REPUBLIK INDONESIA
KEMENTERIAN HUKUM

SURAT PENCATATAN CIPTAAN

Dalam rangka pelindungan ciptaan di bidang ilmu pengetahuan, seni dan sastra berdasarkan Undang-Undang Nomor 28 Tahun 2014 tentang Hak Cipta, dengan ini menerangkan:

Nomor dan tanggal permohonan	:	EC002025111560, 13 Agustus 2025
Pencipta		
Nama	:	**M. Rhifky Wayahdi, Phie Chyan dkk**
Alamat	:	Jl. Umar No. 30-26, Kel. Glugur Darat 1, Medan Timur, Kota Medan, Sumatera Utara, 20238
Kewarganegaraan	:	Indonesia
Pemegang Hak Cipta		
Nama	:	**PT. Mifandi Mandiri Digital**
Alamat	:	Jl. Payanibung, Dalu 10 B, Tanjung Morawa, Kab. Deli Serdang, Sumatera Utara, 20372
Kewarganegaraan	:	Indonesia
Jenis Ciptaan	:	**Buku**
Judul Ciptaan	:	**Logika dan Algoritma Pemrograman: Fondasi Dasar Menjadi Programmer Andal**
Tanggal dan tempat diumumkan untuk pertama kali di wilayah Indonesia atau di luar wilayah Indonesia	:	1 Juli 2025, di Kota Medan
Jangka waktu pelindungan	:	Berlaku selama 50 (lima puluh) tahun sejak Ciptaan tersebut pertama kali dilakukan Pengumuman.
Nomor Pencatatan	:	000951821

adalah benar berdasarkan keterangan yang diberikan oleh Pemohon.
Surat Pencatatan Hak Cipta atau produk Hak terkait ini sesuai dengan Pasal 72 Undang-Undang Nomor 28 Tahun 2014 tentang Hak Cipta.

a.n. MENTERI HUKUM
DIREKTUR JENDERAL KEKAYAAN INTELEKTUAL
u.b
Direktur Hak Cipta dan Desain Industri

Agung Damarsasongko,SH.,MH.
NIP. 196912261994031001

Balai Besar Sertifikasi Elektronik

Disclaimer:
1. Dalam hal pemohon memberikan keterangan tidak sesuai dengan surat pernyataan, Menteri berwenang untuk mencabut surat pencatatan permohonan.
2. Surat Pencatatan ini telah disegel secara elektronik menggunakan segel elektronik yang ditertibkan oleh Balai Besar Sertifikasi Elektronik, Badan Siber dan Sandi Negara.
3. Surat Pencatatan ini dapat dibuktikan keasliannya dengan memindai kode QR pada dokumen ini dan informasi akan ditampilkan dalam browser.

LAMPIRAN PENCIPTA

No	Nama	Alamat
1	M. Rhifky Wayahdi	Jl. Umar No. 30-26, Kel. Glugur Darat 1 Medan Timur, Kota Medan
2	Phie Chyan	Jl. Nuri Lama No 23c, RT/RW: 003/005, Kel. Mariso Mariso, Kota Makassar
3	Agung Yuliyanto Nugroho	Getas Kalongan, Desa Tlogoadi Mlati, Kab. Sleman
4	Amril Mutoi Siregar	Perum BCL Jl Melati 1 Blok A9 No. 18, RT/RW: 006/010, Desa Waluya Cikarang Utara, Kab. Bekasi
5	Hansi Effendi	Jl. Bondo No.4, RT/RW: 003/001, Kel. Air Tawar Barat Padang Utara, Kota Padang
6	Satriawaty Mallu	Dsn. Pangi, RT/RW: 001/001, Desa Pangi Bajo, Kab. Luwu
7	Samsuriah	BTP Blok H Baru No.623, RT/RW: 001/007, Kel. Buntusu Tamalanrea, Kota Makassar
8	Dianta Ginting	Jl. Raya Hankam No.5, RT/RW: 002/003, Kel. Jatimurni Pondokmelati, Kota Bekasi
9	Ni Putu Linda Santiari	Jalan Bedahulu Utama No 14 Banjar Prajasari, Kel. Peguyangan Denpasar Utara, Kota Denpasar
10	Ismi Rizqa Lina	Dusun Krajan II, RT/RW: 005/002, Desa Setail Genteng, Kab. Banyuwangi
11	Waris Marsisno	Jl. Haji Enang B/25, RT/RW: 005/001, Kel. Cisalak Pasar Cimanggis, Kota Depok
12	Fahmi Ruziq	Jl. Sei Musi No. 48D, Kel. Babura Sunggal Medan Sunggal, Kota Medan
13	Mardiah	Jl. Taqwa Lr. Madya III No.95, RT/RW: 053/006, Kel. Sei Selincah Kalidoni, Kota Palembang
14	Anis Fitri Nur Masruriyah	Bubulak Paracis, RT/RW: 013/10, Kel. Tanjungpura Karawang Barat, Kab. Karawang
15	Santi Prayudani	Asr. Widuri Blok Damar Laut No 425, Kel. Harjosari II Medan Amplas, Kota Medan
16	Sri Retnowati	Jerukgulung, RT/RW: 004/004, Desa Jatiluhur Karanganyar, Kab. Kebumen
17	Ari Widiastono	Desa Fiditan Pulau Dullah Utara, Kota Tual
18	Dita Novita Sari	Jl. Budi Utomo No.191, RT/RW: 001/005, Kel. Gadingrejo Gading Rejo, Kab. Pringsewu

19	A.Kachsyfur Djasim Ilyas Paenrongi	Jl. Keberkahan Blok AD No.442 D, RT/RW: 002/004, Kel. Katimbang Biringkanaya, Kota Makassar
20	Triana Dewi Salma	Jl. Wijaya Kusuma gg 23/1, RT/RW: 007/005, Kel. Kudaile Slawi, Kab. Tegal

SERTIFIKAT

PENULIS

No. 017/MMD/Ser/VII/2025

SERTIFIKAT INI DIBERIKAN KEPADA

Ari Widiastono

Sebagai Penulis dalam Buku Berjudul

"Logika dan Algoritma Pemrograman"

Terbitan PT. Mifandi Mandiri Digital

Medan, 01 Juli 2025

MIFANDI MANDIRI DIGITAL
SUMATERA UTARA

Miftahul Jannah

Pimpinan Penerbit

Logika dan Algoritma

Pemrograman

Fondasi Dasar Menjadi Programmer Andal

M Rhifky Wayahdi, Phie Chyan, Agung Yuliyanto Nugroho,
Amril Mutoi Siregar, Hansi Effendi, Satriawaty Mallu, Samsuriah,
Dianta Ginting, Ni Putu Linda Santiari, Ismi Rizqa Lina, Waris Marsisno,
Fahmi Ruziq, Mardiah, Anis Fitri Nur Masruriyah, Santi Prayudani,
Sri Retnowati, Ari Widiastono, Dita Novita Sari,
A.Kachsyfur Djasim Ilyas Paenrongi, Triana Dewi Salma

Logika dan Algoritma Pemrograman

Fondasi Dasar Menjadi Programmer Andal

M Rhifky Wayahdi, Phie Chyan, Agung Yuliyanto Nugroho, Amril Mutoi Siregar, Hansi Effendi, Satriawaty Mallu, Samsuriah, Dianta Ginting, Ni Putu Linda Santiari, Ismi Rizqa Lina, Waris Marsisno, Fahmi Ruziq, Mardiah, Anis Fitri Nur Masruriyah, Santi Prayudani, Sri Retnowati, Ari Widiastono, Dita Novita Sari, A.Kachsyfur Djasim Ilyas Paenrongi, Triana Dewi Salma

PT. MIFANDI MANDIRI DIGITAL

Undang-Undang No. 28 Tahun 2014 Tentang Hak Cipta:

1. Setiap Orang yang dengan tanpa hak melakukan pelanggaran hak ekonomi sebagaimana dimaksud dalam pasal 9 ayat (1) huruf i untuk penggunaan secara komersial dipidana dengan pidana penjara paling lama 1 (satu) tahun dan/atau pidana denda paling banyak Rp. 100.000.000,- (seratus juta rupiah).
2. Setiap Orang yang dengan tanpa hak dan/atau tanpa izin Pencipta atau pemegang Hak Cipta melakukan pelanggaran hak ekonomi pencipta sebagaimana dimaksud dalam Pasal 9 ayat (1) huruf c, huruf d, huruf f, dan/atau huruf h untuk penggunaan secara komersial dipidana dengan pidana penjara paling lama 3 (tiga) tahun dan/atau pidana denda paling banyak Rp. 500.000.000,- (lima ratus juta rupiah).
3. Setiap Orang yang dengan tanpa hak dan/atau tanpa izin pencipta atau pemegang Hak Cipta melakukan pelanggaran hak ekonomi pencipta sebagaimana dimaksud dalam Pasal 9 ayat (1) huruf a, huruf b, huruf e, dan/atau huruf g untuk penggunaan secara komersial dipidana dengan pidana penjara paling lama 4 (empat) tahun dan/ atau pidana denda paling banyak Rp. 1.000.000.000,- (satu miliar rupiah).
4. Setiap Orang yang memenuhi unsur sebagaimana dimaksud pada ayat (3) yang dilakukan dalam bentuk pembajakan, dipidana dengan pidana penjara paling lama 10 (sepuluh) tahun dan/atau pidana denda paling banyak Rp. 4.000.000.000,- (empat miliar rupiah).

Logika dan Algoritma Pemrograman
Fondasi Dasar Menjadi Programmer Andal

M Rhifky Wayahdi, Phie Chyan, Agung Yuliyanto Nugroho, Amril Mutoi Siregar, Hansi Effendi, Satriawaty Mallu, Samsuriah, Dianta Ginting, Ni Putu Linda Santiari, Ismi Rizqa Lina, Waris Marsisno, Fahmi Ruziq, Mardiah, Anis Fitri Nur Masruriyah, Santi Prayudani, Sri Retnowati, Ari Widiastono, Dita Novita Sari, A.Kachsyfur Djasim Ilyas Paenrongi, Triana Dewi Salma

ISBN: 978-634-7226-25-9

Editor : Sarwandi, M.Pd.T
Layout : Miftahul Jannah, M.Kom
Desain sampul : Rifki Ramadan

Penerbit
PT. Mifandi Mandiri Digital

Redaksi & Distributor Tunggal
PT. Mifandi Mandiri Digital
Komplek Senda Residence Jl. Payanibung Ujung D Dalu Sepuluh-B Tanjung Morawa Kab. Deli Serdang Sumatera Utara

Cetakan Pertama, Juli 2025

Kata Pengantar

Di era transformasi digital saat ini, keterampilan dalam logika dan algoritma pemrograman menjadi kemampuan dasar yang sangat penting, bukan hanya bagi calon programmer, tetapi juga bagi siapa pun yang ingin memahami cara kerja teknologi di balik aplikasi, sistem, dan perangkat digital yang digunakan sehari-hari. Logika dan algoritma bukan sekadar teori atau konsep matematis; keduanya adalah fondasi dalam proses berpikir sistematis, pemecahan masalah, dan pengembangan perangkat lunak yang efisien dan andal.

Buku ini hadir sebagai panduan komprehensif yang dirancang untuk membantu pembaca memahami prinsip-prinsip dasar logika dan algoritma pemrograman secara bertahap dan terstruktur. Dimulai dari pengenalan logika dasar, struktur algoritma, hingga penerapan konsep lanjutan seperti rekursi, struktur data, dan algoritma pencarian dan pengurutan, buku ini dilengkapi dengan studi kasus nyata dan ilustrasi yang memudahkan pemahaman. Tidak hanya berfokus pada teori, buku ini juga membimbing pembaca untuk menerapkan konsep algoritmik ke dalam bahasa pemrograman Python secara praktis.

Dengan cakupan yang luas namun disajikan dengan bahasa yang mudah dipahami, buku ini diharapkan menjadi referensi utama bagi pelajar, mahasiswa, guru, maupun siapa saja yang ingin membangun fondasi kuat dalam dunia pemrograman. Semoga kehadiran buku ini dapat membuka jalan bagi pembaca untuk berpikir logis, menyusun solusi secara

algoritmik, dan mengembangkan kemampuan teknis yang dibutuhkan untuk menjadi programmer andal di era digital yang penuh tantangan dan peluang.

Medan, Maret 2025

Penulis

Daftar Isi

BAB 1 PERAN LOGIKA DAN ALGORITMA DALAM PEMROGRAMAN

Pendahuluan

Dalam dunia pemrograman, logika dan algoritma merupakan dua konsep dasar yang tidak dapat dipisahkan. Keduanya menjadi pondasi utama dalam pengembangan perangkat lunak dan aplikasi. Tanpa pemahaman yang baik tentang logika dan algoritma, seorang *programmer* akan kesulitan dalam menyelesaikan masalah secara sistematis dan efisien. Logika memberikan kerangka berpikir sistematis untuk merancang solusi, sementara algoritma menyediakan langkah-langkah terstruktur untuk mencapai tujuan tersebut (Mirkowska & Salwicki, 1987). Bab ini akan membahas peran penting logika dan algoritma dalam pemrograman, serta bagaimana keduanya saling melengkapi untuk menciptakan solusi yang efektif.

Logika berfungsi sebagai alat untuk menilai argumen dan membedakan antara penalaran yang benar dan salah. Dalam konteks ilmu komputer, logika tidak hanya membantu dalam memahami dan merancang algoritma, tetapi juga menjadi dasar bagi pengembangan program komputer yang efisien. C. A. R. Hoare, seorang tokoh terkemuka dalam bidang ini, menekankan bahwa program komputer dapat dipandang sebagai ekspresi matematis, di mana semua sifat program dapat dieksplorasi melalui penalaran deduktif. Dengan demikian,

logika menjadi kunci untuk menciptakan solusi yang tepat dan dapat diandalkan dalam pengembangan perangkat lunak (Caferra, 2011).

Pemrograman, di sisi lain, adalah proses spesifikasi masalah yang diubah menjadi program yang dapat dijalankan. Pemrograman logika, sebagai salah satu pendekatan dalam pemrograman, menekankan pada hasil yang ingin dicapai dari pada langkah-langkah spesifik untuk mencapainya. Dengan menggunakan bahasa yang dekat dengan logika, seperti Prolog atau C/C++, *programmer* dapat lebih fokus pada tujuan akhir, sehingga menciptakan program yang lebih efisien dan mudah dipahami. Logika dan algoritma adalah dua komponen penting dalam pemrograman yang saling melengkapi. Pemahaman yang baik tentang kedua konsep ini akan membantu *programmer* dalam merancang dan mengimplementasikan solusi yang efektif dan efisien. Bab ini akan memberikan gambaran umum tentang peran logika dan algoritma dalam pemrograman, serta contoh sederhana penerapannya.

Pengertian Logika dalam Pemrograman

Logika dalam pemrograman merujuk pada cara berpikir yang sistematis dan terstruktur untuk menyelesaikan masalah dalam pengembangan perangkat lunak. Logika ini mencakup penggunaan aturan-aturan *for*mal, seperti logika proposisional dan predikat, untuk memastikan bahwa program dapat berfungsi dengan benar dan konsisten. Dalam konteks pemrograman, logika juga digunakan untuk memverifikasi kebenaran program, memastikan bahwa setiap langkah yang diambil sesuai dengan spesifikasi yang telah ditetapkan (Mirkowska & Salwicki, 1987). Logika sering kali diimplementasikan melalui struktur kontrol seperti

percabangan (*if-else*) dan perulangan (*loop*).

Dalam konteks ini, logika berfungsi sebagai jembatan antara spesifikasi masalah yang tidak dapat dijalankan dan program yang dapat dieksekusi. Pemrograman logika (*Logic Programming*) berusaha untuk mengurangi kesenjangan ini dengan menggunakan bahasa yang dekat dengan logika, sehingga *programmer* dapat lebih fokus pada apa yang ingin dicapai oleh program, bukan pada bagaimana cara mencapainya. Pendekatan ini memungkinkan *programmer* untuk mengekspresikan solusi dalam bentuk deklaratif, di mana mereka mendefinisikan tujuan dan kondisi yang harus dipenuhi, sementara mesin bertanggung jawab untuk menemukan cara untuk mencapainya.

Dalam pemrograman logika, penekanan diberikan pada apa yang dihitung oleh program, bukan pada langkah-langkah spesifik yang diperlukan untuk melakukan perhitungan tersebut. Hal ini berbeda dengan pemrograman imperatif, di mana *programmer* harus memberikan instruksi langkah demi langkah. Dengan menggunakan logika sebagai dasar, *programmer* dapat menciptakan program yang lebih efisien dan dapat diandalkan, serta lebih mudah untuk dipahami dan dipelihara. Contoh bahasa pemrograman yang menerapkan paradigma ini adalah Prolog, yang memungkinkan pengembang untuk menulis program dengan cara yang lebih intuitif dan mendekati cara berpikir manusia (Caferra, 2011).

Pentingnya logika dalam pemrograman (Loukanova et al., 2021):

1. Ekspresi Kondisi: Logika memungkinkan pengembang untuk mengekspresikan kondisi yang harus dipenuhi untuk menjalankan bagian tertentu dari kode, sehingga program dapat berfungsi secara dinamis.

2. Pengambilan Keputusan: Dengan menggunakan logika, pengembang dapat membuat keputusan dalam program, seperti memilih jalur eksekusi yang berbeda berdasarkan input atau kondisi tertentu.
3. Struktur Kontrol: Logika adalah dasar dari struktur kontrol dalam pemrograman, seperti if statements, *loop*s, dan switch cases, yang mengatur alur eksekusi program.
4. Debugging dan Validasi: Logika membantu dalam proses debugging dengan memungkinkan pengembang untuk memeriksa dan memvalidasi kondisi yang menyebabkan kesalahan atau perilaku yang tidak diinginkan dalam program.
5. Pengembangan Algoritma: Logika berperan penting dalam merancang algoritma yang efisien, karena algoritma sering kali melibatkan serangkaian keputusan logis untuk mencapai solusi yang diinginkan.

Pengertian Algoritma dalam Pemrograman

Algoritma adalah serangkaian langkah-langkah yang terdefinisi dengan baik untuk menyelesaikan suatu masalah atau mencapai suatu tujuan. Algoritma dapat dianggap sebagai resep atau prosedur yang harus diikuti untuk menghasilkan *output* yang diinginkan dari input yang diberikan. Dalam pemrograman, algoritma digunakan untuk mengimplementasikan solusi yang telah dirancang dengan logika. Dalam pemrograman, algoritma berfungsi sebagai blueprint yang menentukan bagaimana data diproses dan diubah menjadi *output* yang diinginkan. Algoritma yang baik harus memiliki sifat-sifat seperti deterministik (setiap langkah jelas dan tidak ambigu), terbatas (memiliki titik akhir), dan efektif (dapat

diselesaikan dalam waktu yang wajar).

Algoritma juga dapat dianggap sebagai interpretasi dari suatu program, di mana setiap langkah dalam algoritma harus memiliki makna yang jelas dalam konteks pemrograman. Misalnya, algoritma Euclid untuk mencari pembagi persekutuan terbesar (GCD) dari dua bilangan dapat diimplementasikan dalam berbagai bahasa pemrograman, namun langkah-langkah logisnya tetap sama. Hal ini menunjukkan bahwa algoritma bersifat independen dari bahasa pemrograman tertentu.

Selain itu, algoritma harus diverifikasi sebelum digunakan untuk memastikan bahwa ia benar-benar menyelesaikan masalah yang dimaksud. Verifikasi ini melibatkan analisis terhadap sifat-sifat algoritma, seperti terminasi (apakah algoritma akan berhenti) dan kebenaran (apakah hasil yang dihasilkan sesuai dengan yang diharapkan). Proses ini sering kali melibatkan penggunaan logika untuk membuktikan bahwa algoritma tersebut memenuhi spesifikasi yang telah ditetapkan (Mirkowska & Salwicki, 1987).

Hubungan antara Logika dan Algoritma

Logika dan algoritma memiliki hubungan yang erat. Logika digunakan untuk merancang solusi, sedangkan algoritma adalah implementasi dari solusi tersebut. Sebuah algoritma yang baik harus didasarkan pada logika yang kuat agar dapat menghasilkan solusi yang efektif dan efisien. Tanpa logika yang benar, algoritma mungkin tidak akan berfungsi sebagaimana mestinya atau bahkan menghasilkan *output* yang salah. Sebaliknya, tanpa algoritma yang terstruktur, logika hanya akan menjadi konsep teoretis yang sulit diimplementasikan dalam program.

Salah satu contoh hubungan ini dapat dilihat dalam proses verifikasi program. Logika digunakan untuk membuktikan bahwa suatu algoritma memenuhi spesifikasi yang telah ditetapkan, seperti kebenaran dan terminasi. Dalam konteks ini, logika berperan sebagai alat untuk menganalisis dan memvalidasi algoritma, memastikan bahwa setiap langkah yang diambil sesuai dengan aturan-aturan logis yang telah ditetapkan.

Keterkaitan logika dan algoritma:

1. Interdependensi: Logika dan algoritma saling bergantung; logika digunakan untuk merumuskan algoritma.
2. Implementasi: Algoritma sering kali terdiri dari serangkaian keputusan logis yang menentukan bagaimana data diproses.

Selain itu, logika juga digunakan dalam proses optimasi algoritma. Dengan memahami logika di balik suatu algoritma, *programmer* dapat mengidentifikasi bagian-bagian yang dapat dioptimalkan untuk meningkatkan efisiensi. Misalnya, dalam algoritma pencarian atau pengurutan, logika digunakan untuk menentukan strategi terbaik yang dapat mengurangi waktu eksekusi atau penggunaan memori (Mirkowska & Salwicki, 1987).

Peran Logika dan Algoritma dalam Pemecahan Masalah

Pemecahan masalah adalah inti dari pemrograman. Logika dan algoritma memainkan peran kunci dalam proses ini. Logika membantu dalam mengidentifikasi masalah dan merancang solusi, sedangkan algoritma digunakan untuk

mengimplementasikan solusi tersebut. Dengan kombinasi yang tepat antara logika dan algoritma, *programmer* dapat menyelesaikan masalah dengan cara yang sistematis dan terukur. Peran logika dan algoritma dalam pemecahan masalah sangat krusial dalam bidang ilmu komputer dan kecerdasan buatan. Logika menyediakan kerangka kerja yang sistematis untuk menganalisis dan menilai argumen, memungkinkan pengembang untuk memahami hubungan antara berbagai proposisi dan deduksi yang dapat diambil dari mereka.

Dengan menggunakan prinsip-prinsip logika, programmer dapat merancang algoritma yang efektif, yang merupakan langkah-langkah terstruktur untuk mencapai solusi dari masalah yang kompleks. Algoritma berfungsi sebagai panduan yang jelas dan terdefinisi untuk menyelesaikan masalah, mengoptimalkan proses pengambilan keputusan, dan memastikan bahwa solusi yang dihasilkan tidak hanya benar tetapi juga efisien. Keterpaduan antara logika dan algoritma memungkinkan pengembang untuk menciptakan sistem yang dapat beradaptasi dan berfungsi dengan baik dalam situasi yang beragam, menjadikannya alat yang sangat berharga dalam pengembangan aplikasi dan sistem cerdas (Caferra, 2011).

Algoritma menyediakan kerangka kerja yang terstruktur untuk mengimplementasikan sebuah solusi. Dengan kombinasi yang tepat antara logika dan algoritma, *programmer* dapat mengembangkan solusi yang efisien, andal, dan mudah dipahami, sehingga memudahkan proses pengembangan dan pemeliharaan perangkat lunak (Mirkowska & Salwicki, 1987). Berikut adalah tiga poin yang menjelaskan keuntungan logika dan algoritma dalam pemecahan masalah:

1. Meningkatkan Efisiensi dan Efektivitas: Logika dan algoritma yang baik dapat meningkatkan efisiensi dan

efektivitas dalam menyelesaikan masalah, memungkinkan program untuk berjalan lebih cepat dan dengan penggunaan sumber daya yang lebih sedikit.
2. Menangani Masalah Kompleks Secara Terstruktur: Dengan menggunakan logika dan algoritma, pengembang dapat menangani masalah kompleks dengan cara yang terstruktur, memecah masalah besar menjadi bagian-bagian yang lebih kecil dan lebih mudah dikelola.
3. Meningkatkan Kemampuan Pemecahan Masalah: Penggunaan logika dan algoritma membantu pengembang untuk mengembangkan keterampilan pemecahan masalah yang lebih baik, karena mereka belajar untuk menganalisis masalah, merumuskan solusi, dan menerapkan pendekatan yang sistematis.

Contoh Penerapan Logika dan Algoritma dalam Pemrograman

Untuk memberikan gambaran yang lebih jelas, mari kita lihat contoh sederhana penerapan logika dan algoritma dalam pemrograman. Misalnya, kita ingin membuat program untuk menentukan apakah suatu bilangan adalah bilangan prima atau bukan. Langkah-langkahnya adalah sebagai berikut.

1. Logika
    a. Sebelum kita melangkah lebih jauh, penting untuk memahami terlebih dahulu apa itu bilangan prima. Bilangan prima adalah bilangan bulat positif yang hanya dapat dibagi oleh 1 dan dirinya sendiri tanpa menghasilkan sisa. Dengan kata lain, bilangan prima tidak memiliki pembagi lain

selain 1 dan bilangan itu sendiri. Contoh bilangan prima yang umum dikenal adalah 2, 3, 5, 7, 11, dan seterusnya.

b. Untuk menentukan apakah suatu bilangan adalah bilangan prima, kita perlu memeriksa apakah bilangan tersebut memiliki pembagi lain selain 1 dan dirinya sendiri. Jika kita menemukan bilangan yang dapat membagi bilangan tersebut tanpa sisa, maka bilangan itu bukanlah bilangan prima.

2. Algoritma

Setelah memahami logika di balik bilangan prima, kita dapat merumuskan algoritma untuk menentukan apakah suatu bilangan adalah bilangan prima atau bukan. Berikut adalah langkah-langkah algoritma yang dapat kita gunakan:

a. Inisialisasi

Mulailah dengan bilangan yang akan diperiksa. Misalnya, kita ingin memeriksa bilangan n.

b. Perulangan

Lakukan perulangan dari 2 hingga akar kuadrat dari bilangan tersebut. Mengapa kita hanya perlu memeriksa hingga akar kuadrat? Karena jika n memiliki pembagi yang lebih besar dari akar kuadratnya, maka pembagi tersebut pasti memiliki pasangan yang lebih kecil dari akar kuadrat. Dengan demikian, kita dapat mengurangi jumlah iterasi yang diperlukan.

c. Pemeriksaan Pembagi

Di dalam perulangan, periksa apakah bilangan n dapat dibagi habis oleh salah satu angka dalam

rentang tersebut. Jika kita menemukan angka yang dapat membagi n tanpa sisa, maka kita dapat menyimpulkan bahwa n bukanlah bilangan prima.

d. Kesimpulan
   Jika tidak ada angka dalam rentang tersebut yang dapat membagi n, maka kita dapat menyimpulkan bahwa n adalah bilangan prima.

Contoh Implementasi

Berikut adalah contoh implementasi algoritma di atas dalam bahasa pemrograman Python:

```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

# Contoh penggunaan
bilangan = 29
if is_prime(bilangan):
    print(f"{bilangan} adalah bilangan prima.")
else:
    print(f"{bilangan} bukan bilangan prima.")
```

Dalam contoh di atas, kita mendefinisikan sebuah fungsi 'is_prime' yang menerima satu parameter, yaitu bilangan yang akan diperiksa. Fungsi ini kemudian menerapkan logika dan algoritma yang telah kita bahas untuk menentukan apakah bilangan tersebut adalah bilangan prima.

Penerapan logika dan algoritma dalam pemrograman sangatlah penting untuk menyelesaikan masalah secara efisien. Dengan memahami konsep dasar seperti bilangan prima, kita dapat mengembangkan algoritma yang tidak hanya efektif tetapi juga mudah dipahami. Contoh di atas adalah salah satu dari sekian banyak penerapan logika dan algoritma dalam pemrograman yang dapat membantu kita dalam menyelesaikan berbagai tantangan yang dihadapi dalam dunia teknologi informasi.

BAB 2 KONSEP DASAR LOGIKA: PROPOSISI, PREDIKAT DAN PENALARAN

Pendahuluan

Dalam Logika merupakan cabang filsafat dan matematika yang mempelajari prinsip-prinsip penalaran yang sahih. Dalam konteks pendidikan tinggi, khususnya di bidang ilmu komputer, matematika, filsafat, dan ilmu kognitif, penguasaan konsep logika sangat penting karena menjadi fondasi bagi berpikir kritis, analitis, dan sistematis. Logika memungkinkan mahasiswa untuk memahami dan menyusun argumen secara runtut, menilai validitas pernyataan, serta melakukan inferensi yang sah.

Logika adalah ilmu yang mempelajari aturan-aturan berpikir yang benar untuk mencapai kesimpulan yang sahih. Secara umum, logika membahas bentuk pemikiran itu sendiri serta hukum-hukum yang mengatur penalaran sehingga menghasilkan kesimpulan yang valid. Beberapa ahli berpendapat bahwa logika adalah metode atau teknik meneliti ketepatan menalar, serta cara berpikir rasional yang berlandaskan bahasa dan kata-kata. Dengan demikian, logika berperan penting sebagai landasan berpikir rasional dan kritis. Sebagai ilmu, logika membantu seseorang berpikir lurus, tepat, dan teratur meningkatkan kemampuan analitis serta menghindari kesalahan berpikir (sesat pikir).

Pendidikan adalah jantung kemajuan peradaban, dan di era yang didominasi oleh inovasi teknologi serta globalisasi,

paradigma pembelajaran terus berevolusi. Abad ke-21 menuntut pendekatan pendidikan yang tidak hanya mentransfer pengetahuan, tetapi juga membekali peserta didik dengan keterampilan kritis, kreativitas, kolaborasi, dan adaptabilitas. Teori dan model pembelajaran modern muncul sebagai jawaban atas kebutuhan ini, menggabungkan wawasan dari psikologi kognitif, neurosains, sosiologi, dan teknologi digital.

Dalam bab ini, kita akan membahas tiga konsep fundamental dalam logika: proposisi, predikat, dan penalaran. Penjelasan akan disertai dengan contoh-contoh *for*mal guna memperkuat pemahaman pembaca terhadap teori yang dijabarkan

Pengertian dan Sejarah Singkat Logika

Logika berasal dari bahasa Yunani logos, yang berarti "kata", "akal", atau "penalaran". Dalam sejarah pemikiran manusia, logika telah berkembang sejak zaman Yunani kuno. Aristoteles adalah tokoh awal yang mengembangkan logika silogistik, yaitu sistem penalaran yang terdiri dari premis dan konklusi. Dalam logika modern, sistem simbolik diperkenalkan oleh George Boole dan Gottlob Frege, yang kemudian menjadi dasar bagi logika matematika dan komputer.

Logika modern terbagi ke dalam dua cabang besar: logika *for*mal dan logika in*for*mal. Logika *for*mal menggunakan simbol dan aturan baku untuk menyusun argumen, sedangkan logika in*for*mal lebih memperhatikan isi dan konteks argumen dalam bahasa alami.

Sejarah logika dimulai sejak zaman Yunani kuno, khususnya dengan karya-karya Aristoteles (384–322 SM), yang dianggap sebagai bapak logika *for*mal. Aristoteles

mengembangkan sistem logika silogistik, yakni sistem yang menyusun argumen melalui dua premis dan satu kesimpulan. Misalnya:

1. Semua manusia akan mati.
2. Socrates adalah manusia.
3. Maka, Socrates akan mati.

Sistem silogistik ini menjadi kerangka utama dalam pembelajaran logika hingga abad pertengahan. Pada masa tersebut, para filsuf dan teolog Eropa, seperti Thomas Aquinas, mengembangkan dan mengajarkan logika sebagai bagian dari trivium (tata bahasa, logika, retorika). elanjutnya, pada awal abad ke-20, tokoh seperti Gottlob Frege, Bertrand Russell, dan Alfred North Whitehead mengembangkan logika predikat sebagai perluasan dari logika proposisional. Mereka memperkenalkan notasi simbolik yang lebih kuat untuk menyatakan kuantifikasi dan relasi antarobjek. Logika predikat ini menjadi fondasi bagi ilmu komputer, linguistik *for*mal, dan teori matematika modern.

Perkembangan logika tidak berhenti pada tataran *for*mal semata. Di abad ke-20, berkembang pula logika non-klasik seperti logika fuzzy, logika modal, dan logika intuitionistik yang bertujuan menangani jenis-jenis penalaran yang tidak dapat dijelaskan secara sempurna dengan logika klasik. Dengan memahami sejarah dan peranan logika, mahasiswa tidak hanya melihatnya sebagai alat teknis, tetapi juga sebagai warisan intelektual yang membentuk cara kita berpikir dan menyusun pengetahuan.

Proposisi: Definisi dan Operasi Dasar

Dalam logika, proposisi adalah pernyataan yang memiliki nilai kebenaran, yaitu dapat bernilai benar (*true*) atau

salah (*false*), tetapi tidak sekaligus keduanya. Proposisi merupakan dasar dari semua bentuk argumen logis, dan digunakan untuk membentuk struktur penalaran *for*mal. Contoh:

1. "2 adalah bilangan genap" → proposisi yang bernilai benar.
2. "Semua kucing bisa terbang" → proposisi yang bernilai salah.

Proposisi berbeda dengan kalimat tanya, seruan, atau perintah karena ketiganya tidak memiliki nilai kebenaran.

Jenis-jenis Proposisi

1. Proposisi Tunggal (*Atomic Proposition*): Proposisi yang tidak mengandung proposisi lain.
   Contoh: Air mendidih pada 100°C.
2. Proposisi Majemuk (*Compound Proposition*): Proposisi yang dibentuk dari dua atau lebih proposisi tunggal menggunakan operator logika.
   Contoh: Air mendidih pada 100°C dan es mencair pada 0°C.

Operator Logika

1. Negasi (¬P)
   a. Membalik nilai kebenaran dari suatu proposisi.
   b. Contoh: Jika P = "Hari ini hujan", maka ¬P = "Hari ini tidak hujan".
2. Konjungsi (P ∧ Q)
   a. Benar jika P dan Q keduanya benar.
   b. Contoh: "2 adalah bilangan genap dan 3 adalah bilangan ganjil".
3. Disjungsi (P ∨ Q)

a. Benar jika salah satu atau kedua proposisi benar.
b. Contoh: “Hari ini cerah atau mendung”.

4. Implikasi ($P \rightarrow Q$)
    a. Dibaca: “Jika P maka Q.” Salah hanya jika P benar dan Q salah.
    b. Contoh: “Jika saya belajar, maka saya lulus ujian”.
5. Biimplikasi ($P \leftrightarrow Q$)
    a. Benar jika P dan Q bernilai sama.
    b. Contoh: “Saya pergi hanya jika kamu pergi”.

Tabel Kebenaran

Tabel kebenaran digunakan untuk menentukan nilai kebenaran dari proposisi majemuk berdasarkan nilai-nilai kebenaran dari komponen-komponennya. Dengan memahami proposisi dan operasi logika dasar ini, mahasiswa dapat menyusun dan mengevaluasi argumen secara sistematis serta memahami bagaimana pengetahuan disusun secara deduktif.

Contoh tabel kebenaran untuk konjungsi ($P \wedge Q$):

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

Logika Predikat dan Kuantifikasi

Dalam logika, proposisi adalah pernyataan yang memiliki nilai kebenaran, yaitu dapat bernilai benar (*true*) atau salah (*false*), tetapi tidak sekaligus keduanya. Proposisi merupakan dasar dari semua bentuk argumen logis, dan digunakan untuk membentuk struktur penalaran *for*mal.

Salah satu elemen utama dalam logika predikat adalah predikat, yaitu fungsi yang menerapkan suatu sifat atau hubungan pada satu atau lebih objek. Contohnya, jika kita punya predikat M(x) yang menyatakan "x adalah manusia", maka kita dapat menyatakan "Socrates adalah manusia" sebagai M(Socrates). Selain predikat, komponen penting lainnya adalah kuantifier (kuantor), yaitu simbol yang menyatakan banyaknya objek yang dimaksud dalam pernyataan:

1. Kuantor Universal (∀)
    a. Menyatakan bahwa suatu pernyataan berlaku untuk semua anggota domain.
    b. Contoh: ∀x M(x) → "Untuk setiap x, x adalah manusia".
2. Kuantor Eksistensial (∃)
    a. Menyatakan bahwa ada setidaknya satu anggota domain yang memenuhi pernyataan.
    b. Contoh: ∃x M(x) → "Ada x sedemikian sehingga x adalah manusia".

Contoh *For*mal: Pernyataan "Semua manusia akan mati" dalam logika predikat dapat ditulis sebagai: ∀x (M(x) → W(x)) Di mana:

1. M(x): x adalah manusia
2. W(x): x akan mati

Hubungan antara Predikat dan Fungsi Logika Predikat dapat memiliki lebih dari satu argumen. Misalnya, L(x, y) menyatakan "x mencintai y". Maka pernyataan "Ali mencintai Dina" ditulis sebagai L(Ali, Dina). Ini menunjukkan bahwa logika predikat juga mampu menangani relasi antarobjek. Dengan menggunakan logika predikat, kita dapat menyusun argumen yang lebih kompleks dan menangkap nuansa makna yang tidak dapat diwakili oleh logika proposisional saja. Logika

predikat menjadi dasar bagi sistem pembuktian matematika modern dan sistem inferensi dalam kecerdasan buatan.

Dalam logika, predikat adalah pernyataan yang mengandung satu atau lebih variabel dan belum memiliki nilai kebenaran hingga variabel-variabel tersebut diisi nilai tertentu. Predikat sering disebut fungsi proposisi. Contohnya, P(x): "x > 0" atau Q(x,y): "x + y = 10". Di sini P dan Q adalah predikat; keduanya bergantung pada nilai variabel x dan y. Sebelum variabel diisi, P(x) dan Q(x,y) bukanlah proposisi karena nilai benarnya belum ditentukan. Perbedaan utama antara predikat dan proposisi adalah status nilai kebenarannya. Proposisi adalah pernyataan lengkap yang sudah pasti bernilai benar atau salah. Sebaliknya, predikat adalah kalimat terbuka (open sentence) yang menjadi proposisi hanya setelah variabel-variabelnya diganti dengan objek tertentu atau disertai kuantor. Misalnya, "x ≥ 10" adalah predikat. Jika x diganti dengan angka 12, kita mendapatkan proposisi "12 ≥ 10" yang bernilai benar; jika x = 5, proposisinya "5 ≥ 10" bernilai salah.

Secara notasi, predikat dilambangkan dengan huruf kapital dan parameter di dalam tanda kurung. Contoh: P(x), Q(x,y), R(x,y,z). Untuk menjadikannya proposisi, variabelnya dapat diikat oleh nilai tertentu atau ditambah dengan kuantor (seperti dibahas di bawah). Intinya, predikat memfokuskan pada sifat atau hubungan yang berlaku pada objek, sedangkan proposisi menyatakan fakta khusus tentang objek (telah ditentukan nilainya). Logika predikat mengenal dua jenis kuantor yang menentukan seberapa luas pernyataan berlaku di semesta pembicaraan:

1. Kuantor universal (∀): Melambangkan "untuk semua" atau "setiap". Pernyataan (∀x) P(x) berarti P(x) benar untuk setiap elemen x dalam semesta diskusi. Simbolnya

adalah “∀”, dibaca “untuk setiap” atau “semua”, Misalnya, (∀x) “x bilangan real, $x^2 \geq 0$” menyatakan “untuk semua bilangan real x, $x^2 \geq 0$”, yang benar karena kuadrat bilangan real tidak pernah negatif. Pernyataan kuantor universal hanya bernilai benar jika benar pada seluruh semesta; jika ada satu kasus pelanggaran, pernyataannya salah.

2. Kuantor eksistensial (∃): Melambangkan “ada” atau “terdapat”. Pernyataan (∃x) Q(x) berarti “ada setidaknya satu elemen x dalam semesta yang membuat Q(x) benar”. Simbolnya “∃”, dibaca “ada” atau “terdapat” Contohnya, (∃x) “x bilangan bulat, $x^2 = 16$” menyatakan “terdapat bilangan bulat x sedemikian sehingga $x^2 = 16$” (yakni $x = 4$ atau -4), yang benar karena minimal satu solusinya ada. Pernyataan eksistensial benar jika setidaknya ada satu contoh pemenuhan, dan salah hanya jika tidak satupun elemen semesta memenuhi sifat tersebut.

Tantangan Implementasi

Ketimpangan Penalaran merupakan proses berpikir yang digunakan untuk menarik kesimpulan berdasarkan informasi atau premis yang tersedia. Dalam logika, penalaran dibedakan menjadi dua jenis utama: penalaran deduktif dan penalaran induktif.

1. Penalaran Induktif
   Penalaran induktif adalah proses menarik kesimpulan umum berdasarkan pengamatan atau data yang terbatas. Berbeda dengan deduksi, induksi tidak menjamin kebenaran mutlak dari kesimpulan, tetapi hanya memberikan probabilitas tinggi bahwa kesimpulan itu

benar.

Contoh:

a. Semua mamalia bernapas dengan paru-paru.
b. Paus adalah mamalia.
c. Maka, paus bernapas dengan paru-paru.

Penalaran ini bersifat valid, karena kesimpulan mengikuti aturan logika dari premis yang diberikan, dan bersifat sound (absah) jika premis-premisnya juga benar secara faktual.

2. Penalaran Deduktif

Penalaran deduktif adalah bentuk penalaran di mana kesimpulan diturunkan secara logis dari premis-premis yang diberikan. Jika premis-premisnya benar dan struktur logiknya valid, maka kesimpulan yang dihasilkan pasti benar.

Contoh:

a. Burung pipit bisa terbang.
b. Burung merpati bisa terbang.
c. Burung elang bisa terbang.
d. Maka, Semua burung bisa terbang.

Penalaran ini tidak valid secara logika deduktif, karena ada burung seperti burung unta yang tidak bisa terbang. Namun, penalaran induktif sangat berguna dalam ilmu pengetahuan untuk membangun hipotesis dan generalisasi berdasarkan observasi.

Validitas dan Kesesatan Logika

Dalam logika *for*mal, penting untuk membedakan antara argumen yang valid dan argumen yang sesat (*fallacy*). Argumen valid adalah argumen yang struktur logiknya benar, sedangkan argumen sesat mungkin tampak logis di permukaan, tetapi

cacat dalam struktur atau asumsi dasarnya.

Contoh kesesatan logika:

1. Post hoc ergo propter hoc (karena setelahnya, maka disebabkannya):
   "Setelah saya memakai kalung ini, saya tidak sakit lagi. Berarti kalung ini menyembuhkan saya".
2. Straw man:
   "Orang itu menolak membeli mobil *list*rik, berarti dia tidak peduli terhadap lingkungan".

Dengan memahami jenis-jenis penalaran dan mengenali struktur argumen yang valid, mahasiswa dapat berpikir lebih kritis, menghindari kesesatan logika, dan menyusun argumen yang kuat dalam berbagai konteks akademik dan praktis.

BAB 3 JENIS-JENIS LOGIKA DALAM PEMOGRAMAN

Pendahuluan

Dalam pemrograman, logika berperan penting dalam pengambilan keputusan dan alur eksekusi suatu program. Berikut adalah beberapa jenis logika yang umum digunakan dalam pemrograman.

Logika Kondisional (*Conditional Logic*)

Logika kondisional sering kali diterapkan menggunakan pernyataan seperti *if-else*, *Switch-Case*, atau ekspresi ternary dalam berbagai bahasa pemrograman. Dengan adanya struktur ini, program dapat menangani berbagai skenario tanpa harus mengeksekusi semua bagian kode secara berurutan.

Contoh Kasus: Menentukan Kategori Usia

Sebuah program diminta untuk mengelompokkan seseorang ke dalam kategori usia berdasarkan umur yang dimasukkan oleh pengguna. Kategorinya sebagai berikut.

Anak-anak: 0 - 12 tahun

Remaja: 13 - 17 tahun

Dewasa: 18 - 59 tahun

Lansia: 60 tahun ke atas

Berikut adalah implementasi dalam bahasa Python menggunakan if-el*if-else*:

```python
# Input umur dari pengguna
umur = int(input("Masukkan umur Anda: "))

# Logika kondisional untuk menentukan kategori usia
if umur >= 0 and umur <= 12:
    kategori = "Anak-anak"
elif umur >= 13 and umur <= 17:
    kategori = "Remaja"
elif umur >= 18 and umur <= 59:
    kategori = "Dewasa"
elif umur >= 60:
    kategori = "Lansia"
else:
    kategori = "Umur tidak valid"

# Output hasil kategori
print(f"Anda termasuk dalam kategori: {kategori}")
```

Penjelasan:

Program meminta pengguna memasukkan usia. Menggunakan if-el*if-else*, program mengevaluasi rentang usia dan menentukan kategori yang sesuai. Jika usia yang dimasukkan tidak valid (misalnya angka negatif), program akan menampilkan pesan kesalahan.

Contoh eksekusi:

```
Masukkan umur Anda: 25
Anda termasuk dalam kategori: Dewasa
```

Logika Perulangan (*Looping Logic*)

Logika ini digunakan untuk mengeksekusi kode secara berulang selama kondisi tertentu terpenuhi. Biasanya diterapkan dengan *for loop*, *while loop*, atau do-*while loop*:

```python
for i in range(5):
    print("Perulangan ke-", i+1)
```

Kasus: Menampilkan Bilangan Ganjil dalam Rentang

Tertentu.

Seorang pengguna ingin menampilkan semua bilangan ganjil dari 1 hingga N, di mana N adalah angka yang dimasukkan oleh pengguna.

```
# Input angka batas dari pengguna
N = int(input("Masukkan angka batas: "))

# Menampilkan bilangan ganjil menggunakan perulangan for
print("Bilangan ganjil dari 1 hingga", N, ":")
for i in range(1, N + 1, 2):
    print(i, end=" ")
```

Penjelasan:

Program meminta pengguna memasukkan angka batas N. Menggunakan perulangan *for*, program mengiterasi dari 1 hingga N, dengan langkah 2 (agar hanya mencetak angka ganjil). Hasilnya ditampilkan dalam satu baris menggunakan end=" " agar lebih rapi.

Contoh Eksekusi:

```
Masukkan angka batas: 10
Bilangan ganjil dari 1 hingga 10:
1 3 5 7 9
```

Logika Boolean (*Boolean Logic*)

Logika boolean bekerja dengan nilai *True* dan *False*, serta menggunakan operator logika seperti AND, OR, dan NOT.

```
a = True
b = False
hasil = a and b  # False
```

Kasus: Verifikasi Login Pengguna

Sebuah sistem meminta pengguna memasukkan username dan password. Jika kombinasi yang dimasukkan sesuai dengan data yang telah ditentukan (username: admin,

password: 12345), maka pengguna diizinkan masuk. Jika tidak, akses ditolak.

Implementasi dalam Python

```
# Data login yang benar
username_benar = "admin"
password_benar = "12345"

# Input dari pengguna
username = input("Masukkan username: ")
password = input("Masukkan password: ")

# Logika Boolean untuk memeriksa apakah username dan password cocok
if username == username_benar and password == password_benar:
    print("Login berhasil! Selamat datang.")
else:
    print("Login gagal! Username atau password salah.")
```

Penjelasan

Program menyimpan username dan password yang benar. Pengguna memasukkan username dan password melalui input. Menggunakan operator AND (and dalam Python), program memeriksa apakah kedua kondisi sesuai. Jika benar, pengguna berhasil login. Jika salah, login ditolak.

Contoh Eksekusi

```
Masukkan username: admin
Masukkan password: 12345
Login berhasil! Selamat datang.
```

```
Masukkan username: user
Masukkan password: 54321
Login gagal! Username atau password salah.
```

Logika Rekursif (*Recursive Logic*)

Logika rekursif adalah teknik pemrograman di mana suatu fungsi memanggil dirinya sendiri untuk menyelesaikan masalah yang lebih kecil.

```python
def faktorial(n):
    if n == 1:
        return 1
    else:
        return n * faktorial(n-1)

print(faktorial(5))  # Output: 120
```

Logika Kombinatorial (*Combinatorial Logic*)

Logika ini digunakan untuk mengkombinasikan beberapa kondisi atau aturan untuk menghasilkan suatu keputusan. Biasanya diterapkan dalam sistem berbasis aturan atau kecerdasan buatan.

Contoh Kasus: Menghitung Jumlah Kombinasi dalam Pemilihan Tim.

Misalkan ada N orang, dan kita ingin memilih R orang dari mereka untuk membentuk sebuah tim. Berapa banyak cara kita bisa memilih tim tersebut? Dalam kombinatorika, jumlah cara memilih R orang dari N tanpa memperhatikan urutan dihitung dengan rumus:

$$C(n,r) = \frac{n!}{r!(n-r)!}$$

Implementasi dalam Python

```python
def faktorial(n):
    """Fungsi rekursif untuk menghitung faktorial"""
    if n == 0 or n == 1:
        return 1
    else:
        return n * faktorial(n - 1)

def kombinasi(n, r):
    """Menghitung kombinasi C(n, r)"""
    if r > n:
        return 0  # Tidak mungkin memilih lebih banyak dari jumlah yang ada
    return faktorial(n) // (faktorial(r) * faktorial(n - r))

# Input dari pengguna
N = int(input("Masukkan jumlah total orang (N): "))
R = int(input("Masukkan jumlah orang yang dipilih (R): "))

# Menampilkan hasil kombinasi
if N < 0 or R < 0:
    print("N dan R harus bilangan positif.")
else:
    print(f"Jumlah kombinasi memilih {R} dari {N} adalah {kombinasi(N, R)}")
```

Penjelasan:

Fungsi faktorial(n) digunakan untuk menghitung faktorial secara rekursif.

Fungsi kombinasi(n, r) menerapkan rumus kombinasi.

Program meminta pengguna memasukkan nilai N dan R, lalu menghitung hasilnya.

Contoh Eksekusi:

```
Masukkan jumlah total orang (N): 5
Masukkan jumlah orang yang dipilih (R): 3
Jumlah kombinasi memilih 3 dari 5 adalah 10
```

Karena $C(5,3) = \frac{5!}{3!(5-3)!} = \frac{5\times4}{2\times1} = 10.$

Logika dalam pemrograman merupakan dasar dalam membangun algoritma dan struktur kontrol yang efisien.

Berbagai jenis logika digunakan sesuai dengan kebutuhan program, antara lain:

1. Logika Kondisional – Memungkinkan program mengambil keputusan berdasarkan kondisi tertentu dengan menggunakan struktur seperti *if-else* atau *Switch-Case*.
2. Logika Perulangan (*Looping Logic*) – Digunakan untuk menjalankan blok kode secara berulang hingga kondisi tertentu terpenuhi, seperti menggunakan *for*, *while*, atau do-*while*.
3. Logika Boolean – Berdasarkan nilai *true* (benar) atau *false* (salah), sering digunakan dalam pernyataan kondisi dan operasi logika seperti AND, OR, dan NOT.
4. Logika Rekursif – Memanfaatkan fungsi yang memanggil dirinya sendiri untuk menyelesaikan masalah yang dapat dipecah menjadi sub-masalah lebih kecil, seperti perhitungan faktorial atau deret Fibonacci.
5. Logika Kombinatorial – Berkaitan dengan penghitungan kombinasi atau permutasi, sering digunakan dalam analisis data, kriptografi, dan kecerdasan buatan.

Setiap jenis logika memiliki perannya masing-masing dalam menyelesaikan berbagai permasalahan pemrograman. Pemahaman yang baik tentang berbagai jenis logika ini memungkinkan pengembang untuk menulis kode yang lebih efisien, fleksibel, dan mudah dipahami.

BAB 4 STRUKTUR DASAR ALGORITMA: DEFENISI DAN KARAKTERISTIK

Pendahuluan

Algoritma dapat dicirikan sebagai kumpulan prosedur atau arahan yang metodis dan berurutan yang bertujuan menyelesaikan masalah tertentu atau melaksanakan tugas yang ditunjuk. Dalam bidang ilmu komputer, algoritma merupakan pilar fundamental rekayasa perangkat lunak, manipulasi data, dan optimalisasi beragam sistem. Mereka menemukan aplikasi di banyak disiplin ilmu, mencakup domain seperti pengambilan informasi, enkripsi data, dan kecerdasan buatan. Seperti yang diartikulasikan oleh Donald Knuth, seorang tokoh terkemuka di bidang ilmu komputer, algoritma harus menunjukkan atribut penting tertentu, termasuk keterbatasan, kepastian, input, *output*, dan efektivitas. Dengan memenuhi kriteria ini, algoritma dapat digunakan secara efektif untuk mengatasi berbagai tantangan dalam domain komputasi (Halimatussya' diyah Purba & Yahfizham Yahfizham, 2023).

Agar suatu algoritma dapat digunakan dengan kemanjuran dalam penyelesaian masalah, ia harus memiliki beberapa karakteristik penting. Pertama, algoritma harus diartikulasikan dengan jelas dan didefinisikan secara komprehensif. Ini mengharuskan langkah-langkah prosedural dapat ditafsirkan oleh agen manusia dan sistem komputasi tanpa tingkat ambiguitas apa pun. Kedua, algoritma harus terorganisir dan logis, memastikan bahwa itu menghasilkan

output yang benar sambil menyelaraskan dengan tujuan awal. Ketiga, algoritma harus menunjukkan efisiensi mengenai pemanfaatan sumber daya, yang mencakup kompleksitas waktu dan kompleksitas ruang. Selanjutnya, algoritma dapat diklasifikasikan sebagai deterministik, menyiratkan bahwa mereka secara konsisten menghasilkan hasil yang identik untuk input yang diberikan, atau non-deterministik, yang dapat menghasilkan hasil yang bervariasi tergantung pada variabel tertentu. Pada akhirnya, algoritma yang kuat harus menunjukkan generalisasi, memungkinkan penerapannya di berbagai skenario dengan karakteristik analog.

Arsitektur dasar dari suatu algoritma berkaitan dengan cara di mana instruksi diatur untuk menyelesaikan tugas tertentu. Struktur dasar ini terdiri dari tiga bentuk utama: struktur sekuensial, struktur percabangan (seleksi), dan struktur perulangan (iterasi). Struktur sekuensial mewakili mode *for*mulasi algoritma yang paling dasar, di mana langkah-langkah dijalankan secara linier dari inisiasi hingga kesimpulan tanpa percabangan atau pengulangan. Contoh klasik dari struktur sekuensial adalah program langsung yang dirancang untuk menghitung luas persegi panjang menggunakan rumus panjang × lebar. Struktur percabangan memfasilitasi pengambilan keputusan berdasarkan kondisi tertentu. Misalnya, dalam algoritma yang dirancang untuk menilai suhu tubuh, jika suhu yang tercatat melebihi 37° C, seseorang diklasifikasikan sebagai mengalami demam. Struktur perulangan digunakan untuk mengulangi instruksi tertentu sampai kondisi terminasi terpenuhi, seperti yang dicontohkan oleh perhitungan total kumulatif bilangan bulat dari 1 hingga n melalui proses iterative (Guerrini & Pisanti, 2024).

Kerangka dasar dari sebuah algoritma memiliki relevansi

yang signifikan dalam domain pemrograman, karena secara fundamental menentukan dinamika operasional suatu program. Pemrograman yang efektif bergantung pada pemanfaatan algoritma yang efisien dan dapat dipahami. Kepatuhan terhadap komposisi struktural yang tepat dalam algoritma meningkatkan kinerja program dan memfasilitasi proses debugging dan pemeliharaan kode. Selain itu, melalui pemahaman komprehensif tentang arsitektur dasar algoritma, seorang *programmer* diberdayakan untuk membangun program yang lebih modular dan terorganisir secara sistematis. Resolusi masalah rumit dapat dicapai dengan mendekonstruksi mereka menjadi segmen yang dapat dikelola yang dapat diatasi secara berurutan melalui penerapan sintesis struktur algoritmik fundamental.

Algoritma menemukan aplikasi tidak hanya dalam ranah pemrograman, tetapi juga di berbagai aspek keberadaan sehari-hari. Contoh ilustratif dari aplikasi tersebut meliputi navigasi, *e-commerce*, keuangan, dan perawatan kesehatan. Dalam konteks navigasi, aplikasi pemetaan digital, dicontohkan oleh Google Maps, menggunakan algoritma pencarian jalur terpendek untuk menyediakan pengguna dengan rute yang paling bijaksana. Di bidang *e-commerce*, sistem rekomendasi memanfaatkan algoritma pembelajaran mesin untuk mengusulkan produk yang selaras dengan preferensi konsumen. Dalam sektor keuangan, algoritma merupakan bagian integral dari perhitungan suku bunga, pemrosesan transaksi perbankan, dan analisis peluang investasi. Di bidang kedokteran, algoritma digunakan untuk memastikan diagnosis berdasarkan data pasien dan gejala gabungan sebelumnya.

Pemahaman komprehensif tentang algoritma, termasuk sifat dan dasar-dasar strukturalnya, sangat diperlukan dalam

bidang pemrograman dan ilmu komputer. Dengan mengatur algoritma yang jelas, efisien, dan masuk akal secara logis, seseorang dapat meningkatkan kinerja sistem dan mencapai hasil pemecahan masalah yang optimal. Akibatnya, memahami prinsip-prinsip dasar algoritma merupakan langkah awal yang penting bagi setiap pengembang perangkat lunak dan ilmuwan data. Selain itu, penerapan algoritma yang meluas dalam kehidupan sehari-hari menggarisbawahi signifikansi terpenting mereka dalam masyarakat kontemporer. Dengan desain struktural yang kuat, algoritma memiliki kemampuan untuk memfasilitasi berbagai aspek keberadaan, mulai dari kemajuan teknologi hingga proses pengambilan keputusan yang ditingkatkan.

Definisi Algoritma

Algoritma dapat didefinisikan sebagai prosedur sistematis atau serangkaian operasi berurutan tertentu yang dibuat dengan cermat untuk mengatasi masalah tertentu atau untuk mencapai tujuan yang ditentukan. Dalam bidang ilmu komputer dan matematika, algoritma berfungsi sebagai landasan fundamental dalam pemrograman dan analisis data. Sangat penting bahwa setiap algoritma memiliki urutan instruksi yang berbeda, logis, dan dapat dieksekusi yang dapat diselesaikan dalam jangka waktu yang terbatas. Gagasan algoritma telah diakui sejak jaman dahulu, dengan salah satu ilustrasi paling awal adalah algoritma Euclidean, yang digunakan untuk memastikan pembagi umum terbesar (GCD) dari dua bilangan bulat. Seiring kemajuan teknologi terus berkembang, algoritma semakin rumit dan menemukan aplikasi di berbagai domain, termasuk kecerdasan buatan, pemrosesan gambar, pengambilan informasi, dan sistem keamanan (Grami,

2023).

Untuk diklasifikasikan sebagai algoritma yang valid, algoritma harus menunjukkan beberapa karakteristik penting. Pertama, ia harus mematuhi prinsip keterbatasan atau kendala, yang menunjukkan bahwa ia diharuskan memiliki sejumlah langkah yang terbatas dan tidak boleh beroperasi tanpa batas. Kedua, setiap komponen dalam algoritma harus secara eksplisit digambarkan, atau memiliki kepastian, untuk mencegah potensi ambiguitas selama fase eksekusi. Ketiga, algoritma memerlukan input, yang mengacu pada data awal yang penting untuk pemrosesan. Keempat, harus menghasilkan *output*, mewakili hasil konklusif dari serangkaian operasi yang dilakukan. Kelima, algoritma harus efektif dan efisien dalam kemampuan pemecahan masalah mereka, memastikan bahwa sumber daya tidak disia-siakan secara berlebihan.

Dalam kehidupan sehari-hari, algoritma digunakan secara luas untuk mengatasi segudang tantangan. Misalnya, selama upaya kuliner, individu biasanya mengikuti resep tertentu yang menyebutkan langkah-langkah prosedural yang diperlukan untuk menyiapkan hidangan. Resep semacam itu dapat dikonseptualisasikan sebagai algoritma, karena memberikan panduan yang jelas dan terstruktur. Ilustrasi lain dalam bidang teknologi adalah algoritma pencarian yang digunakan oleh mesin pencari, seperti Google. Algoritma pencarian ini menggunakan berbagai teknik untuk menemukan informasi terkait berdasarkan kata kunci yang dimasukkan oleh pengguna. Selanjutnya, algoritma diimplementasikan dalam sistem navigasi GPS untuk memastikan rute yang paling efisien ke tujuan tertentu, dengan mempertimbangkan kondisi lalu lintas *real-time*.

Contoh klasik dari algoritma dasar adalah yang

dirancang untuk memastikan apakah bilangan bulat tertentu ganjil atau genap. Algoritma ini dapat diartikulasikan melalui beberapa langkah berurutan:

1. Masukkan bilangan bulat yang akan dievaluasi
2. Bagi bilangan bulat dengan 2
3. Jika hasil bagi menghasilkan sisa 0, maka bilangan bulat diklasifikasikan sebagai genap; sebaliknya, jika tidak, maka bilangan bulat diklasifikasikan sebagai ganjil
4. Sajikan hasilnya. Algoritma ini mudah dan dapat dijalankan dalam berbagai bahasa pemrograman, seperti Python, Java, atau C ++.

Selain algoritma dasar, ada algoritma yang lebih canggih yang digunakan dalam bidang pemrograman dan teknologi. Salah satu algoritma yang paling umum adalah algoritma pencarian biner. Algoritma ini digunakan untuk menemukan elemen dalam daftar yang telah diurutkan sebelumnya dengan secara iteratif membagi daftar menjadi dua segmen sampai elemen yang diinginkan diidentifikasi. Algoritma pencarian biner menunjukkan efisiensi yang lebih besar daripada metodologi pencarian linier, karena memiliki kompleksitas waktu O (log n), menyiratkan bahwa durasi proses pencarian berkurang jauh karena jumlah elemen dalam daftar meningkat.

Algoritma lain yang memiliki signifikansi signifikan dalam domain teknologi termasuk algoritma penyortiran, terutama *Bubble Sort, Quick Sort,* dan *Merge Sort*. Algoritma penyortiran digunakan untuk mengatur data dalam urutan yang telah ditentukan, mencakup pengaturan naik dan turun. Misalnya, dalam plat*for*m *e-commerce*, algoritma penyortiran memfasilitasi pengaturan daftar produk sesuai dengan kriteria seperti harga, *pop*ularitas, atau evaluasi pengguna. *Quick Sort* dan *Merge Sort* diklasifikasikan sebagai algoritma yang lebih

efisien, menunjukkan kompleksitas waktu rata-rata O (n log n), sedangkan *Bubble Sort* dicirikan oleh kinerja yang relatif lebih lambat, menunjukkan kompleksitas waktu O (n^2) dalam kasus yang paling tidak menguntungkan.

Dalam bidang kecerdasan buatan dan pembelajaran mesin, algoritma memainkan peran penting. Misalnya, algoritma regresi linier digunakan untuk memperkirakan nilai yang didasarkan pada keterkaitan antara variabel independen dan dependen. Algoritma klasifikasi, seperti Pohon Keputusan dan K-Nearest Neighbors (KNN), berperan penting dalam mengkategorikan data ke dalam klasifikasi yang berbeda berdasarkan pola yang dibedakan dalam kumpulan data. Selain itu, algoritma jaringan saraf buatan, yang biasa disebut sebagai Jaringan Saraf Buatan, diimplementasikan di berbagai aplikasi kecerdasan buatan, termasuk pengenalan wajah, pemrosesan bahasa alami, dan diagnostik medis.

Dalam disiplin keamanan siber, algoritma digunakan untuk mengenkripsi data, sehingga melindungi informasi sensitif dari akses yang tidak sah. Algoritma kriptografi, termasuk RSA dan AES, memfasilitasi komunikasi yang aman, terutama dalam konteks seperti transaksi perbankan online dan transfer data melalui internet. Algoritma ini berfungsi dengan mengkodekan data menggunakan kunci yang ditunjuk, memastikan bahwa hanya entitas yang memiliki kunci yang benar yang mampu mendekripsi dan mengakses informasi. Pemanfaatan algoritma kriptografi meningkatkan keamanan data yang dikirimkan melalui jaringan, mengurangi risiko yang terkait dengan peretasan dan pencurian informasi.

Dalam bidang analisis data dan big data, algoritma dimanfaatkan untuk secara efektif memproses dan meneliti sejumlah besar informasi. Misalnya, algoritma pengelompokan,

seperti K-Means, digunakan untuk mengelompokkan data berdasarkan kesamaan tertentu. Algoritma ini sering diterapkan dalam analisis pasar untuk menggambarkan segmen pelanggan berdasarkan perilaku pembelian mereka. Selain itu, algoritma rekomendasi, seperti *Collaborative Filtering*, digunakan pada plat*for*m streaming seperti Netflix dan Spotify untuk mengusulkan konten yang selaras dengan preferensi pengguna berdasarkan interaksi historis mereka.

Di tengah kemajuan teknologi yang cepat, algoritma terus berkembang, menjadi semakin canggih. Dalam konteks Revolusi Industri Keempat, algoritma yang didasarkan pada kecerdasan buatan dan pembelajaran mesin semakin terintegrasi di berbagai sektor, termasuk manufaktur, perawatan kesehatan, dan transportasi. Penerapan algoritma yang bijaksana memiliki potensi untuk meningkatkan efisiensi operasional, mengurangi biaya, dan menghasilkan solusi inovatif di berbagai domain. Akibatnya, memperoleh pemahaman yang komprehensif tentang algoritma muncul sebagai kompetensi penting bagi para profesional yang terlibat dalam bidang teknologi informasi dan ilmu komputer.

Karakteristik Algoritma

Karakteristik awal dari suatu algoritma berkaitan dengan keterbatasan atau keterbatasannya. Algoritma harus memiliki jumlah langkah yang terbatas dan tidak boleh dieksekusi tanpa batas. Dengan tidak adanya titik akhir yang pasti, algoritma semacam itu akan membuat program praktis tidak beroperasi. Ilustrasi yang relevan dari penerapan batasan dalam algoritma adalah algoritma pencarian biner. Algoritma ini berfungsi dengan mempartisi daftar pra-disortir menjadi dua segmen, kemudian membandingkan nilai median dengan nilai target.

Jika nilai target lebih rendah dari nilai median, pencarian dilanjutkan di segmen kiri daftar, dan sebaliknya. Proses iteratif ini berlanjut sampai nilai yang diinginkan ditemukan atau sampai daftar berhenti dibagi. Karena algoritma ini memiliki batasan pada jumlah iterasi yang dilakukan, ini mencontohkan algoritma yang memenuhi karakteristik keterbatasan.

Karakteristik selanjutnya adalah kejelasan atau kepastian. Setiap langkah prosedural dalam algoritma harus digambarkan secara eksplisit, tanpa ambiguitas apa pun. Langkah-langkah ini harus dapat dipahami dan dapat ditafsirkan secara seragam oleh semua individu yang terlibat dengannya, termasuk sistem komputasi ketika dijalankan dalam bentuk terprogram. Misalnya, algoritma yang dirancang untuk menghitung luas persegi panjang harus menguraikan langkah-langkah prosedural yang jelas: awalnya, menerima input untuk panjang dan lebar; selanjutnya, mengeksekusi perkalian panjang dengan lebar; akhirnya, menyajikan nilai yang dihasilkan. Kehadiran langkah-langkah ambigu akan mengakibatkan ketidakmampuan algoritma untuk bekerja dengan sukses selama implementasinya (Witten et al., 2017).

Karakteristik ketiga berkaitan dengan keberadaan input yang diperlukan. Setiap algoritma memerlukan data awal yang digunakan dalam fase pemrosesan. Masukan ini dapat terdiri dari nilai tunggal atau beberapa nilai yang akan menjalani pemrosesan lebih lanjut dalam kerangka algoritmik. Misalnya, dalam algoritma penjumlahan yang melibatkan dua angka, input yang diperlukan terdiri dari dua nilai yang harus dijumlahkan. Dengan tidak adanya input, algoritma akan dibuat non-fungsional karena kurangnya data untuk diproses. Algoritma yang efektif harus memiliki kapasitas untuk mengakomodasi beragam input sambil tetap menghasilkan

hasil yang selaras dengan tujuan yang dimaksudkan.

Karakteristik keempat adalah keberadaan *output* yang diperlukan. Setiap algoritma harus menghasilkan setidaknya satu hasil sebagai *output* yang berkaitan dengan masalah yang ingin diatasi. *Output* dari suatu algoritma dapat bermanifestasi sebagai angka, teks, atau bentuk data lain yang relevan dengan proses yang dieksekusi. Misalnya, dalam algoritma untuk mengubah suhu dari Celcius ke Fahrenheit, *output* yang dihasilkan adalah suhu yang dinyatakan dalam satuan Fahrenheit setelah proses konversi. Algoritma tanpa *output* tidak akan menghasilkan utilitas, karena akan gagal memberikan informasi berharga bagi pengguna atau sistem.

Atribut kelima berkaitan dengan konsep efektivitas. Sangat penting bahwa algoritma dapat dieksekusi dalam jangka waktu yang wajar sambil memanfaatkan sumber daya yang tersedia dengan cara yang efisien. Algoritma yang membutuhkan durasi berlebihan atau menghabiskan jumlah memori yang berlebihan untuk menyelesaikan tugas yang belum sempurna dianggap tidak efektif. Misalnya, dalam perhitungan faktorial untuk angka tertentu, algoritma rekursif yang tidak dioptimalkan dapat mengalami penundaan eksekusi yang signifikan karena volume pemanggilan fungsi berulang yang tinggi. Sebaliknya, algoritma iteratif yang digunakan untuk komputasi faktorial menunjukkan efisiensi yang unggul karena mereka hanya memerlukan *loop* tunggal untuk mendapatkan hasil akhir. Akibatnya, algoritma teladan harus mempertimbangkan faktor efisiensi dengan cermat selama implementasinya.

Atribut keenam adalah determinisme. Sebuah algoritma harus secara konsisten menghasilkan hasil yang sama ketika disajikan dengan input yang identik. Dengan kata lain,

algoritma yang dieksekusi di bawah parameter yang sama harus selalu menghasilkan *output* yang sama tanpa adanya ketidakpastian. Misalnya, algoritma penyortiran seperti *Bubble Sort* akan secara konsisten mengembalikan daftar yang diurutkan dalam urutan yang identik ketika diberikan dengan input yang sama. Sebaliknya, algoritma nondeterministik, seperti yang digunakan dalam pembelajaran mesin berbasis pengacakan, dapat menghasilkan hasil yang berbeda meskipun menerima input yang sama, tergantung pada parameter tertentu. Dalam ranah sistem yang memerlukan hasil yang konsisten, algoritma deterministik sangat disukai.

Sebagai kesimpulan, atribut algoritma sangat penting dalam mengevaluasi kualitas dan kemanjuran algoritma dalam mengatasi masalah. Algoritma yang efektif harus memiliki kendala pada langkah-langkah, kejelasan dalam instruksi, input yang sesuai, *output* terkait, efisiensi dalam pemanfaatan sumber daya, dan determinisme dalam memberikan hasil yang konsisten. Dengan memahami karakteristik ini, pengembang perangkat lunak dan ilmuwan data dapat merumuskan algoritma yang lebih efisien dan optimal yang berlaku di seluruh spektrum penggunaan yang luas. Ketika lanskap digital terus berkembang, pemahaman tentang karakteristik algoritmik menjadi semakin penting, mengingat bahwa hampir setiap aspek kehidupan kontemporer didasarkan pada sistem komputasi yang didorong oleh algoritma yang dibuat dengan cermat.

BAB 5 NOTASI ALGORITMIK: *PSEUDOCODE* DAN *FLOWCHART*

Pendahuluan

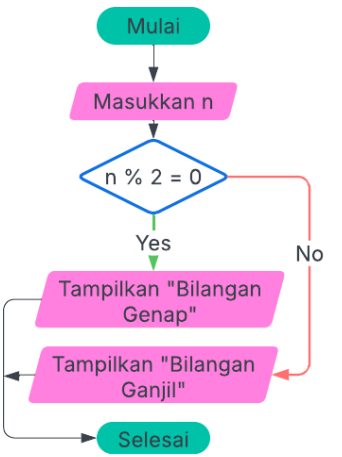
Notasi algoritmik adalah cara untuk merepresentasikan langkah-langkah penyelesaian masalah (algoritma) sebelum diimplementasikannya ke dalam kode program (Cormen et al., 2022). Dalam pemrograman, notasi ini membantu *programmer* merancang solusi secara sistematis, memahami alur logika, dan berkomunikasi dengan tim (Hazzan & Lapidot, 2021). Ada dua jenis notasi algoritmik yang umum digunakan:

1. *Pseudocode*: Representasi berbasis teks yang menyerupai bahasa pemrograman tetapi lebih sederhana dan tidak terikat pada aturan sintaks bahasa tertentu (Chen & Wang, 2021).
2. *Flowchart*: Representasi grafis yang menggunakan simbol-simbol untuk menggambarkan alur proses (Alotaibi & Alotaibi, 2020).

Keduanya memiliki kelebihan masing-masing: *flowchart* lebih visual dan mudah dipahami secara intuitif, sedangkan *Pseudocode* lebih fleksibel dan cocok untuk dokumentasi teknis (Luxton-Reilly & Denny, 2020). Dalam bab ini, kita akan mempelajari kedua jenis notasi ini, bagaimana cara membuatnya, serta kapan menggunakannya dalam pengembangan algoritma.

Berikut ini adalah contoh menentukan bilangan genap atau ganjil menggunakan *Pseudocode* dan *flowchart*.

Pseudocode	*Flowchart*

MULAI
MASUKKAN n
JIKA n % 2 = 0 MAKA
  TAMPILKAN "Bilangan Genap"
SELAIN ITU
  TAMPILKAN "Bilangan Ganjil"
AKHIR JIKA
SELESAI
	

Pengertian *Pseudocode*

Pseudocode adalah cara menulis algoritma dalam bentuk teks yang menyerupai bahasa pemrograman, tetapi tidak terikat pada aturan sintaks ketat seperti bahasa pemrograman sungguhan (Chen & Wang, 2021). *Pseudocode* menggunakan bahasa yang mudah dipahami oleh manusia, sering kali dalam bahasa sehari-hari, untuk menggambarkan langkah-langkah algoritma (Gellenbeck & McConnell, 2020).

Kelebihan *Pseudocode*:

1. Mudah dan cepat ditulis dibandingkan menggambar *flowchart* (Luxton-Reilly & Denny, 2020).
2. Fleksibel, tidak perlu mengikuti aturan sintaks tertentu.
3. Cocok untuk mendokumentasikan algoritma sebelum coding (Robins & Rountree, 2023).

Kekurangan *Pseudocode*:

1. Kurang visual dibandingkan *flowchart*, sehingga mungkin sulit dipahami oleh orang yang lebih suka

representasi grafis (Katai & Toth, 2023).

2. Tidak ada standar baku, sehingga penulisan bisa bervariasi antar *programmer*.

Pseudocode sering digunakan sebagai langkah perantara antara perancangan algoritma dan penulisan kode dalam bahasa pemrograman (Hazzan & Lapidot, 2021).

Berikut ini adalah contoh *Pseudocode* untuk algoritma menghitung faktorial.

ALGORITMA Faktorial

1. MULAI
2. MASUKKAN n
3. SET faktorial = 1
4. JIKA n <= 0 MAKA TAMPILKAN "Faktorial tidak didefinisikan untuk bilangan negatif"
   SELESAI
5. UNTUK i dari 1 hingga n faktorial = faktorial * i
6. TAMPILKAN factorial
7. SELESAI

Struktur Dasar *Pseudocode*

Pseudocode biasanya mengikuti struktur yang mencerminkan elemen dasar pemrograman. Berikut elemen-elemen utama (McCracken & Waters, 2022):

1. Deklarasi Variabel: Menyatakan variabel yang akan digunakan (misalnya, "SET x = 0").
2. Input/*Output*: Mengambil masukan dari pengguna atau menampilkan hasil (misalnya, "MASUKKAN x", "TAMPILKAN hasil").
3. Keputusan (Kondisi): Menggunakan struktur *if-else* (misalnya, "JIKA x > 0 MAKA ... SELAIN ITU ...").
4. Perulangan (*Loop*): Menggunakan struktur seperti *FOR*

atau *WHILE* (misalnya, "UNTUK i dari 1 hingga 10").

5. Proses: Langkah komputasi atau assignment (misalnya, "hasil = x + y").

Contoh sederhana:

Elemen	Contoh Penulisan dalam *Pseudocode*
Deklarasi	SET total = 0
Input	MASUKKAN bilangan
Keputusan	JIKA bilangan > 0 MAKA TAMPILKAN 'Positif'
Perulangan	UNTUK i dari 1 hingga 5, total = total +

Pseudocode yang terstruktur dengan baik membantu *programmer* memahami logika sebelum coding (Gellenbeck & McConnell, 2020).

Contoh *Pseudocode*

Mari kita tuliskan lagi *Pseudocode* untuk algoritma menghitung faktorial (Robins & Rountree, 2023):

ALGORITMA Faktorial

1. MULAI
2. MASUKKAN n
3. SET faktorial = 1
4. JIKA n <= 0 MAKA
5. TAMPILKAN "Faktorial tidak didefinisikan untuk bilangan negatif"
6. SELESAI
7. AKHIR JIKA
8. UNTUK i dari 1 hingga n
9. faktorial = faktorial * i
10. AKHIR UNTUK

11. TAMPILKAN "Faktorial dari ", n, " adalah ", faktorial
12. SELESAI

Penjelasan:
Baris 1-2: Memulai algoritma dan meminta input n.
Baris 3: Inisialisasi variabel faktorial.
Baris 4-7: Memeriksa apakah n valid (n > 0).
Baris 8-10: Perulangan untuk menghitung faktorial.
Baris 11: Menampilkan hasil.
Baris 12: Akhir dari algoritma.

Contoh lain, mencari bilangan maksimum dari tiga bilangan:

ALGORITMA CariMaksimum
1. MULAI
2. MASUKKAN a, b, c
3. SET maks = a
4. JIKA b > maks MAKA
5. maks = b
6. AKHIR JIKA
7. JIKA c > maks MAKA
8. maks = c
9. AKHIR JIKA
10. TAMPILKAN "Bilangan terbesar adalah ", maks
11. SELESAI

Pseudocode seperti ini membantu mempersiapkan langkah-langkah sebelum implementasi kode (Vihavainen & Vikberg, 2020).

Pengertian *Flowchart*

Flowchart (bagan alir) adalah representasi grafis dari

sebuah algoritma atau proses yang menggunakan simbol-simbol standar untuk menggambarkan langkah-langkah penyelesaian masalah (Sedgewick & Wayne, 2022). *Flowchart* biasanya digunakan untuk:

1. Memvisualisasikan alur logika dari sebuah program (Elmqvist & Irani, 2021).
2. Membantu *programmer* atau tim memahami proses secara intuitif.
3. Memudahkan identifikasi kesalahan dalam logika sebelum menulis kode (Vihavainen & Vikberg, 2020).

Kelebihan *Flowchart*:

1. Mudah dipahami, terutama untuk orang yang lebih visual (Alotaibi & Alotaibi, 2020).
2. Membantu merancang alur kerja sebelum coding dimulai.
3. Efektif untuk presentasi atau diskusi tim (Katai & Toth, 2023).

Kekurangan *Flowchart*:

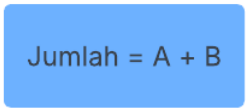
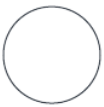
1. Sulit dibuat untuk algoritma yang sangat kompleks karena dapat menjadi terlalu besar atau berantakan (Futschek & Moschitz, 2022).
2. Memerlukan waktu lebih lama untuk menggambar dibandingkan menulis *Pseudocode*.
3. Tidak cocok untuk mendokumentasikan detail teknis.

Flowchart menggunakan simbol-simbol standar yang mewakili jenis langkah tertentu, seperti proses, keputusan, atau input/*output*. Simbol-simbol ini dihubungkan dengan panah untuk menunjukkan alur proses (Malmi & Korhonen, 2021).

Berikut contoh *flowchart* untuk menghitung jumlah dua bilangan.

Simbol-Simbol dalam *Flowchart*

Flowchart menggunakan simbol-simbol standar yang memiliki arti tertentu. Berikut adalah simbol-simbol utama yang sering digunakan (McCracken & Waters, 2022):

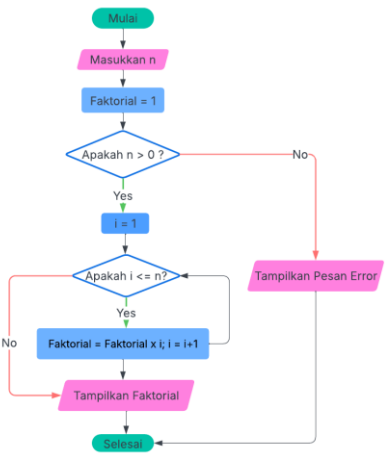
Gambar Simbol	Nama Simbol dan Fungsinya	Contoh Penggunaan
	**Oval**: Menandakan awal atau akhir dari sebuah proses. Biasanya digunakan untuk "Mulai" dan "Selesai".	
	**Persegi**: Menunjukkan langkah proses, seperti perhitungan atau assignment (misalnya, "Jumlah = A + B").	
	**Belah Ketupat**: Digunakan untuk keputu-san, biasanya dengan per-tanyaan yang memiliki jawaban "Ya" atau "Tidak" (misalnya, "Apakah A > B?").	
	**Lingkaran Kecil**: Konektor, digunakan untuk menghubungkan bagian-bagian *flowchart* yang terpisah agar tidak berantakan.	

	**Panah**: Menunjukkan arah alur proses dari satu langkah ke langkah berikutnya.	
	**Parallelogram**: Digunakan untuk input atau *output* (misalnya, "Masukkan nilai A" atau "Tampilkan hasil").	

Setiap simbol harus digunakan sesuai fungsinya agar *flowchart* mudah dipahami oleh orang lain (Dagiene & Stupuriene, 2020).

Contoh *Flowchart*

Mari kita buat *flowchart* untuk menghitung faktorial sebuah bilangan. Faktorial dari bilangan n (ditulis n!) adalah hasil perkalian semua bilangan bulat positif dari 1 hingga n. Contoh: 5! = 5 × 4 × 3 × 2 × 1 = 120 (Sedgewick & Wayne, 2022). Berikut langkah-langkahnya:



Flowchart ini akan menggunakan simbol-simbol seperti oval (awal/akhir), paralelogram (input/*output*), belah ketupat (keputusan), dan persegi (proses). Visualisasi ini membantu memahami alur algoritma secara intuitif (Malmi & Korhonen, 2021).

Perbandingan *Flowchart* dan *Pseudocode*

Flowchart dan *Pseudocode* memiliki tujuan yang sama—membantu merancang algoritma—tetapi berbeda dalam pendekatan (Futschek & Moschitz, 2022). Beberapa perbedaannya dapat dilihat sebagai berikut.

Aspek	*Pseudocode*	*Flowchart*
Kelebihan	Cepat ditulis, fleksibel, cocok untuk dokumentasi teknis (Chen & Wang, 2021).	Visual, mudah dipahami, cocok untuk presentasi atau diskusi (Elmqvist & Irani, 2021).
Kekurangan	Kurang visual, bisa ambigu jika tidak ditulis dengan baik (Luxton-Reilly & Denny, 2020).	Sulit untuk algoritma kompleks, memakan waktu untuk menggambar (Katai & Toth, 2023).
Kapan Digunakan	Saat Anda ingin fokus pada logika tanpa terganggu oleh sintaks bahasa pemrograman.	Saat Anda perlu mempresentasikan alur kepada tim atau memahami logika secara visual.

Dalam praktiknya, kombinasi keduanya sering digunakan: *flowchart* untuk merancang alur secara visual, lalu *Pseudocode* untuk merinci langkah-langkah sebelum coding (Dagiene & Stupuriene, 2020).

Latihan

Berikut ini ada beberapa latihan yang dapat dierjakan untuk lebih memahami tentang *Pseudocode* dan *flowchart*.

1. Buat *flowchart* untuk algoritma yang menentukan apakah sebuah bilangan genap atau ganjil.
2. Tulis *Pseudocode* untuk menghitung rata-rata tiga bilangan yang dimasukkan pengguna.
3. Buat *flowchart* untuk algoritma yang mencetak bilangan 1 hingga 10.
4. Tulis *Pseudocode* untuk algoritma yang memeriksa apakah sebuah tahun adalah tahun kabisat.

Notasi algoritmik seperti *flowchart* dan *Pseudocode* adalah alat penting dalam merancang algoritma (Cormen et al., 2022). *Flowchart* membantu memvisualisasikan alur logika secara grafis, sedangkan *Pseudocode* memungkinkan penulisan langkah-langkah secara tekstual tanpa khawatir tentang sintaks (Gellenbeck & McConnell, 2020). Dengan memahami kedua metode ini, Anda dapat merancang solusi yang lebih terstruktur dan efisien sebelum menulis kode. Kombinasikan keduanya sesuai kebutuhan: gunakan *flowchart* untuk perencanaan visual dan *Pseudocode* untuk dokumentasi teknis (Robins & Rountree, 2023).

BAB 6 STRUKTUR KONTROL DALAM ALGORITMA: *SEQUENCING*, *SELECTION*, *ITERATION*

Pendahuluan

Struktur kontrol dalam algoritma digunakan oleh para pengembang perangkat lunak, *programmer*, dan siapa saja yang merancang atau menulis program komputer. Mereka menggunakan struktur kontrol untuk mengatur alur eksekusi program agar dapat menjalankan tugas secara efisien dan sesuai dengan kebutuhan.

Misalnya, seorang *programmer* yang membuat aplikasi atau sistem informasi akan menggunakan struktur kontrol seperti percabangan (*if-else*) dan pengulangan (*loop*) untuk mengatur proses pengolahan data, pengambilan keputusan otomatis, atau pengulangan tugas tertentu. Selain *programmer*, pelajar yang belajar pemrograman juga mempelajari struktur kontrol ini sebagai bagian dari dasar-dasar pemrograman untuk membangun algoritma yang logis dan efisien.

Secara umum, siapa pun yang mengembangkan algoritma atau menulis kode program—baik itu *programmer* profesional, mahasiswa, pengembang aplikasi, maupun pengembang sistem—menggunakan struktur kontrol dalam algoritma mereka. Fungsi utama penggunaannya adalah untuk memastikan program dapat berjalan secara dinamis dan mampu menangani berbagai situasi serta kondisi yang berbeda.

Struktur kontrol merupakan unsur penting dalam

pemrograman yang memungkinkan pengembang untuk mengendalikan alur eksekusi program sesuai dengan kondisi tertentu. Dalam konteks algoritma, struktur kontrol terbagi menjadi tiga jenis utama: *sequencing*, *selection*, dan *iteration*. Masing-masing memiliki fungsi dan cara kerja yang berbeda dalam menentukan langkah-langkah eksekusi program, sehingga memudahkan dalam menyusun solusi yang efisien dan logis.

Struktur kontrol *sequencing* merupakan proses di mana langkah-langkah dalam algoritma dieksekusi secara berurutan sesuai urutan penulisan. Dalam struktur ini, setiap perintah dieksekusi satu per satu secara berurutan tanpa adanya kondisi tertentu yang mengubah alur eksekusi. *Sequencing* merupakan dasar dari semua algoritma, karena memastikan bahwa program berjalan sesuai urutan yang telah ditentukan, mulai dari awal hingga akhir. Kelebihan dari struktur ini adalah kesederhanaannya dan kemudahan dalam pemahaman alur kerja program.

Selanjutnya, struktur kontrol *selection* digunakan untuk membuat keputusan berdasarkan kondisi tertentu. Dengan menggunakan pernyataan kondisi seperti if, *else*, dan switch, program dapat memilih jalur eksekusi yang berbeda tergantung pada nilai variabel atau hasil evaluasi kondisi. Struktur ini sangat penting dalam pengembangan algoritma yang membutuhkan pengambilan keputusan, seperti menentukan apakah suatu angka termasuk dalam kategori tertentu atau tidak. Melalui *selection*, program menjadi lebih fleksibel dan mampu menangani berbagai situasi yang berbeda secara dinamis.

Struktur Kontrol Algoritma *Sequencing*

Struktur kontrol algoritma *sequencing* berfungsi sebagai dasar utama dalam menjalankan instruksi secara berurutan dari awal hingga akhir. Tujuan utama dari *sequencing* adalah memastikan bahwa setiap langkah dalam algoritma dieksekusi secara sistematis dan sesuai urutan yang telah ditentukan, tanpa melompati atau melewatkan langkah tertentu. Dengan menggunakan struktur ini, proses penyelesaian masalah menjadi lebih terorganisir dan mudah dipahami karena mengikuti alur yang linier dan logis.

Selain itu, *sequencing* membantu memastikan bahwa program berjalan dengan lancar dan konsisten, karena instruksi dieksekusi satu per satu sesuai urutan. Hal ini sangat penting agar hasil yang diperoleh dari algoritma bisa diprediksi dan dapat diandalkan. *Sequencing* juga menjadi fondasi dari pembuatan algoritma yang lebih kompleks, karena struktur ini menyediakan kerangka dasar sebelum menambahkan fitur lain seperti percabangan atau pengulangan.

Secara umum, struktur kontrol *sequencing* sangat penting dalam algoritma karena memberikan kerangka kerja yang sederhana dan terstruktur untuk menjalankan instruksi secara berurutan. Tanpa *sequencing*, proses eksekusi akan kacau dan sulit dikendalikan, sehingga algoritma tidak akan mampu menyelesaikan masalah secara efisien dan efektif. Dengan demikian, *sequencing* adalah langkah awal yang esensial untuk memastikan algoritma berjalan dengan baik dan menghasilkan *output* yang benar.

Contoh Penggunaan Struktur Kontrol Algoritma *Sequencing*

1. *Sequencing* dalam algoritma

Berikut adalah contoh penggunaan struktur kontrol *sequencing* dalam algoritma:
Contoh Kasus:
Menghitung total harga belanja setelah menambahkan pajak.

Algoritma langkah demi langkah:

a. Input harga barang.
b. Hitung pajak 10% dari harga barang.
c. Hitung total harga = harga barang + pajak.
d. Tampilkan total harga.

2. Implementasi dengan struktur *sequencing*:
Setiap langkah dieksekusi secara berurutan sesuai urutan yang telah ditentukan, tanpa adanya percabangan atau pengulangan.

```
plaintext
Input harga_barang
pajak = harga_barang * 0.10
total_harga = harga_barang + pajak
Tampilkan total_harga
```

Langkah-langkahnya dilakukan secara berurutan dari atas ke bawah. *Sequencing* memastikan bahwa setiap instruksi berjalan sesuai urutan tanpa ada yang terlewat, sehingga hasilnya adalah perhitungan total harga secara benar dan terstruktur.
Contoh ini menunjukkan bahwa dalam algoritma sederhana, struktur *sequencing* sangat penting untuk menjamin langkah-langkah dilakukan secara sistematis dan logis.

3. Implementasi dengan Bahasa Pemrograman Pascal
   Contoh kasus sederhana yang menggunakan struktur kontrol *sequencing* dalam bahasa Pascal. *Sequencing* adalah proses eksekusi satu perintah setelah perintah lainnya secara berurutan.
   Contoh Kasus: Menghitung dan menampilkan luas persegi. Program ini akan meminta pengguna memasukkan panjang sisi persegi, kemudian menghitung dan menampilkan luasnya.

```
program HitungLuasPersegi;

uses crt;

var
  sisi, luas: Real;

begin
  clrscr;
// Input: meminta pengguna memasukkan panjang sisi persegi
  Write('Masukkan panjang sisi persegi: ');
  ReadLn(sisi);

  // Proses: menghitung luas persegi (sisi * sisi)
  luas := sisi * sisi;

  // Output: menampilkan hasil luas persegi
  WriteLn('Luas persegi adalah: ', luas:0:2);
  ReadLn;
end.
```

Penjelasan:
Sequencing: Eksekusi program dimulai dari baris clrscr;, kemudian ke Write, ReadLn, proses perhitungan, dan akhirnya menampilkan hasil.
Langkah-langkah berurutan: Input → Pengolahan → *Output*.
Ini adalah contoh sederhana bagaimana struktur *sequencing* bekerja dalam Pascal, di mana setiap perintah dieksekusi secara berurutan sesuai urutan penulisannya.

Contoh Kasus: Menghitung Nilai Akhir Mahasiswa

Deskripsi:
Program akan meminta input nilai tugas, UTS, dan UAS dari pengguna secara berurutan, kemudian menghitung nilai akhir berdasarkan bobot tertentu, dan menampilkan hasilnya.

```
program HitungNilaiAkhirMahasiswa;

uses crt;

var
  tugas, uts, uas, nilaiAkhir: Real;

begin
  clrscr;

  // Input nilai tugas
  Write('Masukkan nilai tugas (0-100): ');
  ReadLn(tugas);
```

```
        // Input nilai UTS
        Write('Masukkan nilai UTS (0-100): ');
        ReadLn(uts);

        // Input nilai UAS
        Write('Masukkan nilai UAS (0-100): ');
        ReadLn(uas);

        // Menghitung nilai akhir dengan bobot: tugas 20%, UTS 30%, UAS 50%
        nilaiAkhir := (tugas * 0.2) + (uts * 0.3) + (uas * 0.5);

        // Menampilkan hasil
        WriteLn('Nilai akhir mahasiswa adalah: ', nilai-Akhir:0:2);

        ReadLn;
      end.
```

Penjelasan:

Sequencing: Program dimulai dari clrscr;. Kemudian mengeksekusi Write dan ReadLn secara berurutan untuk setiap input. Setelah semua input diperoleh, proses penghitungan dilakukan secara berurutan. Akhirnya, hasil ditampilkan dengan WriteLn.

Langkah-langkah berurutan: Input nilai tugas → Input nilai UTS → Input nilai UAS → Hitung nilai akhir → Tampilkan hasil.

Variabel tugas, uts, dan uas diisi secara berurutan, mengikuti alur program. Penggunaan :0:2 pada nilai Akhir

untuk menampilkan dua angka di belakang koma.

Struktur Kontrol Algoritma *Selection*

Struktur kontrol algoritma *selection*, atau percabangan, digunakan untuk membuat keputusan dalam proses eksekusi program. Fungsi utama dari *selection* adalah memungkinkan algoritma memilih jalur tertentu berdasarkan kondisi atau kriteria yang telah ditetapkan. Dengan adanya struktur ini, program dapat menanggapi berbagai situasi secara berbeda, sehingga menjadi lebih fleksibel dan adaptif terhadap data atau kondisi yang berubah-ubah.

Contoh paling umum dari struktur *selection* adalah pernyataan *if-else*, yang memungkinkan program menjalankan blok kode tertentu jika kondisi terpenuhi, dan blok lain jika kondisi tidak terpenuhi. Misalnya, dalam algoritma menentukan apakah seseorang lulus atau tidak berdasarkan nilai ujian, struktur *selection* akan membantu menentukan langkah apa yang harus diambil sesuai hasil nilai tersebut. Tanpa adanya struktur *selection*, program akan selalu menjalankan instruksi yang sama tanpa mempertimbangkan kondisi yang berbeda, sehingga tidak mampu melakukan pengambilan keputusan yang dinamis.

Contoh Penggunaan Struktur Kontrol Algoritma *Selection*

Secara keseluruhan, struktur *selection* sangat penting dalam algoritma karena memberikan kemampuan untuk melakukan pengambilan keputusan otomatis. Ini memungkinkan algoritma untuk lebih cerdas dan efisien dalam menyelesaikan masalah, karena dapat menyesuaikan langkah-langkahnya berdasarkan kondisi yang sedang terjadi. Dengan

demikian, *selection* menjadikan algoritma lebih fleksibel, mampu mengatasi berbagai situasi, dan meningkatkan efektivitas proses penyelesaian masalah. Berikut adalah contoh penggunaan struktur kontrol *selection* (percabangan) dalam algoritma:

1. *Selection* dalam algoritma
   Contoh Kasus
   Menentukan apakah sebuah angka adalah ganjil atau genap.
   Algoritma langkah demi langkah:
   a. Input angka.
   b. Jika angka dibagi 2 sisanya adalah 0, maka angka tersebut genap.
   c. Jika tidak, maka angka tersebut ganjil.
   d. Tampilkan hasilnya.
2. Implementasi dengan struktur *selection* (*if-else*):
   Program akan memilih jalur eksekusi berdasarkan kondisi yang diperiksa.

```
plaintext
Input angka
if (angka % 2 == 0) {
        Tampilkan "Angka genap"
} else {
        Tampilkan "Angka ganjil"
}
```

   Penjelasan:
   Program menggunakan struktur *selection* untuk membuat keputusan. Jika kondisi angka % 2 == 0 terpenuhi, maka program akan menampilkan bahwa

angka tersebut genap. Jika tidak, maka akan menampilkan bahwa angka tersebut ganjil.

Contoh ini menunjukkan bahwa struktur *selection* memungkinkan algoritma untuk melakukan pengambilan keputusan otomatis berdasarkan kondisi tertentu, sehingga hasilnya lebih dinamis dan sesuai dengan data yang diberikan.

Implementasi Struktur Kontrol *Selection* (*If-Else*) dalam Bahasa Pemrograman Pascal

Contoh Kasus: Menghitung Grade Berdasarkan Nilai Siswa. Misalkan kita ingin menentukan grade dari nilai siswa dengan aturan berikut.

Nilai >= 85: Grade A
Nilai >= 70 dan = 55 dan = 85 then
  grade := 'A'
 else if nilai >= 70 then
  grade := 'B'
 else if nilai >= 55 then
  grade := 'C'
 else
  grade := 'D';

 WriteLn('Grade siswa: ', grade);

 // Alternatif: Menggunakan case setelah konversi ke rentang
 case nilai div 10 of
  9, 10: grade := 'A';
  8: grade := 'B';
  7: grade := 'B';

```
    6: grade := 'C';
    5: grade := 'C';
    else grade := 'D';
  end;

  WriteLn('Grade berdasarkan switch case: ', grade);
  ReadLn;
end.
```

Penjelasan:

Program meminta input nilai dari pengguna. Menggunakan *if-else* untuk menentukan grade secara langsung berdasarkan rentang nilai.

Alternatif menggunakan case untuk memetakan nilai ke grade berdasarkan hasil pembagian nilai dengan 10 (nilai div 10). Ini memudahkan pengelompokan nilai dalam kategori tertentu.

Struktur Kontrol Algoritma *Iteration*

Struktur kontrol algoritma *iteration*, atau pengulangan, dalam algoritma digunakan untuk menjalankan bagian tertentu dari program secara berulang-ulang selama kondisi tertentu terpenuhi. Fungsi utama dari *iteration* adalah memudahkan proses pengulangan tugas yang sama tanpa harus menulis kode yang berulang secara manual, sehingga meningkatkan efisiensi dan mengurangi kemungkinan kesalahan.

Contoh paling umum dari struktur *iteration* adalah *loop* seperti *for, while,* dan *do-while*. Misalnya, jika ingin menghitung jumlah dari angka 1 sampai 10, maka dengan *loop*, proses penjumlahan dilakukan berulang kali secara otomatis sampai kondisi terpenuhi. Tanpa adanya struktur *iteration*,

programmer harus menulis kode berulang untuk setiap langkah, yang tentu tidak praktis dan rawan kesalahan.

Secara umum, struktur *iteration* sangat penting dalam algoritma karena membantu mengotomatisasi proses berulang, menghemat waktu, dan meningkatkan keandalan program. Penggunaan *loop* memungkinkan algoritma menyelesaikan tugas yang memerlukan pengulangan langkah tertentu secara otomatis dan terkontrol, sehingga membuat proses pengolahan data dalam jumlah besar menjadi lebih efisien dan terstruktur.

Iteration adalah proses mengulang bagian tertentu dari algoritma sebanyak tertentu, sampai memenuhi kondisi tertentu. Biasanya digunakan untuk melakukan pengulangan tugas yang sama berulang kali, seperti menghitung jumlah, menampilkan data berulang, atau proses lainnya.

Contoh Penggunaan Struktur Kontrol Algoritma *Iteration*

Struktur *Iteration* yang Umum:

1. *For* — digunakan untuk pengulangan dengan jumlah tertentu
2. *While* — digunakan untuk pengulangan selama kondisi tertentu terpenuhi.
3. Repeat..until — digunakan untuk pengulangan sampai kondisi tertentu terpenuhi, minimal satu kali.

Contoh Kasus: Menghitung Jumlah Bilangan dari 1 sampai N

Deskripsi:

Program akan meminta input sebuah angka N, kemudian menggunakan *loop for* untuk menjumlahkan semua bilangan dari 1 sampai N, lalu menampilkan hasilnya.

```
program JumlahBilangan;
```

```
uses crt;

var
  N, i, jumlah: Integer;

begin
  clrscr;

  // Input dari pengguna
  Write('Masukkan nilai N: ');
  ReadLn(N);

  jumlah := 0;

  // Looping using for
  for i := 1 to N do
  begin
    jumlah := jumlah + i;  // Menjumlahkan bilangan dari 1 sampai N
  end;

  // Menampilkan hasil
  WriteLn('Jumlah dari 1 sampai ', N, ' adalah: ', jumlah);

  ReadLn;
end.
```

Penjelasan:

Iteration dengan *for*: *Loop* akan berjalan dari i = 1 sampai i = N.

Pada setiap iterasi, nilai i ditambahkan ke variabel jumlah. Setelah selesai, hasil jumlah semua bilangan dari 1 sampai N ditampilkan.

Contoh Kasus: Menghitung Faktorial dengan *While*

Deskripsi:

Menghitung faktorial dari sebuah angka N menggunakan *loop while*.

```
program Faktorial;

uses crt;

var
  N, i, faktorial: Integer;

begin
  clrscr;

  // Input dari pengguna
  Write('Masukkan angka N: ');
  ReadLn(N);

  i := 1;
  faktorial := 1;

  // Loop menggunakan while
  while i <= N do
  begin
    faktorial := faktorial * i;
    i := i + 1;
```

```
  end;

  // Menampilkan hasil
  WriteLn('Faktorial dari ', N, ' adalah: ', faktorial);

  ReadLn;
end.
```

Intinya:

Iteration memungkinkan menjalankan bagian kode berulang kali hingga kondisi tertentu terpenuhi. Pemilihan struktur *loop* tergantung kebutuhan: *for* untuk jumlah pasti, *while* atau repeat..until untuk kondisi tertentu.

Contoh Kasus: Menghitung Faktorial dari suatu Angka

Deskripsi:

Program ini akan meminta pengguna memasukkan sebuah angka positif, lalu menghitung dan menampilkan faktorial dari angka tersebut menggunakan struktur perulangan *for*.

Program Pascal Menggunakan *for* untuk Faktorial

```
program Faktorial;

uses crt;

var
  n, i, faktorial: LongInt;

begin
```

```
  clrscr;
  Write('Masukkan angka positif: ');
  ReadLn(n);

  // Inisialisasi faktorial
  faktorial := 1;

  // Menggunakan loop for untuk menghitung faktorial
  for i := 1 to n do
    faktorial := faktorial * i;

  WriteLn('Faktorial dari ', n, ' adalah: ', faktorial);
  ReadLn;
end.
```

Penjelasan:

Pengguna diminta memasukkan angka n. Variabel faktorial diinisialisasi dengan 1. *Loop for* i := 1 to n mengalikan faktorial dengan setiap angka dari 1 sampai n. Setelah *loop* selesai, hasil faktorial ditampilkan.

Alternatif: Menggunakan *while* untuk menghitung faktorial

```
program FaktorialWhile;

uses crt;

var
  n, i, faktorial: LongInt;

begin
```

```
  clrscr;
  Write('Masukkan angka positif: ');
  ReadLn(n);

  i := 1;
  faktorial := 1;

  while i <= n do
  begin
    faktorial := faktorial * i;
    i := i + 1;
  end;

  WriteLn('Faktorial dari ', n, ' adalah: ', faktorial);
  ReadLn;
end.
```

Ringkasan:

Perulangan *for* cocok digunakan jika jumlah iterasi pasti. Perulangan *while* cocok digunakan jika jumlah iterasi tergantung kondisi tertentu.

BAB 7 TIPE DATA DAN VARIABEL DALAM ALGORITMA

Pendahuluan

Bab ini mengenalkan berbagai fondasi Bahasa pemrograman baik Bahasa pemrograman c/c++ maupun Bahasa pemrograman java seperti tipe data, variabel. Di dalam lingkungan para pemrograman. Kita akan menganalisis tipe data, variabel dalam lingkup algoritma pemrograman.

Pengenal (Identifier) adalah suatu nama yang digunakan dalam program untuk menyatakan variabel, fungsi, dan lain-lain. Aturan umum yang berlaku bagi pengenal adalah sebagai berikut.

1. Diawali dengan huruf atau simbol garis-bawah (_).
2. Penggunaan yang lain seperti huruf, angka, ataupun simbol underline / garis-bawah (_).
3. Ada perbedaan antara huruf kecil dan besar (Kadir, 2012).

Pada implementasi dan latihan, ada kaidah tidak tertulis dalam pemberian nama. Misalnya, nama variabel dengan menggunakan permulaan huruf kecil dan pada awal kata berikutnya ditulis menggunakan huruf besar. Penggunaan nama konstanta dituliskan sepenuhnya dengan huruf besar.

Pada ciri-ciri yang sempurna dan yang *error* dapat ditampilkan pada tabel berikut ini.

Tabel 7.1 Model/Pola pernyataan peubah

Asli/Benar	Salah	
X	Semester 1	pakai spasi
angka	5 hari	dimulai dengan angka
semester_3	anak"uang	ada tanda "
PERUSAHAAN		

Mengetahui tentang Tipe Data

Dalam program java tersedia banyak macam tipe data dasar/primitif, seperti pada tabel sebagai berikut.

Tabel 7.2 Tipe data primitive

Tipe	Keterangan
Char	Menyatakan sebuah karakter. Contoh sebuah yaitu A, f, 9, atau *
double	Menyatakan bilangan real dengan ketelitian tinggi (15 digit), Dapat menampung bilangan antara 10^{-308} sampai dengan 10^{308}
Float	Menyatakan bilangan real dengan ketelitian rendah (6-7 digit). Dapat menampung bilangan antara 10^{-38} sampai dengan 10^{38}
Byte	Menyatakan bilangan bulat antar -128 sampai dengan +127
Short	Menyatakan bilangan bulat antara -32768 sampai dengan +32767
int	Menyatakan bilangan bulat antara -2147483648 sampai dengan +2147483647
Long	Menyatakan bilangan bulat antara -9223372036854775808 sampai dengan-+9223372036854775807
bolean	Menyatakan nilai logika *true* atau *false*. Nilai *true* berarti benar dan nilai *false* berarti salah

Penetapan tipe data yang benar mesti diselaraskan pada

data yang akan diolah. Misalnya, apabila akan mengoperasikan suatu bilangan bulat, boleh menyeleksi short, int, atau long. Akan tetapi, yang mana tipe data yang akan digunakan? Demi mempermudah dalam memilih tipe data, gunakanlah petunjuk sebagai berikut (Kadir, Algoritma, 2012):

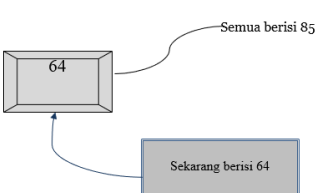
1. memilih tipe data yang membutuhksn penyimpanan yang paling sedikit, akan tetapi
2. bisa menerima semua peluang yang akan terjadi

Penambahan kata unsigned sering dilakukan didepan tipe seperti int atau long. Tujuannya adalah membuat tipe data hanya bisa menampung nilai positif. Misalnya, unsigned short membuat tipe data menampung bilangan antara 0 sampai dengan 65535 (tidak lagi dari -32768 sampai dengan +32767). (Kadir, Algoritma, 2012).

Variabel

Mengetahui tentang Variabel

Variabel yaitu suatu tempat penyimpanan dalam memori komputer untuk menyimpan nilai atau data. Nilai yang disimpan dalam variable bisa berubah-ubah selama program dijalankan. Variabel memiliki nama tipe data, dan nilai yang diubah sesuai dengan kebutuhan program. Gambar 7.1 mengilustrasikan sebuah variabel yang menampung sebuah bilangan.



Gambar 7.1 Variabel berisi sebuah nilai, isinya bisa diubah sewaktu-waktu ketika program dijalankan

Mendeklarasikan Variabel

Apabila Anda akan memakai sebuah variabel dalam program, seharusnya menyatakan variabel tersebut. Pernyataan variabel akan dipakai untuk mengambil posisi di dalam penyimpanan lap*top* akan memastikan jenis tipe data akan bisa tersimpan didalam peubah/variabel.
Model pernyataan peubah: Menggunakan tipedata;
Pernyataan ada pada tabel di bawah ini.

Tabel 7.3 Pernyataan variabel

Pernyataan	Penjelasan
int total;	peubah total dengan tipe **int** (dalam penyimpanan bilangan)
String namaPegawai	Peubah namaPegawai bertipe kelas ***String*** (dalam menampung sederet karakter)
Char huruf;	Peubah huruf bertipe char (dalam menyimpan suatu karakter)
Bool selesai;	Peubah selesai bertipe Bool (tipe yang mengetahui dua kondisi benar atau salah). Kondisi yang benar dinyatakan dengan ***true*** dan keadaan salah dinyatakan dengan ***false***.

Jika ternyata ada peubah yang memiliki tipe yang sama, peubah-peubah itu akan bisa dinyatakan pada suatu penjelasan. Kesimpulannya, antarpeubah harus dipisahkan menggunakan koma. Seperti:

int a, b, c;

ekuivalen dengan

int a;

int b;

int c;

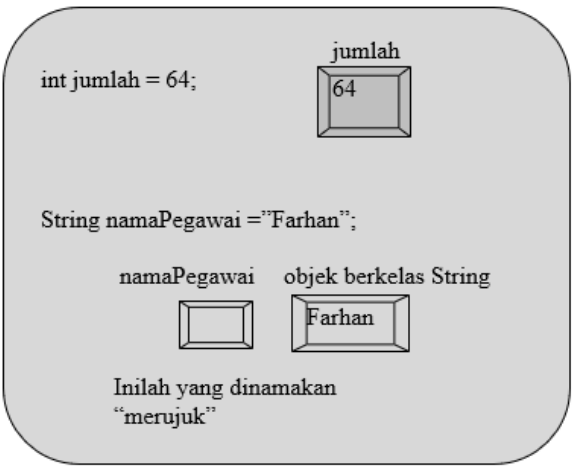
Jika pernyataan, peubah dapat serta-merta ada nilainya, Seperti:

string namaPegawai = “Farhan”;

int junlah = 64;

Seperti contoh pertama, peubah namaPegawai dinyatakan berkelas *string* dan diatur supaya mengikut pada *string* “Farhan”. Seperti contoh kedua, jumlah bertipe int serta diisi dengan 64.

Apabila suatu peubah mempunyai tipe sebuah kelas (contoh namaPegawai) variabel itu menyatakan akan menunjuk ke objek dari kelas tersebut. Gambar 4.2 menjelaskan perbedaan antara variabel yang bertipe kelas dan yang bertipe data primitive (bukan kelas).



Gambar 7.2 Perbedaan variabel bertipe primitif dan variabel bertipe kelas

Menyampaikan Nilai ke Variabel

Penjelasan yang dibutuhkan untuk memberikan nilai ke peubah:

peubah = nilai;

Misalnya:

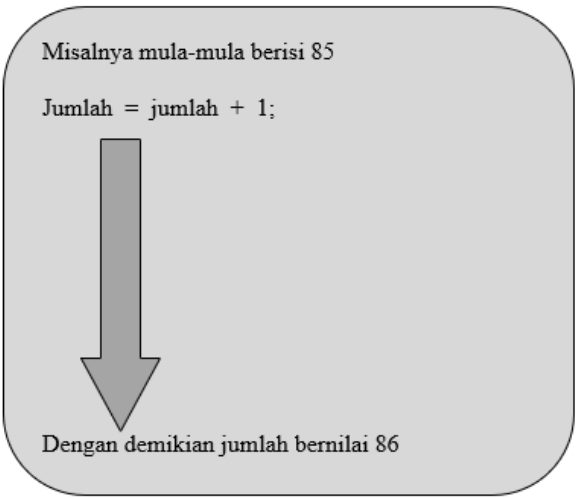
Jumlah = 85;

Menggambarkan penjelasan untuk memberikan suatu nilai 85 pada peubah jumlah.

Penjelasan dapat digambarkan sebagai berikut.

Jumlah = jumlah + 1;

Penjelasan di atas berarti "tambahan isi peubah jumlah dengan 1 dan kemudian hasilnya disimpan pada variabel jumlah". Dengan kata lain, penjelasan tersebut dipakai sehingga menambah nilai peubah jumlah sebesar 1. Pada sebelumnya peubah jumlah bernilai 85, sesudah penjelasan di atas dijalankan maka akan berisi 86.



Gambar 7.3 Ilustrasi penambahan nilai sebesar 1

Di bawah menunjukkan cara trans*for*masi keterangan untuk *Pseudocode* di Java.

Tabel 7.4 Pengkonversian *Pseudocode* keterangan dalam java

Pseudokode	Keterangan Java
x ⟶ 5	x = 5;
x ⟶ x+3	x = x + 3;
luas alas x tinggi	luas = alas * tinggi

Macam-macam Peubah/Variabel

Peubah di java terdapat tiga buah, yaitu 1) peubah instan, 2) variabel kelas, dan 3) variable lokal.

1. Variabel instan adalah variable yang digunakan untuk menyatakan atribut suatu objek.
2. Variabel kelas adalah variabel yang berlaku untuk semua objek berkelas sama.
3. Variabel lokal adalah variabel yang dideklarasikan dalam sebuah metode atau dalam suatu blok pernyataan.

Variabel instan akan dibahas pada *top*ik tersendiri. Adapun contoh variabel lokal dapat dilihat pada program Ucapan.java. Pada pernyataan.

String pesan = “Selamat Belajar Java”;

Variable pesan dideklarasikan dalam metode main. Dengan demikian variabel tersebut hanya dikenal pada main dan terbatas pada bagian yang terletak sesudah pendeklarasian tersebut.

BAB 8 OPERASI ARITMETIKA DAN LOGIKA DALAM ALGORITMA

Pendahuluan

Dalam dunia pemrograman komputer, algoritma merupakan serangkaian instruksi terstruktur yang dirancang untuk menyelesaikan suatu masalah. Algoritma terdiri dari berbagai komponen dasar yang berperan penting dalam memproses data dan menghasilkan *output* yang diinginkan. Di antara komponen-komponen tersebut, operasi aritmetika dan logika menjadi fondasi utama yang memungkinkan algoritma bekerja secara efektif.

Operasi aritmetika dan logika dapat diibaratkan sebagai "bahasa matematis" yang digunakan komputer untuk memproses informasi. Tanpa pemahaman yang baik mengenai kedua operasi ini, seorang *programmer* akan kesulitan dalam mengembangkan algoritma yang efisien dan akurat. Melalui bab ini, kita akan menjelajahi kedua jenis operasi tersebut secara mendalam, menguji peran mereka dalam algoritma, serta memahami bagaimana penggunaannya berdampak pada kinerja program.

Operasi Aritmetika

Pemahaman mendalam tentang operasi aritmetika dan logika memberikan beberapa keuntungan bagi *programmer*:

1. Meningkatkan kemampuan dalam merancang algoritma yang akurat dan efisien.

2. Memudahkan dalam melakukan debugging dan troubleshooting program.
3. Memungkinkan optimasi kode untuk meningkatkan kinerja program.
4. Membantu dalam memahami dan mengimplementasikan algoritma kompleks.
5. Memudahkan kolaborasi dalam pengembangan software.

Catatan Penting: Meskipun sintaksis operasi aritmatika dan logika dapat berbeda antar Bahasa pemrograman, konsep dasarnya tetap sama. Pemahaman konseptual yang kuat akan membantu Anda beradaptasi dengan berbagai Bahasa pemrograman dengan lebih mudah.

Operasi Aritmetika Sederhana

Operasi aritmetika merupakan landasan perhitungan matematis dalam algoritma. Operasi ini memungkinkan program untuk melakukan perhitungan pada data numerik, memproses dan menghasilkan *output* yang diperlukan. Terdapat lima operasi aritmetika dasar yang umum digunakan dalam pemrograman:

Operasi	Operator	Deskripsi	Contoh
Penjumlahan	+	Menambahkan dua nilai	5 + 3 = 8
Pengurangan	-	Mengurangi nilai kedua dari nilai pertama	5 - 2 = 3
Perkalian	*	Mengalikan dua nilai	4 * 2 = 8
Pembagian	/	Membagi nilai pertama dengan nilai kedua	9 / 3 = 3
Modulus	% (mod)	Menghasilkan sisa pembagian	7 % 3 = 1

Operasi-operasi dasar ini menjadi pondasi untuk perhitungan yang lebih kompleks dalam algoritma. Mari kita lihat contoh penggunaan operasi aritmetika dasar dalam

algoritma sederhana:

Contoh 1: Menghitung luas dan keliling persegi panjang

```
// Algoritma untuk menghitung luas dan keliling persegi panjang
1. Mulai
2. Deklarasi variabel: panjang, lebar, luas, keliling
3. Input nilai panjang dan lebar
4. Hitung luas = panjang * lebar
5. Hitung keliling = 2 * (panjang + lebar)
6. Tampilkan luas dan keliling
7. Selesai
```

Selain operasi dasar, terdapat juga operasi aritmetika lanjutan yang sering digunakan dalam algoritma:

1. Eksponensial/Pangkat (^): Mengangkat suatu bilangan ke pangkat tertentu. Contoh: $2^3 = 2\text{^}3 = 8$
2. Akar Kuadrat (sqrt): Menghitung akar kuadrat dari suatu bilangan. Contoh: sqrt(16) = 4
3. Pembulatan (round, ceiling, floor): Membulatkan nilai desimal. round(3.7) = 4 (membulatkan ke bilangan bulat terdekat), ceiling(3.2) = 4 (membulatkan ke atas), floor(3.8) = 3 (membulatkan ke bawah)
4. Logaritma (log): Menghitung logaritma dari suatu bilangan. Contoh: $\log_{10}(100) = 2$
5. Nilai Absolut (abs): Menghasilkan nilai positif dari suatu bilangan. Contoh: abs(-5) = 5

Contoh 2: Menghitung volume dan luas permukaan kerucut

```
// Algoritma untuk menghitung volume dan luas permukaan kerucut
1. Mulai
2. Deklarasi konstanta: PI = 3.14159
3. Deklarasi variabel: jari_jari, tinggi, volume, luas_permukaan, s
4. Input nilai jari_jari dan tinggi
5. Hitung s = sqrt(jari_jari^2 + tinggi^2) // s adalah sisi miring kerucut
6. Hitung volume = (1/3) * PI * jari_jari^2 * tinggi
7. Hitung luas_permukaan = PI * jari_jari * (jari_jari + s)
8. Tampilkan volume dan luas_permukaan
9. Selesai
```

Prioritas Operator Aritmetika

Dalam ekspresi aritmetika yang kompleks, prioritas operator menentukan urutan operasi yang dijalankan. Berikut adalah urutan prioritas operator aritmetika (dari yang tertinggi ke terendah):

1. Tanda kurung ( )
2. Eksponensial/Pangkat ^
3. Perkalian *, pembagian /, dan modulus %
4. Penjumlahan + dan pengurangan –

Praktik Terbaik: Meskipun kita dapat mengandalkan prioritas operator, sangat dianjurkan untuk menggunakan tanda kurung untuk membuat ekspresi lebih jelas dan menghindari kesalahan interpretasi. Misalnya, menulis (2 + 3) * 4 lebih jelas dari pada mengandalkan aturan prioritas.

Operasi dengan prioritas yang sama dijalankan dari kiri ke kanan. Mari kita lihat beberapa contoh.

Contoh 3: Evaluasi ekspresi aritmetika

2 + 3 * 4 = 2 + 12 = 14 (perkalian dilakukan terlebih dahulu)
(2 + 3) * 4 = 5 * 4 = 20 (operasi dalam kurung dilakukan terlebih dahulu)
10 - 6 / 2 + 3 = 10 - 3 + 3 = 10
3 * 2^2 + 5 = 3 * 4 + 5 = 12 + 5 = 17
4 * (5 % 3) - 2 = 4 * 2 - 2 = 8 - 2 = 6

Contoh 4: Algoritma menghitung rata-rata dan standar deviasi

```
// Algoritma untuk menghitung rata-rata dan standar deviasi dari sekumpulan
1. Mulai
2. Deklarasi array: data[n]
3. Deklarasi variabel: n, i, sum, mean, variance, std_dev
4. Input nilai n (jumlah data)
5. For i = 1 to n:
   a. Input data[i]
   b. sum = sum + data[i]
6. mean = sum / n
7. Inisialisasi variance = 0
8. For i = 1 to n:
   a. variance = variance + (data[i] - mean)^2
9. variance = variance / n
10. std_dev = sqrt(variance)
11. Tampilkan mean dan std_dev
12. Selesai
```

Logika dalam Pemrograman

Logika dalam pemrograman merupakan fondasi untuk pengambilan keputusan, pemilihan kondisi, dan kontrol alur program. Operasi logika memungkinkan algoritma untuk bereaksi terhadap berbagai situasi, membuat keputusan berdasarkan kondisi, dan menjalankan instruksi yang sesuai. Pada bagian ini, kita akan mengeksplorasi operator logika dan relasional, serta bagaimana mereka digunakan dalam struktur kontrol algoritma.

Operator Relasional

Operator relasional digunakan untuk membandingkan dua nilai dan menghasilkan nilai boolean (*true* atau *false*). Berikut adalah operator relasional yang umum digunakan:

Operator	Arti	Contoh	Hasil
<	Kurang dari	5 < 10	true
<=	Kurang dari atau sama dengan	5 <= 5	true
>	Lebih dari	5 > 10	false
>=	Lebih dari atau sama dengan	10 >= 5	true
==	Sama dengan	5 == 5	true
!=	Tidak sama dengan	5 != 10	true

Operator Logika

Operator logika digunakan untuk menggabungkan beberapa ekspresi relasional atau boolean, menghasilkan nilai boolean tunggal. Tiga operator logika utama adalah:

Operator	Arti	Penjelasan
AND (&&)	Dan	Menghasilkan true jika kedua operand bernilai true
OR (||)	Atau	Menghasilkan true jika minimal satu operand bernilai true
NOT (!)	Bukan	Membalik nilai boolean, dari true menjadi false atau sebaliknya

Tabel Kebenaran

Tabel kebenaran menunjukkan hasil dari operasi logika untuk semua kemungkinan kombinasi nilai input:

Tabel 8.1 Kebenaran Operator AND

A	B	A AND B
true	true	true
true	false	false
false	true	false
false	false	false

Tabel 8.2 Kebenaran Operator OR

A	B	A OR B
true	true	true
true	false	true
false	true	true
false	false	false

Tabel 8.3 Kebenaran Operator NOT

A	NOT A
true	false
false	true

Evaluasi Short-Circuit

Banyak bahasa pemrograman menerapkan evaluasi short-circuit untuk operator AND dan OR. Ini berarti:

1. Untuk operator AND (&&): Jika operand pertama *false*, operand kedua tidak dievaluasi karena hasilnya sudah pasti *false*.
2. Untuk operator OR (||): Jika operand pertama *true*, operand kedua tidak dievaluasi karena hasilnya sudah pasti *true*.

Evaluasi short-circuit ini dapat meningkatkan efisiensi program dan berguna dalam menghindari *error* potensial, seperti pembagian dengan nol atau akses ke elemen *array* yang tidak ada. Contoh:

```
// Contoh evaluasi short-circuit dengan operator AND
if (i != 0 && 10/i > 2) {
  // Kode dijalankan hanya jika kedua kondisi true
  // Jika i == 0, kondisi kedua tidak dievaluasi, menghindari pembagian dengan nol
}
// Contoh evaluasi short-circuit dengan operator OR
if (array != null || array.length > 0) {
  // Kode dijalankan jika minimal satu kondisi true
  // Jika array == null, kondisi kedua tidak dievaluasi, menghindari NullReferenceException
}
```

Penggunaan Logika dalam Struktur Kontrol

Logika berperan penting dalam struktur kontrol algoritma, terutama dalam:

1. Struktur Seleksi: Menggunakan *if-else*, *Switch-Case*, atau ternary operator untuk memilih eksekusi kode berdasarkan kondisi.
2. Struktur Perulangan: Menggunakan *while*, do-*while*,

atau *for loop* dengan kondisi logis untuk menentukan kapan perulangan berhenti.

Contoh 7: Penggunaan logika dalam struktur seleksi

//Algoritma untuk menentukan kategori nilai mahasiswa

1. Mulai
2. Deklarasi variabel: nilai
3. Input nilai
4. If (nilai >= 90) then
   Tampilkan "Grade: A"
   Else if (nilai >= 80) then
   Tampilkan "Grade: B"
   Else if (nilai >= 70) then
   Tampilkan "Grade: C"
   Else if (nilai >= 60) then
   Tampilkan "Grade: D"
   Else
   Tampilkan "Grade: E"
5. Selesai

Contoh 8: Penggunaan logika dalam struktur perulangan

//Algoritma untuk menemukan bilangan prima dalam rentang tertentu.

1. Mulai
2. Deklarasi variabel: start, end, i, j, isPrime
3. Input start dan end (batas awal dan akhir rentang)
4. *For* i = start to end:
   a. Jika i <= 1, lanjut ke iterasi berikutnya
   b. isPrime = *true*
   c. *For* j = 2 to sqrt(i):
      Jika i % j == 0 (i habis dibagi j) then
      1) isPrime = *false*
      2) Keluar dari *loop* j

d. Jika isPrime == *true* then
   Tampilkan i (bilangan prima ditemukan)
5. Selesai

Logika Bersarang (*Nested Logic*)

Logika bersarang terjadi ketika kita memiliki kondisi di dalam kondisi lain. Ini memungkinkan pembuatan struktur keputusan yang kompleks dan hirarkis. Meskipun berguna, logika bersarang yang terlalu dalam dapat membuat kode sulit dibaca dan dipelihara.

Contoh 9: Logika bersarang

//Algoritma untuk menentukan kualifikasi pinjaman

1. Mulai
2. Deklarasi variabel: usia, pendapatan, masa_kerja, kredit_score
3. Input usia, pendapatan, masa_kerja, kredit_score
4. If (usia >= 21) then
   a. If (pendapatan >= 5000000) then
      1) If (masa_kerja >= 2 OR kredit_score >= 700) then
         Tampilkan "Kualifikasi: Disetujui"
      2) *Else*
         Tampilkan "Kualifikasi: Perlu Review"
   b. *Else*
      Tampilkan "Kualifikasi: Ditolak - Pendapatan Tidak Mencukupi"
5. *Else*
   Tampilkan "Kualifikasi: Ditolak - Usia Di Bawah Minimal"
6. Selesai

Teknik Optimasi Logika

Beberapa teknik yang dapat digunakan untuk mengoptimasi ekspresi logika:

Hukum De Morgan:

1. NOT(A AND B) = NOT(A) OR NOT(B)
2. NOT(A OR B) = NOT(A) AND NOT(B)
3. Urutan evaluasi: Menempatkan kondisi yang lebih mungkin memutuskan hasil akhir di depan (untuk evaluasi short-circuit).
4. Menghindari redundansi: Menghilangkan kondisi yang tidak perlu atau terduplikasi.
5. Menyederhanakan ekspresi logika: Menggunakan aturan aljabar Boolean untuk menyederhanakan ekspresi yang kompleks.

Contoh 10: Optimasi ekspresi logika

```
// Ekspresi logika yang belum dioptimasi:
    if (x > 0 && y > 0 && x > 0) {
      // Kode
    }
    // Ekspresi logika yang dioptimasi (menghilangkan redundansi):
    if (x > 0 && y > 0) {
      // Kode
    }
    // Ekspresi logika yang belum dioptimasi:
    if (!(x >= 5 && y <= 10)) {
      // Kode
    }
    // Ekspresi logika yang dioptimasi (menggunakan hukum De Morgan):
    if (x < 5 || y > 10) {
```

```
    // Kode
}
```

Operasi aritmetika dan logika merupakan komponen fundamental dalam pengembangan algoritma dan pemrograman. Operasi aritmetika memungkinkan algoritma melakukan perhitungan matematis, sedangkan operasi logika memberikan kemampuan untuk membuat keputusan dan mengontrol alur program.

Pemahaman yang baik tentang kedua jenis operasi ini, termasuk prioritas operator, evaluasi ekspresi, dan teknik optimasi, sangat penting bagi *programmer* untuk mengembang-kan algoritma yang efisien, akurat, dan mudah dipelihara. Meskipun sintaksis spesifik mungkin berbeda antar bahasa pemrograman, konsep dasar operasi aritmetika dan logika tetap konsisten, menjadikannya keterampilan fundamental yang dapat ditransfer ke berbagai plat*for*m dan lingkungan pemrograman.

Dengan menguasai operasi aritmetika dan logika, *programmer* dapat merancang solusi algoritma yang lebih kompleks dan efektif untuk mengatasi berbagai masalah komputasi.

BAB 9 PERCABANGAN (*DECISION MAKING*) DALAM ALGORITMA

Pendahuluan

Dalam dunia pemrograman dan algoritma, kemampuan untuk membuat keputusan berdasarkan kondisi tertentu merupakan aspek fundamental. Sebuah program tidak hanya berjalan secara linier dari awal hingga akhir, tetapi sering kali memerlukan percabangan untuk menentukan tindakan yang tepat sesuai dengan situasi yang terjadi. Di sinilah peran penting struktur percabangan dalam algoritma: memungkinkan program untuk "berpikir" dan menyesuaikan perilakunya berdasarkan masukan atau keadaan tertentu.

Struktur percabangan dikenal juga dengan istilah decision making, yaitu proses memilih satu dari beberapa jalur eksekusi berdasarkan hasil evaluasi kondisi logis. Misalnya, dalam kehidupan sehari-hari kita sering mengambil keputusan: jika hari hujan, maka kita membawa payung; jika tidak, kita pergi seperti biasa. Logika yang sama diterapkan dalam algoritma, di mana komputer akan menjalankan perintah yang berbeda tergantung dari hasil kondisi yang diberikan.

Melalui bab ini, pembaca akan mempelajari berbagai bentuk struktur percabangan seperti if, *if-else*, *if-else if-else*, dan *Switch-Case*. Selain itu, akan dibahas pula bagaimana menuliskan percabangan dalam notasi algoritma, *flowchart*, hingga implementasinya dalam bahasa pemrograman. Dengan memahami materi ini, pembaca diharapkan mampu membuat

program yang lebih dinamis, responsif, dan sesuai dengan kebutuhan logika yang kompleks (Downey, 2020; *For*ouzan, 2017).

Pengertian dan Fungsi Percabangan

Percabangan atau decision making adalah struktur logika dalam algoritma yang memungkinkan program untuk memilih satu dari beberapa jalur eksekusi berdasarkan kondisi tertentu. Dalam proses berpikir manusia, percabangan sering kali terjadi secara alami, misalnya ketika seseorang memutuskan untuk membawa payung jika langit mendung, atau memilih belajar lebih lama jika ujian sudah dekat. Prinsip yang sama diterapkan dalam algoritma agar sistem dapat "memutuskan" apa yang harus dilakukan dalam situasi yang berbeda.

Secara teknis, percabangan dalam algoritma bekerja dengan cara mengevaluasi suatu kondisi. Jika kondisi tersebut bernilai benar (*true*), maka program akan menjalankan perintah tertentu; jika kondisi bernilai salah (*false*), maka program akan melewati perintah tersebut atau menjalankan perintah alternatif lainnya. Dengan adanya percabangan, algoritma tidak lagi bersifat linier, melainkan bisa menyesuaikan alurnya secara dinamis tergantung pada masukan atau data yang diterima (Cormen et al., 2022).

Fungsi utama dari percabangan adalah untuk memberikan fleksibilitas dalam logika program. Tanpa percabangan, program hanya akan mengikuti satu jalur tetap tanpa memperhatikan kondisi atau perubahan yang terjadi selama eksekusi. Oleh karena itu, percabangan menjadi salah satu komponen penting dalam menyusun algoritma yang efektif, efisien, dan mampu menangani berbagai kemungkinan. Dalam dunia nyata, fitur-fitur seperti login pengguna, validasi

data, pengambilan keputusan otomatis, dan berbagai sistem berbasis kondisi sangat bergantung pada konsep percabangan ini.

Jenis-jenis Struktur Percabangan

Struktur percabangan dalam algoritma terdiri dari beberapa jenis yang digunakan sesuai dengan kompleksitas kondisi yang ingin diuji. Setiap jenis percabangan memiliki bentuk dan tujuan masing-masing. Berikut adalah beberapa jenis struktur percabangan yang umum digunakan:

Percabangan Tunggal (*If Statement*)

Percabangan ini digunakan ketika kita hanya ingin melakukan suatu aksi jika kondisi tertentu terpenuhi. Jika kondisi tidak terpenuhi, maka tidak ada aksi yang dilakukan dan program akan melanjutkan ke langkah berikutnya. Struktur ini sangat cocok digunakan untuk kondisi sederhana.

Contoh logika: Jika suhu lebih dari 30°C, maka tampilkan "Cuaca Panas".

Pseudocode:

```
IF suhu > 30 THEN
Tulis "Cuaca Panas"
ENDIF
```

Percabangan Ganda (*If-Else Statement*)

Struktur ini digunakan ketika terdapat dua kemungkinan aksi: satu dijalankan jika kondisi bernilai benar, dan yang lain jika kondisi bernilai salah. Percabangan ini memungkinkan program mengeksekusi salah satu dari dua blok perintah.

Contoh logika: Jika nilai lebih dari atau sama dengan 75, maka tampilkan "Lulus", jika tidak tampilkan "Tidak Lulus".

Pseudocode:

```
IF nilai >= 75 THEN
   Tulis "Lulus"
ELSE
   Tulis "Tidak Lulus"
ENDIF
```

Percabangan Bertingkat (*If-Else If-Else Statement*)

Struktur ini digunakan ketika ada lebih dari dua kemungkinan kondisi yang perlu diuji. Program akan mengecek setiap kondisi satu per satu dari atas ke bawah sampai menemukan kondisi yang benar, lalu mengeksekusi perintah terkait dan melewati kondisi lain.

Contoh logika: Jika nilai ≥ 85, tampilkan "A"; jika nilai ≥ 75, tampilkan "B"; jika nilai ≥ 65, tampilkan "C"; jika tidak, tampilkan "D".

Pseudocode:

```
IF nilai >= 85 THEN
   Tulis "A"
ELSE IF nilai >= 75 THEN
   Tulis "B"
ELSE IF nilai >= 65 THEN
   Tulis "C"
ELSE
   Tulis "D"
ENDIF
```

Percabangan *Switch-Case*

Struktur ini merupakan alternatif dari *if-else* bertingkat yang digunakan untuk membandingkan satu variabel dengan beberapa nilai tetap. Struktur ini tersedia di banyak bahasa pemrograman seperti C, C++, dan Java, namun tidak tersedia di

beberapa bahasa seperti Python (Deitel & Deitel, 2019; Liang, 2018).

Contoh logika: Jika hari = 1, tampilkan "Senin"; jika hari = 2, tampilkan "Selasa"; dan seterusnya.

Contoh (dalam gaya umum):

```
SWITCH(hari)
   CASE 1: Tulis "Senin"
   CASE 2: Tulis "Selasa"
   CASE 3: Tulis "Rabu"
   ...
   DEFAULT: Tulis "Hari tidak dikenal"
ENDSWITCH
```

Setiap jenis percabangan memiliki kegunaan tersendiri tergantung pada kompleksitas kondisi dan kejelasan logika yang ingin dicapai. Dengan memilih struktur percabangan yang tepat, algoritma dapat dibuat lebih efisien, mudah dipahami, dan mudah dipelihara.

Notasi Algoritma dan *Flowchart* untuk Percabangan

Agar logika percabangan dalam algoritma mudah dipahami dan divisualisasikan, kita dapat menuliskannya dalam dua bentuk representasi: notasi algoritma dan *flowchart*. Keduanya memiliki tujuan yang sama, yaitu menggambarkan alur logika dari sebuah proses pengambilan keputusan secara sistematis (Sebesta, 2018; Zelle, 2016).

Notasi Algoritma untuk Percabangan

Notasi algoritma adalah cara menyampaikan logika program secara tertulis menggunakan bahasa yang menyerupai bahasa manusia tetapi dengan struktur logis yang teratur. Notasi ini sering digunakan sebelum program dikodekan dalam bahasa pemrograman. Contoh: Menentukan bilangan genap

atau ganjil.

```
Algoritma CekBilangan
Deklarasi:
   bilangan : integer

Deskripsi:
   Input bilangan
   IF bilangan MOD 2 = 0 THEN
      Tulis "Bilangan Genap"
   ELSE
      Tulis "Bilangan Ganjil"
   ENDIF
```

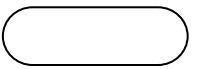
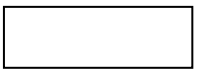
Keterangan:

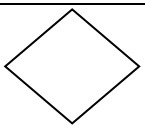
MOD = operasi sisa pembagian
IF dan *ELSE* = menunjukkan percabangan
Tulis = menunjukkan *output* ke layer

Flowchart Percabangan

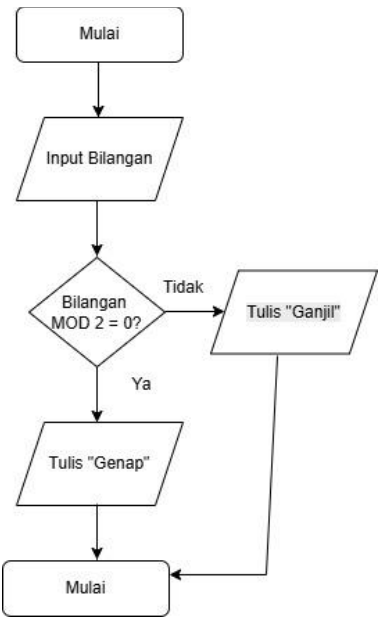
Flowchart adalah representasi grafis dari suatu algoritma menggunakan simbol-simbol standar. *Flowchart* sangat membantu dalam memahami alur proses dan keputusan secara visual. Simbol-simbol yang digunakan:

Tabel 9.1 Simbol *Flowchart* yang digunakan dalam Percabangan

Simbol	Nama	Fungsi
	Terminator	Mulai atau Selesai proses
	Data	Input/*Output* data
	Proses	Proses Operasional

	Decision	Percabangan/Kondisi.
	Flow	Arah alur dalam konsep (prosedur)

Contoh *Flowchart*: Menentukan bilangan genap atau ganjil.



Gambar 9.1 Contoh flowchat percabangan

Flowchart di atas menunjukkan bahwa program melakukan pengecekan apakah bilangan habis dibagi 2. Jika ya, maka *output* "Genap", jika tidak maka "Ganjil".

Dengan menggunakan notasi algoritma dan *flowchart*, perancangan logika program menjadi lebih terstruktur dan mudah dipahami, baik oleh pembuat program maupun oleh orang lain yang membaca atau memeriksa algoritma tersebut.

Implementasi Percabangan dalam Bahasa Pemrograman

Setelah memahami konsep percabangan dalam bentuk notasi algoritma dan *flowchart*, langkah selanjutnya adalah menerapkannya dalam bahasa pemrograman. Percabangan digunakan dalam hampir semua bahasa pemrograman modern seperti Python, C, C++, Java, dan lainnya. Struktur dasarnya serupa, meskipun sintaksisnya berbeda.

Berikut adalah contoh implementasi percabangan dalam beberapa bahasa pemrograman:

Python

Contoh Program: Menentukan Bilangan Genap atau Ganjil.

```
bilangan = int(input("Masukkan
bilangan: "))

if bilangan % 2 == 0:
   print("Bilangan Genap")
else:
   print("Bilangan Ganjil")
```

Penjelasan:

1. input() digunakan untuk membaca data dari pengguna
2. Operator % digunakan untuk mencari sisa pembagian
3. *if* dan *else* digunakan untuk membuat percabangan

Contoh Program: Menentukan Bilangan Genap atau Ganjil.

```c
#include <stdio.h>

int main() {
   int bilangan;

   printf("Masukkan bilangan: ");
   scanf("%d", &bilangan);

   if (bilangan % 2 == 0) {
      printf("Bilangan Genap\n");

   } else {
      printf("Bilangan Ganjil\n");
   }

   return 0;
}
```

Penjelasan:

1. scanf digunakan untuk membaca input dari pengguna
2. *if-else* digunakan untuk menentukan kondisi

Java

Contoh Program: Menentukan Bilangan Genap atau Ganjil.

```java
import java.util.Scanner;

public class CekBilangan {
   public static void main(String[] args)
{
      Scanner input = new
Scanner(System.in);
      int bilangan;

      System.out.print("Masukkan
bilangan: ");
      bilangan = input.nextInt();

      if (bilangan % 2 == 0) {
         System.out.println("Bilangan
Genap");
      } else {
         System.out.println("Bilangan
Ganjil");
      }
   }
}
```

Penjelasan:

1. Scanner digunakan untuk mengambil input dari pengguna
2. Struktur *if-else* serupa dengan bahasa lainnya

Dengan memahami cara kerja dan sintaks percabangan dalam bahasa pemrograman, kita dapat membangun logika yang dinamis dan fleksibel dalam program yang kita buat. Penggunaan percabangan sangat penting dalam aplikasi sehari-hari, mulai dari validasi input, menentukan status pengguna, hingga mengatur logika bisnis dalam aplikasi besar (Malik, 2017; Liang, 2018).

Studi Kasus & Latihan

Untuk memperkuat pemahaman tentang konsep percabangan, berikut ini disajikan beberapa studi kasus beserta latihan soal yang dapat digunakan sebagai bahan praktik atau evaluasi pembelajaran (Kumar & Sharma, 2021; D'Souza & Reddy, 2020).

Studi Kasus 1: Menentukan Nilai Lulus atau Tidak Lulus

Deskripsi: Seorang siswa dinyatakan Lulus jika nilainya ≥ 75. Jika kurang dari itu, maka Tidak Lulus.

Algoritma:

```
Input nilai
IF nilai >= 75 THEN
   Tulis "Lulus"
ELSE
   Tulis "Tidak Lulus"
ENDIF
```

Contoh Implementasi (Python):

```
nilai = int(input("Masukkan
nilai: "))

if nilai >= 75:
   print("Lulus")
else:
   print("Tidak Lulus")
```

Studi Kasus 2: Menentukan Kategori Usia

Deskripsi: Buat program yang mengelompokkan usia seseorang:

1. Usia < 13 → "Anak-anak"
2. Usia 13–17 → "Remaja"
3. Usia 18–59 → "Dewasa"
4. Usia ≥ 60 → "Lansia"

Algoritma:

```
Input usia
IF usia < 13 THEN
   Tulis "Anak-anak"
ELSE IF usia <= 17 THEN
   Tulis "Remaja"
ELSE IF usia <= 59 THEN
   Tulis "Dewasa"
ELSE
   Tulis "Lansia"
ENDIF
```

Latihan Soal

Soal 1:

Buat algoritma dan *flowchart* untuk menentukan apakah suatu tahun merupakan tahun kabisat. (Ketentuan: Tahun kabisat adalah tahun yang habis dibagi 4 dan tidak habis dibagi 100, kecuali juga habis dibagi 400).

Soal 2:

Tulis program untuk menentukan grade nilai siswa dengan ketentuan:

1. A: 85 – 100
2. B: 75 – 84
3. C: 65 – 74
4. D: 50 – 64
5. E: < 50

Soal 3:

Buat program sederhana yang menerima input angka dari 1 sampai 7, dan menampilkan nama hari (Senin sampai Minggu). Jika input tidak sesuai, tampilkan "Hari tidak valid". Pembahasan Singkat Soal 2 (Contoh dalam Python):

```
nilai = int(input("Masukkan nilai: "))

if nilai >= 85:
   print("Grade A")
elif nilai >= 75:
   print("Grade B")
elif nilai >= 65:
   print("Grade C")
elif nilai >= 50:
   print("Grade D")
else:
   print("Grade E")
```

Latihan-latihan ini dapat membantu pembaca mengasah kemampuan berpikir logis dan memahami penerapan percabangan dalam berbagai kasus. Pemahaman yang kuat terhadap percabangan akan menjadi fondasi yang penting sebelum mempelajari struktur logika lainnya, seperti perulangan dan fungsi (Sebesta, 2018; Downey, 2020).

Kesalahan Umum dalam Percabangan

Meskipun struktur percabangan terlihat sederhana,

banyak pemula yang masih sering melakukan kesalahan saat mengimplementasikannya, baik dalam algoritma maupun dalam kode program (Cormen et al., 2022; *For*ouzan, 2017). Kesalahan ini bisa menyebabkan program tidak berjalan seperti yang diharapkan, atau bahkan menghasilkan hasil yang salah. Berikut adalah beberapa kesalahan umum yang sering terjadi dalam penggunaan percabangan:

1. Salah Menulis Kondisi Logika
   Salah satu kesalahan paling umum adalah menulis kondisi logika yang tidak tepat. Misalnya, menggunakan = (operator penugasan) sebagai pengganti == (operator pembanding) dalam bahasa pemrograman seperti C atau Java. Contoh salah (dalam C):

   ```
   if (nilai = 75) {
      printf("Lulus");
   }
   ```

   Kondisi di atas akan selalu dianggap benar karena nilai = 75 adalah penugasan, bukan perbandingan.
2. Tidak Menangani Semua Kemungkinan Kondisi
   Sering kali, program hanya memeriksa sebagian kondisi tanpa menangani kondisi lainnya. Ini dapat menyebabkan program tidak melakukan apa-apa jika kondisi tertentu terjadi. Contoh:

   ```
   umur = 15

   if umur < 13:
      print("Anak-anak")
   elif umur > 17:
      print("Dewasa")
   # Tidak ada kondisi untuk
   umur 13–17 → tidak ada
   output
   ```

   Solusi: tambahkan kondisi *else* atau gunakan <= dan >= dengan cermat.

3. Menulis Struktur *If-else* Bertumpuk Secara Tidak Efisien
   Menumpuk terlalu banyak if tanpa menggunakan *else* if bisa membuat program sulit dibaca dan boros dalam eksekusi. Kurang efisien:

```
if nilai >= 85:
   print("A")
if nilai >= 75 and nilai
< 85:
   print("B")
if nilai >= 65 and nilai
< 75:
   print("C")
```

   Lebih baik:

```
if nilai >= 85:
   print("A")
elif nilai >= 75:
   print("B")
elif nilai >= 65:
   print("C")
```

4. Kurang Teliti dalam Mengatur Urutan Kondisi
   Urutan kondisi sangat penting, terutama jika menggunakan *else* if. Menempatkan kondisi yang lebih umum di awal bisa menyebabkan kondisi spesifik tidak pernah diperiksa. Contoh salah:

```
if nilai >= 65:
   print("C")
elif nilai >= 85:
   print("A")
# Tidak akan pernah
dieksekusi karena
kondisi pertama
sudah memenuhi
```

5. Lupa Menutup Blok Percabangan
   Dalam beberapa bahasa, seperti Python atau Pascal,

indentasi dan penutup blok sangat penting. Lupa menutup blok dapat menyebabkan kesalahan sintaks atau perilaku tak terduga.

Tips Menghindari Kesalahan:

a. Gunakan indentasi yang rapi dan konsisten.
b. Selalu uji semua kemungkinan kondisi, termasuk batas bawah dan atas.
c. Gunakan *else* atau *default* untuk menangani kondisi yang tidak terduga.
d. Gunakan komentar untuk menjelaskan logika percabangan yang kompleks.

Dengan memahami dan menghindari kesalahan-kesalahan umum ini, pembaca akan lebih siap dalam menyusun algoritma yang logis, efisien, dan minim bug.

Struktur percabangan merupakan komponen fundamental dalam perancangan algoritma dan pemrograman, karena memungkinkan program mengambil keputusan berdasarkan kondisi tertentu. Melalui percabangan, sebuah program dapat menjalankan alur logika yang berbeda sesuai dengan input atau situasi yang terjadi.

Dalam bab ini, telah dibahas berbagai jenis percabangan, mulai dari percabangan tunggal (if), percabangan ganda (*if-else*), percabangan bertingkat (*if-else*-if), hingga penggunaan *Switch-Case* untuk menangani banyak kemungkinan nilai. Selain itu, kita juga telah melihat bagaimana percabangan direpresentasikan dalam bentuk algoritma, *flowchart*, serta implementasi nyata dalam berbagai bahasa pemrograman seperti Python, C, dan Java.

Bab ini juga menyoroti kesalahan umum yang sering dilakukan oleh pemula saat menggunakan percabangan, serta memberikan tips untuk menghindarinya. Melalui latihan soal

dan studi kasus, diharapkan pembaca dapat lebih memahami konsep ini secara mendalam dan mampu mengaplikasikannya dalam pembuatan program yang lebih kompleks.

Sebagai bekal untuk materi selanjutnya, pemahaman yang baik tentang percabangan akan sangat membantu ketika kita mulai mempelajari struktur kontrol lain seperti perulangan (*looping*) dan fungsi. Dengan demikian, bab ini menjadi dasar penting dalam perjalanan belajar algoritma dan pemrograman.

BAB 10 PERULANGAN (*LOOPING*) DAN PENERAPANNYA

Pendahuluan

Ketika masih muda pasti pernah diajari tabel angka. Tabel nomor memiliki pola. Menulis tabel dalam suatu ujian memerlukan penulisan, katakanlah, n × diikuti dengan i (i bervariasi dari 1 sampai n) dan kemudian hasil perhitungannya (yaitu n × 1, n × 2, dan seterusnya). Banyak situasi seperti itu mengharuskan untuk mengulang tugas yang diberikan berkali-kali.

Pengulangan ini dapat digunakan untuk menghitung nilai suatu fungsi, untuk mengeprint suatu pola atau untuk sekadar mengulang sesuatu. Bab ini membahas *loop* dan iterasi yang merupakan bagian integral dari pemrograman prosedural. *Looping* berarti mengulang satu set pernyataan sampai suatu kondisi benar. Berapa kali set ini diulang tergantung pada kondisi pengujian. Juga, apa yang harus diulang perlu dicatat dengan pertimbangan matang.

Secara umum, pengulangan sebuah blok memerlukan hal-hal berikut.

1. Memutuskan apa yang akan diulang: himpunan pernyataan.
2. Kondisi pengujian atau berapa kali himpunan pernyataan harus diulang.
3. Kasus khusus di mana *loop* putus (atau berlanjut, keluar dari pernyataan tertentu).

Dalam Python, pengulangan (*looping*) digunakan untuk menjalankan serangkaian perintah secara berulang hingga kondisi tertentu terpenuhi. Jenis *loop* utama yang disediakan Python, yaitu *for*, *while*, dan *for* bersarang

Pengenalan *Looping*

Dalam kehidupan sehari-hari, sering kali kita menemui situasi yang harus dilakukan berulang kali. Misalnya dalam skenario berikut.

"Setiap hari Rabu, pasar yang selalu dikunjungi oleh Ibu (pasar yang sama dengan cerita sebelumnya) selalu tidak menyediakan daging ayam. Maka dari itu, Ibu selalu membeli tempe sebagai gantinya. Pada minggu biasa, Ibu hanya akan memotong 3 balok tempe karena jumlah anggota keluarga adalah 3 orang. Namun, pada minggu lain, Ibu kedatangan keluarga besar untuk makan bersama. Kali ini, Ibu tidak mengetahui total keluarga yang datang. Jadi, setelah memotong 1 balok tempe, Ibu akan selalu mengecek bahwa jumlah tersebut cukup atau tidak."

Pada skenario berikut, Ibu kedatangan keluarga besar di rumahnya dan berencana untuk membuat hidangan makanan berupa tempe. Ibu tidak mengetahui jumlah keluarga yang hadir sehingga setiap kali ada keluarganya yang datang, Ibu akan memotong 1 balok tempe untuk disajikan kepada 1 orang.

Perhatikan pada diagram di atas, Ibu akan selalu melakukan aktivitas berulang untuk memotong tempe hingga kondisinya terpenuhi. Kondisi yang dimaksud adalah jumlah keluarga yang hadir sama dengan jumlah tempe yang disajikan.

Dalam pemrograman, kita juga akan sering menemui masalah serupa yang mengharuskan untuk melakukan kode berulang. Contohnya menampilkan angka 11 hingga 20.

```
print(11)
print(12)
print(13)
print(14)
print(15)
print(16)
print(17)
print(18)
print(19)
print(20)

"""
```

Output:

```
11
12
13
14
15
16
17
18
19
20
"""
```

Pada kode di atas, program menampilkan angka dari 11 hingga 20 menggunakan sintaks yang berulang. Terlihat tidak efektif, bukan? Itulah yang menjadi tujuan dari materi perulangan ini, Anda akan belajar untuk membuat kode program yang efektif dan mudah dibaca oleh *programmer* lain.

Dalam Python, ada beberapa sintaks atau statement untuk melakukan perulangan.

For Loops

For termasuk sintaks dalam Python yang digunakan untuk definite *iteration*. Definite *iteration* merupakan proses perulangan di mana jumlah pengulangannya telah ditentukan secara eksplisit sebelumnya. Perulangan *for* mempunyai *for*mat seperti sebagai berikut.

<iterable> adalah objek dalam Python yang dapat diulang, seperti *list*, tuple, atau *string*. Sementara itu,

<var>variabel yang akan menyimpan elemen berikutnya dari <iterable> setiap kali perulangan berlangsung.

Untuk statement dalam Python bekerja dengan mengulangi elemen koleksi seperti *list, set, dictionary*, dan *string*. Untuk urutan berurutan seperti *list* dan tupel, *for loop* mengambil item satu per satu dalam urutan kemunculannya dalam urutan. Sintaksnya cukup intuitif, dimulai dengan kata kunci *for* kemudian nama variabel yang dipilih untuk digunakan di dalam *loop* untuk item dari urutan, diikuti dengan kata kunci "in", dan terakhir urutan itu sendiri dan titik dua.

```
1 baris = ['pertama','kedua','ketiga','keempat']
2 for item in baris:
3     print(item)
```

```
pertama
kedua
ketiga
keempat
```

Kode di atas benar-benar membaca apa yang dilakukannya untuk setiap item secara berurutan. Kemudian, setiap baris yang diindentasi di bawah definisi *for loop* dieksekusi sesering ada sejumlah item dalam urutan yang sering disebut sebagai *loop body*. Dalam contoh ini, *for loop* hanya mengeprint setiap *string* dalam *list*.

```
1 strg = 'ini string'
2 for i in strg:
3     print(i)
```

```
i
n
i

s
t
r
i
n
g
```

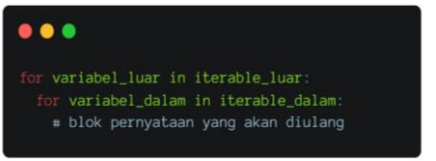
Ketika mengulang satu *string* karena *string* hanyalah urutan karakter, maka itemnya adalah setiap karakter, seperti berikut ini.

```
unor = {'a','b','c','d','e'}
for j in unor:
    print(j)
```

```
b
d
a
e
c
```

For Bersarang

Saat membuat perulangan, sering kali ditemui perulangan di dalam perulangan lainnya, yang dikenal sebagai nested *loop*. Struktur dari nested *loop* adalah sebagai berikut.



Dapat diasumsikan bahwa terdapat dua perulangan, yaitu "perulangan luar" dan "perulangan dalam". Program akan menjalankan "perulangan luar" terlebih dahulu, kemudian melanjutkan ke "perulangan dalam". Pada proses ini, variabel_luar akan mengambil nilai dari “iterable_luar”, sedangkan variabel_dalam akan mengambil nilai dari “iterable_dalam”.

Terkadang kita tidak ingin mengulangi setiap elemen urutan karena alasan tertentu atau mungkin harus menggunakan indeks elemen di dalam *loop*. Ada beberapa cara untuk mencapainya dan di sini akan dilihat fungsi iterator "range" dengan asumsi sudah terbiasa dengan *for loop* secara umum, beberapa contohnya sebagai berikut.

```
for i in range(7):
    print(i)
```

```
0
1
2
3
4
5
6
```

Jadi, dapatkah Anda menebak apa yang terjadi jika kita meneruskan dua argumen ke fungsi? Maka argumen pertama adalah titik awal sedangkan argumen terakhir adalah bilangan bulat setelah nilai terakhir dihasilkan.

Apa yang terjadi jika kita memberikan tiga argument? Argumen pertama adalah awal, yang kedua berhenti dan argumen terakhir adalah ukuran langkah. Dalam hal ini dimulai dari nol berhenti sebelum 10 dan menghasilkan setiap nilai lainnya. Urutan dapat dibalik dengan membalik urutan dua argumen pertama. Sehingga titik awal lebih tinggi dari titik akhir, tetapi kemudian diharuskan memberi tahu fungsi bahwa langkah tersebut seharusnya berupa angka negative.

Kontrol Perulangan

Selain membuat perulangan, kita juga dapat mengontrol perulangan dengan menggunakan beberapa pernyataan di antaranya sebagai berikut.

Break

Pernyataan *break* digunakan untuk menghentikan perulangan secara langsung, sehingga program akan keluar dari perulangan tersebut dan melanjutkan eksekusi ke blok kode berikutnya. Jika perulangan bersarang seperti *for* bertingkat, break akan menghentikan perulangan sesuai dengan tingkat atau posisi perulangan tempat pernyataan tersebut digunakan.

```
for i in range(2):  # Perulangan tingkat pertama
    print("Perulangan luar:", i)
    for j in range(10):  # Perulangan tingkat kedua
        print("Perulangan dalam:", j)
        if j == 1:
            break  # Menghentikan perulangan dalam jika j = 1
```

"""

Output:
Perulangan luar: 0
Perulangan dalam: 0
Perulangan dalam: 1
Perulangan luar: 1
Perulangan dalam: 0
Perulangan dalam: 1
"""

Pada kode di atas, kita melakukan perulangan untuk menampilkan bilangan 0 hingga 1 pada "perulangan luar" atau *for* pertama. Lalu, kita membuat perulangan kedua untuk menampilkan bilangan dari 0 hingga 9. Namun, pada perulangan kedua atau "perulangan dalam" tersebut, kita akan melakukan break jika bertemu angka "1". Alhasil, perulangan dalam hanya akan menampilkan angka hingga 1 saja. Program akan berhenti karena ada statement break yang diberikan jika bertemu angka "1".

Continue

Pernyataan *continue* digunakan untuk menghentikan iterasi saat ini dan langsung melanjutkan ke iterasi berikutnya. Dengan menggunakan *continue*, pernyataan yang terletak setelahnya dalam blok perulangan akan dilewati hingga perulangan berlanjut ke iterasi selanjutnya.

```
for i in range(11):
    if i in [3,5,6,9]:
        continue
    print(i)
```

```
0
1
2
4
7
8
10
```

Sekarang diasumsikan ingin melewatkan beberapa angka arbitrer. Kemudian kata kunci "*continue*" berguna. Ketika pertemuan Python berlanjut di dalam sebuah *loop*, ia akan segera melewati sisa *loop body* dan melompat ke elemen berikutnya dalam urutan. Apabila ingin mengeprint setiap angka dari nol hingga 10 kecuali 3, 5, 6, dan 9 akan dilakukan seperti ini.

Jadi setiap kali rentang menghasilkan salah satu angka dalam *list* pengecualian yang dilihat Python melanjutkannya ke elemen berikutnya dari urutan tanpa mengeksekusi pernyataan print.

Else setelah *For*

Dalam Python, terdapat *else* setelah *for* yang digunakan dalam perulangan pencarian. *Else* ini berfungsi sebagai mekanisme untuk mengeksekusi kode tertentu ketika pencarian dalam perulangan tidak menemukan hasil yang diinginkan.

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 6:
        print("Angka ditemukan! Program berhenti!")
        break
else:
    print("Angka tidak ditemukan.")
"""
Output:
Angka tidak ditemukan.
"""
```

Pada contoh di atas, kita membuat program untuk melakukan pencarian terhadap bilangan 6. Jika bilangan 6 tersebut merupakan elemen atau nilai yang berada *list*, program

akan berhenti dan menampilkan teks "Angka ditemukan! Program berhenti!".

Namun, pada contoh di atas angka 6 bukan merupakan elemen dari *list* maka program akan menampilkan teks "Angka tidak ditemukan". Apa jadinya jika program menemukan angka? Silakan ganti "if num == 6" dengan "if num == 4" atau angka lain yang merupakan nilai yang berada dalam *list*.

Meskipun berada di blok yang berbeda, if dan *else* pada contoh tersebut tetap saling terkait. Pada *else* setelah *for*, bagian *else* tidak akan dijalankan jika kondisi dalam if pernah terpenuhi setidaknya sekali. Dengan kata lain, agar *else* setelah *for* dapat dieksekusi, pernyataan break di dalam if tidak boleh terjadi.

Else setelah *While*

Berbeda dengan *else* setelah *for*, pada pernyataan *else* setelah *while*, blok *else* akan selalu dijalankan ketika kondisi dalam *while* menjadi salah.

```
count = 0
while count < 3:
    print("Dicoding Indonesia")
    count += 1
else:
    print("Blok else dieksekusi karena kondisi pada while salah (3<3 == False).")

"""
Output:
Dicoding Indonesia
Dicoding Indonesia
Dicoding Indonesia
```

Blok *else* dieksekusi karena kondisi pada *while* salah (3<3 == *False*).

"""

Pada contoh di atas, perulangan *while* akan terus terjadi dan *else* tidak akan dieksekusi jika kondisi *while* benar. Kondisi *while* akan terus benar pada kode di atas ketika variabel "count" bertambah dari 1 hingga 2 dan akan berhenti ketika variabel "count" bernilai 3 karena "3<3" adalah *false* atau salah.

Pass

Pernyataan *pass* digunakan ketika Anda membutuhkan sebuah pernyataan atau blok kode, tetapi tidak ingin menjalankan tindakan apa pun dalam program.

```
x = 10
if x > 5:
  pass
else:
  print("Nilai x tidak memenuhi kondisi")

"""
Output:
"""
```

Program di atas tidak menampilkan apa pun karena jika kondisi terpenuhi, program tidak akan melakukan apa pun.

Pernyataan *pass* digunakan dalam situasi di mana Python mengharuskan adanya sebuah pernyataan, tetapi tidak ada tindakan yang perlu dilakukan saat itu. Biasanya, pass berfungsi sebagai *placeholder* untuk menandakan bahwa tidak ada operasi yang dijalankan. Penggunaannya membantu

menjaga struktur kode tetap rapi serta mempermudah penambahan implementasi di masa mendatang.

List Comprehension

Comprehension merupakan salah satu fitur Python yang paling berguna dan *pop*uler. *Comprehension* adalah cara yang sangat Pythonic untuk mengisi urutan dan koleksi dengan lebih sedikit kode.

```
1 boring_list = []
2 for i in range(10):
3     boring_list.append(i*i)
4 print(boring_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Kode mulai dengan memulai *list* kosong kemudian *loop for* berulang melalui beberapa nilai dan di body *loop* ditambahkan perhitungan apa pun dibuat ke *list*. *Comprehension* yang melakukan hal yang persis sama terlihat seperti ini.

```
1 comp_list = [i*i for i in range(10)]
2 print(comp_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Pada dasarnya, *comprehension* hanya *loop for* yang tertutup di dalam tanda kurung. *For loop* baru saja disesuaikan sedikit untuk digabungkan pada satu baris.

While Loops

While adalah salah satu sintaks dalam Python yang digunakan untuk perulangan dengan sifat indefinite *iteration*. Indefinite *iteration* merupakan proses perulangan yang akan terus berjalan hingga memenuhi suatu kondisi tertentu. Struktur dasar perulangan *while* adalah sebagai berikut.

```
while kondisi:
  # blok pernyataan yang akan diulang selama kondisi bernilai True
```

Kondisi adalah sebuah ekspresi yang dievaluasi untuk menentukan apakah bernilai *true* atau *false*. Selama hasil evaluasi tetap *true*, program akan terus berjalan hingga akhirnya bernilai *false*.

1. Mencampur *looping* dan conditional
2. Digunakan ketika jumlah iterasi tidak diketahui
3. Rentan terhadap pengulangan tak terbatas

Ketika hanya diketahui kondisi apa yang harus dipenuhi sebelum keluar dari perulangan, kita menggunakan perulangan *while*. Sementara *loop* adalah campuran dari iterasi dengan kondisional dan hanya digunakan ketika tidak tahu berapa banyak iterasi yang diperlukan dan hanya tahu kondisi apa yang harus dipenuhi untuk mengakhiri *loop*. Sintaksnya hanyalah kata kunci "*while*" diikuti oleh kondisi dan titik dua. Kemudian garis indentasi di bawah ini adalah *loop* body *while*. *Loop* akan terus berulang selama kondisinya benar seperti berikut ini.

```
1 i = 0
2 while i <= 10:
3     print(i)
4     i += 1
```

```
0
1
2
3
4
5
6
7
8
9
10
```

Loop ini akan terus mengeprint i sementara i kurang dari atau sama dengan 10. Tidak seperti *loop for*, *loop while* tidak memiliki urutan untuk memberitahukan berapa banyak iterasi yang harus ada juga tidak memiliki cara untuk secara otomatis menambah ke elemen berikutnya. Dalam contoh dikondisikan iterasi pada ukuran i yang harus dimulai dengan menetapkan i tetapi mungkin yang lebih penting, secara eksplisit menambahkan i di dalam *loop*. Jika tidak, seperti di sini akan

mengimplementasikan infinite *loop*.

```
i = 0
while i < 1:
print('infinite loop because i is still ',i)
```

Ini tidak akan pernah berhenti dengan sendirinya karena tidak akan pernah naik dan mencapai kondisi tersebut. Pada *loop* dapat menghentikan sel yang berjalan dengan mengklik dua kali i.

Terkadang ada situasi di mana mungkin ingin keluar dari satu lingkaran sebelum naik ke siklus berikutnya. Mungkin dalam iterasi terakhir mungkin ingin mengeksekusi hanya setengah dari body sebelum melanjutkan.

```
i = 0
while i <= 10:
    print(i)
    i += 1
```

```
0
1
2
3
4
5
6
7
8
9
10
```

Untuk itu digunakan kata kunci break dan berfungsi seperti di *for loop*. Jika ditemui skrip akan segera keluar dari skrip mengabaikan semua baris di body setelahnya. Perhatikan bahwa kondisi *while* disetel *True* dan tidak akan pernah berubah sehingga dipaksa untuk menulis pernyataan if di body *loop* yang mengeksekusi break jika kondisi terpenuhi. *While loop* juga sesuai dengan kata kunci lanjutan seperti *for loop* seperti berikut ini.

```
ans = 0
while ans!=42:
    ans = input('beri nilai antara 0 dan 100: ')
    ans = int(ans)
    if ans<50:
        print('jawabanmu kurang 50')
        continue
    print(ans)
print('kamu keluar dari loop')
```

```
beri nilai antara 0 dan 100: 23
jawabanmu kurang 50
beri nilai antara 0 dan 100: 87
87
beri nilai antara 0 dan 100: 102
102

beri nilai antara 0 dan 100:
```

Perulangan *while* ini akan terus meminta memasukkan angka sampai saya memberikan angka 42. Jika saya memberikannya 51 [masukkan 51] maka itu hanya akan mencetak nomor itu. Namun, jika memberikan misalnya sesuatu yang lebih rendah dari 50[masukkan 45], maka itu akan memberi tahu persis seperti itu tetapi itu tidak akan mengeprint nomor yang diberikan karena pernyataan lanjutkan melewatkan bagian body lainnya. Jika memasukkan 42 [masukkan 42] dibebaskan dari *loop* dan pernyataan cetak terakhir yang mengatakan demikian dieksekusi.

BAB 11 PENGENALAN STRUKTUR DATA DALAM ALGORITMA

Pendahuluan

Secara umum komputer mengenal berbagai jenis data, misalnya integer (bilangan bulat), karakter, pecahan dan sebagainya. Namun demikian untuk merepresentasikan suatu entitas di dunia nyata sering kali diperlukan penggunaan berbagai jenis data sekaligus. Misal untuk mengetahui karakteristik setiap pelanggan dari suatu gerai penjualan minuman sekurang-kurangnya diperlukan informasi tentang jenis kelamin (bisa berupa karakter atau integer) dan usia (integer) dari setiap pelanggan. Untuk keperluan yang lebih kompleks akan lebih banyak lagi jenis data yang diperlukan. Dalam konteks pemrograman atau pengolahan data dengan komputer, data dari suatu entitas seperti yang telah diuraikan dapat terdiri dari dua atau lebih elemen data. Setiap elemen data merepresentasikan suatu karakteristik tertentu.

Komputer pada dasarnya hanyalah sebuah sistem atau mesin. Komputer tidak akan secara otomatis mampu mengenali suatu elemen merupakan anggota dari data yang mana. Untuk itulah setiap data dan elemen-elemennya yang disimpan atau digunakan komputer perlu diatur. Untuk pengaturan inilah diperlukan struktur data. Secara umum, struktur data dapat diartikan sebagai cara khusus untuk menyimpan dan mengatur data dalam komputer sehingga dapat digunakan secara efisien. Sebenarnya sruktur data merupakan susunan logis dari elemen-

elemen data dengan serangkaian operasi yang diperlukan untuk mengakses elemen tersebut. Model logis atau model matematika dari pengorganisasian data tertentu disebut itulah yang merupakan struktur data. Dengan demikian struktur data dapat didefinisikan sebagai serangkaian aturan dan batasan yang menunjukkan hubungan yang ada antara potongan-potongan data individual yang mungkin terjadi.

Implementasi atau penerapan struktur data dalam komputer akan sangat tergantung dari Bahasa Pemgrograman dan kompiler yang digunakan. Namun demikian tujuan dari penggunaan struktur data pada prinsipnya sama, yaitu untuk meningkatkan efisiensi penyimpanan dan pengaksesan data. Pada bagian berikut akan diuraikan lebih lanjut tentang prinsip dasar, jenis atau klasifikasi struktur data, dan operasi pada struktur data.

Prinsip Dasar Struktur Data

Pengimplementasian struktur data pada umumnya didasarkan pada kemampuan komputer untuk mengambil dan menyimpan data di dalam memorinya. Hal ini ditentukan oleh suatu alamat dari rangkaian bit yang dapat disimpan dalam memori dan dimanipulasi oleh program. Dengan demikian, struktur data record dan *array*, misalnya, akan didasarkan pada penghitungan alamat item data dengan operasi aritmatika, sementara untuk struktur data tertaut (*linked list*) akan didasarkan pada penyimpanan alamat item data dalam struktur itu sendiri.

Sebelum menentukan penggunaan suatu struktur data tertentu, terdapat beberapa hal yang pada umumnya perlu dipertimbangkan. Hal-hal ini antara lain adalah:

1. Volume (seberapa banyak) data yang akan digunakan.

2. Frekuensi dan cara penggunaan data.
3. Apakah datanya bersifat dinamis atau statis.
4. Seberapa banyak tempat penyimpanan yang diperlukan.
5. Waktu yang diperlukan untuk mengakses/mengambil suatu elemen.
6. Kemudahan dari sisi pemrograman.

Pada dasarnya tidak ada suatu struktur data yang dapat memenuhi semua kriteria. Pertimbangan utamanya tetap pada efisiensi, tergantung dari persoalan yang akan dipecahkan. Artinya, bisa saja untuk persoalan yang berbeda digunakan struktur data yang berbeda pula.

Dalam konteks algoritma, pemilihan suatu struktur data pada umumnya didasarkan pada tiga karakteristik, yaitu ketepatan, kompleksitas waktu, dan kompleksitas ruang.

1. Ketepatan mengacu pada keakuratan dan keandalan implementasi struktur data. Dalam hal ini struktur data harus dapat menjalankan operasi seperti yang diharapkan, mampu menangani kasus-kasus ekstrem, dan memberikan menghasilkan hasil yang tepat.
2. Kompleksitas waktu berkaitan dengan waktu yang diperlukan untuk melakukan operasi struktur data. Dengan mengetahui kompleksitas waktunya dapat dengan mudah diperkirakan berapa banyak tambahan waktu yang diperlukan sejalan dengan bertambahnya ukuran input yang diberikan.
3. Kompleksitas ruang berkaitan dengan jumlah memori yang dibutuhkan algoritma untuk menjalankan tugasnya secara efisien pada struktur data tertentu. Idealnya, struktur data harus menggunakan memori sesedikit mungkin. Hal ini sangat penting dalam lingkungan dengan keterbatasan sumber daya atau saat menangani

kumpulan data besar.

Mengoptimalkan ketiga karakteristik ini penting untuk memilih struktur data yang memenuhi persyaratan kinerja dan menjaga keakuratan serta keandalan dalam semua skenario. Pada permasalahan yang memerlukan penyelesaian cepat, mungkin dapat dipilih struktur data dengan kompleksitas waktu yang relatif kecil walaupun harus memerlukan memori yang relatif besar. Sementara pada kasus lain mungkin kompleksitas ruang lebih diutamakan daripada kompleksita waktunya. Begitu juga dengan pilihan ketepatan. Semakin tinggi tinggi ketepatan yang diinginkan pada umumnya akan diiringi dengan semakin besar pula kompleksitas ruang dan waktunya.

Jenis (Klasifikasi) Struktur Data

Secara umum struktur data dapat dikelompokkan menjadi dua jenis, yaitu struktur data primitif dan nonprimitif. Struktur data primitif adalah struktur data dasar yang ada dalam suatu bahasa pemrograman. Struktur data primitif yang tersedia dalam suatu bahasa pemrograman mungkin saja berbeda dengan struktur data primitif pada bahasa pemrograman yang lain. Bahkan dalam suatu bahasa pemrograman tertentu mungkin saja memiliki struktur data primitif yang berbeda, tergantung dari kompiler yang digunakan.

Struktur data primitif digunakan untuk merepresentasikan satu nilai yang sekanjutnya dapat digunakan untuk membangun struktur data yang lebih kompleks. Dalam Bahasa Pemrograman C, misalnya, ada 4 struktur data primitif utama yang digunakan, yaitu int, char, float, dan double. Selain itu dalam C juga terdapat jenis data void.

1. int digunakan untuk menyimpan integer (bilangan bulat), baik prositif maupun negative, seperti -5, 0 dan 10. Jenis data int umumnya memerlukan 2 atau 4 byte untuk penyimpanan di memori, tergantung arsitektur sistem yang digunakan.
2. char digunakan untuk menyimpan karakter tunggal seperti 'A', 'a', atau '$'. Jenis data char memerlukan 1 byte di dalam memori.
3. float digunakan untuk menyimpan bilangan yang memuat titik desimal dengan presisi tunggal (single-precision floating number) seperti -1.96 atau 2.96. Jenis data float memerlukan 4 byte dalam memori.
4. double digunakan untuk menyimpan bilangan yang memuat titik desimal dengan presisi ganda (double-precision floating number). Jenis data double memerlukan 8 byte dalam memori yang memungkinkan penyimpanan data dengan presisi yang lebih tinggi (lebih banyak bilangan di belakang titik desimal) dibandingkan dengan float.
5. void untuk merepresentasikan ketiadaan jenis data. Pada umumnya void digunakan untuk suatu fungsi yang tidak memerlukan pengembalian nilai.

Keempat jenis data primitif dalam bahasa pemrograman C masing-masing dapat dimodifikasi lebih lanjut dengan menggunakan signed, unsigned, short, atau long.

1. signed untuk menentukan bahwa variabel dapat menyimpan nilai positif dan negatif, hal ini merupakan bawaan (*default*) dari jenis data int dan char.
2. unsigned untuk menentukan bahwa variabel hanya dapat menyimpan nilai nonnegatif.
3. short untuk mengurangi ukuran int, yang berpotensi

menghemat memori.

4. long untuk meningkatkan ukuran int atau double, yang memungkinkan rentang nilai yang lebih panjang.

Tabel 11.1 berikut ini merupakan ringkasan tentang jenis-jenis data dalam bahasa pemrograman C umumnya.

Tabel 11.1 Jenis Data Primitif pada Bahasa Pemrograman C

Jenis Data	Ukuran (byte)	Rentang Nilai
char	1	-128 sampai 127 atau 0 sampai 255
unsigned char	1	0 sampai 255
short	2	-32,768 sampai 32,767
unsigned short	2	0 sampai 65,535
int	2 sampai 4	-32,768 sampai 32,767 atau -2,147,483,648 sampai 2,147,483,647
unsigned int	2 sampai 4	0 sampai 65,535 atau 0 sampai 4,294,967,295
long	4 sampai 8	-2,147,483,648 sampai 2,147,483,647 atau lebih
unsigned long	4 sampai 8	0 sampai 4,294,967,295 atau lebih
long long	8	-9,223,372,036,854,775,808 sampai 9,223,372,036,854,775,807
float	4	±3.4E-38 sampai ±3.4E+38
double	8	±1.7E-308 sampai ±1.7E+308
long double	10 sampai 16	±3.4E-4932 sampai ±1.1E+4932

Catatan: Ukuran dan rentang nilai yang tepat dari jenis data, terutama untuk int dan long, dapat bervariasi sesuai dengan kompiler dan arsitektur sistem yang digunakan.

Contoh jenis data primitif seperti yang telah diuraikan adalah khusus untuk bahasa pemrograman C. Dalam bahasa [emrograman lain, *string* dan Boolean seringkali dikategorikan

sebagai jenis data primitif juga.

1. *string* digunakan untuk menyimpan *string*, yaitu serangkaian karakter. Dalam bahasa pemrograman C, *string* pada umumnya diperlakukan sebagai *array* dari karakter.
2. boolean digunakan untuk menyimpan nilai biner yang dapat bermakna benar (*true*) atau salah (*false*).

Suatu variabel yang didefinisikan menggunakan struktur data primitif dirancang untuk hanya memiliki satu nilai. Jika diinginkan untuk menyimpan lebih dari satu nilai, maka struktur data yang digunakan harus nonprimitif. Struktur data nonprimitif adalah struktur data yang dibuat dengan menggunakan sekumpulan data primitif. Penggunaan struktur data nonprimitif dimaksudkan untuk mengelola dan mengorganisir sekumpulan data secara lebih efisien. Keunggulan dari struktur data nonprimitif antara lain adalah dapat menangani berbagai jenis data dan operasi kompleks seperti pencarian, pengurutan, penyisipan, penghapusan, dan berbagai permasalahan lain.

Secara umum struktur data nonprimitif dapat dikelompokkan menjadi dua, yaitu struktur data linier dan struktur data nonlinier. Perbedaan karakteristik dari masing-masing kelompok ini dapat diringkas seperti pada Tabel 11.2 berikut.

Tabel 11.2 Perbedaan Karakteristik Antara Struktur Data Linier dengan Struktur Data Nonlinier

Karakteristik	Struktur Data Linier	Struktur Data Nonlinier
Susunan elemen	Disusun secara sekuensial	Disusun secara hirarkis atau sembarang
Elemen sebelum	Setiap elemen hanya	Suatu elemen dapat

Karakteristik	Struktur Data Linier	Struktur Data Nonlinier
dan sesudah	memiliki satu elemen sebelum dan setelahnya, kecuali eleen pertama dan terakhir	memiliki lebih dari satu elemen sebelumnya atau setelahnya
Level elemen	Hanya ada satu level elemen	Elemen dapat memiliki lebih dari satu level atau tingkatan
Perlintasan	Dapat dilintasi sekali jalan	Tidak dapat dilintasi hanya dengan sekali jalan
Waktu perlintasan	Diperlukan waktu lebih lama karena elemen disusun secara sekuensial	Waktu perlintasan lebih cepat karena elemen disusun secara hirarkis
Penggunaan memori	Kurang efisien	Lebih efisien
Kemudahan implementasi	Lebih mudah	Lebih sulit
Contoh	*Array, stack, queue, linked list*	*Tree, heap, trie, graph, hash table*
Penggunaan utama	Pengolahan citra, pekerjaan pencetakan, mekanisme *undo*	Pengelolaan direktorim jaringan computer, prioritas anttrian

Struktur data linier, sesuai Namanya, merupakan struktur data yang elemen-elemennya disusun secara linier atau sekuensial dalam urutan tertentu dalam satu level. Elemen kedua diletakkan setelah elemen pertama, elemen ketiga setelah elemen kedua dan seterusnya. Oleh karena itu dalam struktur data linier setiap elemen memiliki pendahulu (*predecessor*) dan penerus (*successor*), kecuali elemen pertama (hanya memiliki penerus) dan elemen terakhir (hanya memiliki pendahulu).

Pengaturan susunan elemen seperti ini memungkinkan untuk mengakses elemen satu persatu tanpa terinterupsi dalam sekali jalan atau iterasi, dimulai dari elemen pertama sampai dengan elemen terakhir.

Struktur data linier pada umumnya relatif mudah dan efisien untuk berbagai operasi dasar seperti penjumlahan, pengaksesan atau penghapusan. Akan tetapi ketika program menjadi semakin kompleks, maka keterbatasan struktur data linier akan mulai tampak. Kompleksitas waktu dalam penggunaan struktur data linier pada umumnya akan bertambah seiring dengan penambahan elemen ke dalam struktur data.

Struktur data linier mencakup *array, stack, queue,* dan *linked list* yang akan diuraikan pada bab-bab selanjutnya.

Struktur data nonlinier menyimpan elemen tidak secara berurutan, melainkan secara hierarkis atau acak pada berbagai level (tingkatan). Jadi, dalam struktur data nonlinier, elemen tidak disimpan secara berurutan atau sekuensial. Dalam struktur data nonlinier tidak memungkinkan untuk melakukan penelusuran dalam sekali eksekusi.

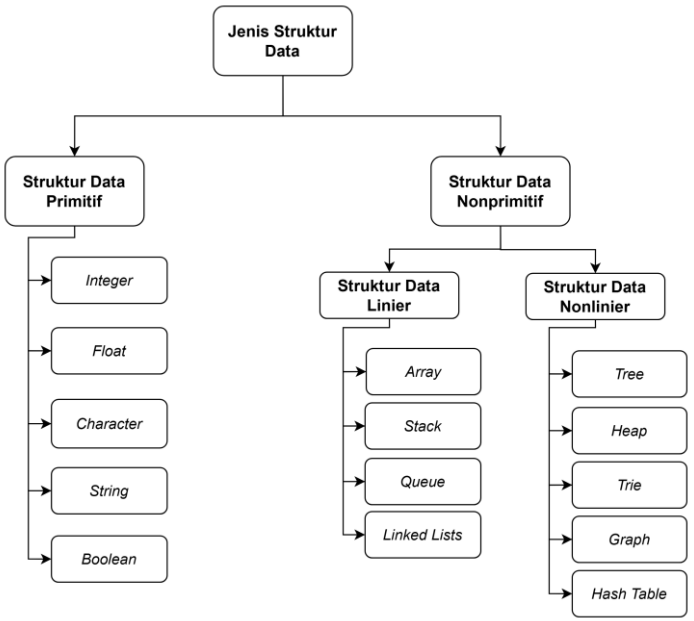
Struktur data nonlinier secara umum relatif lebih sulit untuk diimplementasikan. Namun demikian struktur ini memiliki konerja yang lebih baik untuk operasi yang kompleks. Walaupun data yang digunakan semakin banyak, kompleksitas waktu dan ruang dari struktur data nonlinier akan konstan. Artinya, kinerja dari struktur data nonlinier tidak dipengaruhi oleh banyaknya data yang digunakan.

Penggunaan memori dari struktur data nonlinier lebih efisien dibandingkan dengan struktur data linier. Hal ini dimungkinkan karena dalam struktur data nonlinier dimunkinkan untuk menggunakan koneksi yang sama untuk

berbagi-pakai dengan elemen-elemen data lain.

Struktur data nonlinier mencakup *tree, heap, trie, graph,* dan *hash table*.

Berdasarkan jenis atau klasifikasi struktur data seperti yang telah diuraikan maka dapat disusun jenis struktur data seperti yang disajikan paga Gambar 11.1.



Gambar 11.1 Jenis Struktur Data

Struktur data merupakan cara khusus yang digunakan untuk menyimpan dan mengatur data dalam komputer sehingga dapat digunakan secara efisien. Dalam suatu struktur data terdapat serangkaian aturan dan batasan yang menunjukkan hubungan yang ada antara potongan-potongan atau elemen-elemen data individual yang mungkin terjadi.

BAB 12 *ARRAY* DAN PENGGUNAANNYA DALAM PEMROGRAMAN

Pendahuluan

Dalam pemrograman, struktur data merupakan komponen penting yang memungkinkan pengelolaan data secara efisien. *Array* adalah struktur data yang paling umum dan mudah digunakan (Izadkhah, 2022). Dengan *array*, *programmer* dapat menyimpan sekumpulan data dengan tipe yang sama dalam urutan tertentu, yang membuat akses dan perubahan data lebih mudah. Dengan kemampuannya yang fleksibel, *array* menjadi fondasi bagi banyak algoritma dan aplikasi pemrograman.

Struktur data *array* menyimpan elemen dalam urutan yang diindeks dan memungkinkan setiap elemen untuk diakses secara langsung melalui indeksnya (Xtracts, 2023). Keunggulan utama *array* adalah kemampuannya untuk mengakses elemen secara acak dalam waktu konstan, yaitu O(1), yang berarti waktu akses tidak tergantung pada ukuran *array*. Hal ini membuat *array* sangat efisien untuk operasi seperti pengambilan data atau modifikasi elemen.

Array tidak hanya terbatas pada penyimpanan data satu dimensi. Dalam banyak kasus, *array* multidimensi digunakan untuk merepresentasikan data yang lebih kompleks, seperti matriks atau tabel (Sharma, 2025). Misalnya, *array* dua dimensi sering digunakan dalam pemrosesan gambar, di mana setiap elemen mewakili piksel. Selain itu, *array* juga menjadi

dasar untuk implementasi struktur data lain seperti *stack*, *queue*, dan hash table.

Bab ini akan membahas *array* dan caranya digunakan dalam pemrograman. Kita akan mempelajari cara mendeklarasikan dan menginisialisasi *array*, mengakses dan memodifikasi elemen, serta berbagai operasi yang dapat dilakukan pada *array*. Selain itu, kita juga akan melihat contoh implementasi *array* dalam algoritma-algoritma umum seperti pengurutan dan pencarian. Dengan memahami konsep *array*, diharapkan pembaca dapat mengoptimalkan penggunaannya dalam berbagai skenario pemrograman.

Pengertian dan Karakteristik *Array*

1. Pengertian *Array*
   Salah satu struktur data paling dasar dalam pemrograman adalah *array*, yang digunakan untuk menyimpan sekumpulan elemen dengan tipe data yang sama (Izadkhah, 2022). Elemen-elemen ini disimpan dalam urutan tertentu, dan indeks memungkinkan akses ke elemen-elemen ini. *Array* memungkinkan *programmer* untuk mengelola data secara terstruktur dan efisien, terutama ketika bekerja dengan kumpulan data yang besar.
2. Karekteristik *Array*
   a. Semua elemen dalam *array* harus memiliki tipe data yang sama, baik itu integer, float, *string*, atau tipe data lainnya.
   b. Elemen-elemen *array* disimpan dalam memori secara berurutan. Ini berarti bahwa alamat memori dari elemen pertama hingga elemen terakhir bersifat kontinu, sehingga memudahkan

akses dan pengelolaan data.

c. Elemen-elemen pada *array* bisa diakses dengan memanggil indeksnya, yang biasanya dimulai dari 0.
d. Rata-rata bahasa pemrograman harus menentukan ukuran *array* pada saat deklarasi dan tidak dapat diubah selama runtime. Ini berarti bahwa *array* memiliki ukuran yang tetap setelah dideklarasikan.
e. Salah satu keunggulan utama *array* adalah kemampuannya untuk mengakses elemen secara acak dalam waktu konstan, yaitu O(1). Ini berarti bahwa waktu yang dibutuhkan untuk mengakses elemen tertentu tidak tergantung pada ukuran *array*.

3. Contoh Sederhana *Array*
   Berikut adalah contoh deklarasi *array* menggunakan bahasa pemrograman C:

```
int angka[5] = {1, 2, 3, 4, 5};
```

Dalam pemrograman, *array* merupakan struktur data yang sangat penting, karena kemampuannya untuk menyimpan dan mengelola data secara efisien. Dengan memahami karakteristik *array*, *programmer* dapat memanfaatkannya untuk berbagai keperluan, mulai dari penyimpanan data sederhana hingga implementasi algoritma yang kompleks.

Deklarasi dan Inisialisasi *Array*

1. Deklarasi *Array*
   Deklarasi *array* adalah proses mendefinisikan *array* dalam program sekaligus menentukan tipe data untuk

elemen *array* serta ukuran *array* (Bokare & Suryawanshi, 2024). Deklarasi *array* bervariasi tergantung pada bahasa pemrograman yang digunakan. Berikut adalah contoh deklarasi *array* integer dengan 5 elemen dalam bahasa pemrograman C:

```
int angka[5];
```

int menunjukkan tipe data elemen *array* (integer), angka adalah nama *array*, dan [5] menunjukkan ukuran *array* (5 elemen).

2. Inisialisasi *Array*

   Inisialisasi *array* adalah proses mengisi nilai awal untuk elemen-elemen *array* pada saat deklarasi (Sharma, 2025). Inisialisasi dapat dilakukan secara langsung atau setelah deklarasi. Berikut merupakan contoh inisialisasi *array* menggunakan bahasa pemrograman C:

   a. Inisialisasi langsung dengan nilai:

   ```
   int angka[5] = {1, 2, 3, 4, 5};
   ```

   b. Inisialisasi setelah deklarasi:

   ```
   int angka[5];
   angka[0] = 1;  // Memberikan nilai ke elemen pertama
   angka[1] = 2;  // Memberikan nilai ke elemen kedua
   ```

Deklarasi dan inisialisasi *array* adalah langkah dasar dalam menggunakan *array* dalam pemrograman. Dengan memahami cara mendeklarasikan dan menginisialisasi *array*, *programmer* dapat memulai penggunaan *array* untuk menyimpan dan mengelola data secara efisien. Perbedaan sintaks antara bahasa pemrograman perlu diperhatikan untuk memastikan penggunaan *array* yang tepat.

Operasi Dasar pada *Array*

Array adalah struktur data yang memungkinkan berbagai operasi dasar untuk mengelola dan memanipulasi data. Berikut adalah beberapa operasi dasar yang umum dilakukan pada *array* dalam bahasa C:

1. Mengakses Elemen *Array*
   Elemen *array* dapat diakses dengan memanggil indeksnya. Indeks *array* dimulai dari 0, sehingga indeks 0 untuk elemen pertama, indeks 1 untuk elemen kedua, dan seterusnya.

```c
#include <stdio.h>

int main() {
    int angka[5] = {10, 20, 30, 40, 50};

    // Mengakses elemen ke-3 (indeks 2)
    printf("Elemen ke-3: %d\n", angka[2]);  // Output: 30

    return 0;
}
```

2. Memodifikasi Elemen pada *Array*
   Elemen *array* bisa dimodifikasi dengan memberikan nilai baru ke indeks tertentu.

```c
#include <stdio.h>

int main() {
    int angka[5] = {10, 20, 30, 40, 50};

    // Memodifikasi elemen ke-3 (indeks 2)
    angka[2] = 100;
    printf("Elemen ke-3 setelah modifikasi: %d\n", angka[2]);  // Output: 100

    return 0;
}
```

3. Traversing *Array* (Mengakses Semua Elemen)
   Traversing *array* adalah proses mengakses setiap elemen *array* secara berurutan, biasanya menggunakan *loop*.

```c
#include <stdio.h>

int main() {
    int angka[5] = {10, 20, 30, 40, 50};

    // Traversing array menggunakan loop for
    printf("Elemen array:\n");
    for (int i = 0; i < 5; i++) {
        printf("angka[%d] = %d\n", i, angka[i]);
    }

    return 0;
}
```

Operasi dasar pada *array*, seperti mengakses, memodifikasi, dan traversing adalah hal yang fundamental dalam pemrograman. Dengan memahami operasi-operasi ini, *programmer* dapat memanfaatkan *array* untuk menyelesaikan berbagai masalah pemrograman secara efisien.

Array Satu Dimensi dan Multidimensi

Array dapat dikategorikan berdasarkan dimensinya, yaitu *array* satu dimensi dan *array* multidimensi. Setiap jenis *array* memiliki karakteristik dan penggunaan yang berbeda dalam pemrograman.

1. *Array* 1 Dimensi
   Array 1 dimensi merupakan *array* yang hanya memiliki satu indeks. Ini adalah bentuk *array* yang paling sederhana dan sering digunakan untuk menyimpan data dalam bentuk daftar atau urutan.
   a. Contoh Deklarasi *Array* 1 Dimensi:
      ```c
      int angka[5];
      ```
   b. Inisialisasi *Array* 1 Dimensi:
      ```c
      int angka[5] = {10, 20, 30, 40, 50};
      ```
   c. Contoh Penggunaan:
      Array 1 dimensi sering digunakan untuk

menyimpan daftar nilai, seperti nilai siswa, suhu harian, atau data statistik sederhana.

```c
#include <stdio.h>

int main() {
    int nilai_siswa[5] = {85, 90, 78, 92, 88};

    // Menampilkan nilai siswa
    printf("Nilai siswa:\n");
    for (int i = 0; i < 5; i++) {
        printf("Siswa %d: %d\n", i+1, nilai_siswa[i]);
    }

    return 0;
}
```

2. *Array* Multidimensi

   Array multidimensi merupakan *array* yang memiliki lebih dari satu indeks. *Array* multidimensi yang paling umum adalah *array* dua dimensi (matriks) dan *array* tiga dimensi.

   a. *Array* Dua Dimensi

      Dapat dianggap sebagai "*array* dari *array*" atau tabel yang memiliki baris dan kolom.

      Contoh deklarasi *array* 2 dimensi, mendeklarasikan *array* 2 dimensi 3x3:

      ```c
      int matriks[3][3];
      ```

      Inisialisasi *array* 2 dimensi:

      ```c
      int matriks[3][3] = {
          {1, 2, 3},
          {4, 5, 6},
          {7, 8, 9}
      };
      ```

      Contoh penggunaan, *array* 2 dimensi sering digunakan untuk merepresentasikan matriks, tabel, atau grid, seperti papan permainan, gambar piksel, atau data spreadsheet.

```c
#include <stdio.h>

int main() {
    int matriks[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    // Menampilkan matriks
    printf("Matriks 3×3:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", matriks[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

b. *Array* Tiga Dimensi

Array tiga dimensi dapat dianggap sebagai "*array* dari *array* dari *array*". Ini sering digunakan untuk merepresentasikan data dalam ruang tiga dimensi, seperti volume atau data 3D.

Contoh deklarasi *array* 3 dimensi, mendeklarasikan *array* 3 dimensi 2x3x4:

```c
int volume[2][3][4];
```

Inisialisasi *Array* 3 Dimensi:

```c
int volume[2][3][4] = {
    {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    },
    {
        {13, 14, 15, 16},
        {17, 18, 19, 20},
        {21, 22, 23, 24}
    }
};
```

Contoh Penggunaan, *Array* 3 dimensi dapat diimplementasikan untuk menampilkan data

seperti volume dalam ruang 3D, data voxel dalam pemrosesan gambar 3D, atau data dalam aplikasi simulasi:

```c
#include <stdio.h>

int main() {
    int volume[2][3][4] = {
        {
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {9, 10, 11, 12}
        },
        {
            {13, 14, 15, 16},
            {17, 18, 19, 20},
            {21, 22, 23, 24}
        }
    };

    // Menampilkan volume 3D
    printf("Volume 3D:\n");
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 4; k++) {
                printf("%d ", volume[i][j][k]);
            }
            printf("\n");
        }
        printf("\n");
    }

    return 0;
}
```

Array satu dimensi dan multidimensi adalah struktur data yang sangat berguna dalam pemrograman. *Array* satu dimensi cocok untuk menyimpan data dalam bentuk daftar, sementara *array* multidimensi digunakan untuk merepresentasikan data yang lebih kompleks, seperti matriks atau data 3D. Pemahaman tentang kedua jenis *array* ini memungkinkan *programmer* untuk memilih struktur data yang tepat sesuai dengan kebutuhan aplikasi.

Keunggulan dan Keterbatasan *Array*

Karena kemudahan dan efisiensi, *array* adalah salah satu struktur data yang paling banyak digunakan dalam pemrograman. Namun, seperti halnya struktur data lainnya, *array* memiliki keunggulan dan keterbatasan yang perlu dipahami oleh *programmer*.

1. Keunggulan *Array*
    a. Akses Elemen Secara Acak dalam Waktu Konstan (O(1))
    Salah satu keunggulan utama *array* adalah kemampuannya untuk mengakses elemen secara acak dalam waktu konstan. Artinya, waktu yang dibutuhkan untuk mengakses elemen tertentu (misalnya, elemen ke-5) tidak tergantung pada ukuran *array*.
    Contoh: Dalam *array* int angka[10], mengakses angka[4] membutuhkan waktu yang sama, baik *array* berukuran 10 elemen atau 10.000 elemen.
    b. Penyimpanan Data yang Terstruktur
    Array menyimpan elemen-elemen dalam urutan yang teratur dan berurutan dalam memori. Hal ini memudahkan *programmer* untuk mengelola dan memanipulasi data secara sistematis.
    c. Mudah Diimplementasikan
    Array adalah struktur data yang sederhana dan mudah diimplementasikan dalam hampir semua bahasa pemrograman. Hampir semua bahasa pemrograman menyediakan dukungan bawaan untuk *array*.
    d. Efisiensi dalam Operasi Dasar
    Operasi dasar seperti mengakses, memodifikasi,

dan traversing elemen *array* sangat efisien karena *array* menggunakan indeks untuk mengakses elemen.

e. Dasar untuk Struktur Data Lain
Array sering diimplementasikan sebagai dasar untuk struktur data yang lebih kompleks, seperti *stack*, *queue*, hash table, dan matriks.

2. Keterbatasan *Array*
   a. Ukuran Tetap (Statis)
   Banyak bahasa pemrograman menetapkan ukuran *array* dan tidak dapat diubah selama runtime. Ini dapat menyebabkan masalah jika ukuran *array* terlalu kecil (tidak cukup untuk menyimpan data) atau terlalu besar (membuang memori).
   b. Tidak Fleksibel untuk Penambahan atau Penghapusan Elemen
   Array tidak dirancang untuk menambah atau menghapus elemen secara dinamis. Jika elemen perlu ditambahkan atau dihapus, *programmer* harus membuat *array* baru dan menyalin data dari *array* lama, yang dapat memakan waktu dan memori.
   c. Pemborosan Memori
   Jika *array* dialokasikan dengan ukuran yang lebih besar dari yang diperlukan, memori yang tidak digunakan akan terbuang. Sebaliknya, jika *array* terlalu kecil, program mungkin tidak dapat menyimpan semua data yang diperlukan.
   d. Keterbatasan dalam Menyimpan Tipe Data Berbeda
   Hanya elemen dengan tipe data yang sama dapat

disimpan dalam *array*. Jika *programmer* perlu menyimpan tipe data yang berbeda (misalnya, integer dan *string*), *array* tidak dapat digunakan tanpa trik tertentu seperti menggunakan *array* of struct.

e. Kesulitan dalam Menyisipkan atau Menghapus Elemen di Tengah *Array*
Menghapus atau menempatkan elemen di tengah *array* membutuhkan pergeseran elemen lain, yang dapat memakan waktu, terutama dalam *array* yang besar.

3. Contoh Kasus Keterbatasan *Array*
Misalnya, jika Anda memiliki *array* int angka[5] = {10, 20, 30, 40, 50}; dan ingin menyisipkan nilai 25 di antara 20 dan 30, Anda harus:
   a. Membuat *array* baru yang lebih besar ukurannya.
   b. Menyalin elemen-elemen sebelum titik penyisipan.
   c. Menyisipkan elemen baru.
   d. Menyalin elemen-elemen setelah titik penyisipan.

   Proses ini memakan waktu dan tidak efisien, terutama untuk *array* yang besar.

Array adalah struktur data yang sangat berguna dan efisien untuk banyak skenario pemrograman, terutama ketika ukuran data diketahui dan tidak berubah. Namun, *array* juga memiliki keterbatasan, seperti ukuran tetap dan kurangnya fleksibilitas dalam penambahan atau penghapusan elemen. Oleh karena itu, *programmer* perlu mempertimbangkan keunggulan dan keterbatasan *array* saat memilih struktur data yang tepat untuk aplikasi mereka.

Contoh Implementasi *Array*

Array adalah struktur data yang sangat serbaguna dan dapat digunakan dalam berbagai skenario pemrograman. Berikut adalah beberapa contoh implementasi *array* dalam bahasa C yang menunjukkan bagaimana *array* dapat digunakan untuk menyelesaikan masalah nyata.

1. Menghitung Rata-Rata Nilai dalam *Array*
   Contoh ini menunjukkan bagaimana *array* dapat digunakan untuk menyimpan sekumpulan nilai dan menghitung rata-ratanya.

```c
#include <stdio.h>

int main() {
    int nilai[5] = {85, 90, 78, 92, 88};
    int jumlah = 0;
    float rata_rata;

    // Menghitung jumlah nilai
    for (int i = 0; i < 5; i++) {
        jumlah += nilai[i];
    }

    // Menghitung rata-rata
    rata_rata = (float)jumlah / 5;

    printf("Jumlah nilai: %d\n", jumlah);
    printf("Rata-rata nilai: %.2f\n", rata_rata);

    return 0;
}
```

   Output:

```
Jumlah nilai: 433
Rata-rata nilai: 86.60
```

2. Mencari Nilai Maksimum dan Minimum dalam *Array*
   Contoh ini menunjukkan bagaimana *array* dapat digunakan untuk menemukan nilai maksimum dan minimum dalam sekumpulan data.

```c
#include <stdio.h>

int main() {
    int angka[5] = {10, 20, 30, 40, 50};
    int maksimum = angka[0];
    int minimum = angka[0];

    // Mencari nilai maksimum dan minimum
    for (int i = 1; i < 5; i++) {
        if (angka[i] > maksimum) {
            maksimum = angka[i];
        }
        if (angka[i] < minimum) {
            minimum = angka[i];
        }
    }

    printf("Nilai maksimum: %d\n", maksimum);
    printf("Nilai minimum: %d\n", minimum);

    return 0;
}
```

Output:

```
Nilai maksimum: 50
Nilai minimum: 10
```

Contoh-contoh di atas menunjukkan bagaimana *array* dapat digunakan dalam berbagai skenario pemrograman, mulai dari perhitungan sederhana hingga implementasi struktur data yang lebih kompleks seperti *stack*. Dengan memahami cara mengimplementasikan *array*, *programmer* dapat memanfaatkannya untuk menyelesaikan berbagai masalah secara efisien.

BAB 13 *STRING* DAN OPERASINYA DALAM ALGORITMA

Pendahuluan

Pada bab ini, akan dibahas mengenai definisi *string*, operasi dasar pada *string*, struktur data pendukung *String*, dan perbandingan antar struktur data yang dapat menyimpan *string*. Kita akan mulai dari definisi *string*.

String adalah suatu tipe data non numerik yang tersusun dari satu atau kumpulan char atau sub*string*. *String* dapat berupa huruf (a-z, A-Z), angka (0-9), tanda baca (@, #, dan lain-lain), spasi () (Nyhoff & Nyhoff, 2017).

Sub*string* adalah karakter yang berurutan dalam *string*, masing-masing sub*string* atau elemen penyusun *string* memiliki indeks. Elemen *string* yang memiliki index 0 dikenal dengan istilah prefix. Elemen *string* yang memiliki index terakhir dikenal dengan istilah suffix (Laaksonen, 2017). Salah satu bidang yang meneliti tentang kumpulan *string* atau text adalah text mining.

Text mining adalah disiplin ilmu yang menggabungkan ilmu bahasa dan ilmu komputer dengan teknik statistik dan pembelajaran mesin. Text mining digunakan dalam analisas teks dan mengubah teks dari bentuk yang tidak terstruktur menjadi bentuk yang lebih terstruktur Cielen, D., Meysman, A. D. B., & Ali, M. (2016).

Operasi Dasar *String*

Sebelum masuk pada *top*ik algoritma *String*, ada baiknya kita belajar mengenai operasi dasar yang terdapat pada *string*:

1. *String Concatenation*:
   String dapat digabungkan untuk membentuk susunan *string* yang lebih panjang. Pada proses *concatenation*, urutan *string* pada proses *concatenation* sangat mempengaruhi hasil dari susunan setelah penggabungan *string*. Pada umumnya, *string concatenation* menggunakan operator "+" berikut merupakan contoh dari *string concatenation*:
   Algo + ritma
   Hasil: Algoritma
2. *String Hashing*
   String hashing dapat digunakan untuk memeriksa apakah dua *string* sama secara efisien dengan membandingkan nilai hashnya. Nilai hash adalah bilangan bulat yang dihitung dari karakter-karakter *string*. Jika dua *string* sama maka nilai hashnya juga sama, yang memungkinkan untuk membandingkan *string* berdasarkan nilai hash-nya. Salah satu cara untuk menghitung nilai hash adalah dengan polynomial *hashing* mengikuti persamaan berikut.

   $$(s[0]An-1 + s[1]An-2 + \cdots + s[n-1]A0) \bmod B,$$

   Dimana:
   s[0].... s[n – 1] adalah kode karakter, A dan B adalah konstanta yang telah ditentukan sebelumnya. n adalah jumlah *string*.
   Contoh: Misalkan kita memiliki *string* S = "CAT" dan kita ingin menghitung hash-nya menggunakan polynomial *hashing*. Dimana jumlah *string* (n) adalah 3,

ASCII: C = 67, A = 65, T = 84. Nilai A =3 dan B = 97, maka:

$(67 .3^2 + 65 .3^1 + 84 .3^0) mod\ 97$

$603 + 195 + 84 = 882$

$882\ mod\ 97 = 9$

3. *String Matching*

   Sring *matching* adalah proses pencarian apakah suatu *string*/ pola teks/ pattern terdapat dalam teks. *String matching* biasanya digunakan untuk: Pencarian kata kunci/keyword dalam dokumen. Algoritma dalam teks seperti search engine, dan plagiarism checkher.

   Contoh: Perhatikan *string* yang tersimpan pada hash table berikut. Value pada hash table ini adalah text "Algoritma" yang memiliki key atau index dimulai dari 0 sampai dengan 8. Masing-masing value memiliki index atau key yang berbeda.

Text	A	l	g	o	**r**	i	t	m	a
Index	0	1	2	3	**4**	5	6	7	8

   String yang akan dicari: r

   string "r" ditemukan pada index ke-4

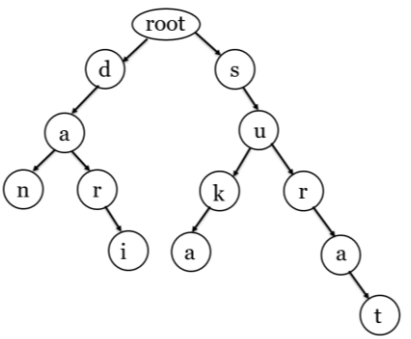
   pada contoh hanya menggunakan satu *string* saja. Namunn, dalam *string matching* tidak terbatas pada satu karakter saja, bisa juga *string*/pola teks/pattern yang dicari merupakan satu kata utuh bahkan lebih.

   Algoritma *String matching* yang dapat digunakan antara lain Tries dan Ternary Search.

Struktur Data untuk Menyimpan *String*

String dapat disimpan dalam representasi struktur data dengan tujuan untuk mempermudah jalannya proses algoritma *string* (Karumanchi, 2016).

1. *Hashing* Table
   Hash table adalah suatu struktur data yang tersusun atas key dan value, hasing table dapat menyimpan *string* sebagai value yang mana tiap value memiliki key sendiri. Contoh penyimpanan *string* dalam hasing table telah dijelaskan pada bagian sebelumnya yaitu *string matching*.
2. Trie
   Trie (dibaca: try) diambil dari kata "retreival" merupakan struktur data berbentuk tree yang juga dikenal sebagai prefix tree. Trie dapat menyimpan kumpulan *string*, terutama untuk operasi pencarian berdasarkan prefix (awalan). Root pada trie biasanya kosong, contoh representasi *string* yang disimpan dalam trie ditunjukkan pada gambar 1. Kata yang memiliki prefix sama akan berbagi parent yang sama juga. Contohnya adalah kata "dari" serta "dan" sama sama memiliki prefiks "da" maka kedua kata tersebut akan berbagi parent "d" dan "a" lalu akan terpisah pada cabang yang berbeda. Kata selanjutnya adalah "suka" dan "surat" yang sama-sama memiliki prefiks "su" kedua kata ini juga berbagi parent node yang sama.



Gambar 13.1 Visualisasi Trie

3. *Binary Search* Tree
   Binary Search Tree adalah struktur data yang dapat digunakan untuk mengatur dan menyimpan data yang diurutkan. Setiap simpul dalam *Binary Search* Tree memiliki paling banyak dua child node yaitu child node kiri dan child node kanan, dengan child node kiri berisi nilai yang lebih kecil dari node induk dan child node kanan berisi nilai yang lebih besar dari node induk. Struktur hierarkis ini memungkinkan operasi pencarian, penyisipan, dan penghapusan yang efisien pada data yang tersimpan di tree.
4. Ternary search trees
   Representasi ternary search tree merupakan penggabungan dari keunggulan *Binary Search* Tree yaitu efisiensi memori dan keunggulan trie dalam efisiensi waktu. Operasi dasar dalam Ternary search tree adalah *insert* atau menambahkan kata/*string* per huruf ke dalam tree. Search atau pencarian huruf per huruf, sesuai cabang kiri/tengah/kanan.
   Perbandingan antara Hash table, Trie, *Binary Search* Tree (BST), dan Ternary Search Tree (TST) ditunjukkan pada tabel 13.1

Tabel 13.1 Perbandingan Struktur data penyimpan *string*

Aspek	***Hash* Table**	**Trie**	**BST**	**TST**
Struktur Dasar	*Array* dengan fungsi *hash*	Pohon per karakter	Pohon biner	Pohon biner per karakter (3 cabang)
Pencarian Parsial (Prefix)	Tidak didukung	Sangat efisien	Tidak bisa	Didukung

Aspek	***Hash* Table**	Trie	BST	TST
Kecepatan Pencarian	O(1) (ideal), tapi tidak urut	O(k) (k = panjang *string*)	O(log n) (jika seimbang)	O(k log n) (lebih lambat dari Trie)
Penggunaan Memori	Hemat (relatif)	Boros (butuh banyak node per huruf)	Hemat	Sedang (lebih hemat dari Trie)
Urutan *Output* Terurut	Tidak bisa	Bisa	Bisa (in-order traversal)	Bisa (in-order traversal)
Dinamis & Skalabilitas	Re-sizing berdasarkan load factor	Dinamis	Dinamis	Dinamis (mudah tumbuh/menyusut)

BAB 14 STRUKTUR DATA *LIST*, *STACK* DAN *QUEUE*

Pendahuluan

Struktur data adalah cara atau metode untuk menyimpan dan mengatur data dalam komputer agar dapat digunakan secara efisien (Canning et al., 2023). Setiap jenis struktur data memiliki karakteristik dan kegunaan tertentu, tergantung pada bagaimana data diakses, disisipkan, atau dihapus. Beberapa struktur data dasar yang umum digunakan dalam pemrograman adalah *list* (daftar), *stack* (tumpukan), dan *queue* (antrian), yang masing-masing menyimpan kumpulan elemen dan menyediakan operasi tertentu untuk memanipulasinya.

Memahami struktur data sangat penting dalam pemrograman karena pilihan struktur data yang tepat dapat meningkatkan efisiensi dan performa program secara signifikan (Ren & Li, 2025). Selain itu, struktur data juga membantu *programmer* dalam menyelesaikan masalah secara sistematis, seperti pengelolaan memori, pencarian data, dan pengurutan. Dengan pemahaman yang baik, seorang *programmer* dapat menulis kode yang lebih bersih, terstruktur, dan mudah dikembangkan.

List, *stack*, dan *queue* memiliki perbedaan dalam cara mereka menyimpan dan mengakses data (Soni, 2024). *List* bersifat fleksibel dan memungkinkan akses langsung ke elemen mana pun berdasarkan indeks. *Stack* mengikuti prinsip LIFO (*Last In, First Out*), di mana elemen terakhir yang dimasukkan

akan menjadi elemen pertama yang diambil. Sementara itu, *queue* menggunakan prinsip FIFO (*Last In, First Out*), sehingga elemen yang pertama masuk akan menjadi elemen pertama yang keluar. Perbedaan inilah yang membuat masing-masing struktur data cocok digunakan untuk situasi tertentu dalam pemrograman.

List (Daftar)

Pengertian dan Konsep Dasar *List*

List atau daftar adalah salah satu struktur data dasar yang digunakan untuk menyimpan sekumpulan nilai atau elemen secara berurutan (Ren & Li, 2025). Elemen-elemen dalam *list* dapat berupa berbagai jenis data, seperti angka, huruf, kata, atau bahkan *list* lainnya. *List* memungkinkan kita menyimpan banyak data dalam satu variabel dan memudahkan akses ke setiap elemen berdasarkan posisinya di dalam daftar.

List biasanya bersifat dinamis, artinya jumlah elemen di dalamnya dapat bertambah atau berkurang sesuai kebutuhan. Elemen-elemen dalam *list* memiliki indeks yang dimulai dari 0. Misalnya, jika kita memiliki *list* berisi ["apel", "pisang", "jeruk"], maka "apel" berada di indeks 0, "pisang" di indeks 1, dan "jeruk" di indeks 2. Dengan indeks ini, kita bisa mengambil, mengganti, atau menghapus elemen tertentu dengan mudah.

Sebagai contoh sederhana, kita bisa menggunakan *list* untuk menyimpan daftar buah yang akan dibeli di pasar: ["mangga", "semangka", "pepaya"]. Dengan *list* ini, kita bisa menambahkan buah baru seperti "nanas", menghapus "semangka", atau mengecek apakah "mangga" ada di dalam daftar. Kegunaan *list* yang fleksibel ini membuatnya sangat umum digunakan dalam pemrograman sehari-hari, bahkan

dalam program-program kecil sekalipun.

Operasi Dasar pada *List*

Agar dapat memanfaatkan *list* secara efektif, penting memahami operasi-operasi dasar yang dapat dilakukan terhadapnya (Canning et al., 2023; Ren & Li, 2025). Operasi-operasi ini meliputi penambahan elemen, penghapusan, pengaksesan, dan pengecekan elemen dalam *list*. Berikut ini adalah penjelasan mengenai operasi dasar pada *list* beserta contoh dalam bentuk *Pseudocode* dan implementasi Python:

1. Menambahkan elemen, berarti memasukkan data baru ke dalam *list* yang sudah ada. Penambahan umum dilakukan di bagian akhir menggunakan *append*, atau disisipkan di posisi tertentu menggunakan *insert*.

Pseudocode Menambahkan Elemen
Deklarasi *list* buah ← ["apel", "jeruk"] Tambahkan "mangga" ke akhir *list* buah.

Contoh kode dan hasil pada python

Kode python dan hasil

```
# Menambahkan Elemen
buah = ["apel", "jeruk"]
buah.append("mangga")
print(buah)  # Output: ['apel', 'jeruk', 'mangga']
```

```
['apel', 'jeruk', 'mangga']
```

2. Menyisipkan elemen, berarti menambahkan elemen baru pada posisi tertentu, bukan di akhir *list*. Operasi ini menggunakan fungsi *insert*.

Pseudocode menyisipkan elemen
Deklarasi *list* buah ← ["apel", "jeruk"] Sisipkan "pisang" pada indeks ke-1.

Contoh kode dan hasil pada python

Kode python dan hasil

```
# Menyisipkan Elemen
buah = ["apel", "jeruk"]
buah.insert(1, "pisang")
print(buah)  # Output: ['apel', 'pisang', 'jeruk']
```

```
['apel', 'pisang', 'jeruk']
```

3. Menghapus elemen berdasarkan nilai, untuk menghapus elemen berdasarkan nilai, gunakan fungsi "*remove*". Fungsi ini akan menghapus elemen pertama yang cocok dengan nilai yang diberikan.

Pseudocode Menghapus Elemen Berdasarkan Nilai
Deklarasi *list* buah ← ["apel", "jeruk", "apel"] Hapus elemen pertama yang bernilai "apel".

Contoh kode dan hasil pada python

Kode python dan hasil

```
# Menghapus Elemen Berdasarkan Nilai
buah = ["apel", "jeruk", "apel"]
buah.remove("apel")
print(buah)  # Output: ['jeruk', 'apel']
```

```
['jeruk', 'apel']
```

4. Menghapus elemen berdasarkan indeks (*pop*), dengan menggunakan fungsi "*pop*()" menghapus elemen terakhir secara *default*, atau elemen pada indeks tertentu jika diberikan parameter.

Pseudocode Menghapus Elemen Berdasarkan Indeks (*Pop*)
Deklarasi *list* buah ← ["apel", "jeruk", "mangga"] Hapus elemen terakhir.

Contoh kode dan hasil pada python

Kode python dan hasil

```
# Menghapus Elemen Berdasarkan Indeks
buah = ["apel", "jeruk", "mangga"]
buah.pop()
print(buah)  # Output: ['apel', 'jeruk']
```

```
['apel', 'jeruk']
```

5. Mengakses dan mengubah elemen, pada akses elemen dilakukan dengan indeks, dimulai dari nol. Untuk mengubah elemen, cukup memberikan nilai baru pada indeks tertentu.

Pseudocode Mengakses dan Mengubah Elemen
Deklarasi *list* buah ← ["apel", "jeruk"] Tampilkan elemen di indeks ke-1 Ubah elemen di indeks ke-0 menjadi "semangka".

Contoh kode dan hasil pada python

Kode python dan hasil

```
# Mengakses dan Mengubah Elemen
buah = ["apel", "jeruk"]
print(buah[1])          # Output: 'jeruk'
buah[0] = "semangka"
print(buah)             # Output: ['semangka','jeruk']
```

```
jeruk
['semangka', 'jeruk']
```

6. Menghitung jumlah elemen dalam *list* dapat dihitung menggunakan fungsi "len()", yang akan mengembalikan total elemen di dalam *list*.

Pseudocode Menghitung Jumlah Elemen
Deklarasi *list* buah ← ["apel", "jeruk", "mangga"] Hitung jumlah elemen dalam *list* buah.

Contoh kode dan hasil pada python

Kode python dan hasil

```
# Menghitung Jumlah Elemen
buah = ["apel", "jeruk", "mangga"]
print(len(buah))  # Output: 3
```

```
3
```

7. Mengecek keberadaan elemen, untuk memeriksa apakah suatu nilai terdapat di dalam *list*, gunakan operator "in". Hasil pemeriksaan berupa nilai boolean *True* atau *False*.

Pseudocode g. Mengecek Keberadaan Elemen
Deklarasi *list* buah ← ["apel", "jeruk"] Periksa apakah "jeruk" ada dalam *list*.

Contoh kode dan hasil pada python

Kode python dan hasil

```
# Mengecek Keberadaan Elemen
buah = ["apel", "jeruk"]
print("jeruk" in buah)  # Output: True
```

```
True
```

Representasi *List* dalam Bahasa Pemrograman

Dalam Python, *list* direpresentasikan sebagai struktur data yang mampu menyimpan kumpulan nilai dalam satu variabel. Nilai-nilai tersebut dapat berupa angka, *string*, bahkan tipe data campuran atau *list* lain. Python menggunakan tanda kurung siku [] untuk menyatakan *list*, dan setiap elemen dipisahkan dengan tanda koma. Python memberikan fleksibilitas tinggi karena *list* bersifat dinamis, artinya ukuran *list* dapat berubah selama program berjalan. Selain itu, *list* diakses menggunakan indeks yang dimulai dari 0. Misalnya, buah[0] akan mengakses elemen pertama yaitu "apel". Python juga mendukung indeks negatif untuk mengakses elemen dari belakang, misalnya buah[-1] akan mengembalikan "mangga".

List dalam Python juga menyediakan banyak fungsi bawaan seperti *append*(), *insert*(), *remove*(), *pop*(), *sort*(), dan *reverse*() yang memudahkan pengelolaan data. Representasi *list* ini sangat intuitif dan efisien, menjadikannya salah satu struktur data paling umum digunakan dalam berbagai aplikasi pemrograman. Contoh sederhana daftar buah dan aksesnya ditunjukkan pada Gambar 14.1.

```
# daftar data buah, sayur dan warna
data_buah_sayur = [
    ["apel", "pisang", "jeruk"],
    ["bayam", "wortel", "brokoli"],
    ["merah", "kuning", "hijau", "oranye"]
]

# Akses data berdasarkan indeks
print(data_buah_sayur[0][1])
print(data_buah_sayur[1][2])
print(data_buah_sayur[2][-1])
```

```
pisang
brokoli
oranye
```

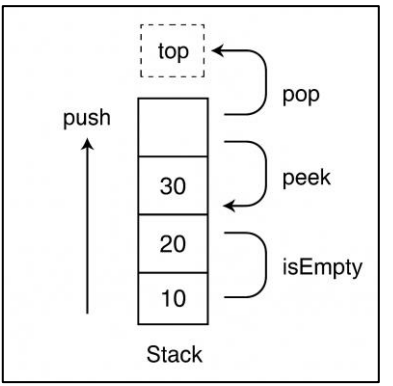
Gambar 14.1 Contoh Sederhana dalam Python

Stack (Tumpukan)

Pengertian dan Konsep Dasar *Stack*

Stack atau tumpukan adalah salah satu struktur data linear yang mengikuti prinsip LIFO (*Last In, First Out*), artinya elemen yang terakhir dimasukkan adalah elemen pertama yang akan dikeluarkan. *Stack* hanya memperbolehkan penyisipan dan penghapusan elemen dari satu sisi, yaitu bagian atas (*top*).

Struktur ini dapat diibaratkan seperti tumpukan piring: piring yang terakhir ditumpuk di atas adalah piring pertama yang diambil. Karena sifatnya yang membatasi akses hanya pada elemen terakhir, *stack* banyak digunakan dalam situasi di mana pengolahan data dilakukan secara berurutan dan berbalik arah, seperti pemanggilan fungsi rekursif, pembatalan perintah (*undo*), dan konversi ekspresi matematika. Ilustrasi *Stack* ditunjukkan pada Gambar 14.2.



Gambar 14.2 Ilustrasi Tumpukan

Operasi Dasar pada *Stack*

Agar dapat memanfaatkan *stack* secara efektif, penting memahami operasi-operasi dasar yang dapat dilakukan di atasnya. Operasi ini memungkinkan penyisipan, penghapusan,

pengaksesan, serta pemeriksaan kondisi *stack*. Berikut adalah penjelasan dari masing-masing operasi dasar yang biasa digunakan pada struktur data *stack*:

1. push (elemen), menambahkan elemen ke atas *stack*. Operasi ini menempatkan data baru di posisi teratas.
2. *pop*(), menghapus dan mengembalikan elemen yang berada di atas *stack*. Jika *stack* kosong, maka operasi ini akan menimbulkan kondisi underflow.
3. *peek*() atau *top*(), mengakses elemen teratas *stack* tanpa menghapusnya. Berguna untuk melihat isi *stack* saat ini.
4. isEmpty(), mengecek apakah *stack* kosong. Mengembalikan nilai boolean *True* jika *stack* tidak memiliki elemen.

Representasi *Stack* dalam Bahasa Pemrograman

Dalam Python, struktur *list* dapat digunakan secara langsung sebagai *stack*. Operasi *append*() digunakan untuk push, dan *pop*() digunakan untuk menghapus elemen terakhir (*top*). Implementasi berbasis *array* seperti ini mudah dilakukan, tetapi memiliki batasan jika digunakan dalam bahasa dengan alokasi memori statis, seperti C.

```
stack = []
stack.append(10)
stack.append(20)
print(stack.pop())  # Output: 20
```

```
20
```

Alternatif lain adalah menggunakan *linked list*, di mana setiap node berisi data dan referensi ke node berikutnya. Keuntungan dari pendekatan ini adalah alokasi memori dinamis, sehingga ukuran *stack* tidak tetap. Implementasi ini cocok untuk penggunaan yang memerlukan fleksibilitas dalam ukuran *stack* dan efisiensi memori.

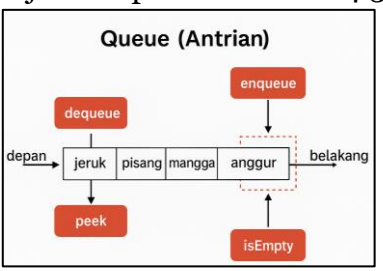
```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class StackLinkedList:
    def __init__(self):
        self.top = None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node

    def pop(self):
        if self.top is None:
            return "Stack Kosong"
        data = self.top.data
        self.top = self.top.next
        return data

    def peek(self):
        return self.top.data if self.top else "Stack Kosong"

    def isEmpty(self):
        return self.top is None
```

Queue (Antrian)

Pengertian dan Konsep Dasar *Queue*

Queue atau antrian adalah struktur data linear yang mengikuti prinsip FIFO (*Last In, First Out*). Artinya, elemen yang pertama masuk adalah elemen pertama yang keluar, seperti barisan orang menunggu giliran di loket (Canning et al., 2023; Soni, 2024). Proses penambahan elemen dilakukan dari bagian belakang (*rear*), sedangkan penghapusan dilakukan dari depan (*front*).

Queue banyak digunakan dalam pemrograman sistem, jaringan, dan proses penjadwalan (*job scheduling*), seperti pada antrean pencetakan dokumen, penanganan event pada sistem operasi, serta simulasi sistem pelayanan (*customer service*). Ilustrasi *Stack* ditunjukkan pada Gambar 14.3.



Gambar 14.3 Ilustrasi Antrian

Operasi Dasar pada *Queue*

Struktur data *queue* digunakan untuk mengelola data dalam urutan yang mengikuti prinsip FIFO (*Last In, First Out*). Agar dapat digunakan secara efektif, perlu dipahami beberapa operasi dasar yang memungkinkan untuk menambah, menghapus, memeriksa, dan mengakses elemen. Berikut adalah penjelasan dari masing-masing operasi dasar pada *queue*:

1. en*queue*(elemen), menambahkan elemen ke bagian belakang antrian. Elemen yang baru ditambahkan akan menunggu giliran untuk diproses.
2. *dequeue*(), menghapus dan mengembalikan elemen dari bagian depan antrian. Jika *queue* kosong, maka kondisi underflow terjadi.
3. *peek*() atau *front*(), mengakses elemen paling depan tanpa menghapusnya. Berguna untuk mengetahui elemen mana yang akan diproses selanjutnya.
4. isEmpty(), mengecek apakah antrian kosong. Jika kosong, kembalikan *True*; jika tidak, kembalikan *False*.

Representasi *Queue* dalam Bahasa Pemrograman

Dalam penggunaannya, *queue* memiliki beberapa operasi penting. Operasi en*queue* digunakan untuk menambahkan elemen baru di belakang antrian, sementara *dequeue* digunakan untuk menghapus elemen dari depan antrian. Selain itu, terdapat operasi *peek* yang memungkinkan melihat elemen paling depan tanpa menghapusnya, dan isEmpty yang memeriksa apakah antrian sedang kosong. Semua operasi ini penting untuk mengelola data secara terstruktur, seperti dalam simulasi antrean kasir, pengelolaan tugas cetak (*printer queue*), atau pemrosesan data streaming.

```python
def enqueue(queue, elemen):
    queue.append(elemen)

def dequeue(queue):
    if len(queue) == 0:
        print("Underflow")
        return None
    return queue.pop(0)

def peek(queue):
    if len(queue) == 0:
        print("Queue kosong")
        return None
    return queue[0]

def isEmpty(queue):
    return len(queue) == 0

# Contoh penggunaan
queue = []
enqueue(queue, 'apel')
enqueue(queue, 'pisang')
enqueue(queue, 'jeruk')

print("Elemen terdepan:", peek(queue))  # Output: apel
print("Hapus elemen:", dequeue(queue))  # Output: apel
print("Queue sekarang:", queue)         # Output: ['pisang', 'jeruk']
print("Apakah kosong?", isEmpty(queue)) # Output: False
```

```
Elemen terdepan: apel
Hapus elemen: apel
Queue sekarang: ['pisang', 'jeruk']
Apakah kosong? False
```

Kode Python di atas mendefinisikan empat fungsi utama untuk mengelola *queue*: en*queue* menambahkan elemen ke belakang *queue* menggunakan *append()*, *dequeue* menghapus elemen dari depan menggunakan *pop*(0) sambil memeriksa kondisi kosong agar menghindari *error* (*underflow*), *peek* memeriksa elemen terdepan tanpa menghapusnya, dan isEmpty mengembalikan nilai boolean untuk mengecek apakah *queue* kosong. Contoh penggunaan di akhir menunjukkan bagaimana tiga elemen ('apel', 'pisang', 'jeruk') dimasukkan ke *queue*, lalu fungsi *peek* digunakan untuk melihat elemen pertama ('apel'), *dequeue* digunakan untuk menghapusnya, dan akhirnya kondisi *queue* diperiksa untuk memastikan masih ada elemen yang tersisa.

BAB 15 ALGORITMA *SORTING*: *BUBBLE SORT*, *SELECTION* SORT, *INSERTION SORT*

Pendahuluan

Dalam ilmu komputer, proses pengurutan (*sorting*) data memiliki peran penting karena menjadi dasar bagi berbagai algoritma lain yang mengandalkan keterurutan data. Contoh nyata adalah algoritma *Binary Search*, yang memerlukan data dalam kondisi terurut (*sorted*) agar dapat beroperasi secara optimal. Di samping itu, pengurutan (*sorting*) juga memiliki penerapan luas dalam kehidupan sehari-hari, seperti dalam penyusunan daftar nama, nomor telepon, dan jenis data lain yang sering digunakan sehari-hari.

Pengurutan (*sorting*) merupakan metode untuk menyusun data yang awalnya tidak beraturan menjadi teratur berdasarkan kriteria atau aturan tertentu. Terdapat dua jenis metode pengurutan (*sorting*) data yang umum digunakan, yaitu:

1. *Ascending Order* (Pengurutan Menaik)
   Pengurutan menaik merupakan proses penataan data dimulai dari nilai yang paling rendah hingga mencapai nilai yang paling tinggi.
2. *Descending Order* (Pengurutan Menurun)
   Pengurutan menurun adalah proses pengaturan data dari nilai tertinggi ke nilai terendah.

Tiga algoritma pengurutan (*sorting*) yang paling mendasar, tetapi sangat penting adalah *Bubble Sort*, *Selection*

Sort, dan *Insertion Sort*. Ketiga algoritma pengurutan (*Sorting*) tersebut mudah dipahami dan diimplementasikan. Namun demikian, ketiga algoritma diatas memiliki keterbatasan masing-masing. Oleh sebab itu, memahami cara kerja masing-masing algoritma sangatlah penting agar dapat menentukan algoritma pengurutan (*Sorting*) yang paling tepat sesuai kebutuhan.

Bubble Sort

Pengertian *Bubble Sort*

Disebut "Bubble" karena dalam proses pengurutan, elemen-elemen data secara perlahan berpindah ke posisi yang benar, menyerupai gelembung yang naik ke permukaan dalam segelas minuman bersoda. *Bubble Sort* adalah algoritma pengurutan yang bergerak berulang dari awal hingga akhir data untuk membandingkan semua nilai yang saling bersebelahan. Selama melakukan perbandingan jika ditemukan urutan nilai data yang tidak sesuai berdasarkan kondisi tertentu (*Ascending Order* atau *Descending Order*) maka tukar ke dua posisi elemen data tersebut sampai semua nilainya sudah terurut.

Kelebihan:

1. Mudah diimplementasikan secara ilustrasi dan pengkodeaan.
2. Tidak mengubah urutan yang memiliki nilai sama.

Kekurangan:

1. Termasuk kategori algoritma pengurutan yang memiliki per*for*ma paling tidak optimal.
2. Proses pengurutan data dalam jumlah besar cenderung mengalami penurunan kinerja yang drastis, terutama saat jumlah data yang diolah meningkat karena

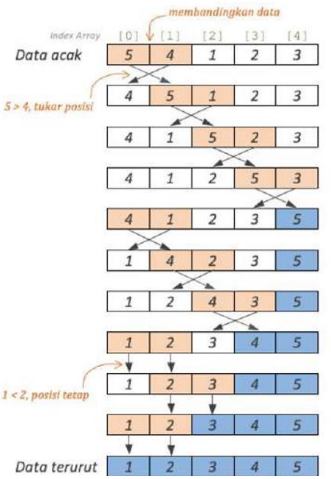
dilakukan secara bertahap atau satu per satu.

Cara Kerja *Bubble Sort*

Berikut adalah cara kerja dari *Bubble Sort*:

1. Bandingkan nilai pertama dengan nilai kedua.
2. Jika nilai pertama lebih besar, tukar posisinya.
3. Teruskan ke nilai berikutnya.
4. Ulangi proses diatas hingga nilai tersusun urut.

Ilustrasi *Bubble Sort*



Gambar 15.1 Ilustrasi cara kerja Bubble Sort

Algoritma *Bubble Sort* (*Pseudocode*)

```
bubbleSort(arr)
        n = panjang(arr)
untuk i dari 0 sampai n-1
                untuk j dari 0 sampai n-i-1
                        jika arr[j] > arry[j+1]
                                tukar(arr[j], arr[j+1])
```

Implementasi *Bubble Sort* dalam Python

```
def bubble_sort(arr):
n = len(arr) # Menyimpan panjang data
for i in range(n):
# Flag untuk mengecek apakah terjadi pertukaran
        swapped = False
        for j in range(0, n - i - 1):
# Bandingkan nilai bersebelahan
        if arr[j] > arr[j + 1]:
# Tukar jika nilai sebelumnya lebih besar
        arr[j], arr[j + 1] = arr[j + 1], arr[j]
        swapped = True
# Jika tidak ada pertukaran, array sudah terurut, berhenti
        if not swapped:
                break
return arr
```

Analisis Kompleksitas

Kompleksitas *Bubble Sort* adalah sebagai berikut:

Tabel 15.1 Kompleksitas *Bubble Sort*

Kasus	*Time Complexity*	Penjelasan
Best Case	O(n)	Pada data sudah terurut cukup satu proses jika menggunakan flag (*swapped*).
Average Case	$O(n^2)$	Pada data acak banyak perbandingan dan pertukaran tetap dilakukan.
Worst Case	$O(n^2)$	Pada data terurut terbalik memerlukan jumlah maksimal perbandingan dan *swap*.
Space Complexity	O(1)	Hanya menggunakan variabel sementara untuk pertukaran.

Selection Sort

Pengertian *Selection Sort*

Merupakan perpaduan antara pengurutan dan pencarian data. *Selection Sort* adalah algoritma pengurutan yang pada setiap proses membandingkan semua nilai untuk mencari nilai terkecil atau terbesar (*Ascending Order* atau *Descending Order*), lalu menempatkan nilai tersebut pada posisi yang sesuai, dan proses ini berlanjut hingga semua nilai terurut. Sepanjang proses berlangsung, perbandingan dan perubahan hanya dilakukan pada indeks yang digunakan sebagai acuan, sementara pertukaran nilai dilakukan setelah proses selesai.

Kelebihan:

1. Sederhana sehingga mudah diterapkan.
2. Pertukaran nilai hanya dilakukan satu kali dalam setiap proses sehingga waktu pengurutan menjadi lebih efisien.
3. Kinerja tidak terpengaruh oleh urutan awal nilai (acak, terurut, atau terbalik).

Kekurangan:

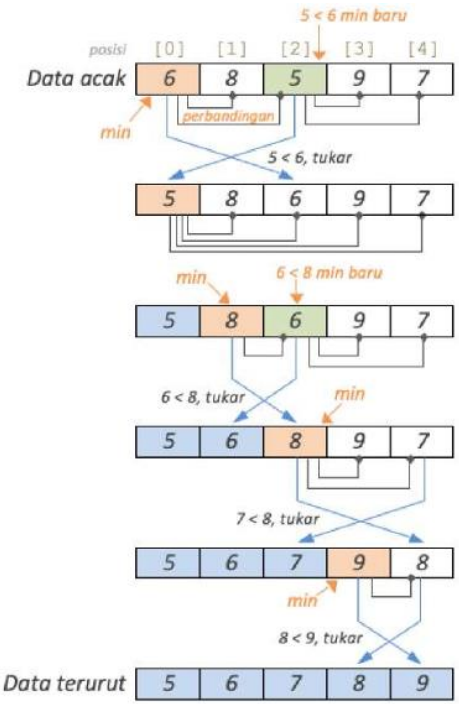
1. Posisi relatif nilai yang sama bisa berubah setelah pengurutan.
2. Kinerja menurun seiring bertambahnya jumlah data

Cara Kerja *Selection Sort*

Berikut adalah cara kerja dari *Selection Sort*:

1. Temukan nilai terkecil atau terbesar.
2. Ganti nilai tersebut dengan nilai yang diposisi awal.
3. Ulangi proses diatas untuk bagian yang belum terurut.
4. Lanjutkan proses hingga seluruh nilai tersusun urut.

Ilustrasi *Selection Sort*



Gambar 15.2 Ilustrasi Cara Kerja Selection Sort

Algoritma *Selection Sort* (*Pseudocode*)

```
selectionSort(arr)
        n = pjg(arr)
untuk i dari 0 sampai n-1
                min_index = i
                untuk j dari i+1 sampai n
                        jika arr[j] < arry[min_index]
                                min_index = j
                        tukar(arr[i], arr[min_index])
```

Implementasi *Selection Sort* dalam Python

```
def selection_sort(arr):
n = len(arr) # Menyimpan panjang data
for i in range(n):
# Asumsikan nilai terkecil ada di posisi i
    for j in range(i + 1, n):
    if arr[j] < arr[min_index]:
            min_index = j
# Bandingkan elemen bersebelahan
            if arr[j] > arr[j + 1]:
# Tukar nilai terkecil dengan nilai pada posisi i
            arr[i], arr[min_index] = arr[min_index], arr[i]
return arr
```

Analisis Kompleksitas

Kompleksitas *Selection Sort* adalah sebagai berikut:

Tabel 15.2 Kompleksitas *Selection Sort*

Kasus	*Time Complexity*	Penjelasan
Best Case	$O(n^2)$	Pada data sudah terurut, tetap dilakukan perbandingan untuk setiap nilai.
Average Case	$O(n^2)$	Pada data acak tetap membutuhkan pencarian nilai minimum di setiap proses.
Worst Case	$O(n^2)$	Pada data terurut terbalik tetap melakukan jumlah maksimal perbandingan.
Space Complexity	O(1)	Hanya menggunakan variabel sementara untuk pertukaran.

Insertion Sort

Pengertian *Insertion Sort*

Asal kata *insert*ion adalah *insert* yang artinya memasukkan atau menyisipkan. Jadi *Insertion Sort* adalah algoritma pengurutan dengan cara mengambil nilai pada data, selanjutnya nilai tersebut akan disisipkan pada posisi yang seharusnya. Pada penyisipan nilai, maka nilai-nilai lain akan bergeser ke belakang.

Kelebihan:

1. Memiliki struktur yang sederhana.
2. Bekerja cukup efisien jika jumlah datanya sedikit atau hampir tersusun.
3. Nilai-nilai yang identik atau sama tetap dipertahankan urutan awalnya.

Kekurangan:

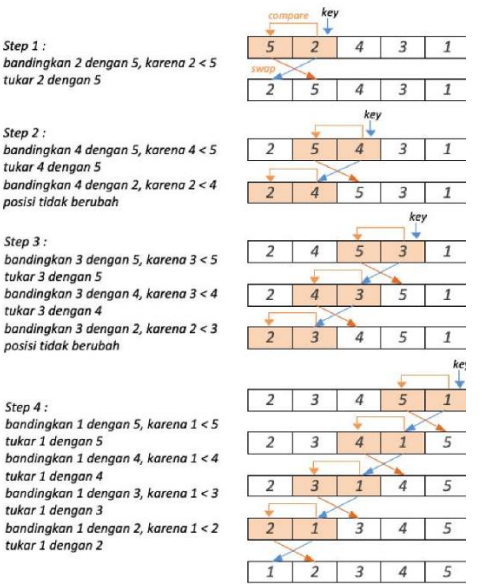
1. Menjadi lambat saat menangani dataset yang besar.
2. Hampir setiap penyisipan memerlukan pemindahan banyak nilai sebelumnya.

Cara Kerja *Insertion Sort*

Berikut adalah cara kerja dari algoritma *Insertion Sort*:

1. Pengurutan dimulai dari nilai kedua dengan membandingkannya terhadap nilai sebelumnya.
2. Jika nilai sebelumnya lebih besar, geser nilai tersebut dan sisipkan/tempatkan nilai saat ini pada posisi yang tepat.
3. Lanjutkan proses hingga seluruh nilai tersusun urut.

Ilustrasi *Insertion Sort*



Gambar 15.3 Ilustrasi Cara Kerja Insertion Sort

Algoritma *Insertion Sort* (*Pseudocode*)

```
insertionSort(arr)
        n = panjang(arr)
untuk i dari 0 sampai n-1
                key = arr[i]
                j-i-1
                sementara j >=0 dan arr[j] > key
arr[j+1] - arry[j]
                        j=j-1
arr[j+1] = key
```

Implementasi *Insertion Sort* dalam Python

```
def insertion_sort(arr):
for i in range(1, len(arr)): # Mulai dari nilai kedua sampai akhir data
        key = arr[i] # Simpan nilai saat ini sebagai 'key'
        j = i – 1 # Inisialisasi indeks sebelah kiri dari 'key'
# Pindahkan nilai yang lebih besar dari key ke satu posisi di depan
        while j >= 0 and arr[j] > key:
            arr[j +1] = arr[j]
            j -= 1
# Tempatkan key di posisi yang tepat
        arr[j + 1] = key
return arr
```

Analisis Kompleksitas

Kompleksitas *Insertion Sort* adalah sebagai berikut:

Tabel 15.3 Kompleksitas *Selection Sort*

Kasus	*Time Complexity*	Penjelasan
Best Case	O(n)	Pada data sudah terurut, hanya perlu satu perbandingan tiap nilai.
Average Case	$O(n^2)$	Pada data acak menyebabkan pergeseran nilai cukup banyak di setiap proses.
Worst Case	$O(n^2)$	Pada data terurut terbalik semua nilai haris digeser setiap terjadi penyisipan.
Space Complexity	O(1)	Hanya menggunakan variabel sementara untuk pertukaran.

Dari ketiga algoritma *Sorting* yang telah dijelaskan sebelumnya, yaitu *Bubble Sort*, *Selection Sort* dan *Insertion Sort* dapat dilihat perbandingan ketiganya sebagai berikut.

Tabel 15.4 Perbandingan Ketiga Algoritma *Sorting*

Aspek	*Bubble Sort*	*Selection Sort*	*Insertion Sort*
Mudah dipahami	Cocok untuk pemula	Logika sederhana	Struktur intuitif

Stabilitas	Stabil (urutan nilai sama tetap)	Tidak Stabil	Stabil (urutan nilai sama tetap)
Optimalisasi Eksekusi	Bisa dihentikan lebih awal dengan *flag*	Tidak bisa dihentikan lebih awal walau sudah terurut	Cepat jika data hampir terurut, tetapi tidak jika data sangat acak
Efisiensi	Kurang efisien meskipun untuk data yang berjumlah kecil dan tidak cocok untuk data yang berjumlah besar	Tidak terlalu efisien, bahkan untuk data yang berjumlah kecil dan lambat saat menangani data yang berjumlah besar	Sangat efisien untuk data kecil, tetapi kurang optimal untuk data yang berjumlah besar
Jumlah Pertukaran	Melakukan terlalu banyak pertukaran nilai	Pertukaran sedikit, tetapi terlalu banyak melakukan perbandingan	Pergeseran terlalu banyak saat nilai tidak terurut

Dari tabel diatas dapat disimpulkan untuk data kecil atau hampir terurut, *Insertion Sort* menjadi pilihan paling efisien dan stabil, sehingga dalam aplikasi dunia nyata, *Insertion Sort* seringa kalo menjadi pilihan yang efisien. Sedangkan *Bubble Sort* dan *Selection Sort* lebih cocok untuk pembelajaran konsep dasar *Sorting*.

BAB 16 PENGENALAN REKURSI DALAM ALGORITMA

Pendahuluan

Dalam dunia pemrograman, terdapat berbagai pendekatan untuk menyelesaikan permasalahan komputasi. Salah satu pendekatan yang sangat penting dan sering digunakan adalah rekursi. Jika sebuah fungsi memanggil dirinya sendiri untuk menyelesaikan masalah secara langsung atau tidak langsung, ini disebut rekursi. Metode ini sangat efektif dalam menyelesaikan masalah yang dapat dipecah menjadi submasalah yang lebih kecil yang memiliki struktur yang sama dengan masalah awal.

Sebelum memahami lebih jauh bagaimana rekursi bekerja, penting untuk mengetahui bahwa teknik ini tidak hanya digunakan dalam contoh-contoh sederhana, tetapi juga diterapkan dalam berbagai bidang pemrograman. Misalnya, dalam matematika, rekursi digunakan untuk menghitung nilai faktorial suatu bilangan atau menghasilkan deret angka Fibonacci. Selain itu, dalam dunia pemrograman yang lebih kompleks, rekursi juga sangat berguna untuk memproses struktur data seperti pohon dan grafik. Struktur-struktur ini memiliki bentuk bercabang dan bertingkat, sehingga pendekatan rekursif sangat cocok digunakan karena mampu menelusuri data secara menyeluruh dengan cara yang efisien dan elegan (Fathansyah, 2007; Wahyono, 2016).

Namun demiki/an, penggunaan rekursi juga menuntut

pemahaman yang mendalam, terutama terkait cara kerja *stack* pemanggilan fungsi, batasan memori, dan risiko seperti *stack overflow* jika kondisi berhenti (*base case*) tidak ditentukan secara benar (Sedgewick, R., & Wayne, 2011). Oleh karena itu, bab ini akan membahas secara menyeluruh mengenai konsep dasar rekursi, jenis-jenis rekursi, perbandingannya dengan iterasi, serta penerapan praktisnya dalam berbagai kasus. Diharapkan setelah mempelajari bab ini, pembaca mampu menerapkan rekursi secara tepat dan efisien dalam menyelesaikan permasalahan pemrograman yang dihadapi.

Pengertian Rekursi

Rekursi adalah metode dalam matematika dan pemrograman di mana sebuah fungsi didefinisikan dengan memanggil dirinya sendiri. Dalam konteks pemrograman, rekursi diimplementasikan melalui fungsi yang memanggil dirinya sendiri untuk menyelesaikan submasalah dari masalah utama (Herlambang, 2018). Jadi, fungsi tersebut akan terus bekerja menyelesaikan bagian-bagian kecil itu sampai mencapai kondisi paling sederhana, lalu mulai menggabungkan hasilnya untuk mendapatkan jawaban akhir.

Dalam pembelajaran algoritma dasar, rekursi sangat membantu untuk menyelesaikan masalah-masalah yang memiliki pola berulang. Misalnya, saat komputer harus mencari sebuah file di dalam banyak folder yang saling bersarang, rekursi dapat digunakan karena proses pencarian di dalam setiap folder bisa dilakukan dengan cara yang sama. Rekursi dalam konsep fundamental yaitu ilmu komputer yang memungkinkan pemecahan masalah yang kompleks dengan membaginya menjadi bagian-bagian yang lebih kecil dan lebih mudah dikelola. Pendekatan ini membimbing sistem komputasi

untuk menyelesaikan tugas secara bertahap dan berulang hingga mencapai kondisi dasar atau inti masalah (Vishwakarma et al., 2024). Begitu juga dalam menghitung faktorial sebuah bilangan, di mana hasilnya diperoleh dengan mengalikan bilangan tersebut dengan faktorial dari bilangan sebelumnya. Pola-pola seperti ini cocok diselesaikan menggunakan rekursi karena setiap langkahnya mirip dengan langkah sebelumnya (Sedgewick & Wayne, 2011).

Komponen Rekursi

Sebuah fungsi rekursif terdiri dari dua komponen utama:

1. Kasus Basis (*Base case*): kondisi penghentian dalam fungsi rekursif yang tidak akan memanggil fungsi itu sendiri lagi. Tanpa *base case*, program akan terus memanggil fungsi secara berulang tanpa pernah berhenti, yang pada akhirnya akan menyebabkan *stack overflow*, yaitu kondisi ketika memori untuk menyimpan pemanggilan fungsi habis (Alfina, Adi Ahmad, Yeni Yanti, 2024).
   Contoh: Misalnya kita ingin menghitung faktorial dari sebuah bilangan n. Dalam bentuk rekursif, kita tahu bahwa:

   ```
   faktorial(n)= n * faktorial(n-1)
   ```

   Tapi ini harus berhenti di suatu titik, yaitu saat n == 1, karena

   ```
   faktorial(1) = 1
   ```

   Jadi, n == 1 adalah kasus basis yang menghentikan

rekursi.

2. Kasus Rekursif (*Recursive Case*): Bagian di mana ini merupakan suatu fungsi memanggil dirinya sendiri dengan parameter yang lebih sederhana atau dapat dikatakan lebih kecil. Tujuannya adalah agar setiap pemanggilan membawa fungsi itu lebih dekat ke kasus basis, sehingga rekursi bisa berakhir dengan hasil yang benar.
   Contoh lanjutan: Masih menggunakan contoh faktorial:

   ```
   faktorial(n)= n * faktorial(n-1)
   ```

   Di sini, faktorial(n-1) adalah kasus rekursif, karena fungsi memanggil dirinya sendiri dengan nilai n yang lebih kecil. Jika awalnya n = 5, maka ia akan memanggil faktorial(4), lalu factorial(3), dan seterusnya, hingga mencapai *base case* faktorial(1).

Perbandingan Rekursi dan Iterasi

Dalam dunia pemrograman, rekursi dan iterasi merupakan dua pendekatan fundamental dalam penyelesaian masalah. Rekursi melibatkan fungsi yang memanggil dirinya sendiri untuk menyelesaikan submasalah hingga mencapai kondisi dasar, sedangkan iterasi menggunakan struktur kontrol seperti *loop* untuk mengulangi serangkaian instruksi sampai kondisi tertentu terpenuhi. Pendekatan ini sering digunakan dalam suatu masalah di mana dapat dipecah menjadi sub-masalah yang serupa, seperti traversal pohon atau perhitungan faktorial. Metode rekursif lebih efisien digunakan untuk perhitungan deret Fibonacci dengan ukuran kecil. Namun, seiring bertambahnya ukuran input, metode iteratif

menunjukkan kinerja yang lebih stabil dan efisien (Lutfina & Ramadhan, 2019). Rekursi memiliki keunggulan dalam menyederhanakan kode untuk masalah yang memiliki struktur berulang atau hierarkis, seperti traversing pohon atau implementasi algoritma *Divide and Conquer*. Sedangkan Iterasi adalah metode pemrograman yang menggunakan struktur pengulangan seperti *for*, *while*, atau do-*while* untuk mengulang serangkaian instruksi hingga kondisi tertentu terpenuhi (Endres et al., 2021). Iterasi lebih efisien dalam penggunaan memori karena tidak memerlukan pemanggilan fungsi berulang yang dapat memperdalam *stack call*.

Dengan demikian, pemilihan antara rekursi dan iterasi harus disesuaikan dengan karakteristik masalah yang dihadapi serta pertimbangan efisiensi dan keterbacaan kode.

Contoh Rekursi

1. Faktorial
   Faktorial dari sebuah bilangan bulat positif n, yang ditulis sebagai n!, adalah hasil perkalian dari semua bilangan bulat dari 1 hingga n (Afrizal Zein & Chrisantus Tristianto, 2025).
   Rumus Faktorial:
   $$\boldsymbol{n}! = \boldsymbol{n} \times (\boldsymbol{n} - \mathbf{1})!$$
   Dengan kondisi dasar (*base case*):
   $$0! = 1$$
   Implementasi dalam Bahasa Python:
   python

```
def faktorial(n):
    if n == 0:
        return 1
    else:
        return n * faktorial(n - 1)
```

Penjelasan:

Jika n == 0, fungsi akan berhenti (*base case*) dan mengembalikan 1.

Jika n > 0, fungsi akan memanggil dirinya dengan n-1, dan terus berlanjut hingga mencapai n = 0.

Misalnya: faktorial(4) akan menjadi 4 * 3 * 2 * 1 * 1 = 24.

2. Pengertian Deret Fibonacci

Deret Fibonacci yaitu urutan angka yang mana dari setiap angka merupakan penjumlahan dari dua angka sebelumnya. Pola Fibonacci dimulai dari 0 dan 1 (Alfina, Adi Ahmad, Yeni Yanti, 2024).

Rumus Fibonacci:

$$\boldsymbol{F(n) = F(n-1) + F(n-2)}$$

Dengan kondisi dasar:

$$F(0) = 0, \quad F(1) = 1$$

Implementasi dalam Python:

python

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1)
+ fibonacci(n - 2)
```

Penjelasan:
Fungsi ini akan terus memanggil dirinya dua kali, dengan n-1 dan n-2, hingga mencapai *base case* n == 0 atau n == 1.
Misalnya: fibonacci(4) akan menghasilkan 3, karena fibonacci(4) = fibonacci(3) + fibonacci(2) = 2 + 1 = 3.

Kelebihan dan Kekurangan Rekursi

Kelebihan Rekursi

1. Kode Lebih Ringkas dan Elegan
   Dalam kasus-kasus tertentu, seperti perhitungan faktorial, pencarian file dalam direktori bersarang, atau perulangan struktur pohon, rekursi memungkinkan kita menulis solusi dalam bentuk kode yang lebih singkat dan lebih mudah dibaca.
2. Cocok untuk Masalah Bertingkat atau Berulang
   Permasalahan seperti perhitungan deret Fibonacci atau pemrosesan folder dalam folder cocok dengan rekursi karena dapat dipecah menjadi submasalah serupa secara alami.
3. Mengikuti Cara Pikir Matematika
   Banyak rumus matematis didefinisikan secara rekursif, dan penerapannya dalam kode menjadi lebih intuitif jika menggunakan pendekatan rekursi.

Kekurangan Rekursi

1. Risiko *Stack Overflow*
   Setiap kali fungsi memanggil dirinya sendiri, sistem menyimpan informasi pada *stack*. Tanpa kondisi dasar yang tepat, pemanggilan yang terus-menerus bisa

menyebabkan kehabisan memori (*stack overflow*).

2. Efisiensi Rendah pada Kasus Tertentu
   Pada kasus seperti deret Fibonacci, pemanggilan rekursif yang tidak dioptimalkan bisa menyebabkan pengulangan perhitungan yang sama berkali-kali. Hal ini membuat eksekusi menjadi lambat.
3. Lebih Sulit untuk Debugging
   Karena fungsi saling memanggil dirinya sendiri, urutan eksekusi bisa menjadi sulit diikuti. Ini menyulitkan pemula dalam melacak kesalahan.
4. Tidak Semua Masalah Cocok Diselesaikan dengan Rekursi
   Beberapa permasalahan justru lebih baik diselesaikan dengan iterasi karena lebih efisien dalam penggunaan memori dan waktu komputasi (Afrizal Zein & Chrisantus Tristianto, 2025).

Studi Kasus & Latihan Soal

Studi Kasus 1: Menghitung Jumlah Elemen dalam Struktur Data Bersarang (*Nested list*)

Seorang pengembang aplikasi pendidikan membuat fitur yang memungkinkan pengguna mengelompokkan materi ke dalam subkategori yang bersarang (*nested list*). Karena kedalaman subkategori tidak diketahui secara pasti, dibutuhkan fungsi untuk menghitung jumlah total elemen (materi) dalam semua lapisan *list* tersebut. Pendekatan rekursif sangat cocok karena setiap *sublist* dapat dianggap sebagai masalah yang sama (menghitung elemen) namun dalam skala lebih kecil.

Masalah:

Buat fungsi rekursif untuk menghitung total elemen dalam *list* yang bisa berisi elemen atau *list* lain.

Deskripsi Kasus: *List* seperti [1, [2, 3], [4, [5, 6]]] perlu dihitung jumlah elemennya tanpa memperhatikan kedalaman.

Studi Kasus 2: Pencarian File dalam Struktur Folder

Dalam sistem operasi, struktur folder dan file sering kali bersifat hierarkis dan bersarang. Saat ingin mencari file dengan nama tertentu, pendekatan rekursi digunakan untuk menelusuri folder dan subfolder secara menyeluruh.

Masalah:

Implementasikan algoritma rekursif untuk mencari apakah file tertentu ada dalam struktur folder yang direpresentasikan sebagai dictionary bersarang (*dict of dict/list*).

Deskripsi Kasus:

Struktur folder bisa berupa dictionary bersarang seperti:

Contoh Implementasi Python:

```
folder = {
   "Documents": {
      "Resume.docx": None,
      "Photos": {
         "photo1.jpg": None
      }
   },
   "Music": {
      "song.mp3": None
   }
}
```

Studi Kasus 3: Perhitungan Nilai Faktorial dalam Statistik Kombinatorial

Dalam analisis kombinatorik, faktorial digunakan untuk menghitung jumlah kemungkinan penyusunan. Misalnya, untuk menghitung permutasi dan kombinasi dalam survei terhadap sejumlah responden. Rekursi sering dipakai untuk menghitung faktorial secara efisien.

Masalah:

Gunakan pendekatan rekursif untuk menghitung nilai faktorial dari suatu bilangan bulat positif, yang selanjutnya bisa digunakan dalam rumus kombinasi (nCr) atau permutasi (nPr).

Deskripsi Kasus:

Dalam kombinasi (nCr), kita butuh faktorial. Misalnya, 5! = 5 × 4 × 3 × 2 × 1.

Contoh Implementasi Python:

```
def faktorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * faktorial(n - 1)
```

Latihan Soal

Soal 1: Penjumlahan Angka

Buat fungsi rekursif untuk menghitung jumlah dari bilangan 1 sampai n. Implementasi Python:

```
def jumlah(n):
    # Lengkapi fungsi ini
    pass
```

Contoh: jumlah(4) harus menghasilkan 10 (karena 1 + 2 + 3 + 4 = 10)

Soal 2: Deret Fibonacci

Buat fungsi rekursif untuk menghitung bilangan ke-n dalam deret Fibonacci. Implementasi Python:

```
def fibonacci(n):
    # Lengkapi fungsi ini
    pass
```

Contoh: fibonacci(6) menghasilkan 8 (karena urutannya: 0, 1, 1, 2, 3, 5, 8).

Bab ini telah membahas secara menyeluruh mengenai konsep rekursi dalam algoritma, mulai dari pengertian dasar, struktur komponen, perbandingan dengan iterasi, hingga penerapan praktis dalam berbagai kasus nyata. Rekursi adalah teknik pemrograman yang memungkinkan suatu fungsi memanggil dirinya sendiri untuk menyelesaikan submasalah yang serupa dengan permasalahan awal. Pendekatan ini sangat relevan digunakan ketika masalah yang dihadapi memiliki sifat bertingkat, berulang, atau menyerupai struktur bercabang seperti pohon atau *nested list*.

Secara umum, rekursi terdiri dari dua komponen utama: *base case* sebagai titik berhenti, dan recursive case sebagai proses pemanggilan ulang dengan parameter yang semakin mendekati kondisi dasar. Tanpa *base case* yang tepat, rekursi bisa menimbulkan masalah seperti *stack overflow* akibat pemanggilan fungsi tanpa henti.

Perbandingan antara rekursi dan iterasi juga menunjukkan bahwa masing-masing pendekatan memiliki kelebihan dan kekurangan tersendiri. Rekursi unggul dalam hal keterbacaan dan kesesuaian dengan masalah yang berstruktur hierarkis, sementara iterasi lebih efisien dari sisi memori dan kecepatan eksekusi, terutama pada input berskala besar.

Dalam bab ini juga ditunjukkan bahwa rekursi bukan

sekadar teori, tetapi merupakan solusi nyata dalam kehidupan sehari-hari di dunia pemrograman, seperti menghitung faktorial, mencari file dalam struktur folder yang bersarang, serta menjelajahi elemen-elemen dalam *nested list*. Selain itu, latihan soal yang disediakan membantu pembaca untuk menerapkan langsung konsep yang telah dipelajari. Dengan memahami dan menguasai rekursi, pembaca diharapkan tidak hanya mampu menulis program yang ringkas, tetapi juga memiliki pemahaman yang lebih dalam terhadap cara kerja algoritma.

BAB 17 ALGORITMA PENCARIAN: *LINEAR SEARCH* DAN *BINARY SEARCH*

Pendahuluan

Dalam bidang ilmu komputer, pencarian data merupakan salah satu proses fundamental yang sangat sering digunakan dalam berbagai aplikasi. Proses ini melibatkan usaha untuk menemukan posisi atau keberadaan suatu nilai tertentu di dalam sekumpulan data, seperti daftar angka, kumpulan nama, atau database.

Agar proses pencarian bisa dilakukan dengan efisien dan tidak memakan banyak waktu, dibutuhkan algoritma yang tepat. Pemilihan algoritma pencarian ini tergantung pada bagaimana struktur data disusun dan seberapa besar ukuran datanya.

Dua algoritma pencarian yang paling mendasar namun sangat penting adalah *Linear Search* dan *Binary Search*:

1. *Linear Search* bekerja dengan cara yang sederhana, yaitu memeriksa satu per satu setiap elemen dari awal hingga akhir hingga menemukan nilai yang dicari. Meskipun mudah diimplementasikan dan tidak memerlukan kondisi khusus pada data, algoritma ini menjadi kurang efisien ketika data berjumlah besar, karena harus melakukan pemeriksaan berulang.
2. *Binary Search*, di sisi lain, menggunakan pendekatan pembagian ruang pencarian secara bertahap. Dengan memanfaatkan fakta bahwa data telah diurutkan terlebih

dahulu, algoritma ini bisa mempersempit ruang pencarian menjadi setengah bagian di setiap langkahnya, sehingga jauh lebih cepat dibandingkan *Linear Search* dalam kasus data besar.

Masing-masing algoritma memiliki kelebihan dan keterbatasan tersendiri. Oleh karena itu, penting untuk memahami cara kerja keduanya agar dapat memilih metode pencarian yang paling sesuai dengan kebutuhan dan kondisi data yang sedang dihadapi.

Linear Search

Definisi *Linear Search*

Linear Search, atau dikenal juga sebagai pencarian sekuensial, adalah metode pencarian sederhana di mana proses pencarian dilakukan dengan memeriksa setiap elemen dalam struktur data satu per satu, dimulai dari elemen pertama hingga elemen terakhir.

Tujuan dari metode ini adalah untuk mencocokkan elemen yang dicari (target) dengan elemen yang ada dalam daftar (*array* atau *list*). Jika ditemukan kecocokan, maka algoritma akan mengembalikan indeks dari elemen tersebut. Jika tidak ditemukan hingga akhir, maka algoritma akan mengembalikan nilai -1, yang menandakan bahwa elemen tersebut tidak ada dalam daftar.

Kelebihan:

1. Sederhana dan mudah diimplementasikan.
2. Tidak memerlukan data yang terurut.

Kekurangan:

1. Kurang efisien untuk dataset besar.
2. Waktu pencarian cenderung lama jika elemen berada di

akhir atau tidak ada.

Ilustrasi Pencarian *Linear Search*

Misalkan terdapat sebuah *array*:

ini
CopyEdit
Data = [15, 28, 36, 49, 55]
Target = 36

Proses *Linear Search* dilakukan sebagai berikut:

1. Bandingkan 15 dengan 36 → tidak sama → lanjut
2. Bandingkan 28 dengan 36 → tidak sama → lanjut
3. Bandingkan 36 dengan 36 → sama → pencarian selesai, elemen ditemukan di indeks ke-2

Jika misalnya target adalah 99 (yang tidak ada dalam *array*), maka pencarian akan dilakukan hingga akhir dan dikembalikan nilai -1.

Algoritma *Linear Search* (*Pseudocode*)

text
CopyEdit
function LinearSearch(arr, target):
  for i from 0 to length(arr) - 1:
    if arr[i] == target:
      return i
  return -1 // jika tidak ditemukan

Langkah-langkah algoritma:

1. Iterasi dilakukan dari indeks pertama (0) hingga terakhir (n-1).
2. Setiap elemen dibandingkan dengan nilai target.

3. Jika ditemukan kecocokan, indeks dikembalikan.
4. Jika tidak ditemukan, algoritma mengembalikan -1.

Implementasi *Linear Search* dalam Python

```
python
CopyEdit
def linear_search(arr, target):
  for i in range(len(arr)):
    if arr[i] == target:
      return i
  return -1  # jika target tidak ditemukan
```

Contoh penggunaan:

```
python
CopyEdit
data = [15, 28, 36, 49, 55]
target = 36

hasil = linear_search(data, target)
print("Elemen ditemukan di indeks:", hasil)
```

Output:

```
yaml
CopyEdit
Elemen ditemukan di indeks: 2
```

Analisis Kompleksitas

Kondisi	Kompleksitas Waktu (Time Complexity)
Best Case	O(1) – Jika elemen berada di posisi awal
Average Case	O(n/2) ≈ O(n)
Worst Case	O(n) – Jika elemen di akhir atau tidak ada

Kompleksitas Ruang (Space Complexity): O(1) – Tidak membutuhkan ruang tambahan karena hanya menggunakan

variabel indeks.

Binary Search

Definisi *Binary Search*

Binary Search atau pencarian biner adalah algoritma pencarian yang efisien dan cepat, khusus digunakan untuk data yang telah terurut. Algoritma ini bekerja berdasarkan prinsip "*Divide and Conquer*" atau membagi masalah menjadi dua bagian yang lebih kecil.

Cara kerja *Binary Search*:

1. Bandingkan nilai tengah dari *array* dengan nilai target.
2. Jika sama, pencarian selesai.
3. Jika target lebih kecil dari nilai tengah, pencarian dilanjutkan di bagian kiri *array*.
4. Jika target lebih besar dari nilai tengah, pencarian dilakukan di bagian kanan *array*.
5. Proses ini diulang sampai nilai ditemukan atau pencarian gagal.

⚠ Catatan penting: *Binary Search* hanya dapat diterapkan pada *array* atau daftar yang sudah diurutkan secara menaik atau menurun.

Ilustrasi Pencarian *Binary Search*

Misalnya kita memiliki *array* yang sudah terurut secara menaik:

ini
CopyEdit
Data = [10, 20, 30, 40, 50, 60, 70]
Target = 50
Langkah-langkah pencarian biner:

1. Tengah = 40 → 50 > 40 → cari di kanan
2. Tengah = 60 → 50 < 60 → cari di kiri
3. Tengah = 50 → cocok → ditemukan di indeks ke-4

Algoritma *Binary Search*

🔁 Versi Iteratif (tanpa rekursi)

text

CopyEdit

```
function BinarySearch(arr, target):
    low = 0
    high = length(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        else if arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1  // jika tidak ditemukan
```

🔁 Versi Rekursif

text

CopyEdit

```
function BinarySearchRecursive(arr, low, high, target):
    if low > high:
        return -1
    mid = (low + high) // 2
    if arr[mid] == target:
        return mid
    else if arr[mid] > target:
        return BinarySearchRecursive(arr, low, mid - 1,
```

```
target)
        else:
            return BinarySearchRecursive(arr, mid + 1, high,
target)
```

Implementasi dalam Python

📌 Versi Iteratif

```
python
CopyEdit
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

📌 Versi Rekursif

```
python
CopyEdit
def binary_search_recursive(arr, low, high, target):
    if low > high:
        return -1
    mid = (low + high) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] > target:
```

```
        return binary_search_recursive(arr, low, mid - 1,
target)
    else:
        return binary_search_recursive(arr, mid + 1, high,
target)
```

Contoh Penggunaan:

```
python
CopyEdit
data = [10, 20, 30, 40, 50, 60, 70]
target = 50

hasil = binary_search(data, target)
print("Elemen ditemukan di indeks:", hasil)
```

Output:

```
yaml
CopyEdit
Elemen ditemukan di indeks: 4
```

Kompleksitas Waktu dan Ruang

Kondisi	Kompleksitas Waktu
Best Case (tengah)	O(1)
Average Case	O(log n)
Worst Case	O(log n)

Kompleksitas Ruang:

1. Iteratif: O(1) – karena tidak menggunakan pemanggilan fungsi secara bertingkat.
2. Rekursif: O(log n) – karena ada tumpukan (*stack*) pemanggilan fungsi yang bertambah setiap iterasi rekursif.

Keunggulan dan Kekurangan *Binary Search*

Keunggulan:

1. Sangat efisien untuk data besar.
2. Jauh lebih cepat dari *Linear Search* (khususnya saat data banyak).

Kekurangan:

1. Hanya bekerja pada data yang terurut.
2. Tidak fleksibel jika data sering berubah atau tidak stabil urutannya.

Perbandingan Linear dan *Binary Search*

Ketika berbicara tentang pencarian elemen dalam sebuah struktur data seperti *array* atau *list*, terdapat dua pendekatan umum yang digunakan, yaitu *Linear Search* dan *Binary Search*. Meskipun keduanya bertujuan untuk menemukan data tertentu, perbedaan mendasar antara keduanya terletak pada cara kerja, efisiensi, serta syarat-syarat penggunaannya.

Perbedaan Cara Kerja

Linear Search, atau pencarian sekuensial, bekerja dengan cara memeriksa setiap elemen dalam *array* satu per satu dari awal sampai akhir. Jika elemen yang dicari ditemukan, maka proses dihentikan. Namun jika tidak, maka pencarian akan terus berlanjut hingga seluruh elemen diperiksa. Sifatnya sangat sederhana dan tidak memerlukan prasyarat apa pun terhadap susunan data.

Sebaliknya, *Binary Search* bekerja dengan prinsip *Divide and Conquer*, yaitu membagi ruang pencarian menjadi dua bagian secara terus-menerus. Pada setiap langkahnya, algoritma ini memeriksa elemen di tengah *array*. Jika elemen

tersebut cocok dengan target, pencarian selesai. Jika tidak, algoritma akan menentukan sisi mana (kiri atau kanan) dari *array* yang harus diproses lebih lanjut, berdasarkan nilai perbandingan. Hal ini hanya bisa dilakukan jika data sudah dalam keadaan terurut.

Kecepatan dan Efisiensi

Dalam praktiknya, *Linear Search* cukup lambat, terutama saat diterapkan pada kumpulan data yang sangat besar. Hal ini karena jumlah perbandingan yang dilakukan akan bertambah seiring bertambahnya jumlah elemen. Dalam kasus terburuk, *Linear Search* mungkin perlu memeriksa seluruh elemen sebelum mengetahui bahwa elemen yang dicari tidak ada.

Sebaliknya, *Binary Search* dikenal jauh lebih efisien. Karena setiap langkahnya memangkas separuh dari data yang tersisa, jumlah langkah pencariannya akan jauh lebih sedikit. Misalnya, jika ada 1.000 elemen, *Binary Search* hanya membutuhkan sekitar 10 langkah (karena 2^10 = 1024), jauh lebih cepat dibandingkan *Linear Search* yang bisa memerlukan hingga 1.000 langkah.

Kompleksitas Algoritma

Dalam analisis kompleksitas waktu:

1. *Linear Search* memiliki kompleksitas waktu O(n), karena dalam kasus terburuk ia harus memeriksa semua n elemen.
2. *Binary Search* memiliki kompleksitas waktu O(log n), karena proses pencarian membagi dua jumlah elemen setiap kali.

   Artinya, *Linear Search* tumbuh secara linear seiring

bertambahnya data, sementara *Binary Search* tumbuh secara logaritmik, yang jauh lebih efisien untuk dataset besar.

Persyaratan terhadap Data

Linear Search tidak memiliki syarat khusus terhadap susunan data. Ia bisa bekerja pada data yang acak, tidak berurutan, bahkan data yang belum lengkap. Ini membuatnya sangat fleksibel, terutama dalam situasi real-time atau saat data terus berubah.

Sebaliknya, *Binary Search* memerlukan data yang sudah diurutkan terlebih dahulu. Jika digunakan pada data yang belum terurut, hasil pencariannya bisa salah atau bahkan menimbulkan *error* logika. Oleh karena itu, sebelum menerapkan *Binary Search*, biasanya dilakukan proses pengurutan terlebih dahulu, seperti menggunakan algoritma *Quick Sort* atau *Merge Sort*.

Kecocokan Berdasarkan Ukuran dan Struktur Data

Linear Search sangat cocok digunakan pada:

1. Dataset yang kecil, karena proses pencarian yang sederhana tidak memberikan beban komputasi besar.
2. Situasi di mana data tidak terurut dan tidak mungkin untuk diurutkan terlebih dahulu.
3. Akses data dilakukan secara sekuensial, misalnya dalam struktur *linked list* atau file stream.

Binary Search sangat cocok digunakan pada:

1. Dataset yang besar dan stabil, di mana data tidak sering berubah dan bisa disimpan dalam urutan tetap.
2. Situasi di mana kecepatan pencarian sangat penting.
3. Struktur data yang memungkinkan akses langsung ke indeks, seperti *array* atau *list* di Python.

Kesulitan Implementasi

Dari sisi implementasi, *Linear Search* sangat mudah. Hanya membutuhkan satu perulangan dan satu perbandingan. Ini membuatnya sangat cocok untuk pemula yang baru belajar algoritma.

Sebaliknya, *Binary Search* lebih kompleks. Implementasinya memerlukan pemahaman tentang indeks batas atas (high) dan batas bawah (low), serta nilai tengah (mid). Selain itu, ada dua cara implementasi: versi iteratif dan versi rekursif, yang masing-masing memiliki kelebihan dan tantangan tersendiri, terutama dalam menghindari kesalahan logika seperti infinite *loop* atau indeks yang salah.

Penggunaan dalam Dunia Nyata

Linear Search biasa digunakan dalam situasi di mana:

1. Data diakses secara acak atau dinamis
2. Jumlah data kecil
3. Prioritasnya adalah kemudahan, bukan kecepatan

Binary Search digunakan dalam:

1. Sistem pencarian cepat seperti pada search engine indexing, database, atau aplikasi mobile yang mengakses daftar besar data.
2. Situasi yang memerlukan respons real-time dengan akurasi tinggi.
3. Pencarian nama, ID, atau data numerik yang telah disusun.

BAB 18 PEMROGRAMAN MODULAR: FUNGSI DAN PROSEDUR DALAM ALGORITMA

Pendahuluan

Di era teknologi informasi yang terus berkembang, aplikasi perangkat lunak semakin kompleks dan beragam. Tantangan utama dalam pengembangan perangkat lunak modern adalah bagaimana menyusun program yang rapi, terstruktur, dan mudah diubah atau diperbaiki seiring waktu. Untuk menjawab tantangan tersebut, konsep pemrograman modular menjadi sangat relevan. Pendekatan ini memecah program besar menjadi bagian-bagian kecil atau modul, di mana setiap modul memiliki tanggung jawab spesifik. Dua elemen utama dalam modularisasi adalah fungsi dan prosedur.

1. Fungsi bertugas untuk memproses input dan mengembalikan hasil berupa nilai, sedangkan
2. Prosedur menjalankan serangkaian perintah tanpa mengembalikan nilai secara langsung.

Pendekatan modular ini membantu menjaga kejelasan struktur program, memudahkan pemeliharaan, dan mendukung pengembangan berkelanjutan, terutama dalam tim pengembang. Bab ini akan menguraikan konsep-konsep dasar tersebut secara mendalam dan memberikan banyak contoh praktis.

Dasar-Dasar Pemrograman Modular

Definisi dan Konsep Dasar Pemrograman Modular

Pemrograman modular adalah paradigma pemrograman yang menekankan pembagian program ke dalam komponen-komponen kecil yang independen. Setiap komponen—baik fungsi maupun prosedur—dibuat untuk menangani tugas tertentu dan dapat diintegrasikan kembali (reusability) ke dalam sistem lain. Kelebihan pendekatan ini meliputi:

1. Keterbacaan kode: Struktur program yang jelas memudahkan pengembang dalam membaca dan memahami kode.
2. Kemudahan pengelolaan: Setiap modul dapat diuji dan diperbaiki secara terpisah.
3. Kolaborasi tim: Pengembang dapat bekerja pada modul-modul yang berbeda secara paralel.
4. Reusabilitas: Modul yang sudah dibuat dapat dipakai ulang di proyek atau bagian lain dari aplikasi.

Sejarah dan Perkembangan Pemrograman Modular

Konsep pemrograman modular bermula dari upaya untuk mengurangi kompleksitas program komputer yang semakin besar di era mainframe. Pada awalnya, pemrograman dilakukan dalam bentuk monolitik, yang menyebabkan kode sulit untuk diorganisir dan dikelola. Seiring waktu, pendekatan struktural mulai diterapkan, yang kemudian berkembang menjadi paradigma yang lebih modern—yaitu pemrograman modular. Saat ini, konsep ini bahkan telah meluas menjadi paradigma lain seperti pemrograman berorientasi objek (OOP), di mana objek-objek mewakili modul modular dengan atribut dan metode yang memungkinkan organisasi kode secara

natural.

Fungsi: Definisi, Struktur, dan Penerapan

Pengertian Fungsi

Dalam pemrograman, fungsi adalah blok kode mandiri yang melakukan operasi tertentu dan mengembalikan hasil (*output*) kepada pemanggil. Fungsi dirancang untuk menerima data (parameter), memproses data tersebut, dan mengembalikan nilai yang dapat digunakan dalam operasi berikutnya.

Struktur Umum Fungsi

Secara umum, fungsi memiliki komponen-komponen berikut:

1. Nama Fungsi: Identitas unik untuk fungsi agar dapat dipanggil.
2. Parameter: Nilai atau data yang diteruskan ke dalam fungsi.
3. Blok Instruksi: Rangkaian perintah yang dieksekusi untuk menghasilkan keluaran.
4. Nilai Kembali (Return Value): Hasil akhir dari proses fungsi.

   Contoh *Pseudocode* Fungsi:

   arduino
   CopyEdit

```
fungsi HitungLuas(panjang, lebar)
  luas = panjang * lebar
  return luas
end fungsi
```

Karakteristik dan Keunggulan Fungsi

1. Reusabilitas: Setelah didefinisikan, fungsi dapat dipanggil berulang kali di berbagai bagian program.
2. Modularitas: Fungsi membantu memecah masalah kompleks menjadi sub-masalah yang lebih mudah diselesaikan.
3. Transparansi: Proses pemrosesan data dapat diisolasi dan diuji secara mandiri.
4. Integrasi: Fungsi dapat digabungkan untuk membentuk alur pemrosesan data yang kompleks.

Contoh Implementasi Fungsi dalam Berbagai Bahasa Pemrograman

1. Python:
   ```
   python
   CopyEdit
   def hitung_luas(panjang, lebar):
       return panjang * lebar

   hasil = hitung_luas(5, 10)
   print("Luas:", hasil)
   ```
2. C:
   ```
   c
   CopyEdit
   int hitungLuas(int panjang, int lebar) {
       return panjang * lebar;
   }

   int main() {
       int hasil = hitungLuas(5, 10);
   ```

```
    printf("Luas: %d", hasil);
    return 0;
}
```

3. Java:

```
java
CopyEdit
public class Hitung {
    public static int hitungLuas(int panjang, int lebar) {
        return panjang * lebar;
    }

    public static void main(String[] args) {
        int hasil = hitungLuas(5, 10);
        System.out.println("Luas: " + hasil);
    }
}
```

Prosedur: Definisi, Struktur, dan Penerapan

Pengertian Prosedur

Prosedur merupakan blok kode yang dirancang untuk menjalankan serangkaian instruksi atau tugas tanpa mengembalikan nilai secara eksplisit ke pemanggil. Fokus utama prosedur adalah menjalankan aksi—misalnya, mencetak pesan, mengubah nilai variabel global, atau melakukan operasi input/*output*.

Struktur Dasar Prosedur

Struktur prosedur menyerupai fungsi, hanya saja tidak ada nilai pengembalian. Elemen utamanya meliputi:

1. Nama Prosedur: Identitas yang memungkinkan

pemanggilan prosedur.
2. Parameter (Opsional): Input yang dibutuhkan untuk menjalankan tugas.
3. Blok Instruksi: Rangkaian aksi yang dijalankan oleh prosedur.
   Contoh *Pseudocode* Prosedur:
   lua
   CopyEdit
   prosedur CetakPesan(pesan)
     output pesan
   end prosedur

Karakteristik Prosedur

1. Tidak Menghasilkan Nilai: Fokus pada efek samping (side effects) seperti mencetak ke layar.
2. Fokus pada Proses: Lebih menekankan pada urutan perintah atau kegiatan.
3. Kepentingan Tindakan: Ideal untuk operasi yang tidak memerlukan feedback berupa nilai.

Contoh Implementasi Prosedur dalam Berbagai Bahasa

1. Python:
   python
   CopyEdit
   def cetak_pesan(pesan):
     print(pesan)

   cetak_pesan("Selamat Datang!")
2. C:
   c
   CopyEdit

```
void cetakPesan(char pesan[]) {
    printf("%s", pesan);
}

int main() {
    cetakPesan("Selamat Datang!");
    return 0;
}
```

3. Java:

```
java
CopyEdit
public class Cetak {
    public static void cetakPesan(String pesan) {
        System.out.println(pesan);
    }

    public static void main(String[] args) {
        cetakPesan("Selamat Datang!");
    }
}
```

Perbandingan Fungsi dan Prosedur

Untuk lebih memahami perbedaan kedua konsep ini, berikut adalah tabel perbandingan:

Aspek	Fungsi	Prosedur
Nilai Kembali	Mengembalikan nilai (return value)	Tidak mengembalikan nilai
Penggunaan	Digunakan dalam perhitungan atau pengolahan data	Digunakan untuk menjalankan aksi atau tugas
Reusabilitas	Bisa digunakan	Umumnya dipanggil untuk

Aspek	Fungsi	Prosedur
	berulang kali di dalam ekspresi	aksi yang terpisah
Contoh Penggunaan	hasil = hitung_luas(5, 10)	cetak_pesan("Selesai")

Perbedaan mendasar ini membuat fungsi lebih sering digunakan dalam operasi matematis atau logika dimana hasil perhitungan dibutuhkan kembali, sedangkan prosedur lebih cocok untuk tugas yang menghasilkan efek samping atau berinteraksi dengan lingkungan eksekusi (misalnya, antarmuka pengguna).

Konsep Parameter dan Scope

Parameter Masukan dan Keluaran

Baik fungsi maupun prosedur dapat menerima parameter sebagai input. Parameter tersebut memungkinkan modul untuk menjadi dinamis dan fleksibel. Ada dua jenis parameter:

1. Parameter Masukan (Input Parameters): Digunakan untuk mengirimkan data ke dalam fungsi/prosedur.
2. Parameter Keluaran (*Output* Parameters): Dalam beberapa bahasa, prosedur dapat mengembalikan nilai dengan mengubah nilai parameter atau melalui mekanisme lain, meskipun tidak menggunakan keyword return.
   Contoh:
   pseudo
   CopyEdit
   fungsi Tambah(a, b)

```
    return a + b
end fungsi
```

Di sini, a dan b merupakan parameter masukan.

Scope dan Lifetime

Konsep scope (cakupan) menentukan di mana variabel atau parameter dapat diakses dalam program.

1. Variabel lokal, yang dideklarasikan dalam fungsi atau prosedur, hanya dapat diakses di dalam blok kode tersebut.
2. Variabel global dapat diakses dari seluruh bagian program, namun penggunaannya perlu diatur agar tidak mengakibatkan konflik atau kesalahan data.

Teknik Pengembangan Modul yang Baik

Agar pemrograman modular dapat berjalan optimal, ada beberapa prinsip dan best practices yang perlu diterapkan:

1. Kohesi Tinggi (*High Cohesion*)
   Setiap modul harus memiliki satu tanggung jawab utama dan mengelompokkan fungsi-fungsi yang terkait. Hal ini meminimalkan ketergantungan internal dan mempermudah pemeliharaan.
2. Kopling Rendah (*Low Coupling*)
   Modul yang berbeda hendaknya memiliki ketergantungan minimal satu sama lain. Dengan demikian, perubahan di satu modul tidak akan memberikan dampak besar ke modul lainnya.
3. Penggunaan Nama yang Deskriptif
   Nama fungsi dan prosedur harus mencerminkan tugas yang dilakukan, sehingga memudahkan *programmer*

lain untuk memahami tujuan modul tersebut tanpa perlu membaca semua baris kode.

4. Dokumentasi dan Komentar
   Memberikan dokumentasi dan komentar di dalam kode adalah hal esensial untuk memastikan bahwa alur program dan logika di balik setiap fungsi atau prosedur mudah dipahami, terutama ketika program dikembangkan secara tim.
5. Pengujian Unit (*Unit Testing*)
   Setiap modul harus dapat diuji secara independen untuk meminimalkan kesalahan dan memastikan bahwa setiap bagian berfungsi sesuai spesifikasi.

Studi Kasus dan Penerapan dalam Proyek Nyata

Studi Kasus 1: Sistem Informasi Desa

Pada proyek pengembangan sistem informasi desa, pemrogram modular dapat diterapkan sebagai berikut.

1. Modul Input Data: Prosedur untuk menerima data transaksi warga atau data aset desa.
2. Modul Perhitungan: Fungsi untuk menghitung total pengeluaran, pendapatan, dan saldo.
3. Modul Pelaporan: Prosedur yang mengubah data perhitungan menjadi laporan yang dapat dicetak atau ditampilkan melalui antarmuka pengguna.
   Contoh *Pseudocode* untuk Sistem Informasi:
   pseudo
   CopyEdit
   fungsi HitungSaldo(pemasukan, pengeluaran)
   saldo = pemasukan - pengeluaran
   return saldo

```
end fungsi

prosedur InputDataTransaksi()
   input pemasukan
   input pengeluaran
   saldo = HitungSaldo(pemasukan, pengeluaran)
   output "Saldo akhir: ", saldo
end prosedur

// Program utama
InputDataTransaksi()
```

Studi Kasus 2: Aplikasi Kalkulator Matematika

Dalam pengembangan aplikasi kalkulator, fungsi-fungsi matematika seperti penjumlahan, pengurangan, perkalian, dan pembagian dapat dikemas sebagai modul mandiri. Sementara, prosedur digunakan untuk mengelola interaksi pengguna (UI) dan menampilkan hasil perhitungan.

Contoh *Pseudocode* Kalkulator:

```
pseudo
CopyEdit
fungsi Tambah(a, b)
   return a + b
end fungsi

fungsi Kurang(a, b)
   return a - b
end fungsi

fungsi Kali(a, b)
   return a * b
```

```
end fungsi

fungsi Bagi(a, b)
  jika b == 0
    output "Error: Pembagian dengan nol!"
    return null
  else
    return a / b
  end jika
end fungsi

prosedur TampilkanHasil()
  input a, b
  hasilTambah = Tambah(a, b)
  hasilKurang = Kurang(a, b)
  hasilKali = Kali(a, b)
  hasilBagi = Bagi(a, b)
  output "Hasil Penjumlahan: ", hasilTambah
  output "Hasil Pengurangan: ", hasilKurang
  output "Hasil Perkalian: ", hasilKali
  output "Hasil Pembagian: ", hasilBagi
end prosedur

// Program Utama
TampilkanHasil()
```

Konsep Lanjutan: Rekursi dan Modularitas

Rekursi dalam Fungsi

Rekursi adalah teknik di mana sebuah fungsi memanggil dirinya sendiri untuk menyelesaikan masalah yang lebih kecil

dari suatu masalah besar. Teknik ini sangat efektif dalam menyelesaikan masalah yang mempunyai pola berulang, seperti perhitungan faktorial, deret Fibonacci, ataupun traversing struktur data seperti pohon atau graf.

Contoh Fungsi Rekursif Faktorial:

```
pseudo
CopyEdit
fungsi Faktorial(n)
  jika n == 0 atau n == 1
    return 1
  else
    return n * Faktorial(n - 1)
  end jika
end fungsi
```

Parameter *Default* dan Parameter Variabel

Dalam beberapa bahasa pemrograman modern, kita dapat menggunakan parameter *default* atau parameter variabel untuk membuat fungsi lebih fleksibel. Hal ini sangat berguna dalam menghindari overload *function* dan membuat kode lebih ringkas.

Contoh Parameter *Default* (Python):

```
python
CopyEdit
def sapa(nama, sapaan="Halo"):
  print(sapaan, nama)

# Panggilan fungsi
sapa("Budi")       # Output: Halo Budi
sapa("Andi", "Hai")  # Output: Hai Andi
```

Error Handling dalam Modul

Dalam pengembangan program, tidak jarang terjadi *error* atau pengecualian. Dengan membagi kode ke dalam fungsi dan prosedur, kita dapat menerapkan teknik penanganan *error* secara terpusat agar tidak mengganggu keseluruhan alur program. Teknik ini sering disebut sebagai exception *handling* dan membantu memastikan program tetap berjalan dengan baik walaupun terjadi kesalahan di salah satu modul.

Best Practices dan Tantangan dalam Pemrograman Modular

1. Pengelolaan Dependensi
   Pastikan setiap modul memiliki sedikit mungkin ketergantungan (dependencies) pada modul lainnya. Hal ini meminimalkan dampak perubahan pada satu modul terhadap keseluruhan sistem.
2. Konsistensi Desain
   Gunakan pola desain (design patterns) dan standar penulisan kode yang konsisten di seluruh modul. Hal ini akan mempermudah pemeliharaan dan kolaborasi dalam tim.
3. Dokumentasi yang Baik
   Setiap fungsi dan prosedur hendaknya didokumentasikan dengan baik. Sertakan komentar dan instruksi penggunaan yang jelas agar orang lain (atau Anda di masa depan) dapat dengan mudah memahami tujuan dan cara kerja modul tersebut.
4. Pengujian Otomatis
   Implementasikan unit testing untuk setiap modul. Teknik pengujian ini memungkinkan Anda mendeteksi

error sejak dini dan memastikan bahwa setiap fungsi dan prosedur bekerja sesuai dengan spesifikasi.

Rangkuman dan Arahan Pengembangan Lebih Lanjut

Sebagai penutup, pengembang didorong untuk selalu mencari cara-cara inovatif dalam mengimplementasikan modularisasi melalui:

1. Eksplorasi konsep pemrograman berorientasi objek untuk membangun modul yang lebih kompleks.
2. Penerapan pattern design seperti factory, singleton, dan observer untuk mengatur integrasi antar modul.
3. Mengasah kemampuan dengan latihan soal yang telah disediakan serta mengembangkan proyek mini untuk menerapkan semua konsep yang telah dipelajari.

Melalui pemahaman mendalam tentang fungsi dan prosedur, setiap pengembang akan memiliki alat dan teknik yang cukup untuk mengatasi permasalahan kompleks dalam dunia pengembangan perangkat lunak modern.

BAB 19 PEMROSESAN DATA DENGAN ALGORITMA GREEDY & *DIVIDE AND CONQUER*

Pendahuluan

Volume data global yang dihasilkan oleh berbagai sumber, mulai dari transaksi bisnis hingga sensor *Internet of Things* (IoT), telah meningkat secara eksponensial dalam beberapa tahun terakhir. International Data Corporation (IDC) memproyeksikan bahwa total data di seluruh dunia akan tumbuh dari 33 zettabyte pada tahun 2018 menjadi 175 zettabyte pada tahun 2025. Pertumbuhan masif ini menjadikan pemrosesan data yang efektif dan efisien sebagai kebutuhan yang semakin mendesak. Pemrosesan data tidak hanya mencakup penyimpanan, tetapi juga mencakup pembersihan, trans*for*masi, analisis, dan penarikan informasi berharga dari himpunan data besar. Untuk mencapai hal tersebut, peran algoritma yang andal dan optimal menjadi krusial dalam memastikan data dapat diolah dengan cepat dan akurat.

Pemilihan metode algoritmik yang tepat berperan penting dalam mengatasi tantangan pemrosesan data skala besar. Algoritma yang kurang efisien dapat menyebabkan waktu eksekusi yang lama, penggunaan sumber daya yang berlebihan, atau bahkan kegagalan dalam menyelesaikan pemrosesan ketika data berukuran sangat besar. Oleh karena itu, pengembangan strategi algoritma yang cerdas menjadi salah satu fokus utama di bidang ilmu komputer dan rekayasa perangkat lunak. Dua pendekatan klasik yang menonjol dalam

perancangan algoritma adalah algoritma Greedy dan algoritma *Divide and Conquer*. Keduanya menawarkan paradigma pemecahan masalah yang berbeda, namun sama-sama bertujuan meningkatkan efisiensi pemrosesan data dan penyelesaian masalah kompleks.

Algoritma Greedy (algoritma tamak) merupakan teknik algoritmik yang selalu mengambil keputusan optimal secara lokal pada setiap langkah penyelesaian masalah, dengan harapan keputusan-keputusan lokal tersebut akhirnya menghasilkan solusi yang optimal secara global. Pendekatan greedy bersifat heuristik, artinya algoritma ini rakus memilih opsi terbaik saat itu juga tanpa meninjau kembali keputusan sebelumnya. Strategi ini terbukti sangat efektif untuk berbagai permasalahan optimasi, terutama yang memenuhi sifat pilihan lokal optimal dan substruktur optimal. Misalnya, algoritma greedy dapat diterapkan pada masalah penjadwalan (seperti memilih sekumpulan kegiatan sehingga tidak terjadi konflik jadwal) atau kompresi data (contohnya algoritma Huffman untuk pengkodean optimal). Meskipun algoritma greedy tidak selalu menjamin solusi optimal untuk semua jenis masalah, keunggulannya terletak pada kesederhanaan dan kecepatan komputasi. Dalam konteks pemrosesan data, algoritma greedy sering digunakan ketika dibutuhkan keputusan cepat yang mendekati optimal, misalnya dalam pengambilan keputusan real-time atau alokasi sumber daya secara cepat.

Di sisi lain, algoritma *Divide and Conquer* (bagi dan taklukkan) menawarkan pendekatan yang berbeda dengan memecah suatu masalah besar menjadi sub-masalah yang lebih kecil dan lebih mudah dikelola. Setiap sub-masalah diselesaikan secara rekursif, kemudian solusi-solusi parsial tersebut digabungkan kembali untuk membentuk solusi akhir bagi

masalah semula Paradigma *Divide and Conquer* telah melahirkan berbagai algoritma fundamental dalam ilmu komputer. Contoh paling klasik adalah algoritma *Merge Sort* untuk pengurutan data, di mana himpunan data dipecah menjadi bagian-bagian yang lebih kecil sebelum diurutkan dan digabungkan kembali secara efisien. Demikian pula, algoritma *Quick Sort* dan pencarian biner (*Binary Search*) merupakan implementasi konsep *Divide and Conquer* yang memanfaatkan rekursi untuk mencapai kompleksitas waktu yang jauh lebih baik daripada metode naif. Keunggulan utama pendekatan ini terletak pada kemampuannya mengurangi kompleksitas masalah: sebuah tugas yang pada skala penuh mungkin membutuhkan waktu komputasi kuadratik dapat dipecahkan menjadi sub-tugas yang masing-masing diselesaikan dalam waktu linier atau log-linear, menghasilkan efisiensi keseluruhan yang tinggi. Dalam ranah pemrosesan data, *Divide and Conquer* sangat relevan ketika menangani dataset besar, karena pembagian tugas ke dalam unit-unit kecil tidak hanya mempermudah pemecahan, tetapi juga memungkinkan eksekusi paralel pada sistem komputasi modern.

Kedua pendekatan algoritmik di atas – greedy dan *Divide and Conquer* – memiliki relevansi yang tinggi bagi mahasiswa, peneliti, maupun praktisi. Bagi mahasiswa dan akademisi, memahami teori di balik algoritma ini sangat penting sebagai dasar untuk menganalisis kompleksitas dan korektness (kebenaran) suatu algoritma. Sementara itu, bagi praktisi di industri, kemampuan mengimplementasikan algoritma greedy dan *Divide and Conquer* secara efektif dapat meningkatkan kinerja aplikasi, khususnya dalam pemrosesan data skala besar dan pemecahan masalah optimasi di dunia nyata. Analisis per*for*ma kedua jenis algoritma ini juga

memberikan wawasan tentang situasi seperti apa masing-masing pendekatan lebih unggul, sehingga pengembang sistem dapat menentukan strategi yang tepat sesuai karakteristik masalah. Misalnya, untuk masalah routing atau penjadwalan real-time, pendekatan greedy yang cepat dan sederhana bisa lebih disukai; sedangkan untuk pengolahan dataset besar yang membutuhkan ketelitian, algoritma *Divide and Conquer* mungkin memberikan hasil yang lebih optimal.

Selain pemahaman konseptual, implementasi praktis dari algoritma-algoritma tersebut tidak kalah penting. Bahasa pemrograman Python telah menjadi salah satu alat utama dalam penerapan algoritma dan pengolahan data karena sifatnya yang mudah dibaca dan ditulis. Python mendukung prototyping algoritma dengan sintaks yang ringkas sehingga sering digunakan sebagai pseudo-code yang dapat dieksekusi. Dalam konteks pemrosesan data, Python juga menyediakan ekosistem pustaka yang kaya (seperti NumPy untuk komputasi numerik, pandas untuk manipulasi data, dan scikit-learn untuk pembelajaran mesin) yang dapat memanfaatkan algoritma greedy maupun *Divide and Conquer* secara efisien. Kemudahan implementasi ini memungkinkan mahasiswa dan peneliti untuk menguji teori algoritma dengan cepat, sementara praktisi dapat langsung mengaplikasikan algoritma dalam proyek data science atau pengembangan perangkat lunak. Mengingat betapa pentingnya jembatan antara teori dan praktek, pembahasan dalam tulisan ini menekankan tidak hanya pada landasan teori algoritma greedy dan *Divide and Conquer*, tetapi juga pada contoh implementasi keduanya menggunakan Python. Dengan demikian, diharapkan pembaca mendapatkan pemahaman yang utuh tentang cara kerja masing-masing algoritma dan bagaimana menggunakannya untuk memecahkan masalah

pemrosesan data secara nyata.

Algoritma Greedy (*Greed Algorithm*)

Algoritma greedy merupakan salah satu strategi desain algoritma yang paling sederhana dan *pop*uler untuk menyelesaikan persoalan optimisasi. Ide pokok dari pendekatan greedy adalah melakukan pilihan yang terbaik pada saat ini tanpa menunda atau mempertimbangkan konsekuensi jangka panjang dari pilihan tersebut. Dengan kata lain, algoritma greedy selalu memilih opsi yang tampak optimal secara lokal pada setiap langkah, dengan harapan bahwa rangkaian keputusan lokal optimal tersebut akan membentuk solusi yang optimal secara global (atau setidaknya mendekati optimal).

Karakteristik utama masalah yang dapat dipecahkan secara optimal dengan algoritma greedy adalah adanya struktur optimal submasalah dan properti greedy choice. Struktur optimal submasalah berarti solusi optimal global dapat dibangun dari solusi optimal submasalah-submasalahnya. Sementara properti greedy choice berarti pilihan lokal optimal pada setiap langkah akhirnya mengarah pada solusi global optimalrepositori.uin-alauddin.ac.id. Jika kedua properti ini terpenuhi, algoritma greedy akan memberikan solusi optimal. Contoh klasik masalah yang memenuhi kriteria tersebut adalah *activity selection problem* (pemilihan interval kegiatan maksimal) dan pengkodean Huffman untuk kompresi data, yang keduanya memiliki bukti *for*mal bahwa pilihan greedy menghasilkan solusi optimal.

Dalam operasi algoritma greedy, biasanya terdapat tiga komponen penting:

1. Himpunan Kandidat: Kumpulan elemen yang berpotensi menjadi bagian dari solusi. Algoritma akan melakukan

iterasi atas himpunan kandidat ini dan pada setiap langkah akan memilih salah satu elemen (kandidat) untuk dimasukkan ke solusi sementara.

2. Fungsi Seleksi (*Selection Function*): Fungsi atau heuristik yang digunakan untuk menentukan kandidat mana yang paling baik atau optimal secara lokal pada setiap langkah. Fungsi seleksi ini didasarkan pada kriteria optimasi masalah. Contohnya, pada masalah penjadwalan interval, fungsi seleksi dapat berupa memilih kegiatan dengan waktu selesai paling awal (*earliest finish time*) di antara kandidat yang tersisa.
3. Fungsi Kelayakan (*Feasible Function*): Sebuah prosedur atau aturan untuk mengecek apakah keputusan (pilihan kandidat) yang diambil masih menghasilkan solusi yang layak atau valid. Jika penambahan elemen kandidat tertentu ke dalam solusi sementara melanggar kendala masalah, maka elemen tersebut dianggap tidak layak dan diabaikan.
4. Fungsi Obyektif: Fungsi yang menghitung nilai atau cost dari solusi yang dibangun. Pada masalah optimasi, biasanya berupa fungsi yang harus diminimalkan atau dimaksimalkan. Algoritma greedy akan menambahkan elemen kandidat selama penambahannya meningkatkan (untuk maksimasi) atau tidak merusak (untuk minimasi) nilai fungsi obyektif secara lokal.

Dengan kerangka di atas, algoritma greedy membangun solusi selangkah demi selangkah. Pada setiap iterasi, dipilih satu elemen dari himpunan kandidat berdasarkan fungsi seleksi yang memberikan kontribusi terbaik terhadap solusi sementara. Jika elemen tersebut layak (tidak membuat solusi menjadi tidak valid), elemen dimasukkan ke solusi. Proses ini berulang hingga

tidak ada lagi kandidat yang dapat dipilih atau hingga solusi telah lengkap.

Sebagai ilustrasi konseptual, pertimbangkan persoalan penukaran uang koin (*coin change problem*) untuk mendapatkan sejumlah nilai dengan jumlah koin paling sedikit. Pendekatan greedy akan selalu mengambil koin dengan denominasi terbesar terlebih dahulu yang tidak melebihi nilai yang tersisa. Misalnya, untuk menukar Rp37.000 dengan koin tersedia {Rp25k, Rp10k, Rp5k, Rp1k}, algoritma greedy akan memilih koin 25k (menyisakan 12k), lalu 10k (menyisakan 2k), kemudian 1k dua kali. Solusi greedy ini menghasilkan empat koin. Pada mata uang standar, solusi ini optimal; namun untuk set denominasi tertentu, pendekatan greedy bisa gagal memberikan solusi optimal global (misalnya bila tersedia koin 4k dan 3k untuk menukar 6k, algoritma greedy akan memilih 4k+1k+1k = 3 koin, padahal solusi optimal adalah 3k+3k = 2 koin). Contoh ini menunjukkan bahwa keberhasilan algoritma greedy sangat bergantung pada sifat masalahnya.

Kelebihan dan Kekurangan Algoritma Greedy

Kelebihan: Pendekatan greedy memiliki sejumlah keunggulan. Pertama, algoritma greedy umumnya mudah diimplementasikan karena mengikuti logika "pilih yang terbaik saat ini" secara langsung. Kedua, greedy cenderung efisien secara waktu untuk banyak kasus, sering kali berjalan dalam kompleksitas linear atau polynomial time yang relatif kecil. Sebagai contoh, banyak algoritma greedy bekerja dalam kompleksitas O(n log n) atau O(n), terutama jika tahap pemilihan kandidat dapat dipercepat dengan struktur data yang tepat (misalnya menggunakan heap/priority *queue* untuk memilih elemen terbaik dengan cepat). Huffman coding,

misalnya, bekerja dalam kompleksitas O(n log n) dengan n adalah jumlah simbol, karena pada setiap langkah memilih dua frekuensi terkecil. Demikian pula, algoritma greedy untuk masalah penjadwalan atau seleksi aktivitas dapat diselesaikan dalam O(n log n) (dengan *Sorting* awal) atau O(n) (dengan struktur data terurut). Ketiga, solusi yang dihasilkan greedy untuk banyak masalah terkenal ternyata optimal atau mendekati optimal. Misalnya, algoritma Dijkstra untuk pencarian lintasan terpendek dan algoritma Prim/Kruskal untuk minimum spanning tree adalah algoritma greedy yang terbukti menghasilkan solusi optimal pada graf berbobot.

Kekurangan: Keterbatasan utama algoritma greedy adalah myopic nature-nya, yaitu sifat rakus yang tidak melihat konsekuensi jangka panjang. Karena keputusan diambil semata-mata berdasarkan informasi lokal terbaik, algoritma greedy dapat terperangkap dalam solusi sub-optimal (optimal lokal tetapi bukan optimal global) apabila masalah yang diselesaikan tidak memenuhi properti *greedy choice*. Sebagai contoh, untuk 0/1 Knapsack Problem (masalah knapsack dengan barang tidak dapat dibagi), strategi greedy (misalnya memilih barang dengan rasio nilai/berat tertinggi terlebih dahulu) tidak selalu memberikan solusi optimal – diperlukan algoritma yang lebih cermat seperti *dynamic programming* untuk mendapatkan solusi optimal. Dengan kata lain, greedy tidak menjamin optimalitas global kecuali pada masalah-masalah khusus yang sudah terbukti memiliki struktur yang cocok.

Selain itu, algoritma greedy tidak menyimpan state sebelumnya; keputusan yang sudah diambil tidak dapat diubah kemudian. Hal ini berarti bila di tengah jalan ternyata pilihan yang diambil sebelumnya kurang tepat, algoritma greedy tidak

memiliki mekanisme backtracking atau peninjauan ulang. Kontras dengan metode seperti *dynamic programming* yang eksplisit mempertimbangkan semua kemungkinan subsolusi (dengan menyimpan hasil submasalah), greedy hanya fokus pada keputusan saat ini. Akibatnya, cakupan aplikasi greedy lebih terbatas dibandingkan metode algoritmik lain yang lebih komprehensif. Meski demikian, karena kesederhanaannya, pendekatan greedy sangat berguna sebagai heuristik untuk menghasilkan solusi cepat yang cukup baik pada masalah-masalah kompleks di mana solusi optimal susah dicari. Dalam praktik pemrosesan data skala besar, algoritma greedy sering dipakai ketika kebutuhan waktu nyata (real-time) lebih diutamakan daripada jaminan optimalitas, atau sebagai tahap awal yang memberikan solusi yang kemudian dapat ditingkatkan dengan metode lain.

Algoritma *Divide and Conquer*

Algoritma *divide-and-conquer* adalah paradigma desain algoritma yang menyelesaikan masalah dengan membagi masalah tersebut menjadi beberapa sub-masalah yang lebih kecil, menyelesaikan setiap sub-masalah secara rekursif, lalu menggabungkan hasil solusi sub-masalah menjadi solusi akhir masalah semula. Secara umum, strategi *divide-and-conquer* dapat dijabarkan dalam tiga langkah berulang:

1. *Divide* (Pembagian): Bagi masalah menjadi dua atau lebih sub-masalah yang berukuran lebih kecil dan serupa dengan masalah semula. Pembagian ini biasanya dilakukan hingga sub-masalah mencapai ukuran yang cukup kecil untuk diselesaikan secara langsung (*base case*).
2. *Conquer* (Penyelesaian): Selesaikan setiap sub-masalah

secara rekursif. Jika sub-masalah sudah sangat kecil atau sederhana, maka diselesaikan secara langsung tanpa rekursi (ini merupakan *base case* dari rekursi).

3. *Combine* (Penggabungan): Gabungkan solusi-solusi dari sub-masalah yang telah diselesaikan pada langkah sebelumnya menjadi solusi komprehensif untuk masalah awal.

Paradigma *divide-and-conquer* efektif digunakan pada banyak masalah komputasi karena dua alasan utama. Pertama, membagi masalah besar menjadi bagian yang lebih kecil dapat mengurangi kompleksitas per sub-masalah, sehingga total pekerjaan yang dilakukan menjadi lebih terstruktur. Kedua, sub-masalah yang terpisah memungkinkan penggunaan rekursi dan potensi paralelisasi. Secara matematis, kompleksitas algoritma *divide-and-conquer* sering dapat dianalisis menggunakan persamaan rekurensi (recurrence relation) dan teorema rekurensi (seperti Master Theorem) untuk menentukan orde kompleksitas waktu, en.wikipedia.org.

Sebagai contoh, algoritma *Merge Sort* membagi masalah pengurutan berukuran n menjadi dua sub-masalah berukuran n/2, dan operasi penggabungan memakan waktu linear O(n); dari ini dapat diturunkan rekurensi $T(n) = 2T(n/2) + O(n)$ yang memberikan solusi $T(n) = O(n \log n)$. Algoritma *divide-and-conquer* melahirkan banyak algoritma terkenal. Selain *Merge Sort*, contoh lainnya adalah *Quick Sort*, *Binary Search*, Karatsuba algorithm untuk perkalian bilangan besar, algoritma Strassen untuk perkalian matriks, pencarian closest pair of points pada bidang, algoritma Fast Fourier Trans*for*m (FFT), dan banyak lagi, en.wikipedia.org.

Semua algoritma tersebut menggunakan prinsip yang sama: mengecilkan lingkup permasalahan secara rekursif.

Perbedaan penting antara *divide-and-conquer* dan pendekatan lain adalah sifat rekursif dan kebutuhan untuk menggabungkan hasil. Berbeda dengan algoritma greedy yang membangun solusi secara langsung, *divide-and-conquer* sering kali tidak menghasilkan solusi parsial yang bermakna hingga tahap penggabungan. Sebagai ilustrasi, dalam *Merge Sort*, pembagian menghasilkan potongan *array* yang tidak urut, dan baru setelah penggabungan akhir solusi terurut diperoleh. Demikian pula pada *Quick Sort*, pemilihan pivot membagi *array*, namun urutan akhir baru terbentuk setelah seluruh rekursi selesai. Dalam konteks pemrosesan data skala besar, *divide-and-conquer* sangat bermanfaat karena sub-masalah dapat didistribusikan ke berbagai unit komputasi. Paradigma ini sejalan dengan model komputasi paralel dan terdistribusi seperti MapReduce (Dean & Ghemawat, 2004), di mana data yang sangat besar dibagi menjadi potongan-potongan yang diproses secara parallel (*conquer*) di banyak node, lalu hasil-hasil parsial dikombinasikan. Dengan kata lain, *divide-and-conquer* tidak hanya konsep algoritmik teoretis, tetapi juga mendasari arsitektur pemrosesan data modern untuk big data, karena sifatnya yang scalable. Pembagian masalah juga membantu dalam penggunaan cache memory pada pemrosesan sekuensial: memproses sub-masalah kecil seringkali lebih cache-friendly dan efisien daripada memproses langsung masalah berukuran besar.

Kelebihan dan Kekurangan Algoritma *Divide-and-conquer*

Kelebihan: Algoritma *divide-and-conquer* menawarkan kerangka yang kuat untuk merancang algoritma yang efisien.

Dengan memecah masalah besar menjadi sub-masalah, algoritma menjadi lebih mudah dikelola dan dianalisis. Setiap sub-masalah yang lebih kecil dapat difokuskan secara terpisah, kadang dengan teknik yang lebih sederhana. Strategi ini memungkinkan penggunaan ulang (*reuse*) solusi sub-masalah sehingga tidak ada pekerjaan yang dilakukan berulang-ulang untuk bagian masalah yang sama. *Divide-and-conquer* juga sangat mendukung paralelisasi: karena sub-masalah biasanya independen, kita dapat mengeksekusi penyelesaian sub-masalah pada beberapa prosesor secara simultan, kemudian menggabungkan hasilnya. Hal ini krusial dalam pemrosesan data skala besar di lingkungan komputasi modern (misalnya *cluster computing* atau *multi-core processors*). Sebagai contoh, algoritma *Merge Sort* mudah diparalelkan dengan memberikan setiap separuh *array* ke thread terpisah untuk diurutkan, lalu menggabungkan dua hasil tersebut. Skalabilitas inilah yang membuat *divide-and-conquer* menjadi pilihan alami untuk algoritma big data. Selain itu, analisis kompleksitas algoritma *divide-and-conquer* umumnya lebih terstruktur. Dengan metode rekurensi, kita dapat memperkirakan time complexity dengan lebih sistematis. Banyak algoritma *divide-and-conquer* yang memiliki kompleksitas optimal atau mendekati optimal untuk kelas masalahnya. Sebagai contoh, kompleksitas $O(n \log n)$ untuk pengurutan (*Merge Sort*, *Quick Sort* rata-rata) diketahui optimal untuk algoritma berbasis perbandingan menurut teori lower bound *Sorting* (Knuth, 1998). Juga, algoritma berbasis *divide-and-conquer* seringkali mudah dimodifikasi menjadi algoritma *decrease-and-conquer* (kasus khusus di mana pembagian menghasilkan satu sub-masalah, seperti *Binary Search*) atau dioptimalkan lebih lanjut menjadi algoritma *dynamic programming* apabila sub-masalah

tumpang tindih.

Kekurangan: *Overhead* rekursi dan penggabungan merupakan salah satu kekurangan pendekatan *divide-and-conquer*. Setiap kali masalah dipecah, dibutuhkan memori tambahan untuk memegang hasil sementara sub-masalah, dan ada *overhead* pemanggilan rekursif. Pada beberapa kasus, *overhead* ini bisa signifikan. Misalnya, *Quick Sort* dalam kasus terburuk (pivot terburuk) membelah data sedemikian rupa sehingga salah satu sub-masalah berukuran hampir n, dan yang lain sangat kecil, sehingga degenerasi ke $T(n) = T(n-1) + T(0) + O(n)$ yang kompleksitasnya $O(n^2)$. Meskipun kasus terburuk jarang terjadi jika pivot dipilih secara acak atau median, ini menunjukkan sensitivitas per*for*ma *divide-and-conquer* terhadap cara pembagian sub-masalah. Selain itu, algoritma *divide-and-conquer* seperti *Merge Sort* memerlukan memori tambahan untuk tahap penggabungan (misalnya *array* tambahan sebesar n untuk menggabungkan dua *Subarray* terurut). Penggunaan memori ekstra ini kadang menjadi kelemahan jika ruang memori terbatas; *Quick Sort* tidak membutuhkan *array* tambahan besar (*in-place*), namun algoritma lain mungkin membutuhkan.

Dalam implementasi nyata, rekursi yang dalam juga bisa menyebabkan penggunaan *stack* yang besar dan potensi *stack overflow* bila optimisasi tail-recursion tidak diterapkan oleh bahasa pemrograman. Meskipun demikian, banyak bahasa modern atau perancang algoritma yang mengubah algoritma *divide-and-conquer* rekursif menjadi bentuk iteratif untuk keperluan praktis. Sebagai contoh, *Merge Sort* dan *Quick Sort* dapat diimplementasikan secara iteratif.

Terakhir, tidak semua masalah cocok untuk *divide-and-conquer*. Jika sub-masalah tidak independen (misal terjadi

tumpang tindih sub-masalah yang signifikan) tanpa penyelesaian yang disimpan, *divide-and-conquer* murni akan mengulang komputasi berulang kali. Pada situasi seperti itu, pendekatan *dynamic programming* lebih tepat karena menyimpan hasil sub-masalah. Contohnya, Fibonacci sequence dapat diselesaikan dengan *divide-and-conquer* (rekurens Fibonacci), namun ini sangat tidak efisien ($T(n)$ eksponensial) karena banyak sub-perhitungan yang diulang. Barulah dengan memoization atau DP, perhitungan Fibonacci dapat efisien O(n). Dengan demikian, penting untuk mengidentifikasi struktur masalah: apakah cukup dipecah saja (*divide-and-conquer*) atau perlu dipecah dan disimpan (*dynamic programming*).

Secara ringkas, *divide-and-conquer* unggul dalam hal struktur dan paralelisasi, namun memiliki *overhead* dan syarat kemandirian sub-masalah. Dalam pemrosesan data skala besar, pemahaman trade-off ini penting: kadang kombinasi antara greedy dan *divide-and-conquer* juga digunakan (misalnya algoritma hibrida yang menjalankan *divide-and-conquer* hingga input kecil, lalu menggunakan algoritma greedy atau sederhana untuk menyelesaikan sub-masalah kecil dengan cepat).

Studi Kasus Algoritma Greedy: Kompresi Data Huffman

Sebagai contoh penerapan algoritma greedy dalam pemrosesan data, kita akan membahas pengkodean Huffman (Huffman coding) untuk kompresi data. Huffman coding adalah algoritma kompresi lossless yang menghasilkan kode biner dengan panjang bit variabel untuk setiap simbol input,

sedemikian rupa sehingga simbol yang lebih sering muncul diberi kode lebih pendek daripada simbol yang jarang muncul. Algoritma Huffman terbukti menghasilkan kode prefix optimal untuk sekumpulan simbol dan frekuensi yang diberikan (Huffman, 1952). Pendekatan yang digunakan oleh Huffman coding bersifat greedy: pada setiap langkah, algoritma memilih dua simbol (atau gabungan simbol) dengan frekuensi terendah untuk digabungkan. Penggabungan ini mengurangi jumlah simbol dengan membuat pohon biner, di mana dua simbol terjarang dijadikan anak dari sebuah simpul baru yang frekuensinya merupakan jumlah frekuensi keduanya. Proses ini diulang terus hingga tersisa satu simpul akar yang mencakup semua simbol. Hasil akhirnya adalah sebuah pohon Huffman, di mana setiap daun merepresentasikan simbol asli, dan jalur dari akar ke daun memberikan kode biner untuk simbol tersebut (dengan konvensi, misal turun kiri = bit 0, turun kanan = bit 1).

Sebagai ilustrasi, misalkan terdapat enam karakter dengan frekuensi kemunculan: a:45, b:13, c:12, d:16, e:9, f:5 (contoh ini berjumlah total 100, seperti dalam Huffman, 1952). Algoritma Huffman akan bekerja sebagai berikut.

1. Langkah 1: Pilih dua simbol terjarang: f (5) dan e (9). Gabungkan menjadi simpul baru dengan frekuensi 14. Simpul baru ini tidak memiliki kode (bukan simbol asli).
2. Langkah 2: Dari daftar simbol dan simpul baru {a:45, b:13, c:12, d:16, [e+f]:14}, pilih dua dengan frekuensi terendah: c (12) dan b (13). Gabungkan menjadi simpul baru dengan frekuensi 25.
3. Langkah 3: Daftar sekarang: {a:45, d:16, [e+f]:14, [b+c]:25}. Pilih dua terendah: [e+f] (14) dan d (16). Gabungkan menjadi simpul baru frekuensi 30.

4. Langkah 4: Daftar: {a:45, [b+c]:25, [d+e+f]:30}. Pilih dua terendah: [b+c] (25) dan [d+e+f] (30). Gabungkan jadi simpul 55.
5. Langkah 5: Daftar: {a:45, [b+c+d+e+f]:55}. Terakhir, gabungkan a (45) dan [b+...+f] (55) menjadi akar dengan frekuensi 100.

Hasilnya adalah pohon biner dengan total frekuensi 100 di akar. Setiap cabang kiri/kanan mewakili bit 0/1. Misalnya, dalam satu konstruksi Huffman, karakter 'a' mungkin menjadi daun tunggal di satu cabang (misal kiri) sehingga kodenya 0, sedangkan karakter lain di cabang kanan memperoleh kode yang lebih panjang. Kode Huffman akhirnya mungkin: a = 0; c = 100; b = 101; f = 1100; e = 1101; d = 111. (Perlu dicatat bahwa kode dapat bervariasi tergantung penggabungan, namun panjangnya tetap sama; kode dengan panjang lebih pendek diberikan ke frekuensi lebih besar). Jika kita bandingkan, sebelum kompresi setiap karakter membutuhkan 3 bit (karena ada 6 simbol, butuh setidaknya 3 bit tetap untuk merepresentasikan, misal a=000, b=001, dll, total 300 bit untuk 100 karakter). Dengan kode Huffman di atas, total bit yang diperlukan = 451 + 133 + 123 + 163 + 94 + 54 = 224 bit, jauh lebih kecil dari 300 bit (terjadi penghematan 76 bit, sekitar 25% lebih efisien). Inilah keuntungan Huffman: memanfaatkan frekuensi untuk mengurangi ukuran penyimpanan. Algoritma Huffman memerlukan struktur data priority *queue* (mis. min-heap) untuk dengan cepat memilih dua simbol terendah secara berulang. Kompleksitas waktunya adalah O(n log n) untuk n simbol (karena setiap pengambilan minimum dari heap O(log n) dan dilakukan ~n kali). Berikut ini implementasi Python untuk algoritma Huffman, dengan penjelasan langkah per langkah:

```
import heapq  # menggunakan heapq sebagai priority queue min-heap

# Kelas simpul Huffman
class HuffmanNode:
    def __init__(self, char, freq, left=None, right=None):
        self.char = char        # karakter (None untuk simpul internal)
        self.freq = freq        # frekuensi kemunculan karakter
        self.left = left        # anak kiri (sub-pohon)
        self.right = right      # anak kanan (sub-pohon)
    def __lt__(self, other):
        return self.freq < other.freq  # perbandingan berdasarkan frekuensi (agar bisa dimasukkan ke heap)

def build_huffman_tree(freqs):
    """
    Membangun pohon Huffman dari kamus frekuensi `freqs`.
    `freqs` adalah dict {char: freq}.
    Mengembalikan root dari pohon Huffman.
    """
    # Inisialisasi min-heap dengan simpul Huffman untuk setiap karakter
    heap = []
    for char, fr in freqs.items():
        node = HuffmanNode(char, fr)
        heap.append(node)
    heapq.heapify(heap)  # membentuk heap dari list

    # Gabungkan simpul hingga tersisa satu (akar)
```

```
    while len(heap) > 1:
        # Pop dua simpul dengan frekuensi terkecil
        node1 = heapq.heappop(heap)  # simpul terkecil pertama
        node2 = heapq.heappop(heap)  # simpul terkecil berikutnya
        # Buat simpul baru dengan freq gabungan
        merged = HuffmanNode(char=None, freq=node1.freq + node2.freq, left=node1, right=node2)
        heapq.heappush(heap, merged)  # masukkan kembali ke heap
    # Akhirnya heap berisi satu elemen yaitu akar pohon Huffman
    return heap[0]

def generate_huffman_codes(node, prefix="", code_dict=None):
    """
    Menelusuri pohon Huffman untuk membentuk kode biner masing-masing karakter.
    `node` adalah root pohon Huffman,
    `prefix` adalah bit code prefiks (rekursif dipassing),
    `code_dict` adalah dict hasil {char: code}.
    """
    if code_dict is None:
        code_dict = {}
    # Jika node daun (karakter tidak None), simpan kode prefix saat ini
    if node.char is not None:
        code_dict[node.char] = prefix if prefix != "" else "0"
        # (Kasus prefix kosong terjadi bila pohon hanya
```

```
punya satu simbol)
        else:
            # Rekursif telusuri anak kiri (tambah "0" pada
prefix)
            generate_huffman_codes(node.left, prefix + "0",
code_dict)
            # Rekursif telusuri anak kanan (tambah "1" pada
prefix)
            generate_huffman_codes(node.right, prefix + "1",
code_dict)
        return code_dict

    # Contoh penggunaan:
    freqs = {'a': 45, 'b': 13, 'c': 12, 'd': 16, 'e': 9, 'f': 5}
    root = build_huffman_tree(freqs)          # membangun
pohon Huffman
    codes = generate_huffman_codes(root)          #
menghasilkan kode Huffman untuk tiap karakter
    print(codes)  # {'a': '0', 'c': '100', 'b': '101', 'f': '1100', 'e':
'1101', 'd': '111'}
```

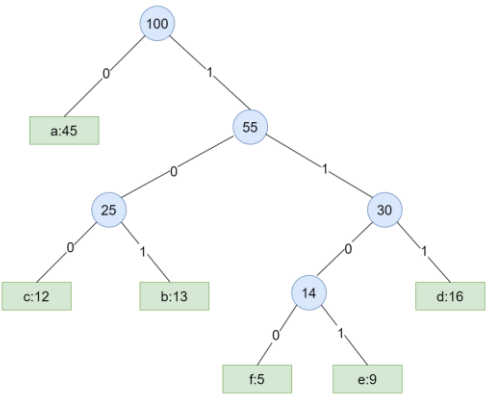
Pada implementasi di atas, kelas HuffmanNode merepresentasikan simpul dalam pohon Huffman, yang dapat berupa simpul daun (mengandung karakter) atau simpul internal (dengan char=None dan memiliki *left*/*right*). Operator kurang dari (__lt__) didefinisikan agar Python dapat membandingkan dua HuffmanNode berdasarkan frekuensinya, sehingga cocok dimasukkan ke dalam heapq (heapq di Python membutuhkan elemen yang dapat dibandingkan).

Fungsi build_huffman_tree membangun pohon Huffman dengan memasukkan semua karakter sebagai node awal ke dalam min-heap, kemudian berulang kali

menggabungkan dua node berfrekuensi terendah. Tiap penggabungan menghasilkan node internal baru dengan frekuensi penjumlahan keduanya, dan node baru ini dimasukkan kembali ke heap. Proses berlanjut hingga hanya satu node yang tersisa di heap, yaitu root dari pohon Huffman lengkap.

Fungsi generate_huffman_codes melakukan penelusuran depth-first pada pohon Huffman yang dihasilkan. Setiap kali turun ke kiri, fungsi menambahkan "0" pada prefix kode, dan ke kanan menambahkan "1". Jika mencapai simpul daun (node dengan karakter tidak None), berarti kita telah membentuk kode biner lengkap untuk karakter tersebut, dan kode disimpan ke code_dict. Dalam contoh di atas, hasil akhirnya disimpan dalam codes, yang berisi pemetaan karakter ke kode bit: misalnya 'a': '0', 'b': '101', dan seterusnya, sesuai dengan pohon Huffman yang terbentuk.

Output kode (mapping karakter ke bit) yang dicetak pada contoh di atas adalah: {'a': '0', 'c': '100', 'b': '101', 'f': '1100', 'e': '1101', 'd': '111'}. Kode ini sepadan dengan pohon Huffman yang telah diilustrasikan sebelumnya, di mana karakter 'a' mendapatkan kode paling pendek (0) karena frekuensinya terbesar, sedangkan 'f' dan 'e' memperoleh kode terpanjang (1100 dan 1101) karena frekuensi mereka paling kecil. Perlu dicatat, kode Huffman selalu memiliki sifat prefix-free, artinya tidak ada kode satu simbol yang merupakan prefiks dari kode simbol lain, sehingga penafsiran bit-bit hasil kompresi tidak ambigu.



Gambar 19.1 Pohon Huffman

Contoh pohon Huffman yang dihasilkan dari frekuensi karakter {a:45, b:13, c:12, d:16, e:9, f:5}. Setiap daun hijau merepresentasikan karakter beserta frekuensinya. Setiap simpul internal biru menunjukkan total frekuensi dari subpohon kiri dan kanan. Misalnya, simpul bertanda 55 berasal dari penggabungan subpohon dengan frekuensi 25 (b+c) dan 30 (d+e+f). Dengan pembacaan bit (kiri=0, kanan=1), didapat kode Huffman: a=0; b=101; c=100; d=111; e=1101; f=1100.

Keoptimalan algoritma Huffman sebagai solusi greedy telah dibuktikan secara matematis (misalnya dengan exchange argument atau induksi). Intuisi optimalitasnya: penggabungan dua simbol terjarang pada tahap awal jelas tidak akan mengorbankan optimalitas, karena dua simbol paling jarang tersebut pada akhirnya harus menempati posisi terdalam di pohon (memiliki bit paling panjang). Menggabungkan mereka lebih awal memastikan keduanya tidak terpisah dalam cabang yang berbeda yang akan menghasilkan panjang kode lebih panjang. Dengan cara greedy seperti ini, Huffman coding menjamin rata-rata panjang kode terpendek untuk distribusi

frekuensi yang diberikan. Algoritma Huffman banyak dipakai dalam berbagai aplikasi kompresi, seperti *for*mat JPEG, MP3, dan algoritma kompresi lainnya, sebagai salah satu tahap enkoding entropi.

Studi kasus Huffman ini menunjukkan bagaimana algoritma greedy dapat diaplikasikan pada pemrosesan data (teks, berkas, dan lain-lain) dalam skala besar untuk mengurangi ukuran penyimpanan. Pada data berukuran sangat besar, algoritma Huffman masih dapat diandalkan karena kompleksitasnya O(n log n). Misalnya, untuk membangun kamus kode Huffman untuk sebuah korpus teks raksasa, kita cukup menghitung frekuensi tiap simbol (misal byte 0-255), lalu menerapkan Huffman. Dengan 256 simbol, proses penggabungan heap dilakukan 255 kali saja, sangat efisien. Untuk data dengan lebih banyak simbol (misal dalam kompresi kata atau frasa), kompleksitas tetap skala linearitmik. Oleh karena itu, Huffman coding tetap relevan dalam era big data sebagai algoritma kompresi dasar sebelum menggunakan teknik kompresi lanjutan.

Studi Kasus Algoritma *Divide-and-Conquer*: *Merge Sort* untuk Pengurutan Data Besar

Salah satu aplikasi paling terkenal dari algoritma *divide-and-conquer* adalah algoritma *Merge Sort* untuk mengurutkan data. Pengurutan (*Sorting*) merupakan operasi pemrosesan data yang sangat umum dan fundamental – mulai dari mengurutkan catatan transaksi, *log* pengguna, hingga data saintifik dalam jumlah besar. Di era big data, mengurutkan dataset berukuran ratusan juta atau milyaran elemen adalah tugas yang menantang. Algoritma *Merge Sort* menyediakan

solusi yang terstruktur dengan kompleksitas optimal O(n log n) untuk masalah ini, dan sangat cocok untuk pemrosesan data dalam skala besar karena mudah diparalelkan dan diimplementasikan dalam lingkungan terdistribusi.

Merge Sort bekerja dengan prinsip *divide-and-conquer*: *array* (atau *list*) yang akan diurutkan dipecah menjadi dua bagian kira-kira sama besar (langkah *divide*), masing-masing bagian diurutkan secara rekursif menggunakan *Merge Sort* (langkah *conquer*), kemudian dua bagian yang telah terurut tersebut digabung (*merge*) menjadi satu *array* terurut penuh (langkah *combine*). Proses pembagian berlanjut hingga unit terkecil, yaitu *array* berukuran 1 (atau 0) yang secara trivial sudah terurut. Langkah penggabungan dua *array* terurut dilakukan dengan cara *merge* dua *list* terurut, yang dapat dilakukan dalam linear time terhadap total ukuran dua *list*.

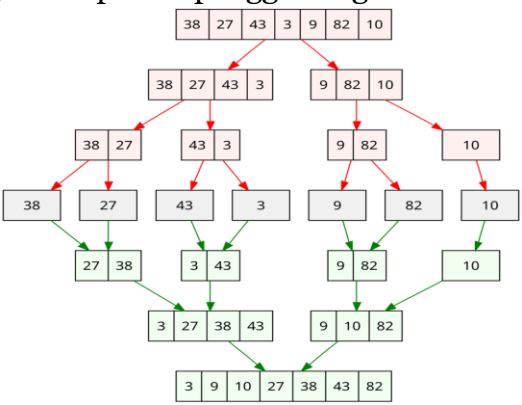
Sebagai ilustrasi, misalkan kita memiliki *array* berikut yang belum terurut: [38, 27, 43, 3, 9, 82, 10]. *Merge Sort* akan memecah *array* ini secara rekursif dan mengurutkan potongan-potongannya:

1. *Array* awal dipecah menjadi dua *Subarray*: [38, 27, 43, 3] dan [9, 82, 10].
2. *Subarray* [38, 27, 43, 3] dipecah lagi menjadi [38, 27] dan [43, 3].
3. *Subarray* [38, 27] dipecah menjadi [38] dan [27] – keduanya ukuran 1 sehingga dianggap terurut. Lalu digabungkan menjadi [27, 38].
4. *Subarray* [43, 3] dipecah menjadi [43] dan [3], kemudian digabungkan terurut menjadi [3, 43].
5. Kini dua hasil [27, 38] dan [3, 43] digabungkan menjadi [3, 27, 38, 43].
6. Sementara itu, *Subarray* [9, 82, 10] dipecah menjadi [9]

dan [82, 10].

7. *Subarray* [82, 10] dipecah menjadi [82] dan [10], kemudian digabungkan menjadi [10, 82].
8. Lalu [9] digabung dengan [10, 82] menjadi [9, 10, 82].
9. Terakhir, dua hasil besar [3, 27, 38, 43] dan [9, 10, 82] digabung menjadi satu *array* terurut penuh: [3, 9, 10, 27, 38, 43, 82].

Proses di atas divisualisasikan pada Gambar berikut, di mana panah merah menunjukkan proses pembagian dan panah hijau menunjukkan proses penggabungan kembali:



Gambar 19.2 Merge Sort algorithm Diagram

Visualisasi algoritma *Merge Sort* dengan pendekatan *divide-and-conquer* pada *array* [38, 27, 43, 3, 9, 82, 10]. Kotak berwarna merah muda menunjukkan *Subarray* yang dipecah menjadi elemen-elemen tunggal (langkah *divide*, panah merah). Kotak hijau menunjukkan hasil penggabungan *Subarray* terurut (langkah *combine*, panah hijau). Hasil akhir di bagian bawah adalah *array* terurut sepenuhnya.

Dari ilustrasi tersebut, terlihat bahwa *Merge Sort* melakukan $\log_2 n$ tingkatan pembagian (untuk n=7, kedalaman sekitar 3 pembagian), dan di setiap tingkatan

melakukan kerja penggabungan total sebesar n (penggabungan setiap elemen tepat sekali per tingkat). Hal ini konsisten dengan kompleksitas $O(n \log n)$ yang dimiliki *Merge Sort*. Kompleksitas tersebut bersifat worst-case maupun average-case, karena *Merge Sort* selalu membagi dua secara seimbang terlepas dari urutan data awal (berbeda dengan *Quick Sort* yang rata-ratanya $O(n \log n)$ tetapi worst-case $O(n^2)$ apabila pivot selalu buruk). *Merge Sort* juga merupakan algoritma stable *Sort*, artinya urutan elemen dengan nilai yang sama akan dipertahankan seperti semula, karena proses penggabungan tidak menukar urutan elemen yang dianggap sama. Stabilitas ini penting dalam beberapa aplikasi, misalnya saat mengurutkan menurut kunci tertentu tetapi mempertahankan urutan semula untuk kunci yang bernilai sama.

Implementasi *Merge Sort* secara rekursif di Python dapat dilakukan dengan cukup ringkas. Berikut kode Python untuk *Merge Sort*, disertai penjelasan per baris:

```
def merge_Sort(arr):
    """Mengurutkan list `arr` menggunakan algoritma Merge Sort."""
    # Base case: jika panjang array 0 atau 1, kembalikan array (sudah terurut)
    if len(arr) <= 1:
        return arr
    # Tentukan titik tengah untuk membagi array
    mid = len(arr) // 2
    # Rekursif: urutkan separuh kiri dan separuh kanan
    left_sorted = merge_Sort(arr[:mid])
    right_sorted = merge_Sort(arr[mid:])
    # Gabungkan dua subhasil terurut
    return merge(left_sorted, right_sorted)
```

```
def merge(left, right):
    """Menggabungkan dua list terurut (`left` dan `right`) menjadi satu list terurut."""
    merged = []
    i = j = 0  # indeks pengiterasian untuk left dan right
    # Iterasi selama masih ada elemen di kedua list
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])      # ambil elemen lebih kecil dari left
            i += 1
        else:
            merged.append(right[j])   # ambil elemen lebih kecil dari right
            j += 1
    # Jika masih ada sisa elemen di salah satu list, tambahkan semuanya
    if i < len(left):
        merged.extend(left[i:])  # tambahkan sisa elemen di left (jika ada)
    if j < len(right):
        merged.extend(right[j:])  # tambahkan sisa elemen di right (jika ada)
    return merged

# Contoh penggunaan:
data = [38, 27, 43, 3, 9, 82, 10]
Sorted_data = merge_Sort(data)
print(Sorted_data)  # Output: [3, 9, 10, 27, 38, 43, 82]
```

Fungsi *merge_Sort* di atas bekerja secara rekursif.

Pertama, ia mengecek *base case*: bila panjang *array* 0 atau 1, ia langsung mengembalikan *array* tersebut (karena *array* kosong atau satu elemen sudah terurut). Jika tidak, *array* dibagi menjadi dua bagian *left* dan *right* berdasarkan indeks tengah. Kemudian, *merge_Sort* memanggil dirinya sendiri untuk bagian kiri dan kanan (dengan ukuran kurang lebih setengah). Setelah mendapatkan *left_sorted* dan *right_sorted* (dua *list* terurut hasil rekursi), langkah terakhir adalah menggabungkan keduanya dengan memanggil fungsi *merge*.

Fungsi *merge*(*left, right*) melakukan penggabungan dua *list* terurut. Variabel indeks i dan j digunakan untuk melintasi *left* dan *right*. Selama masih ada elemen yang belum ditangani di kedua *list*, fungsi membandingkan elemen terkecil yang tersisa (yaitu *left*[i] dan *right*[j]) dan mengambil yang lebih kecil untuk ditempatkan ke *list* hasil *merge*d. Indeks yang elemen dari *list*-nya terambil akan ditingkatkan. Setelah salah satu *list* habis (semua elemennya sudah masuk ke *merge*d), elemen-elemen sisa dari *list* yang lain dapat langsung ditambahkan (karena sisanya sudah pasti lebih besar dari semua elemen sebelumnya dan tetap terurut pada tempatnya). Proses *merge* ini berjalan dalam waktu linear terhadap total panjang *left* + *right*.

Sebagai contoh, untuk data = [38, 27, 43, 3, 9, 82, 10] seperti di atas, pemanggilan *merge_Sort*(data) akan mengembalikan *Sorted*_data = [3, 9, 10, 27, 38, 43, 82]. Hasil ini dicetak pada baris terakhir. Dapat dilihat bahwa algoritma telah berhasil mengurutkan data tersebut. Dalam implementasi nyata, kita mungkin tidak mencetak hasil di tengah proses, namun di sini ditunjukkan *output* akhir untuk memastikan kebenaran algoritma.

Merge Sort dalam hal penggunaan memori

membutuhkan ruang tambahan yang proporsional dengan ukuran n (pada implementasi di atas, saat proses *merge* berlangsung, dibutuhkan *list* tambahan *merge*d yang total maksimal sebesar n, di samping rekursi yang menggunakan *stack* hingga depth log n). Hal ini merupakan trade-off antara waktu dan ruang: *Merge Sort* menggunakan ruang ekstra tetapi menjaga waktu tetap optimal. Pada data skala besar, apabila data tidak muat di memori, *Merge Sort* dapat dimodifikasi menjadi external *Merge Sort*, yakni dengan membagi data menjadi potongan-potongan yang muat di memori, mengurutkan setiap potongan (misal dengan *Merge Sort* juga), lalu menggabungkan potongan-potongan tersebut dari disk. External *Merge Sort* digunakan dalam banyak sistem manajemen basis data dan mesin MapReduce untuk mengurutkan data yang ukurannya lebih besar dari memori utama.

Keunggulan lain *Merge Sort* adalah sifatnya yang deterministik (waktu berjalan tidak tergantung pada data awal, berbeda dengan *Quick Sort* yang per*for*manya bisa sangat baik atau buruk tergantung input/pivot). Dalam konteks pemrosesan paralel, *Merge Sort* juga sederhana untuk diimplementasikan: masing-masing separuh bisa diurutkan secara paralel, dan operasi *merge* terakhir digabungkan. Pada sistem terdistribusi, algoritma serupa digunakan untuk mengurutkan data besar: misalnya Hadoop MapReduce melakukan *Sort*/shuffle data antar node menggunakan prinsip *divide* (split data ke node), *conquer* (urutkan lokal di masing-masing node, sering dengan *Merge Sort* karena dataset per node cukup besar), dan *combine* (menggabungkan *output* terurut dari semua node).

Sebagai catatan, algoritma *Quick Sort* juga merupakan algoritma *divide-and-conquer* untuk *Sorting* yang sangat

terkenal. *Quick Sort* biasanya lebih cepat dalam praktik (konstanta lebih kecil, operasi swap *in-place*, cache-friendly), namun tidak stabil dan memiliki worst-case yang lebih buruk. Implementasi *Quick Sort* melibatkan pemilihan pivot, partisi *array* berdasarkan pivot, dan rekursi pada partisi kiri dan kanan. Dalam konteks big data, *Quick Sort* paralel juga ada, tetapi *Merge Sort* sering lebih disukai karena deterministik dan penggabungan *output* lebih cocok dengan model pemrograman terdistribusi. Banyak pustaka standar (termasuk Python *Sorted*() atau *list.Sort*()) menggunakan varian Tim*Sort*, yang merupakan penggabungan *Merge Sort* dengan algoritma lain, menunjukkan pentingnya konsep penggabungan (*merge*) dalam *Sorting* skala besar.

Efektivitas dan Efisiensi: Perbandingan Greedy vs *Divide-and-conquer*

Setelah meninjau kedua pendekatan diatas beserta studi kasusnya, penting untuk membahas efektivitas dan efisiensi algoritma greedy dan *divide-and-conquer* pada berbagai konteks masalah. Meskipun keduanya bertujuan meningkatkan kinerja pemrosesan data, cara kerja dan kondisi ideal penggunaannya berbeda secara fundamental.

1. Optimalitas Solusi: Algoritma *divide-and-conquer* umumnya dirancang untuk menghasilkan solusi optimal atau tepat untuk masalah yang diselesaikannya (contoh: *Merge Sort* selalu menghasilkan urutan yang benar dan optimal dalam hal terurut). Sementara itu, algoritma greedy bisa jadi hanya menghasilkan solusi mendekati optimal untuk beberapa masalah tertentu apabila syarat optimalitas greedy tidak terpenuhi. Dalam studi kasus

yang dibahas, Huffman coding dengan greedy memberikan solusi optimal (kode prefix paling efisien) karena masalahnya memang memenuhi properti optimal untuk greedy. Namun, jika kita beralih ke masalah lain seperti penjadwalan tugas dengan bobot (weighted scheduling) atau knapsack 0/1, solusi greedy tidak akan optimal – di situ harus digunakan DP atau backtracking. Jadi efektivitas greedy sangat bergantung pada struktur masalah: untuk masalah dengan struktur matroid atau yang memenuhi greedy-choice property, greedy sangat efektif; di luar itu greedy menjadi heuristik. Sebaliknya, *divide-and-conquer* cenderung lebih umum menjamin optimalitas karena ia memecahkan semua sub-masalah secara lengkap. Misalnya, *Binary Search* selalu menemukan posisi yang benar (tepat) dalam *list* terurut, *Quick Sort* selalu menghasilkan *list* terurut (meski waktu bisa bervariasi).

2. Kompleksitas Waktu: Dari perspektif kompleksitas, kedua pendekatan dapat menghasilkan algoritma dengan kompleksitas yang setara tergantung masalahnya. Untuk masalah *Sorting*, baik pendekatan greedy (belum ada greedy murni untuk *Sorting* selain trik seperti bucket *Sort* jika data tertentu) maupun *divide-and-conquer* mencapai O(n log n). Untuk masalah kompresi, algoritma Huffman O(n log n) cukup efisien. Namun, algoritma greedy biasanya memiliki logika sederhana sehingga *overhead* konstanta kecil, sedangkan *divide-and-conquer* melibatkan *overhead* rekursi dan penggabungan. Contohnya, *Merge Sort* dibanding algoritma *selection Sort* (yang notabene bukan greedy, tapi sederhana) memperlihatkan pada input kecil

mungkin *selection Sort* lebih cepat karena *overhead* rekursi *Merge Sort*. Akan tetapi, pada input besar, kompleksitas lebih rendah *Merge Sort* akan mendominasi sehingga lebih cepat. Greedy seringkali bisa diimplementasi dalam bentuk iteratif linear atau n log n tanpa recursion, sehingga lebih ringan *overhead*.
Pada data skala besar, kompleksitas asimtotik menjadi faktor utama. *Divide-and-conquer* unggul pada banyak masalah karena mampu menurunkan kompleksitas drastis (misal dari O(n^2) menjadi O(n log n) atau O(n log n) menjadi O(n)). Namun ada juga kasus di mana greedy memberikan lompatan efisiensi. Contoh: untuk masalah minimum spanning tree, algoritma Kruskal (greedy) atau Prim (greedy) keduanya berjalan O(E log V) dan optimal. Tidak ada algoritma *divide-and-conquer* murni yang mengalahkan itu secara signifikan. Sebaliknya, untuk perkalian bilangan besar, algoritma *divide-and-conquer* (Karatsuba) mengurangi kompleksitas dari O(n^2) menjadi ~O(n^1.58), jauh lebih baik untuk n besar. Ini menunjukkan bahwa pilihan pendekatan sangat tergantung pada karakteristik masalah dan solusi yang diinginkan.

3. Kompleksitas Ruang: Algoritma greedy umumnya menggunakan sedikit memori tambahan di luar input (*in-place*) karena membangun solusi sepotong demi sepotong. Sebaliknya, *divide-and-conquer* kadang memerlukan ruang tambahan. Dalam Huffman coding (greedy), kita membutuhkan struktur data (heap dan tree) seukuran jumlah simbol – ini cukup kecil relatif terhadap data asli yang dikompresi. Dalam *Merge Sort* (*divide-and-conquer*), kita membutuhkan *array*

tambahan sebesar n untuk melakukan *merge*. Pada skala besar, penggunaan memori ekstra bisa menjadi kendala. Namun, sering kali hal ini dapat diatasi: misalnya, dalam implementasi parallel/distribusi, memori ekstra tersebar di banyak mesin. Juga, untuk masalah-masalah yang sangat besar (lebih besar dari RAM), *divide-and-conquer* dapat diimplementasikan secara eksternal (contoh: external *Merge Sort* di disk) sedangkan algoritma greedy kadang sulit jika keputusan lokal perlu informasi global (harus streaming).

4. Paralelisasi dan Skalabilitas: Pada konteks data sangat besar, biasanya kita memanfaatkan komputasi paralel. *Divide-and-conquer* umumnya lebih mudah diparalelkan karena sub-masalah relatif independen. Seperti telah dibahas, *Merge Sort* dapat dipecah pengurutannya di dua core. Banyak algoritma *divide-and-conquer* lain juga demikian (contoh: algoritma FFT memproses rekursi terpisah yang bisa parallel). Sementara itu, algoritma greedy sering bersifat serial: keputusan langkah i biasanya memengaruhi input untuk langkah i+1, sehingga sulit mem-parallel-kan prosesnya. Contohnya, Huffman coding secara alami sulit diparalelkan di tahap konstruksi kode (karena kita butuh global heap), meskipun jika dataset dibagi misal per simbol per mesin, akhirnya tetap perlu digabung secara global. Dalam penjadwalan tugas, algoritma greedy akan menjadwalkan satu per satu; jika kita coba parallel, kita harus hati-hati supaya keputusan tidak konflik. Oleh sebab itu, dalam sistem real-time scheduling, sering diterapkan algoritma greedy yang berjalan iteratif cepat, tapi untuk parallel scheduling biasanya dikombinasikan

dengan heuristik lain atau local search.

5. Kesederhanaan Implementasi dan Pemeliharaan: Algoritma greedy umumnya menang dalam hal sederhana dan jelas. Kode greedy (seperti Huffman di atas, atau Dijkstra, dan lain-lain) relatif mudah diikuti langkahnya karena linier dari awal ke akhir. Sedangkan algoritma *divide-and-conquer*, terutama yang sangat rekursif, bisa lebih sulit di-debug atau diverifikasi kebenarannya, meskipun konsepnya jelas. Dalam praktik pengembangan perangkat lunak, greedy kadang lebih disukai jika kriteria optimalitas tidak ketat, karena implementasi cepat dan kinerjanya cukup. Namun untuk kasus yang memerlukan hasil benar-benar optimal (seperti komputasi ilmiah, perbankan, dsb.), algoritma *divide-and-conquer* atau algoritma pasti lainnya lebih diandalkan.
6. Fleksibilitas terhadap Perubahan Masalah: Ketika spesifikasi masalah berubah, algoritma greedy yang sangat tergantung pada heuristik tertentu mungkin perlu dirancang ulang. *Divide-and-conquer* cenderung lebih fleksibel karena memecah masalah dengan cara umum. Misalnya, jika *for*mat data sedikit berubah, *Merge Sort* tetap bisa jalan; namun sebuah heuristik greedy mungkin tidak langsung cocok dan harus disesuaikan.

Sebagai kesimpulan dari pembahasan efektivitas dan efisiensi: algoritma greedy unggul dalam kesederhanaan dan kecepatan pada masalah-masalah dengan struktur yang sesuai, dan sering dipakai untuk solusi heuristik cepat dalam skenario data besar saat pendekatan pasti terlalu lambat. Algoritma *divide-and-conquer* unggul dalam kemampuan menyelesaikan masalah secara optimal dengan kompleksitas yang terukur dan

dukungan yang baik untuk paralelisasi, menjadikannya tulang punggung bagi banyak algoritma skala besar. Dalam banyak sistem nyata, kedua pendekatan ini bisa saling melengkapi. Sebagai contoh, dalam algoritma pengurutan eksternal, kita mungkin menggunakan *divide-and-conquer* untuk mengurutkan chunk data, lalu menggunakan pendekatan greedy untuk memilih chunk mana yang digabung lebih dahulu tergantung ketersediaan memori/disk. Atau dalam jaringan komputer, protokol routing menggunakan algoritma greedy (seperti hot-potato routing) untuk kecepatan, tetapi memastikan globalnya stabil dengan pembagian zona (*divide-and-conquer* dalam desain jaringan).

Inti yang dapat diambil adalah: pemahaman mendalam tentang sifat masalah akan mengarahkan kita memilih strategi algoritmik yang tepat. Pemrosesan data skala besar membutuhkan algoritma yang tidak hanya cepat secara big-O, tetapi juga efisien dalam implementasi praktis. Greedy dan *divide-and-conquer* adalah dua alat penting dalam perbendaharaan tersebut, dan ahli pemrosesan data sebaiknya mengenali kapan harus menggunakan masing-masing pendekatan untuk mendapatkan hasil terbaik.

BAB 20 PENGENALAN BAHASA PEMROGRAMAN DAN IMPLEMENTASI ALGORITMA

Pendahuluan

Setiap algoritma pada dasarnya hanyalah sebuah rencana logis—langkah demi langkah untuk menyelesaikan masalah tertentu. Namun, selama ide itu belum diwujudkan dalam bentuk kode yang dapat dijalankan, ia tetaplah abstrak. Bahasa pemrograman hadir sebagai jembatan utama antara logika dan eksekusi, antara ide dan solusi nyata.

Bab ini memperkenalkan bagaimana algoritma yang telah dipelajari sebelumnya dapat diterapkan secara langsung menggunakan Python. Dengan sintaks yang bersih dan mendekati *Pseudocode*, Python menjadi alat yang efisien untuk mempraktikkan berbagai algoritma, mulai dari pencarian, pengurutan, hingga pemrosesan data sederhana. Fokus utama bukan lagi pada konsep, tetapi pada bagaimana ide-ide tersebut diubah menjadi instruksi program yang berjalan.

Melalui praktik ini, pembaca diajak untuk tidak hanya memahami struktur kode, tetapi juga mulai membangun kebiasaan menulis program yang rapi, modular, dan mudah dikembangkan. Dengan fondasi ini, Anda akan lebih siap melangkah ke tahap berikutnya dalam dunia pemrograman—apakah itu membangun aplikasi, menganalisis data, atau mengembangkan solusi berbasis teknologi yang lebih kompleks.

Mengenal Bahasa Pemrograman Python

Di antara berbagai bahasa pemrograman yang ada—dari C++ yang kompleks hingga JavaScript yang fleksibel—Python menonjol sebagai bahasa yang ramah bagi pemula sekaligus andal bagi profesional. Dikenal dengan sintaks yang ringkas dan mudah dibaca, Python dirancang agar kode terasa seperti bahasa manusia (Li, 2025). Filosofi ini menjadikannya pilihan utama dalam pendidikan ilmu komputer, serta digunakan luas dalam pengembangan web, data science, kecerdasan buatan, hingga otomasi. Cukup satu baris seperti print("Halo"), Anda sudah bisa mulai menulis program yang berjalan.

Dibandingkan dengan Java yang membutuhkan beberapa baris kode hanya untuk mencetak teks sederhana, Python memungkinkan Anda langsung menulis logika utama tanpa beban sintaks yang rumit. Karena bersifat interpreted, Python mengeksekusi kode baris demi baris, sehingga memudahkan pemula mendeteksi kesalahan sejak awal (Sweigart, 2025). Dukungannya terhadap berbagai paradigma—imperatif, prosedural, hingga fungsional—menjadikannya fleksibel untuk berbagai pendekatan pemrograman.

Sebagai ilustrasi bagaimana Python begitu dekat dengan cara berpikir algoritmik, perhatikan contoh berikut, kita ingin menentukan bilangan terbesar dari tiga nilai. Dalam Python, kita dapat menulis:

```
a = 12
b = 9
c = 17

terbesar = max(a, b, c)
print("Bilangan terbesar adalah:", terbesar)
```

Kode Python bersifat jelas, ringkas, dan mudah dipahami—bahkan oleh mereka yang belum pernah menulis program sebelumnya. Fungsi-fungsi bawaan seperti max() menunjukkan betapa praktis dan siap pakainya bahasa ini, bahkan untuk kebutuhan dasar. Lebih dari sekadar sintaks, Python membentuk cara berpikir yang sistematis dan elegan, menjadikannya alat yang ideal untuk mewujudkan algoritma menjadi program yang berjalan dan bermanfaat.

Python vs Bahasa Lain

Python dikenal dengan sintaks yang bersih dan menyerupai bahasa manusia, menjadikannya pilihan utama di banyak universitas, perusahaan teknologi, dan komunitas pemula. Meski demikian, Python bukan satu-satunya bahasa yang digunakan dalam pengembangan perangkat lunak. Bahasa seperti C++, Java, dan JavaScript juga memainkan peran penting. Untuk memberikan perspektif yang lebih luas, bagian berikut menyajikan perbandingan sintaks dasar antara Python dan beberapa bahasa pemrograman *pop*uler lainnya.

1. Struktur Percabangan: if Statement

Bahasa	**Sintaks**
Python	`if x > 0:\n print("Positif")`
Java	`if (x > 0) {\n System.out.println("Positif");\n}`
C++	`if (x > 0) {\n cout << "Positif";\n}`

   Pada Python, tanda kurung kurawal {} digantikan dengan indentasi (tab/spasi), yang membuat kode lebih rapi dan terbaca. Tidak ada tanda titik koma ; dan ekspresi logika tidak perlu dibungkus dalam tanda kurung () kecuali untuk kejelasan.

2. Struktur Perulangan: *for Loop*

Bahasa	Sintaks
Python	`for i in range(5):\n print(i)`
Java	`for (int i = 0; i < 5; i++) {\n System.out.println(i);\n}`
C++	`for (int i = 0; i < 5; i++) {\n cout << i;\n}`

Python menggunakan perulangan *for* berbasis koleksi atau urutan (*range, list*, dsb.), yang lebih intuitif dibandingkan struktur *for* tradisional dengan tiga parameter. Hal ini membuat Python sangat cocok untuk manipulasi data atau struktur iteratif.

3. Fungsi dan Pemanggilannya

Bahasa	Sintaks
Python	`def sapa():\n print("Halo")\nsapa()`
Java	`void sapa() {\n System.out.println("Halo");\n}` *(dalam class)*
C++	`void sapa() {\n cout << "Halo";\n}`

Membuat fungsi di Python sangat sederhana. Tidak diperlukan deklarasi tipe kembalian (*void, int*, dsb.), dan tidak wajib menggunakan struktur kelas jika tidak dibutuhkan. Ini sangat berguna dalam tahap awal pembelajaran algoritma.

Python menawarkan sintaks yang ringkas, jelas, dan sejalan dengan cara berpikir manusia. Hal ini memungkinkan pembelajar untuk lebih fokus pada logika algoritmik tanpa terjebak dalam kompleksitas teknis bahasa. Meski demikian, pemahaman lintas bahasa tetap penting sebagai bekal beradaptasi di dunia pengembangan perangkat lunak yang beragam. Bab ini tidak dimaksudkan untuk memberi gambaran bahwa Python adalah titik awal yang kuat dan dengan menguasainya, Anda akan lebih mudah memahami bahasa pemrograman lain di kemudian hari.

Menerjemahkan Algoritma ke Python

Setelah memahami sintaks dasar Python, langkah berikutnya adalah menerjemahkan algoritma yang telah disusun secara logis—baik dalam narasi maupun *Pseudocode*—ke dalam kode yang dapat dijalankan. Proses ini dikenal sebagai translasi algoritma, dan merupakan keterampilan inti dalam pemrograman. Python sangat ideal untuk tahap ini karena sintaksnya menyerupai *Pseudocode*, sehingga proses konversi dari logika ke instruksi program menjadi lebih mulus dan intuitif.

1. Contoh 1: Mencari nilai maksimum dari tiga bilangan
   a. *Pseudocode*

```
Mulai
  Masukkan tiga bilangan: a, b, c
  Jika a > b dan a > c maka
      maksimum = a
  Jika b > a dan b > c maka
      maksimum = b
  Jika tidak maka
      maksimum = c
  Tampilkan maksimum
Selesai
```

   b. Implementasi dalam Python

```
a = int(input("Masukkan bilangan pertama: "))
b = int(input("Masukkan bilangan kedua: "))
c = int(input("Masukkan bilangan ketiga: "))

if a > b and a > c:
    maksimum = a
elif b > a and b > c:
    maksimum = b
else:
    maksimum = c
print("Bilangan terbesar adalah:", maksimum)
```

Kita bisa melihat bahwa struktur logika dari *Pseudocode* tetap dipertahankan. Python hanya membutuhkan penyesuaian pada sintaks seperti penggunaan *if, elif*, dan *else*, serta tanda titik dua (:) dan indentasi untuk menandai blok kode.

2. Contoh 2: Menghitung Faktorial (Versi Iteratif dan Rekursif)
    a. Versi Iteratif
        1) *Pseudocode*

```
Masukkan nilai n
Set hasil = 1
Untuk i dari 1 sampai n:
    hasil = hasil * i
Tampilkan hasil
```

        2) Python

```
n = int(input("Masukkan angka: "))
hasil = 1

for i in range(1, n + 1):
    hasil *= i

print("Faktorial dari", n, "adalah",
hasil)
```

    b. Versi Rekursif
        1) *Pseudocode*

```
Fungsi faktorial(n):
    Jika n == 0 atau n == 1:
        Kembalikan 1
    Jika tidak:
        Kembalikan n * faktorial(n -
1)
```

        2) Python

```
def faktorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * faktorial(n - 1)
```

```
    n = int(input("Masukkan angka: "))
    print("Faktorial dari", n, "adalah",
faktorial(n))
```

3. Pola Translasi Umum

   Agar proses translasi lebih sistematis, berikut pola umum yang bisa Anda ikuti:

   a. Identifikasi input – unakan input() atau parameter fungsi.
   b. Definisikan variabel awal – Tetapkan nilai *default* jika perlu.
   c. Gunakan struktur kontrol – *if, for, while* sesuai kebutuhan algoritma.
   d. Tulis logika sesuai urutan langkah – Ikuti *Pseudocode* dengan runut.
   e. Tampilkan hasil akhir – Gunakan print() untuk mencetak ke layar.

Dengan berlatih menerjemahkan algoritma menjadi kode Python, Anda tidak hanya mengembangkan keterampilan teknis, tetapi juga membiasakan diri dengan cara berpikir sistematis yang menjadi fondasi dalam pemrograman tingkat lanjut.

Implementasi Algoritma *Pop*uler dalam Python

Pengurutan dan pencarian adalah dua jenis algoritma dasar yang telah dibahas secara konseptual pada bab-bab sebelumnya. Keduanya sering digunakan dalam berbagai proses pengolahan data (Bhasin, 2023). Dalam bagian ini, fokus kita adalah mengimplementasikan algoritma tersebut menggunakan Python—tanpa mengulas kembali teori dasarnya. Sebagai pelengkap, satu algoritma tambahan yang sederhana namun relevan dalam berbagai konteks juga disajikan. Dengan

pendekatan praktis, bagian ini menunjukkan bagaimana ide-ide algoritmik diwujudkan menjadi kode Python yang bersih, efisien, dan mudah dijalankan.

1. *Binary Search*

   Binary Search merupakan metode pencarian yang efisien untuk data yang sudah terurut. Berbeda dengan *Linear Search* yang memeriksa elemen satu per satu, algoritma ini membagi ruang pencarian menjadi dua di setiap langkah, sehingga jauh lebih cepat untuk dataset berukuran besar (Ren & Li, 2025). Berikut implementasi versi iteratifnya dalam Python:

```
def binary_search(data, target):
    awal = 0
    akhir = len(data) - 1

    while awal <= akhir:
        tengah = (awal + akhir) // 2
        if data[tengah] == target:
            return tengah
        elif data[tengah] < target:
            awal = tengah + 1
        else:
            akhir = tengah - 1

    return -1

# Contoh penggunaan
angka = [1, 3, 5, 7, 9, 11]
print("Hasil:", binary_search(angka, 7))
```

2. *Bubble Sort*

   Bubble Sort adalah salah satu algoritma pengurutan paling sederhana. Meski kurang efisien untuk dataset besar, algoritma ini sering digunakan sebagai pengantar karena logikanya mudah dipahami dan memperlihatkan proses pertukaran elemen secara eksplisit. Berikut implementasinya dalam Python:

```
def bubble_Sort(data):
    n = len(data)
    for i in range(n):
        for j in range(0, n - i - 1):
            if data[j] > data[j + 1]:
               data[j],data[j+1] = data[j+1],data[j]

# Contoh penggunaan
angka = [64, 34, 25, 12, 22, 11, 90]
bubble_Sort(angka)
print("Hasil:", angka)
```

3. *Insertion Sort*

   Insertion Sort bekerja seperti seseorang yang menyusun kartu di tangan—mengambil satu per satu dan menempatkannya di posisi yang tepat dalam kumpulan yang telah terurut. Algoritma ini relatif efisien untuk dataset kecil atau hampir terurut. Berikut implementasi dalam Python:

```
def insertion_Sort(data):
    for i in range(1, len(data)):
        elemen_kunci = data[i]
        j = i - 1
        while j >= 0 and data[j] > elemen_kunci:
            data[j + 1] = data[j]
            j -= 1
        data[j + 1] = elemen_kunci

# Contoh penggunaan
angka = [12, 11, 13, 5, 6]
insertion_Sort(angka)
print("Hasil:", angka)
```

4. *Palindrome Checker*

   Algoritma ini *pop*uler karena sering digunakan dalam pemeriksaan teks, validasi input, dan tantangan logika sederhana. *Palindrom* adalah kata atau kalimat yang terbaca sama dari depan maupun belakang, seperti

“kasur rusak” atau “level”. Berikut implementasinya dalam Python:

```
def is_palindrome(text):
    text = text.lower().replace(" ", "")
    return text == text[::-1]

# Contoh penggunaan
kata = input("Masukkan kata/kalimat: ")
if is_palindrome(kata):
    print("Palindrome!")
else:
    print("Bukan palindrome.")
```

Dengan mempraktikkan implementasi berbagai algoritma ini, anda tidak hanya memahami bagaimana sebuah logika dijalankan oleh komputer, tetapi juga mulai mengembangkan kepekaan terhadap efisiensi kode dan gaya penulisan yang baik. Python memberikan kemudahan untuk mengekspresikan algoritma dengan cara yang bersih dan intuitif, menjadikannya alat yang ideal untuk tahap awal penguasaan pemrograman.

Praktik Baik dalam Pengkodean Algoritma dengan Python

Menulis algoritma yang benar baru setengah dari praktik pemrograman yang baik. Sisanya bergantung pada bagaimana kode ditulis—apakah rapi, mudah dipahami, dan siap dikembangkan lebih lanjut. Dalam praktik nyata, program tidak hanya dijalankan oleh mesin, tetapi juga dibaca dan dimodifikasi oleh manusia. Untuk itu, berikut beberapa prinsip dasar yang perlu diperhatikan saat mengimplementasikan algoritma dalam Python.

1. Penamaan yang Deskriptif

   Gunakan nama variabel dan fungsi yang jelas dan

mencerminkan isi atau fungsinya. Hindari nama yang terlalu singkat seperti a, b, atau x kecuali dalam konteks yang sangat lokal dan sementara.

Contoh kurang baik:

```
def f(x):
    return x*x + 2*x + 1
```

Contoh Baik:

```
def hitung_kuadrat_bergeser(nilai):
    return nilai * nilai + 2 * nilai + 1
```

2. Komentar seperlunya

   Komentar membantu menjelaskan bagian-bagian penting dalam kode, terutama logika yang tidak langsung terlihat dari baris program. Namun, komentar yang berlebihan justru dapat mengganggu keterbacaan.

   Contoh baik:

```
# Menentukan bilangan terbesar dari tiga angka
maksimum = max(a, b, c)
```

3. Struktur Modular

   Pisahkan bagian logika ke dalam fungsi-fungsi kecil yang melakukan tugas spesifik. Ini memudahkan debugging, pengujian, dan pemeliharaan kode.

```
def cari_maksimum(a, b, c):
    return max(a, b, c)

def main():
    a = int(input("Masukkan bilangan pertama: "))
    b = int(input("Masukkan bilangan kedua: "))
    c = int(input("Masukkan bilangan ketiga: "))
    print("Bil terbesar:",cari_maksimum(a,b,c))
main()
```

4. Manfaatkan Fitur Python secara Bijak

   Python memiliki banyak fitur bawaan yang mempercepat pengembangan, seperti *slicing*, *list comprehension*, dan pustaka standar (*math, collections*, dll). Gunakan fitur ini untuk menyederhanakan logika, namun tetap

perhatikan keterbacaan.

Dengan membiasakan diri menulis algoritma secara terstruktur dan bersih, Anda akan menjadi *programer* yang efisien. Python sebagai bahasa yang ekspresif memberi ruang besar untuk menulis algoritma dengan cara yang elegan. Tinggal bagaimana kita menggunakannya dengan baik.

Bab ini memperlihatkan bagaimana konsep-konsep algoritmik yang telah dipelajari sebelumnya dapat diimplementasikan secara nyata menggunakan Python. Dengan sintaks yang sederhana dan ekspresif, Python menjadi alat yang efektif untuk menerjemahkan logika menjadi program yang berjalan. Lebih dari sekadar memahami sintaks, menguasai pemrograman berarti membentuk cara berpikir yang sistematis, mampu memecah masalah, dan menyusunnya kembali menjadi solusi. Python memberikan fondasi yang kuat untuk memulai perjalanan ini—dari latihan dasar hingga pengembangan aplikasi yang lebih kompleks. Sisanya bergantung pada kemauan untuk terus mencoba, memperbaiki, dan belajar sepanjang proses.

Daftar Pustaka

Abdillah, A. (2014). Inovasi dan Pengembangan Produk UKM Handikraf untuk Pasar Pariwisata di Bali. Program Studi Pariwisata, Jurusan Administrasi Bisnis FIA UB.

Adiningsih, S. (2003). The Indonesia Business Rop in AFTA, Indonesia Business Perspective, Volume V, No. 3, PT. Harvest International Indonesia.

Afiff, F. (2012). Pilar-Pilar Ekonomi Kreatif. Artikel. Universitas Bina Nusantara: Jakarta.

Afrizal Zein & Chrisantus Tristianto. (2025). Algoritma dan Struktur Data Menggunakan Pemogramman Python. EUREKA MEDIA AKSARA.

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Boston, MA: Addison Wesley.

Alfina, Adi Ahmad, Yeni Yanti, & M. (2024). Algoritma dan Pemrograman. In Algorima dan Pemrograman (Issue 1). CV. Naskah Aceh.

Alotaibi, H., & Alotaibi, M. (2020). The effectiveness of *flowchart*s in enhancing algorithmic thinking among computer science students. International Journal of Emerging Technologies in Learning (iJET), 15(11), 4-18. https://doi.org/10.3991/ijet.v15i11.13567

Barakbah, A. R., Karlita, T., & Ahsan, A. S. (2013). Logika dan Algoritma. Politeknik Elektronika Negeri Surabaya.

Bhasin, H. (2023). Python Programming Using Problem Solving. Mercury Learning and In*for*mation.

Bhasin, H., 2018. Python basics: a self-teaching Introduction. Mercury Learning and In*for*mation.

Bokare, M. M., & Suryawanshi, A. (2024). C Language Foundation Textbook *for* BCA UG Course. Sankalp Publication.

Caferra, R. (2011). Logic *for* Computer Science and Artificial Intelligence. John Wiley & Sons, Inc. ISBN 978-1-84821-301-2.

Canning, J., Broder, A., & La*for*e, R. (2023). Data Structures & Algorithms in Python.

Chauhan, Y., & Duggal, A. (2020). Different *Sorting* algorithms comparison based upon the time complexity. International Journal of Research and Analytical Reviews (IJRAR), 7(3), 114–118. https://www.researchgate.net/publication/344280789_Different_*Sorting*_Algorithms_comparison_based_upon_the_Time_Complexity

Chen, Y., & Wang, L. (2021). *Pseudocode* as a pedagogical tool in introductory programming education. Education and In*for*mation Technologies, 26(3), 2873-2890. https://doi.org/10.1007/s10639-020-10391-2

Cielen, D., Meysman, A. D. B., & Ali, M. (2016). Introduction to data science. Manning Shelter Island.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). Cambridge, MA: MIT Press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.

D'Souza, D., & Reddy, R. (2020). A beginner's approach to control structures in programming languages. International Journal of Scientific Research in Computer Science, Engineering and In*for*mation Technology, 6(3), 245–251.

Dagiene, V., & Stupuriene, G. (2020). Bebras challenge and algorithmic thinking development. Olympiads in In*for*matics, 14, 21-34. https://doi.org/10.15388/ioi.2020.03

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design and Implementation, 137–150.

Deitel, P. J., & Deitel, H. M. (2019). C How to Program (10th ed.). Pearson Education.

Dicoding, 2025. Academi: Memulai Pemrograman dengan Python.https://www.dicoding.com/academies/86/tutorials/4769

Downey, A. B. (2020). Think Python: How to Think Like a Computer Scientist (2nd ed.). O'Reilly Media.

Elmqvist, N., & Irani, P. (2021). Visualizing algorithms: A survey of tools and techniques. IEEE Transactions on Visualization and Computer Graphics, 27(6), 2985-3001. https://doi.org/10.1109/TVCG.2020.2975790

Endres, M., Weimer, W., & Kamil, A. (2021). An Analysis of Iterative and Recursive Problem Per*for*mance. SIGCSE 2021 - Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, 321–327. https://doi.org/10.1145/3408877.3432391

Ennis, R. H. (2011). The nature of critical thinking: An outline of critical thinking dispositions and abilities. University of Illinois. Retrieved from http://faculty.education.illinois.edu/rhennis/documents/TheNatureofCriticalThinking_51711_000.pdf

Fathansyah, R. (2007). Struktur Data dan Algoritma dengan Bahasa C. Bandung: In*for*matika.

*For*ouzan, B. A. (2017). Foundations of Computer Science (4th ed.). Cengage Learning.

Futschek, G., & Moschitz, J. (2022). Teaching computational thinking with *flowchart*s and *Pseudocode*: A classroom experiment. ACM Transactions on Computing Education, 22(4), 1-22. https://doi.org/10.1145/3517131

Gellenbeck, E., & McConnell, J. (2020). Using *Pseudocode* to bridge the gap between problem-solving and coding. Journal of Computing Sciences in Colleges, 35(6), 92-100.

Ginting, D. (2023). Modul Perkuliahan: Mekanika Komputasi - Operator Logika dan Relasional. Jakarta: Program Studi Magister Teknik Mesin, Universitas Mercu Buana.

Ginting, D. (2023). Modul Perkuliahan: Mekanika Komputasi - Variable dan *Array*. Jakarta: Program Studi Magister Teknik Mesin, Universitas Mercu Buana.

Goel, K., Dwivedi, P., & Sharma, O. (2023). Per*for*mance analysis of various *Sorting* algorithms: Comparison and optimization. 2023 11th International Conference on Intelligent Systems and Embedded Design (ISED), 1–5. https://doi.org/10.1109/ISED56462.2023.10444609

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2022). Data structures & algorithms in Java (6th ed.). Wiley.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H., 2014. Data Structures and Algorithms in Java™. 6th Edition. Wiley.

Govier, T. (2010). A practical study of argument (7th ed.). Wadsworth Cengage Learning.

Grami, A. (2023). Algorithms. In Discrete Mathematics (pp. 211–230). *Else*vier. https://doi.org/10.1016/B978-0-12-820656-0.00012-5

Guerrini, V., & Pisanti, N. (2024). Algorithms Foundations. In Reference Module in Life Sciences. *Else*vier. https://doi.org/10.1016/B978-0-323-95502-7.00008-7

Halimatussya'diyah Purba, & Yahfizham Yahfizham. (2023). Konsep Dasar Pemahaman Algoritma Pemrograman. Jurnal Arjuna: Publikasi Ilmu Pendidikan, Bahasa Dan Matematika, 1(6), 290–301. https://doi.org/10.61132/arjuna.v1i6.356

Hazzan, O., & Lapidot, T. (2021). Guide to teaching computer science: An activity-based approach (2nd ed.). Springer. https://doi.org/10.1007/978-3-030-39360-1

Herlambang, S. (2018). Implementasi Fungsi Rekursif Dalam Algoritma dan Perbandingannya dengan Fungsi Iteratif. 6. https://in*for*matika.stei.itb.ac.id/~rinaldi.munir/Matdis/2008-2009/Makalah2008/Makalah0809-079.pdf

Hidayat, A., Agustin, N., Misriati, T., Prayudani, S., Sibyan, H., Aryanti, R., Hidayat, R., Iswanto, M. E., Taufiqurrochman, M., Zein, A. A., Lase, Y. Y., Erlinda, S.,

& Wilda, A. N. (2024). Struktur data menggunakan bahasa pemrograman C++ (T*undo*, Ed.). Malang: Future Science.
Hoare, C. A. R. (1962). Quick*Sort*. The Computer Journal, 5(1), 10–16. (contoh algoritma *divide-and-conquer* untuk *Sorting*)
https://www.geeks*for*geeks.org/
https://www.khanacademy.org/computing/computer-science/algorithms
https://www.tutorialspoint.com/
Huffman, D. A. (1952). A Method *for* the Construction of Minimum-Redundancy Codes. Proceedings of the IRE, 40(9), 1098-1101.
Hurley, P. J. (2015). A concise introduction to logic (12th ed.). Cengage Learning.
Izadkhah, H. (2022). Problems on Algorithms: A Comprehensive Exercise Book *for* Students in Software Engineering. Germany: Springer International Publishing.
Jay Wengrow,2017. "A Common-Sense Guide to Data Structures and Algorithms: Level Up Your Core Programming Skills", Pragmatic Bookshelf; 1 edition.
Jon Kleinberg, Éva Tardos, 2006. "Algorithm Design", 1st Edition, Pearson; 1 edition (March 26, 2005).
Kadir, A. (2012). Algoritma & Pemrograman menggunakan C & C++ (1 ed., Vols. -). (B. R. W, Ed., & -, Trans.) Yogyakarta: ANDI.
Kadir, A. (2012). Algortima & pemrograman menggunakan Java (1 ed., Vols. -). (F. S. Suyantoro, Ed., & -, Trans.) Yogyakarta: ANDI.
Kadir, A. (2020). Dasar Logika Pemrograman Komputer (Edisi Terbaru). Elex Media Komputindo.
Karumanchi, N. (2016). Data structures and algorithmic thinking with
Karumanchi, N., 2016. Data Structure and Alorithmic Thinking with Python. CareerMonk Publications.

Karumanchi, N., 2017. Data Structures And Algorithms Made Easy. CareerMonk Publications.
Katai, Z., & Toth, L. (2023). Enhancing programming skills with visual tools: The role of *flowchart*s. Computers & Education, 194, 104681. https://doi.org/10.1016/j.compedu.2022.104681
Kleinberg, J., & Tardos, É. (2006). Algorithm Design. Addison-Wesley. (Greedy algorithms chapter)
Knuth, D. E. (1997). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Reading, MA: Addison Wesley Longman.
Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: *Sorting* and Searching (2nd ed.). Addison-Wesley.
Kumar, R., & Sharma, S. (2021). Analyzing the learning curve of programming control structures among beginners. Journal of Computer Applications, 44(2), 112–119.
Laaksonen, A. (2017). Guide to competitive programming: Learning and improving algorithms through contests. Springer.
Li, T. (2025). Research on the Application of Python in Big Data Analysis. Journal of Artificial Intelligence and In*for*mation, 2, 37–41.
Liang, Y. D. (2018). Introduction to Java Programming and Data Structures (11th ed.). Pearson.
Loukanova, R., Lumsdaine, P. L., & Muskens, R. (2021). Logic and Algorithms in Computational Linguistics. Springer. Studies in Computational Intelligence. ISBN 978-3-031-21779-1.
Lutfina, E., & Ramadhan, F. L. (2019). Perbandingan Kinerja Metode Iteratif dan Metode Rekursif dalam Algoritma *Binary Search*. Seminar Nasional APTIKOM, 2019.
Luxton-Reilly, A., & Denny, P. (2020). The impact of *Pseudocode* on novice *programmer*s' understanding of algorithms. Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science

Education, 568-574. https://doi.org/10.1145/3341525.3387409
Malik, D. S. (2017). C++ Programming: From Problem Analysis to Program Design (8th ed.). Cengage Learning.
Malik, D.S., 2018. “Pengantar Algoritma dan Pemrograman”, Boston: Cengage Learning.
Malik, K. (2020). Logika dan Algoritma Pemrograman Komputer. Pustaka Nurja.
Malmi, L., & Korhonen, A. (2021). Visual algorithm simulation: A review of tools and their educational impact. ACM Transactions on Computing Education, 21(2), 1-28. https://doi.org/10.1145/3445989
McCracken, M., & Waters, R. (2022). Data structures and algorithms using Python (3rd ed.). Pearson Education.
Mirkowska, G., & Salwicki, A. (1987). Algorithmic Logic. D. Reidel Publishing Company. ISBN 90-277-1928-4.
Mitchell, J. C. (2002). Concepts in Programming Languages. Cambridge University Press.
Moore, B. N., & Parker, R. (2012). Critical thinking (10th ed.). McGraw-Hill Education.
Munir Rinaldi, 2005. “Algoritma dan Pemrograman dalam Bahasa Pascal dan C”, In*for*matika Bandung.
Nugroho Adi, 2009, “Algoritma dan struktur data dengan C#”, Pustaka ID, (1st edition).
Nugroho, A., & Kurniawan, F. (2022). Optimasi pencarian pada basis data terdistribusi. Jurnal Teknologi Informasi dan Komunikasi, 10(1), 45–56. https://doi.org/10.1234/jtik.v10i1.12345
Nyhoff, J. L., & Nyhoff, L. R. (2017). Processing: An introduction to programming. CRC Press.
Prasetya, R. A., & Wijaya, A. (2023). Penerapan algoritma *Binary Search* untuk pengindeksan dokumen elektronik. Jurnal In*for*matika dan Sistem Informasi, 9(3), 212–224. https://doi.org/10.5678/jisi.v9i3.67890

Prim, R. C. (1957). Shortest Connection Networks And Some Generalizations. The Bell System Technical Journal, 36(6), 1389–1401. (contoh algoritma greedy pada graf)
Putri, M. P., Barovih, G., Azdy, R. A., Yuniansyah, Saputra, A., Sriyeni, Y., Rini, A., & Admojo, F. T. (2022). Algoritma dan Struktur Data. Bandung: Widina Bhakti Persada.
Python. Careet Monk Publications.
Raharjo, B., & Santoso, D. (2021). Algoritma dan struktur data. Andi Offset.
Raharjo, B., & Santoso, D. (2021). Algoritma dan Struktur Data. Yogyakarta: Andi Offset.
Ramalho, L. (2015). Fluent Python. Sebas*topo*l, CA: O'Reilly Media.
Ren, J., & Li, A. (2025). *Binary Search*. In Silicon Valley Python Engineer Interview Guide: Data Structure, Algorithm, and System Design (pp. 215–225). Springer.
Ren, J., & Li, A. (2025). Data Structure, Algorithm, and System Design. Springer Nature Singapore. https://doi.org/10.1007/978-981-96-3201-5
Rizvi, Q. M., Rai, H., & Jaiswal, R. (2024). *Sorting* algorithms in focus: A critical examination of *Sorting* algorithm per*for*mance. International Conference on Emerging Trends in IoT & Computing Technologies-2023, 1–5. https://www.researchgate.net/publication/378962637_*Sorting*_Algorithms_in_Focus_A_Critical_Examination_of_*Sorting*_Algorithm_Per*for*mance
Robins, A., & Rountree, J. (2023). Learning to program: The role of scaffolding with *Pseudocode* and *flowchart*s. Computer Science Education, 33(1), 45-67. https://doi.org/10.1080/08993408.2022.2034567
Rosa, A. S. (2018). Logika Algoritma dan Pemrograman Dasar. Modula.
Sahu, U. (2024). Comparison among *selection Sort*, *Bubble Sort*, and *Insertion Sort*. Interview Kickstart Blog. Retrieved from

https://interviewkickstart.com/blogs/learn/comparison-among-bubble-*Sort*-*selection*-*Sort*-and-*insert*ion-*Sort*
Salmon, M. H. (2012). Introduction to logic and critical thinking (6th ed.). Cengage Learning.
Schäfer, A. (2023). Efficient algorithms *for* big data analytics. Springer.
Sebesta, R. W. (2018). Concepts of Programming Languages (12th ed.). Pearson.
Sebesta, R. W. (2019). Concepts of Programming Languages (12th ed.). Pearson.
Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Boston, MA: Addison-Wesley Professional.
Sedgewick, R., & Wayne, K. (2020). Algorithms (4th ed.). Addison-Wesley.
Sedgewick, R., & Wayne, K. (2022). Algorithms (5th ed.). Addison-Wesley Professional.
Sharma, P. B. (2025). Principles Of Programming Language Paradigms. OrangeBooks Publication.
Soni, M. (2024). Mastering Algorithms & Data Structures.
Steven S S. Skiena, 2010. "The Algorithm Design Manual 2nd" Springer;
Sweigart, A. (2025). Automate the Boring Stuff with Python. No Starch Press.
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cli*for*d Stein, 2022. "Introduction to Algorithms", The MIT Press, (5th Edition).
Vierdansyah, D., Sihombing, R., & Damanik, M. (2023). Perbandingan Analisis *Bubble Sort* dan *Insertion Sort* pada Pengurutan Data Sewa Kios. COELITE: Journal of Computer Engineering, Electronics and In*for*mation Technology, 2(1), 1–6. https://doi.org/10.17509/coelite.v2i1.57091
Vihavainen, A., & Vikberg, T. (2020). Algorithm visualization and its effect on learning outcomes in programming courses. Journal of Educational Technology Systems,

48(4), 492-510. https://doi.org/10.1177/0047239520904701
Vishwakarma, A., Agrwal, H., Thakur, S. S., Bairagi, T., Dahariya, Y., Mahilang, H., Gohil, C., & Kumar, P. K. (2024). Recursion Algorithms: Principles, Applications, And Innovations. 04, 8263–8270.
Wahyono, T. (2016). Struktur Data: Konsep dan Implementasi dalam Bahasa C/C++. Yogyakarta: Andi.
Weiss, M. A., Data structures and algorithm analysis in C++, 4th Edition. Pearson.
Wengrow, J., 2020. A Common-Sense Guide to Data Structures and Algorithms. 2nd Edition. The Pragmatic *Programmer*s, LLC.
Wirth, N. (1976). Algorithms + Data Structures = Programs. Englewood Cliffs, NJ: Prentice-Hall.
Witten, I. H., Frank, E., Hall, M. A., & Pal, C. J. (2017). Algorithms. In Data Mining (pp. 91–160). *Else*vier. https://doi.org/10.1016/B978-0-12-804291-5.00004-0
Xtracts, C. (2023). PHP Basics - A Brief Guide. Mocktime Publication.
Yanti, F., & Eriana, E. S. (2024). Algoritma *Sorting* dan searching. Purbalingga: CV. Eureka Media Aksara.
Zelle, J. M. (2016). Python Programming: An Introduction to Computer Science (3rd ed.). Franklin, Beedle & Associates.
Zhang, Y., & Li, X. (2021). A comparative analysis of search algorithms: Linear vs *Binary Search* in large-scale data. Journal of Computer Science, 15(2), 105–118. https://doi.org/10.1016/j.jcs.2021.01.005

Tentang Penulis

M. Rhifky Wayahdi, lahir di Medan, 05 Februari 1993, merupakan anak pertama dari tiga bersaudara. Penulis merupakan alumni Program Sarjana (S-1) di Universitas Potensi Utama pada Jurusan Sistem Informasi dan lulus tahun 2015. Penulis melanjutkan studi Program Magister (S-2) Teknik In*for*matika di Universitas Sumatera Utara dan lulus tahun 2019. Kemudian saat ini Penulis sedang melanjutkan Pendidikan Doktoral (S-3) Ilmu Komputer di Universitas Sumatera Utara mulai 2024 sampai sekarang (on-going). Berkarir sebagai dosen dimulai dari tahun 2020 di Universitas Battuta. Mulai tahun 2023-sekarang diberi kepercayaan menjadi Dekan Fakultas Teknologi di Universitas Battuta.

Phie Chyan, ST, M.Cs, Lahir di Makassar 13 April 1981, Setelah menyelesaikan pendidikan menengah di SMA Katholik Cendrawasih Makassar Tahun 1996. Penulis melanjutkan pendidikan tinggi S1 di Universitas Atma Jaya Makassar program studi Teknik Elektro Kemudian S2 di Universitas Gadjah Mada Program Studi Ilmu Komputer dan Doktor (S3) di Universitas Hasannudin di Program Studi Elektro konsetrasi Teknik In*for*matika. Buku ini merupakan salah satu karya dari penulis sesuai bidang minat dan ilmu dari penulis dengan tujuan untuk berbagi ilmu pengetahuan kepada masyarakat.

Agung Yuliyanto Nugroho, M.Kom., M.Par Ketertarikan penulis terhadap ilmu komputer dimulai pada tahun 2015 silam. Hal tersebut membuat penulis melanjutkan pendidikan ke Perguruan Tinggi dan berhasil menyelesaikan studi S1 di prodi Teknik In*for*matika Universitas Teknologi Yogyakarta pada tahun 2018. Dua tahun kemudian, penulis menyelesaikan studi S2 di prodi Teknik In*for*matika Program Pasca Sarjana Universitas Amikom Yogyakarta dan juga prodi Magister Pariwisata di Sekolah Tinggi Pariwisata Ambarrukmo Yogyakarta. Atas dedikasi dan kerja keras dalam membuat suatu karya, Republik Indonesia Kementerian Hukum Dan Hak Asasi Manusia sudah mencatat ada kurang lebih 100 karya yang sudah tercatat di surat pencatatan ciptaan sebagai salah satu kontribusi dalam melindungi hak kekayaan intelektual.

Dr. Amril Mutoi Siregar, M. Kom. lahir di Ujung Padang, Sumatera Utara. Sekolah SD sampai SMP diselesaikan di kota kelahirannya. Kemudian pada tahun 1994 melanjutkan Pendidikan di SMA PGRI 4 Jakarta Jurusan IPA. Tahun 2004 Melanjutkan Pendidikan Program S1 di STMIK MIC Cikarang pada jurusan Teknik In*for*matika. Tahun 2014 melanjutkan Pendidikan Program S2 di President University Cikarang Jurusan Teknik In*for*matika. Tahun 2020 melanjutkan Pendidikan Program S3 di IPB university jurusan Ilmu Komputer. Penulis konsentrasi mengajar dan penelitian di bidang: Machine Learning, Deep Learning, data mining, Data Science dan Algoritma.

Dr. Ir. Hansi Effendi, S.T., M.Kom., adalah seorang akademisi dan peneliti di bidang pendidikan teknologi dan ilmu komputer. Ia menyelesaikan pendidikan S1 di bidang Teknik Elektro di Universitas Andalas, kemudian meraih gelar S2 dalam Magister Ilmu Komputer di UPI YPTK Padang. Untuk memperdalam keahliannya, ia menyelesaikan studi S3 di Program Doktor Pendidikan Teknologi dan Kejuruan, Universitas Negeri Yogyakarta. Dengan pengalaman lebih dari 20 tahun di dunia pendidikan, Hansi Effendi telah berkontribusi sebagai dosen di Universitas Negeri Padang sejak tahun 2002. Fokus penelitiannya meliputi pengembangan metode pembelajaran inovatif, dengan penekanan pada pendekatan seperti Pembelajaran Berbasis Masalah (PBL) dan Pembelajaran Berbasis Proyek (PjBL). Ia juga aktif terlibat dalam berbagai proyek penelitian yang bertujuan meningkatkan kualitas pendidikan melalui pendekatan berbasis teknologi.

Satriawaty Mallu, Dosen di STMIK Profesional Makassar, menyelesaikan S1 Teknik In*for*matika Universitas Handayani, S2 Ilmu Komputer Universitas Gadjah Mada dan Insinyur Profesi UMI Makassar, sebagai Dosen salah satu kewajiban kita adalah menulis, buku ini adalah salah satu karya dan Insya Allah secara konsisten akan disusul dengan buku-buku berikutnya. Pokok bahasan buku yang ditulis semata-mata untuk berbagi ilmu pengetahuan.

Samsuriah, S.Kom., M.T, buku ini adalah slah satu karya dan Insya Allah secara konsisten akan disusul dengan buku-buku berikutnya. Pokok bahasan buku yang ditulis semata-mata untuk berbagi ilmu pengetahuan. Lulus S1 di STMIK Handayani Makassar tahun 2012, kemudian menyelesaikan pendidikan S2 di program studi Teknik Elektro dengan konsentrasi Teknik In*for*matika pada Fakultas Teknik Universitas Hasanuddin Makassar pada tahun 2015. Pada tahun 2015 Menjadi Dosen di Prodi DIII Manajemen In*for*matika Sekolah Tinggi Manajemen In*for*matika dan Komputer (STMIK) Profesional Makassar hingga saat ini, kemudian desember 2024 pindah di Jurusan/Prodi Ilmu Komputer sekaligus diberikan amanah sebagai Koordinator Program Studi di Program Studi Ilmu Komputer. Penulis mengajarkan beberapa mata kuliah seperti, Sistem Operasi, Teknik Riset Operasi, Pemrograman Berorientasi Objek, dll. Aktif melaksanakan penelitian terutama yang berhubungan dengan bidang ilmu yang di tekuni yaitu komputer serta melakukan kerja sama penelitian dengan berbagai bidang ilmu yang lain. Selain itu penulis telah menerbitkan artikel pada berbagai jurnal nasional.

Dr. Dianta Ginting, EMBA. lahir di desa Kutatengah, Kabanjahe, Kabupaten Karo, Sumatera Utara. Pendidikan dasar (SD) diselesaikan di desa kelahirannya, kemudian melanjutkan ke SMP Negeri 1 Kabanjahe. Pada tahun 1999, beliau menyelesaikan pendidikan menengah atas di SMA Negeri 2 Kabanjahe. Tahun 2002, Dr. Dianta Ginting melanjutkan pendidikan S1 di

bidang Geofisika di Institut Teknologi Bandung (ITB). Setelah menyelesaikan studi sarjana, beliau melanjutkan ke jenjang magister (S2) pada tahun 2009 di Chungbuk National University, Korea Selatan. Ketertarikan dalam pengembangan keilmuan membawanya kembali melanjutkan studi doktoral (S3) pada tahun 2015 di Kyung Hee University, Korea Selatan.

Ni Putu Linda Santiari, S.Kom., M.Kom. Lahir di Badung, 07 Oktober 1990. Lulus S2 di Program Studi Magister Teknik In*for*matika Universitas Amikom Yogyakarta tahun 2016. Saat ini sebagai Dosen di Institut Teknologi dan Bisnis STIKOM Bali pada Fakultas In*for*matika dan Komputer.

Ismi Rizqa Lina, S.Mat., M.at merupakan seorang Dosen prodi Sains Data Universitas Insan Cita Indonesia. Ia menyelesaikan Pendidikan S1 Matematika di UIN Maulana Malik Ibrahim Malang tahun 2027, kemudian melanjutkan Pendidikan S2 Matematikan di Universitas Brawijaya tahun 2020. Buku Algoritma dan Pemrograman merupakan buku ke 2 yang ditulis setelah menulis buku Pengantar Statistik. Selain itu ia juga fokus penelitian di bidang matematika dan Machine Learning.

Dr. Drs. Waris Marsisno, M.Stat. merupakan dosen tetap pada Program Studi Komputasi Statistik, Politeknik Statistika STIS. Bagian yang ditulis dalam buku ini merupakan salah satu karya dan semoga secara konsisten akan disusul dengan buku-buku berikutnya. Pokok bahasan buku yang ditulis semata-mata untuk berbagi ilmu pengetahuan.

Fahmi Ruziq, lahir di Banda Aceh, 16 Juni 1989, merupakan anak pertama dari empat bersaudara. Penulis merupakan alumni Program Sarjana (S-1) di Universitas Serambi Mekkah pada Jurusan Teknik In*for*matika dan lulus tahun 2010. Penulis melanjutkan studi Program Magister (S-2) Teknik In*for*matika di Universitas Sumatera Utara dan lulus tahun 2020. Berkarir sebagai dosen dimulai dari tahun 2020 di Universitas Battuta. Mulai tahun 2022-sekarang diberi kepercayaan menjadi Ketua Program Studi Sistem Informasi Fakultas Teknologi di Universitas Battuta.

Mardiah, S.SI., M.Kom Lahir di Bandung 6 November 1993, penulis menempuh pendidikan Sarjana di bidang Sistem Informasi di Universitas Sriwijaya, kemudian melanjutkan studi Magister Komputer di Institut Pertanian Bogor. Buku ini adalah salah satu karya dan Insya Allah secara konsisten akan disusul dengan buku-buku berikutnya. Pokok bahasan buku yang ditulis bertujuan untuk meneruskan ilmu yang dititipkan dan

salah satu bentuk usaha mendapatkan amal jariyah dari Allah. Barakallah fiikum.

Anis Fitri Nur Masruriyah, adalah seorang pendidik dan peneliti yang aktif di dunia pendidikan tinggi di Indonesia. Ia lahir pada tahun yang tidak disebutkan dan telah aktif mengajar di salah satu universitas di Indonesia sejak tahun 2019. Penulis adalah seorang lulusan dari Institut Pertanian Bogor (IPB) meraih gelar M.Kom pada tahun 2018. Sebelumnya, pada tahun 2015 memperoleh gelar S.Kom dari Universitas Brawijaya. Selama karirnya di dunia pendidikan, Penulis telah aktif terlibat dalam penelitian dan pengabdian kepada masyarakat sejak tahun 2019. Penulis juga telah berkontribusi dalam upaya pengabdian masyarakat untuk memanfaatkan pengetahuannya demi kemajuan masyarakat dan negara. Sebagai seorang pendidik dan peneliti, buku ini adalah salah satu karya dan inshaa allah secara konsisten akan disusul dengan buku-buku berikutnya. Pokok bahasan buku yang ditulis semata-mata untuk berbagi ilmu pengetahuan.

Santi Prayudani, lahir di Kotamadya Binjai Sumatera Utara pada tanggal 28 Maret 1986. Penulis menempuh pendidikan S1 di Prodi Ilmu Komputer Universitas Sumatera Utara pada tahun 2004. Kemudian melanjutkan kembali pendidikan S2 di Prodi Teknik In*for*matika Universitas Sumatera Utara pada tahun 2011. Memulai karir sebagai guru di SDS Al Azhar Medan pada tahun 2010. Kemudian mengajar juga sebagai dosen di AMIK

Harapan dan Universitas Pembangunan Panca Budi dari tahun 2011 sampai tahun 2014. Saat ini penulis diberi amanah oleh negara untuk mengabdi sebagai dosen di Politeknik Negeri Medan dari tahun 2015.

Sri Retnowati, S.Pd., M.Pd. Lahir di Kebumen, 22 September 1994. Lulus S2 di Program Studi Pendidikan Matematika Universitas Sebelas Maret tahun 2020. Saat ini sebagai Dosen di Universitas Insan Cita Indonesia Program Studi Sains Data.

Ari Widiastono, S.Kom., MT.,MH, buku ini adalah salah satu karya dan inshaa allah secara konsisten akan disusul dengan buku-buku berikutnya. Pokok bahasan buku yang ditulis semata-mata untuk berbagi ilmu pengetahuan.

Dita Novita Sari, S.Kom., M.T.I. Lahir di Gadingrejo, 06 November 1988. Lulus S2 di Program Teknik In*for*matika Institut In*for*matika dan Bisnis Darmajaya Lampung tahun 2015. Saat ini sebagai Dosen di Institut Bakti Nusantara Lampung Program Studi Sistem Informasi.

A.Kachsyfur Djasim Ilyas Paenrongi, buku ini adalah salah satu karya dan inshaa allah secara konsisten akan disusul dengan buku-buku berikutnya. Pokok bahasan buku yang ditulis semata-mata untuk berbagi ilmu pengetahuan.

Triana Dewi Salma, M.T., Penulis lahir di Tuban pada tanggal 2 September 1994. Saat ini, penulis menjabat sebagai Dekan Fakultas Sains dan Bisnis serta dosen tetap pada Program Studi In*for*matika di Universitas LIA. Penulis memperoleh gelar Sarjana pada bidang In*for*matika dari Universitas Muhammadiyah Surakarta dan melanjutkan pendidikan Magister In*for*matika di Institut Teknologi Bandung. Dorongan utama dalam berkarya adalah keyakinan bahwa ilmu pengetahuan harus terus berkembang dan memberi manfaat jangka panjang. Buku ini adalah salah satu karya dan inshaa allah secara konsisten akan disusul dengan buku-buku berikutnya. Pokok bahasan buku yang ditulis semata-mata untuk berbagi ilmu pengetahuan.

Buku Logika dan Algoritma Pemrograman: Fondasi Dasar Menjadi Programmer Andal disusun untuk membantu pembaca membangun pemahaman yang kuat terhadap konsep logika dan algoritma, dua elemen krusial dalam dunia pemrograman. Melalui pendekatan sistematis dan progresif, buku ini membekali pembaca dengan kemampuan berpikir logis, menyusun langkah-langkah penyelesaian masalah secara terstruktur, serta mengimplementasikannya ke dalam kode program. Isi buku mencakup berbagai topik penting mulai dari proposisi dan logika predikat, struktur kontrol algoritma, tipe data, operasi aritmetika dan logika, hingga konsep modularitas, rekursi, dan algoritma pemrosesan data seperti greedy dan divide-and-conquer. Disertai dengan contoh-contoh pseudocode, flowchart, dan implementasi nyata dalam bahasa Python, buku ini cocok digunakan baik sebagai bahan ajar di kelas maupun panduan belajar mandiri. Dirancang untuk pemula hingga menengah, buku ini menjadi referensi tepat bagi pelajar, mahasiswa, dosen, atau siapa pun yang ingin memulai karier sebagai programmer profesional. Dengan memahami fondasi logika dan algoritma secara menyeluruh, pembaca akan lebih siap menghadapi tantangan dunia pemrograman yang menuntut efisiensi, ketepatan, dan kreativitas dalam menyelesaikan masalah digital masa kini.

DITERBITKAN OLEH
PT. MIFANDI MANDIRI DIGITAL
Jln Payanibung Ujung D
Dalu Sepuluh-B, Tanjung Morawa
Kab. Deli Serdang Sumatera Utara

ISBN 978-634-7226-25-9