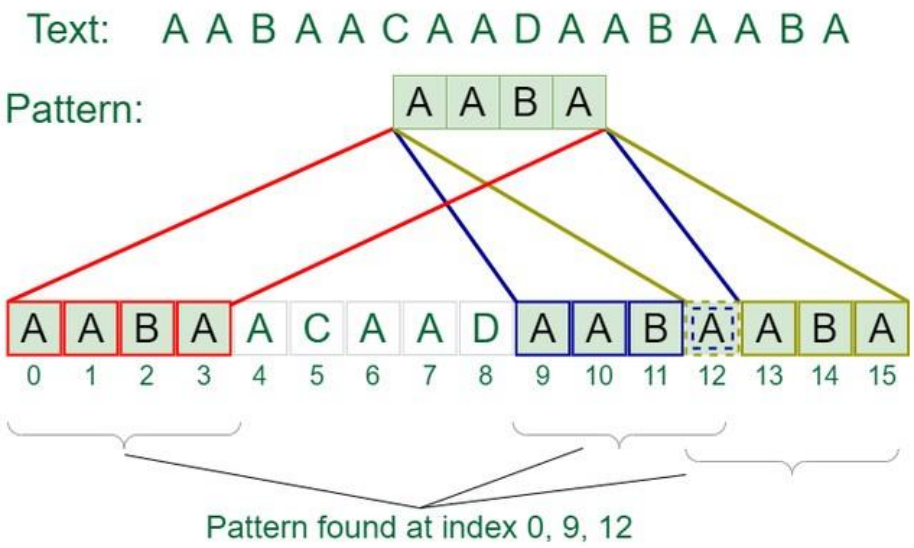
- **Pernyataan Masalah:** Algoritma KMP digunakan untuk mencari pola dalam string dengan cara yang efisien..
- **Prinsip Kerja**: Algoritma mencari substring dalam string dengan menghindari pemeriksaan ulang karakter yang sudah cocok, menggunakan informasi pola sebelumnya.
- **Penerapan:**

1. Digunakan dalam mesin pencari dan pengedit teks.
2. Contoh nyata: Menemukan kata kunci dalam dokumen teks besar.

- **Langkah-langkah:**

1. Buat LPS (*Longest Prefix Suffix*) array untuk pola.
2. Mulai telusuri teks dengan dua pointer: satu untuk teks (i) dan satu untuk pola (j).
3. Jika teks[i] sama dengan pola[j], pindahkan kedua pointer ke depan.
4. Jika j mencapai panjang pola, pola ditemukan di teks.
5. Jika teks[i] tidak cocok dengan pola[j], gunakan LPS untuk menentukan posisi baru pointer pola (j). Jika j = 0, pindahkan pointer teks saja.
6. Ulangi langkah ini sampai teks selesai ditelusuri.

- **Kelebihan**: Efisien untuk pencarian substring panjang dengan waktu yang lebih cepat dibandingkan pencarian biasa.
- **Kekurangan**: Memerlukan waktu tambahan untuk membangun tabel prefix yang bisa memperlambat untuk teks kecil.



4. Quick Select Algorithm

- **Pernyataan Masalah:** Algoritma ini digunakan untuk menemukan elemen ke-k terkecil dalam array tanpa harus mengurutkan seluruh array.
- **Prinsip Kerja**: Memilih pivot, mempartisi array, dan fokus pada bagian yang mengandung elemen ke-k.
- **Penerapan:**

1. Digunakan dalam pemrosesan data statistik.
2. Digunakan dalam penerapan analisis data besar.

3. Contoh nyata: Menentukan median dari dataset besar.

- **Langkah-langkah:**

1. Pilih elemen pivot secara acak dari array.
2. Pisahkan array menjadi dua bagian: elemen lebih kecil dari pivot di kiri, dan elemen lebih besar di kanan.
3. Periksa posisi pivot:
   - Jika posisi pivot sama dengan indeks yang dicari, proses selesai. Jika posisi pivot lebih kecil dari indeks yang dicari, cari di bagian kanan.
   - Jika posisi pivot lebih besar dari indeks yang dicari, cari di bagian kiri.
4. Ulangi proses hingga elemen yang dicari ditemukan.
5. Kembalikan elemen di posisi yang diminta sebagai hasil akhir.

- **Kelebihan**: Efisien untuk mencari elemen tanpa mengurutkan seluruh array. **Kekurangan**: Bisa melambat dalam kasus terburuk.

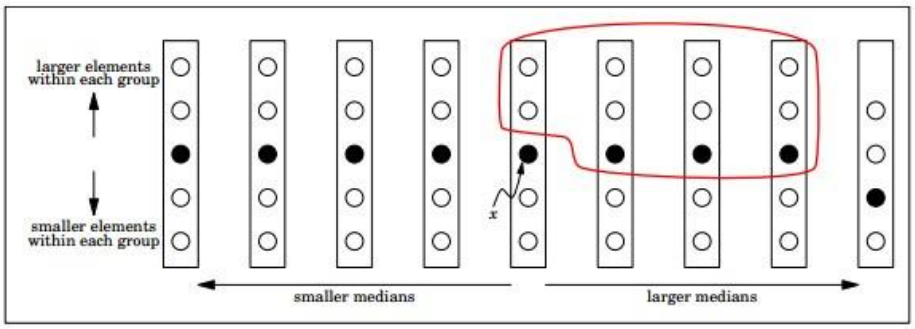


Figure 1.1. Imagine laying out the groups side by side, ordered with respect to their medians. Also imagine that each group is sorted from greatest to least, top to bottom. Then each of the elements inside the red curve is guaranteed to be greater than x.

5. Boyer-Moore Majority Vote Algorithm

- **Pernyataan Masalah:** Algoritma ini digunakan untuk menemukan elemen mayoritas dalam array (elemen yang muncul lebih dari separuh total elemen).
- **Prinsip Kerja**: Menggunakan kandidat dan penghitung untuk menemukan elemen mayoritas dalam satu lintasan array.
- **Penerapan:**

1. Digunakan dalam analisis data untuk menemukan nilai yang dominan.
2. Diterapkan dalam pemilihan suara atau survei.
3. Contoh nyata: Menentukan kandidat pemenang dalam pemilihan dengan suara mayoritas.

- **Langkah-langkah:**

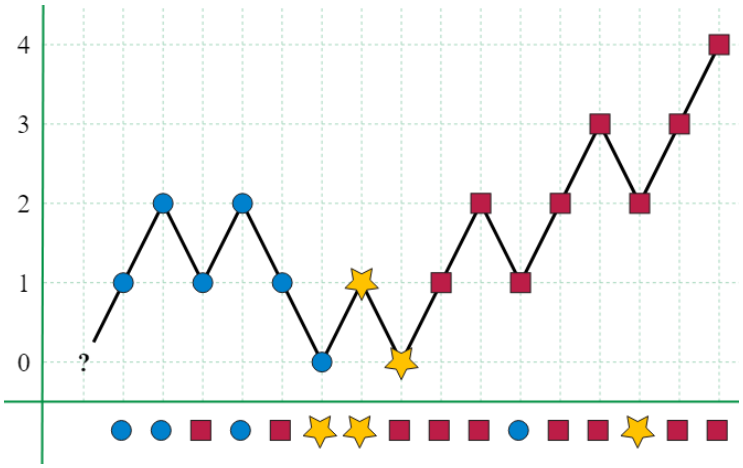
1. Mulai dengan variabel **kandidat** dan **count**.
2. Iterasi array: jika **count** 0, ganti kandidat dengan elemen saat ini.

3. Jika elemen cocok dengan **kandidat**, tambahkan **count**. Jika tidak, kurangi **count**.
4. Setelah iterasi selesai, **kandidat** adalah elemen yang paling sering muncul.
5. Cek lagi dengan menghitung berapa kali "calon terbanyak" muncul di array:

   * Jika jumlahnya lebih dari setengah total elemen, berarti itu adalah elemen terbanyak.

   *Jika tidak, berarti tidak ada elemen yang mendominasi.

- **Kelebihan**: Sangat efisien dan memerlukan ruang yang sedikit.
- **Kekurangan**: Hanya dapat digunakan untuk mencari mayoritas yang lebih dari 50% dalam array.



Graphs Algorithms

- **Pengertian:**

Algoritma yang digunakan untuk menyelesaikan masalah pada struktur graf, yaitu kumpulan titik (node) yang terhubung oleh garis (edge).

- **Fungsinya:**
  Menyelesaikan masalah seperti rute perjalanan, jaringan komputer, atau pencarian jalur optimal.

1. Kruskal's Algorithm

- **Pernyataan Masalah:** Algoritma Kruskal digunakan untuk menemukan pohon rentang minimum (MST) dalam graf berbobot.
- **Prinsip Kerja**: Mengurutkan sisi berdasarkan bobot, lalu menambahkan sisi ke pohon jika tidak membentuk siklus.
- **Penerapan:**

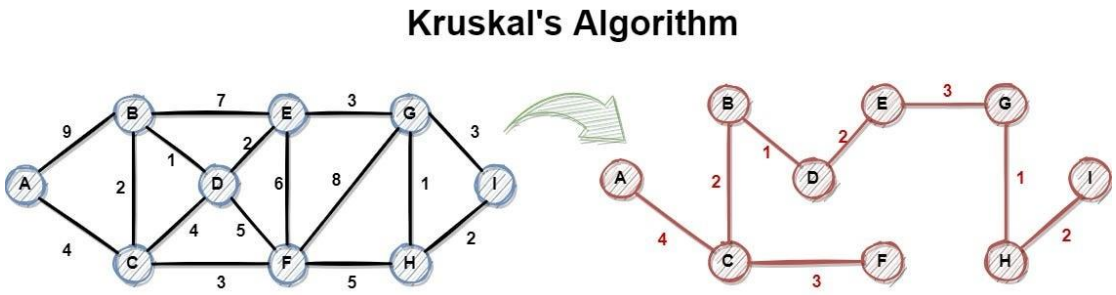
1. Digunakan untuk membangun jaringan seperti jaringan listrik atau telekomunikasi dengan biaya minimal.
2. Digunakan dalam perancangan rute jaringan.

3. Contoh nyata: Menghubungkan kota-kota dengan biaya konstruksi jalan minimum.

- **Langkah-langkah:**

1. Urutkan semua sisi graf berdasarkan bobotnya, dari yang terkecil hingga terbesar.
2. Pilih sisi dengan bobot terkecil dan tambahkan ke MST jika tidak membentuk siklus. Gunakan struktur data *union-find* untuk memeriksa siklus.
3. Ulangi hingga semua node terhubung atau jumlah sisi mencapai V−1V - 1 (dengan VV adalah jumlah node).
4. Kembalikan pohon MST dengan total bobot terkecil.

- **Kelebihan**: Mudah digunakan untuk menemukan jarak terpendek di graf yang memiliki banyak simpul.
- **Kekurangan**: Bisa memakan banyak waktu dan ruang jika grafnya sangat besar.



2. Dijkstra's Algorithm

- **Pernyataan Masalah:** Algoritma Dijkstra digunakan untuk menemukan jalur terpendek dari satu simpul ke simpul lainnya dalam graf berbobot positif.
- **Prinsip Kerja**: Memulai dari simpul sumber, memperbarui jarak terpendek, dan memilih simpul dengan jarak minimum hingga selesai.
- **Penerapan:**

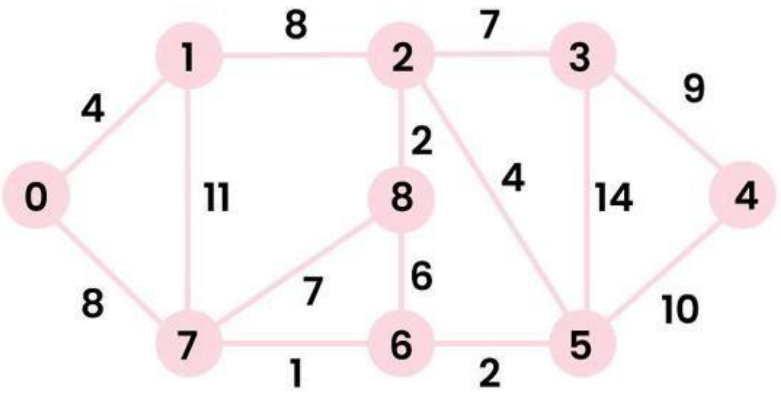
1. Digunakan dalam sistem navigasi GPS untuk mencari rute tercepat.
2. Digunakan dalam penerapan peta dan pencarian rute.
3. Contoh nyata: Menentukan rute tercepat dalam penerapan seperti Google Maps.

- **Langkah-langkah:**

1. Tentukan jarak awal untuk semua simpul (set 0 untuk simpul asal, tak terhingga untuk yang lain).
2. Pilih simpul dengan jarak terkecil, lalu perbarui jarak ke simpul-simpul yang terhubung.
3. Perbarui jarak ke simpul yang terhubung dengan simpul saat ini jika melalui simpul ini lebih pendek.
4. Tandai simpul ini sebagai sudah dikunjungi.
5. Ulangi hingga menemukan jalur terpendek untuk semua simpul.

6. Hasilnya adalah jarak terpendek dari simpul awal ke semua simpul.

- **Kelebihan**: Cepat untuk graf berbobot positif.
- **Kekurangan**: Tidak menangani bobot negatif.



Working of Dijkstra's Algorithm

3. Bellman-Ford Algorithm

- **Pernyataan Masalah:** Algoritma ini digunakan untuk menemukan jalur terpendek antara satu titik sumber dengan semua titik lainnya dalam suatu grafik
- **Prinsip Kerja**
  Algoritma memperbarui jarak terpendek untuk setiap simpul dengan merelaksasi semua sisi graf secara berulang, lalu memeriksa siklus berbobot negatif.
- **Penerapan:**

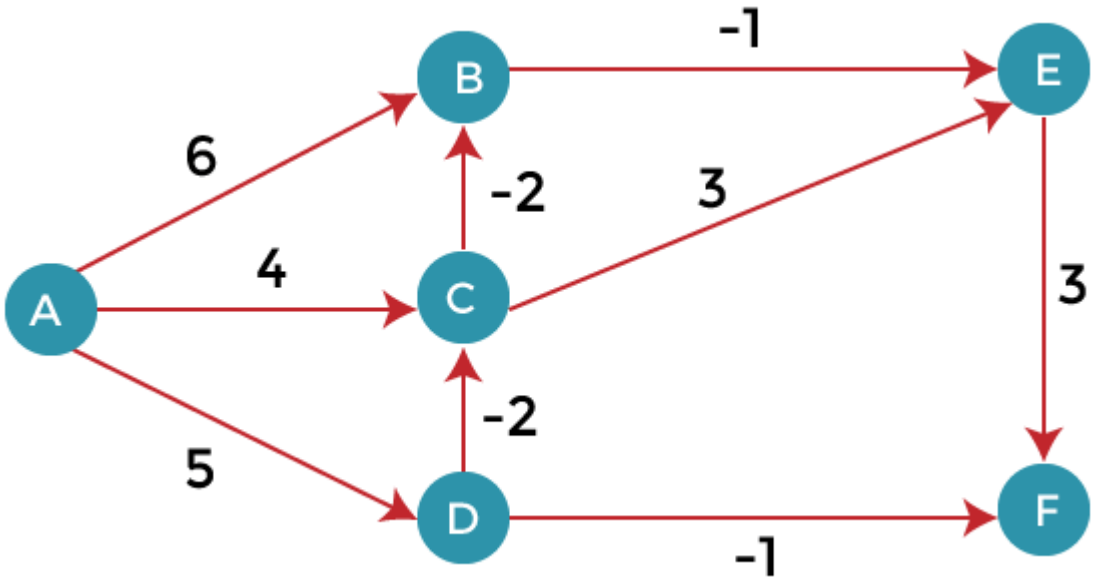
1. Digunakan dalam analisis jaringan transportasi.
2. Menentukan biaya transportasi minimum dengan diskon khusus.
3. Menemukan kemungkinan menghasilkan keuntungan melalui pertukaran mata uang.

- **Langkah-langkah:**

1. Set jarak semua simpul ke tak hingga, kecuali simpul awal (0).
2. Ulangi perbarui jarak sisi sebanyak jumlah simpul - 1.
3. Jika ada jalur lebih pendek, perbarui jarak.
4. Cek siklus negatif: jika jarak masih bisa diperbarui, laporkan.
5. Hasil: jarak terpendek dari simpul awal ke semua simpul.

- **Kelebihan & Kekurangan**

1. **Kelebihan**: Dapat menangani bobot negatif.
2. **Kekurangan**: Kurang efisien dibanding algoritma Dijkstra untuk graf tanpa bobot negatif.



4. Floyd-Warshall Algorithm

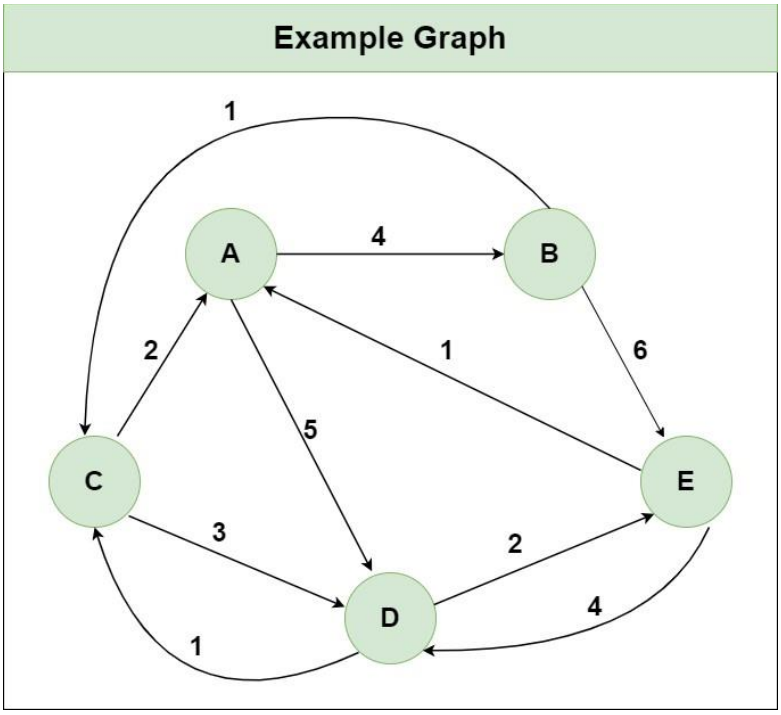
- **Pernyataan Masalah:** Algoritma ini digunakan untuk menemukan jalur terpendek antara semua pasangan simpul dalam graf berbobot.
- **Prinsip Kerja :** Algoritma memeriksa setiap pasangan simpul dengan mempertimbangkan semua simpul sebagai perantara untuk menemukan jalur terpendek.
- **Penerapan:**

1. Digunakan dalam analisis jaringan komunikasi
2. Pencarian rute terpendek di jaringan transportasi
3. Deteksi hubungan terpendek dalam graf sosial

- **Langkah-langkah:**

1. Buat matriks jarak awal berdasarkan bobot sisi graf, isi dengan ∞ jika tidak ada jalur langsung.
2. Set jarak dari simpul ke dirinya sendiri menjadi 0.
3. Lewati setiap simpul sebagai perantara.
4. Perbarui jarak antar dua simpul jika melewati perantara menghasilkan jalur lebih pendek.
5. Matriks akhir menunjukkan jarak terpendek antara semua pasangan simpul.

- **Kelebihan**: Sangat efektif untuk menemukan jarak terpendek di graf dengan banyak pasangan simpul, terutama untuk graf yang padat.
- **Kekurangan**: Memerlukan ruang dan waktu yang lebih banyak jika graf sangat besar.



5. Topological Sort

- **Pernyataan Masalah:** Algoritma ini digunakan untuk menyusun urutan tugas berdasarkan dependensi dalam graf berarah tanpa siklus (DAG).
- **Prinsip Kerja**: Algoritma memproses simpul dengan derajat masuk 0, menambahkannya ke urutan, dan mengulangi proses hingga semua simpul selesai diproses.
- **Penerapan:**

1. Digunakan dalam penjadwalan tugas.
2. Menentukan urutan kompilasi proyek perangkat lunak.
3. Mengatur urutan pengambilan mata kuliah sesuai prasyarat, seperti mata kuliah dasar sebelum lanjut.

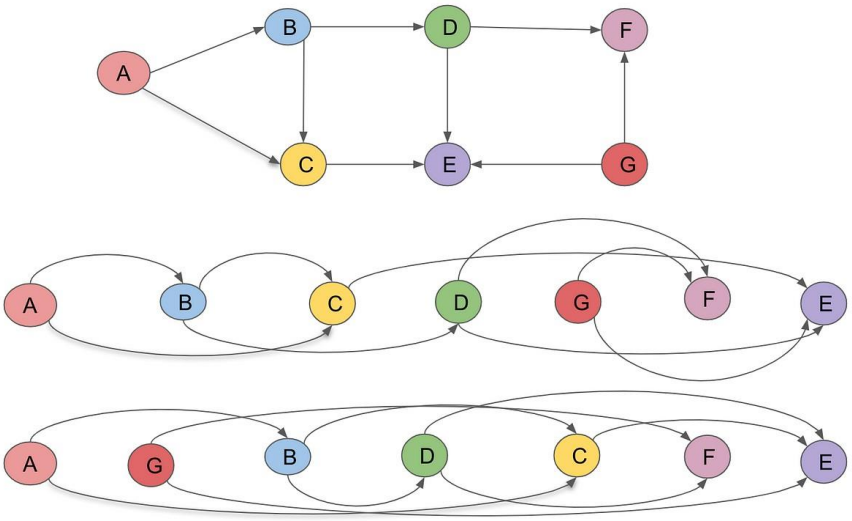
- **Langkah-langkah:**

1. Hitung jumlah sisi yang masuk ke setiap simpul.
2. Masukkan simpul yang tidak memiliki sisi masuk ke dalam antrian.
3. Ambil simpul dari antrian, tambahkan ke hasil, dan kurangi jumlah sisi masuk pada simpul-simpul yang terhubung.
4. Jika sisi masuk simpul-simpul terhubung jadi 0, masukkan simpul tersebut ke antrian.
5. Ulangi hingga semua simpul diproses.
6. Urutan simpul yang diproses adalah hasil topological sort.

○ **Kelebihan & Kekurangan**

2. **Kelebihan**: Mudah diimplementasikan untuk DAG.

3. **Kekurangan**: Tidak dapat diterapkan untuk graf dengan siklus.



Topological Sort

6. Flood Fill Algorithm

- **Pernyataan Masalah:** Flood Fill adalah algoritma untuk mengisi area dengan warna tertentu, sering digunakan dalam grafika komputer seperti aplikasi paint.
- **Prinsip Kerja:** Algoritma ini bekerja dengan memeriksa suatu piksel, mengisi warna, dan melanjutkan ke piksel tetangga hingga area yang diinginkan terisi.
- **Penerapan:**

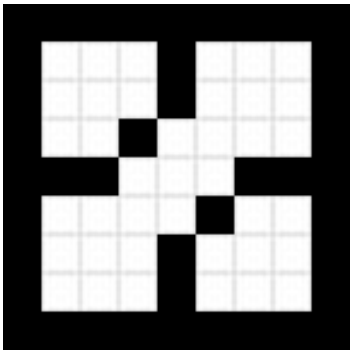
1. Digunakan dalam perangkat lunak pengeditan gambar.
2. Membatasi area permainan berdasarkan zona.
3. Mengisi warna pada gambar.

- **Langkah-langkah:**

1. Mulai dari titik awal di grid.
2. Ganti warna titik awal menjadi warna target.
3. Periksa pixel di atas, bawah, kiri, dan kanan. Jika warnanya sama dengan warna awal, ganti dengan warna target.
4. Ulangi langkah 3 untuk setiap pixel baru hingga semua area terhubung selesai diwarnai.

- **Kelebihan & Kekurangan**

1. **Kelebihan**: Sederhana dan efektif untuk area kecil.
2. **Kekurangan**: Rentan terhadap stack overflow pada area besar saat menggunakan metode rekursif.



7. Lee Algorithm

- **Pernyataan Masalah:** Algoritma ini digunakan menemukan jalur terpendek dalam graf berbentuk grid (misalnya, labirin).
- **Prinsip Kerja:** Algoritma memulai dari posisi awal dan menyebar ke sel tetangga (atas, bawah, kiri, kanan) secara bertahap, sambil menghitung langkah hingga mencapai tujuan.
- **Penerapan:**

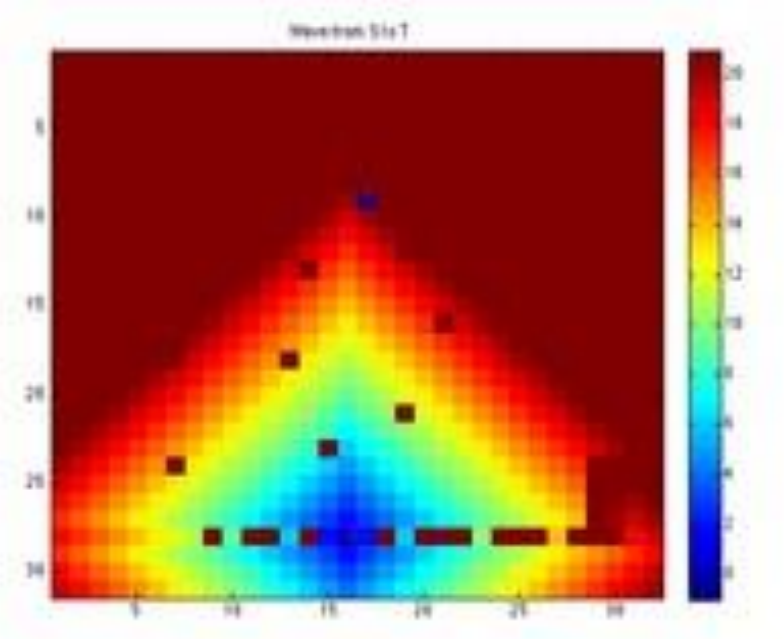
1. Digunakan dalam penerapan navigasi berbasis grid.
2. Perancangan jalur robot dalam lingkungan berbasis grid.
3. Menemukan konektivitas dalam gambar biner.

- **Langkah-langkah:**

1. Mulai dari titik awal dan beri tanda angka 0 di sana.
2. Cek semua titik di sekitarnya (atas, bawah, kiri, kanan) dan beri angka 1.
3. Ulangi dengan memberi angka yang lebih besar untuk titik-titik berikutnya sampai mencapai tujuan.
4. Jika sudah sampai tujuan, angka di sana adalah jarak terpendek.

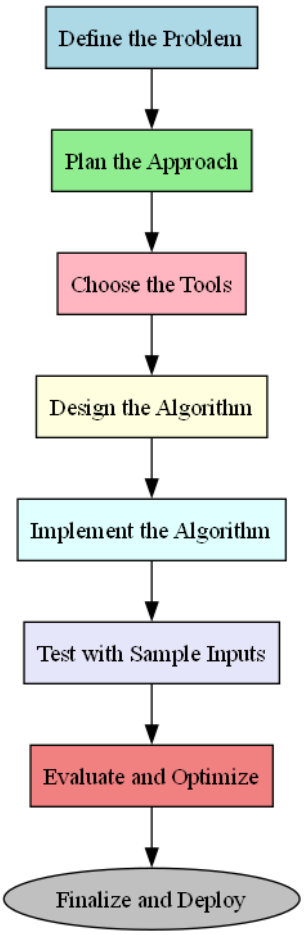
- **Kelebihan & Kekurangan**

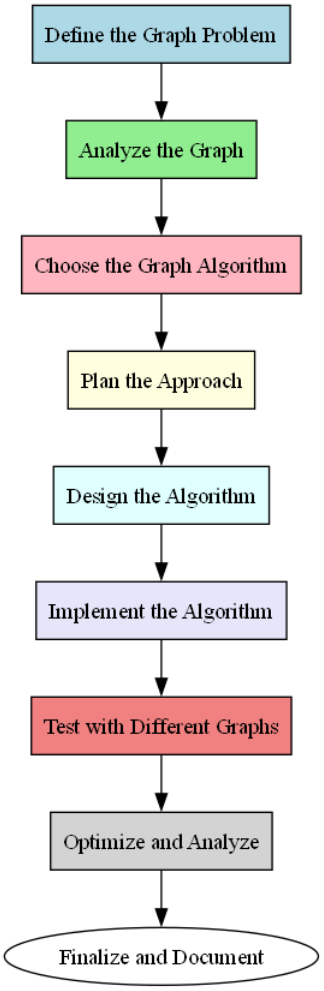
1. **Kelebihan**: Memberikan solusi optimal untuk grid berbobot seragam.
2. **Kekurangan**: Tidak efisien untuk grid besar karena membutuhkan memori tinggi.

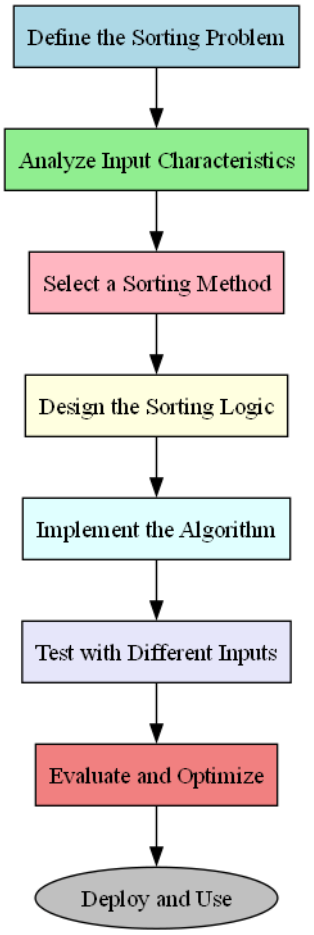


TASK 2

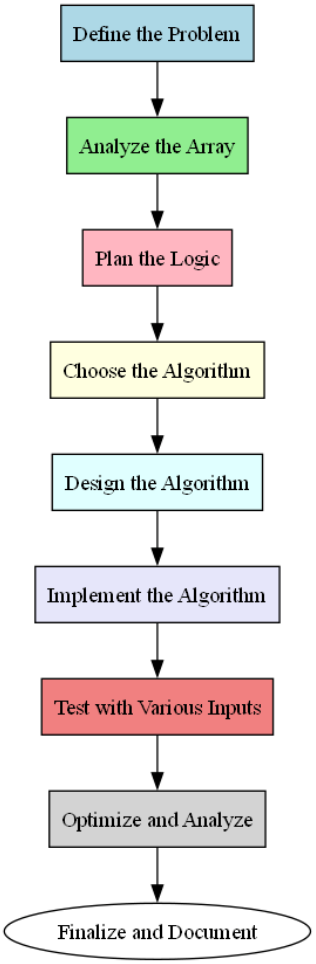
The flowchart to visualize the step-by-step process for each algorithm:

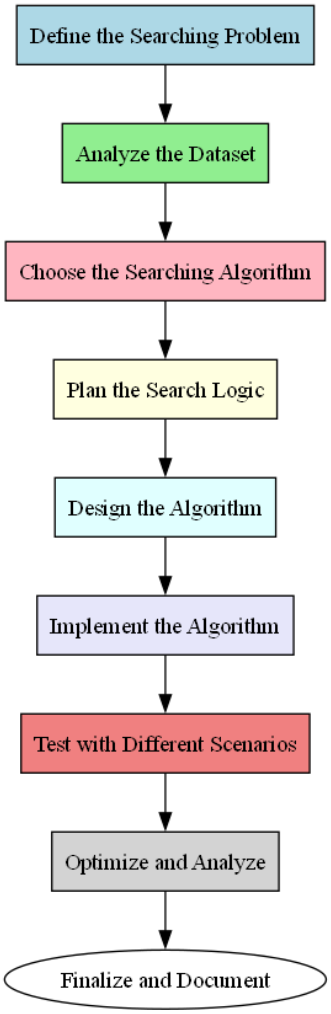
Basic Algorithm	

Graph Algorithm	

Sorting Algorithm	

Arrays Algorithm



Searching Algorithm	

TASK 3

Algorithms	Time Complexity			Space complexity
	Best	Average	Worst	
Linear Search	O(1)	$O(n)$	$O(n)$	O(1)
Binary Search	O(1)	$O(\log n)$	$O(\log n)$	O(1)
Jump Search	O(1)	$O(\sqrt{n}$	$O(\sqrt{n}$	O(1)
Exponential Search	O(1)	$O(\log i)$	$O(\log n)$	O(1)
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Kadane's Algorithm	$O(n)$	$O(n)$	$O(n)$	O(1)
Floyd's Cycle Detection	$O(n)$	$O(n)$	$O(n)$	O(1)
KMP Algorithm	$O(m+n)$	$O(m+n)$	$O(m+n)$	$O(m)$
Quick Select	$O(n)$	$O(n)$	$O(n^2)$	O(1)
Breadth-First Search (BFS)	$O(\mathcal{V}+E)$	$O(\mathcal{V}+E)$	$O(\mathcal{V}+E)$	$O(\mathcal{V})$
Depth-First Search (DFS)	$O(\mathcal{V}+E)$	$O(\mathcal{V}+E)$	$O(\mathcal{V}+E)$	$O(\mathcal{V})$
Dijkstra's Algorithm	$O(\mathcal{V}^2)$	$O(\mathcal{V}^2)$	$O(\mathcal{V}^2)$	$O(\mathcal{V})$
Bellman-Ford Algorithm	$O(\mathcal{V}E)$	$O(\mathcal{V}E)$	$O(\mathcal{V}E)$	$O(\mathcal{V})$
Kruskal's Algorithm	$O(E \log E)$	$O(E \log E)$	$O(E \log E)$	$O(E)$

TASK 4

Python Implementation:

1. Basic Algorithms:

- Euclid's Algorithm untuk menghitung GCD

```
def euclids_algorithm(self, a, b):
    """
    Euclid's Algorithm to calculate GCD (Greatest Common Divisor)
    :param a: First number
    :param b: Second number
    :return: GCD of a and b
    """
    while b != 0:
        a, b = b, a % b
    return a
```

- Huffman Coding untuk analisis frekuensi karakter

```
def huffman_coding(self, data):
    """
    Huffman Coding to find character frequencies (simple example)
    :param data: Input string
    :return: Frequency dictionary
    """
    from collections import Counter
    frequency = Counter(data)
    return sorted(frequency.items(), key=lambda x: x[1])
```

- Union-Find Algorithm untuk pengelolaan himpunan terpisah

```
def union_find(self, n):
    """
    Simple Union-Find implementation
    :param n: Number of elements
    :return: Parent array after union-find operations
    """
    parent = [i for i in range(n)]

    def find(x):
        if parent[x] == x:
            return x
        return find(parent[x])

    def union(x, y):
        root_x = find(x)
        root_y = find(y)
        if root_x != root_y:
            parent[root_y] = root_x

    return parent
```

Algoritma ini mendemonstrasikan dasar logika pemrograman yang efisien untuk menyelesaikan masalah sehari-hari.

2.

Sorting Algorithms:

- Bubble Sort

```
def bubble_sort(self, arr):
    """
    Bubble Sort Algorithm
    :param arr: List of numbers to sort
    :return: Sorted list
    """
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

- Merge Sort

```
def merge_sort(self, arr):
    """
    Merge Sort Algorithm
    :param arr: List of numbers to sort
    :return: Sorted list
    """
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        self.merge_sort(L)
        self.merge_sort(R)

        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
    return arr
```

- Quick Sort

```
def quick_sort(self, arr):
    """
    Quick Sort Algorithm
    :param arr: List of numbers to sort
    :return: Sorted list
    """
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return self.quick_sort(left) + middle + self.quick_sort(right)
```

Sorting ini penting untuk berbagai aplikasi seperti pengelolaan data dalam database atau pemrosesan big data.

3.

Array Algorithms:

- Kadane's Algorithm untuk mencari subarray dengan jumlah maksimum

```python
def kadanes_algorithm(self, nums):
    """
    Kadane's Algorithm to find the maximum subarray sum
    :param nums: List of integers
    :return: Maximum sum
    """
    max_current = max_global = nums[0]
    for i in range(1, len(nums)):
        max_current = max(nums[i], max_current + nums[i])
        if max_current > max_global:
            max_global = max_current
    return max_global
```

- KMP Algorithm untuk pencocokan pola string

```python
def kmp_algorithm(self, text, pattern):
    """
    Knuth-Morris-Pratt (KMP) Algorithm for pattern matching
    :param text: The main text
    :param pattern: The pattern to find
    :return: Starting index of the pattern in text, -1 if not found
    """
    def compute_lps(pattern):
        lps = [0] * len(pattern)
        length = 0
        i = 1
        while i < len(pattern):
            if pattern[i] == pattern[length]:
                length += 1
                lps[i] = length
                i += 1
            elif length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
        return lps

    lps = compute_lps(pattern)
    i = j = 0
    while i < len(text):
        if text[i] == pattern[j]:
            i += 1
            j += 1
        if j == len(pattern):
            return i - j
        elif i < len(text) and text[i] != pattern[j]:
            if j != 0:
                j = lps[j - 1]
            else:
                i += 1
    return -1
```

- Floyd’s Cycle Detection untuk mendeteksi siklus pada linked list

```python
def floyd_cycle_detection(self, head):
    """
    Floyd's Cycle Detection Algorithm
    :param head: Head of the linked list
    :return: True if a cycle is detected, False otherwise
    """
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

Algoritma ini sangat relevan untuk pemrosesan array dan manipulasi data string.

Graph Algorithms:

- BFS

4.

```python
def bfs(self, graph, start):
    """
    Breadth-First Search (BFS)
    :param graph: Graph represented as adjacency list
    :param start: Starting node
    :return: List of visited nodes in BFS order
    """
    visited = set()
    queue = deque([start])
    result = []

    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            result.append(node)
            queue.extend(graph[node])

    return result
```

- DFS

```python
def dfs(self, graph, start, visited=None):
    """
    Depth-First Search (DFS)
    :param graph: Graph represented as adjacency list
    :param start: Starting node
    :param visited: Set of visited nodes
    :return: List of visited nodes in DFS order
    """
    if visited is None:
        visited = set()
    visited.add(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            self.dfs(graph, neighbor, visited)
    return visited
```

- Dijkstra's Algorithm

```python
def dijkstra(self, graph, start):
    """
    Dijkstra's Algorithm for shortest path
    :param graph: Graph represented as adjacency list
    :param start: Starting node
    :return: Shortest distance to all nodes
    """
    import heapq
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
```

Graph sering digunakan dalam pemodelan jaringan, seperti jaringan komputer atau peta.

Searching Algorithms:

- Linear Search

5.

```python
def linear_search(self, arr, target):
    """
    Linear Search Algorithm
    :param arr: List of elements
    :param target: Element to search
    :return: Index of the target if found, else
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

- Binary Search

```python
def binary_search(self, arr, target):
    """
    Binary Search Algorithm
    :param arr: Sorted list of elements
    :param target: Element to search
    :return: Index of the target if found, else -1
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Algoritma pencarian sangat mendasar dan penting dalam pengolahan data.

For more information: https://github.com/neshiahilton/Group_4_2024_UAS_SIPW1001