RESUME

PERANCANGAN DAN ANALISIS ALGORITMA

DOSEN PENGAMPU:

Widya Darwin S.Pd,MPdt

OLEH:

Elsa Rahma Hidayani

21346006

Informatika

PROGRAM STUDI INFORMATIKA

JURUSAN ELEKTRONIKA

FAKULTAS TEKNIK

UNIVERSITAS NEGERI PADANG

2023

DAFTAR ISI

BAB I
PENGANTAR ANALISIS ALGORITMA

A. Pendahuluan

Pengantar Analisis Algoritma Analisis algoritma adalah bidang studi yang berkaitan dengan penelitian, perancangan, dan analisis efisiensi algoritma. Algoritma sendiri merupakan langkah-langkah sistematis yang digunakan untuk memecahkan masalah atau menyelesaikan tugas tertentu. Dalam pengembangan perangkat lunak, analisis algoritma menjadi sangat penting karena dapat mempengaruhi kinerja dan efisiensi dari sebuah program.

B. Pengertian Algoritma

1. Definisi Algoritma Algoritma dapat didefinisikan sebagai serangkaian instruksi langkah demi langkah yang digunakan untuk menyelesaikan masalah atau mencapai tujuan tertentu. Algoritma harus memiliki langkah-langkah yang jelas, dapat dipahami oleh mesin atau manusia, serta menghasilkan output yang diharapkan. Algoritma merupakan dasar dalam pengembangan perangkat lunak dan memainkan peran penting dalam proses pemrograman.

2. Karakteristik Algoritma yang Baik Sebuah algoritma yang baik memiliki beberapa karakteristik yang harus dipenuhi, antara lain:

- Kejelasan: Langkah-langkah dalam algoritma harus jelas dan dapat dipahami dengan baik.
- Efisiensi: Algoritma harus dirancang sedemikian rupa sehingga dapat menyelesaikan masalah dengan menggunakan sumber daya yang minimal.
- Kelengkapan: Algoritma harus mencakup semua langkah yang diperlukan untuk menyelesaikan masalah yang diberikan.
- Kesederhanaan: Algoritma yang sederhana lebih mudah dimengerti, dianalisis, dan diimplementasikan.

3. Contoh Algoritma Sederhana Berikut ini adalah contoh algoritma sederhana untuk mencari bilangan terbesar dari dua bilangan:

4. Masukkan dua bilangan ke dalam variabel A dan B.

5. Jika A lebih besar dari B, maka cetak A sebagai bilangan terbesar.

6. Jika B lebih besar dari A, maka cetak B sebagai bilangan terbesar.

7. Jika A dan B sama besar, maka cetak "Bilangan A dan B sama besar."

C. Dasar Algoritma

1. Langkah-langkah Dasar dalam Merancang Algoritma Dalam merancang algoritma, terdapat beberapa langkah dasar yang harus diikuti, antara lain:

- Memahami masalah: Langkah pertama adalah memahami dengan baik masalah yang ingin diselesaikan atau tugas yang ingin dilakukan.

- Analisis masalah: Mengidentifikasi masalah yang lebih kecil yang perlu dipecahkan dalam menyelesaikan masalah yang lebih besar.
- Perancangan algoritma: Merancang langkah-langkah yang diperlukan untuk menyelesaikan masalah yang diidentifikasi sebelumnya.
- Implementasi algoritma: Menerjemahkan algoritma yang dirancang ke dalam bahasa pemrograman yang dapat dieksekusi oleh komputer.

2. Representasi Algoritma: Pseudocode dan Flowchart Pseudocode dan flowchart adalah dua metode yang umum digunakan untuk merepresentasikan algoritma:

- Pseudocode adalah metode penulisan algoritma menggunakan bahasa yang mirip dengan bahasa pemrograman, namun lebih fokus pada logika algoritma daripada sintaksis.
- Flowchart adalah representasi visual menggunakan simbol-simbol grafis untuk menggambarkan langkah-langkah algoritma.

3. Struktur Kontrol dalam Algoritma: Urutan, Pemilihan, Perulangan Struktur kontrol adalah bagian penting dalam merancang algoritma yang meliputi:

- Urutan: Langkah-langkah dalam algoritma dieksekusi secara berurutan dari atas ke bawah.
- Pemilihan: Memilih langkah atau tindakan berdasarkan kondisi yang ditentukan.
- Perulangan: Mengulangi langkah-langkah tertentu hingga kondisi yang ditentukan terpenuhi.

D. Problem Solving

1. Pengertian Problem Solving Problem solving merupakan proses untuk memecahkan masalah atau menemukan solusi yang efektif untuk suatu permasalahan. Dalam konteks analisis algoritma, problem solving mencakup pemahaman masalah, analisis masalah, dan merancang algoritma yang dapat menyelesaikan masalah tersebut.

2. Pendekatan dalam Problem Solving Ada beberapa pendekatan yang dapat digunakan dalam problem solving, antara lain:

- Pendekatan sistematis: Menggunakan langkah-langkah yang terstruktur untuk mengidentifikasi masalah, menganalisis penyebabnya, dan merancang solusi yang tepat.
- Pendekatan kreatif: Menggunakan pemikiran kreatif dan berpikir out-of-the-box untuk menemukan solusi yang inovatif dan tidak konvensional.

3. Tahapan dalam Problem Solving Proses problem solving melibatkan beberapa tahapan, yaitu:

- Analisis masalah: Memahami dengan baik masalah yang sedang dihadapi, mengidentifikasi faktor penyebab, dan mengumpulkan informasi yang relevan.

- Perancangan solusi: Merancang langkah-langkah yang diperlukan untuk menyelesaikan masalah berdasarkan analisis yang telah dilakukan.
- Implementasi solusi: Mengimplementasikan solusi yang telah dirancang ke dalam bentuk algoritma yang dapat dieksekusi oleh komputer.
- Evaluasi solusi: Mengevaluasi solusi yang diimplementasikan untuk memastikan bahwa masalah telah diselesaikan dengan baik.

4. Contoh Penerapan Problem Solving dalam Algoritma Sebagai contoh, dalam penyelesaian masalah matematika, langkah-langkah problem solving meliputi pemahaman soal, pemecahan masalah ke dalam langkah-langkah yang lebih kecil, dan merancang algoritma matematika yang tepat untuk mencapai solusi yang benar.

BAB II
ANALISIS FRAMEWORK

A. Pengukuran Ukuran Input

Pengukuran ukuran input adalah langkah penting dalam analisis algoritma karena ukuran input mempengaruhi kinerja dan kompleksitas algoritma. Berikut adalah beberapa alasan mengapa pengukuran ukuran input penting:

1. Memahami Kompleksitas Masalah:
    - Dengan mengukur ukuran input, kita dapat memahami kompleksitas masalah yang ingin kita pecahkan menggunakan algoritma. Misalnya, jika kita ingin mengurutkan sebuah array, ukuran inputnya bisa diukur dalam jumlah elemen yang ada dalam array tersebut. Pengukuran ini membantu kita memahami sejauh mana algoritma harus bekerja untuk menyelesaikan masalah dengan ukuran input yang berbeda-beda.
2. Membandingkan Algoritma:
    - Dengan mengukur ukuran input, kita dapat membandingkan kinerja antara dua algoritma yang berbeda. Dalam analisis algoritma, kita sering membandingkan algoritma berdasarkan waktu eksekusi atau kompleksitas ruang yang diperlukan untuk input yang sama. Misalnya, algoritma A mungkin lebih efisien daripada algoritma B untuk input dengan ukuran tertentu, tetapi hal ini mungkin berubah ketika ukuran input meningkat.
3. Prediksi Skalabilitas:
    - Dengan mengukur ukuran input, kita dapat memprediksi seberapa baik algoritma akan berperforma saat ukuran input diperbesar. Dalam pengembangan perangkat lunak, skalabilitas adalah hal yang penting karena sering kali kita harus menangani input yang semakin besar seiring waktu. Dengan mengukur ukuran input dan menganalisis kinerja algoritma berdasarkan ukuran input tersebut, kita dapat memperkirakan seberapa baik algoritma tersebut akan berperforma di masa depan.

Faktor-faktor yang dapat mempengaruhi ukuran input termasuk jumlah elemen, ukuran file, jumlah simpul dalam graf, atau dimensi dari struktur data yang digunakan. Dalam analisis algoritma, kita harus memperhatikan faktor-faktor ini ketika memilih dan menganalisis algoritma untuk menyelesaikan suatu masalah.

Dengan memahami pentingnya pengukuran ukuran input dan faktor-faktor yang mempengaruhinya, kita dapat melakukan analisis yang lebih akurat dan dapat mengambil keputusan yang lebih baik dalam pemilihan dan perancangan algoritma.

B. Waktu Eksekusi

1. Pengertian Waktu Eksekusi: Waktu eksekusi adalah ukuran yang digunakan untuk mengukur seberapa efisien atau cepat sebuah algoritma menyelesaikan tugasnya. Dalam

konteks analisis algoritma, waktu eksekusi mengacu pada jumlah operasi atau langkah yang diperlukan oleh algoritma untuk menyelesaikan masalah dengan input tertentu.

Pentingnya waktu eksekusi dalam analisis algoritma:

- Waktu eksekusi membantu kita memahami seberapa efisien sebuah algoritma dalam menyelesaikan masalah dengan ukuran input yang berbeda.
- Waktu eksekusi memungkinkan kita membandingkan kinerja antara dua algoritma yang berbeda untuk input yang sama.
- Dalam pengembangan perangkat lunak, waktu eksekusi menjadi faktor kunci dalam memilih algoritma yang tepat untuk aplikasi yang spesifik.

2. Order of Growth (Urutan Pertumbuhan): Order of growth (urutan pertumbuhan) menggambarkan bagaimana waktu eksekusi algoritma berubah seiring dengan pertumbuhan ukuran input. Konsep ini membantu kita menganalisis kompleksitas waktu algoritma.

Notasi Big O:

- Notasi Big O (O) digunakan untuk mewakili urutan pertumbuhan algoritma.
- O(f(n)) menggambarkan batas atas dari waktu eksekusi algoritma ketika ukuran input menjadi sangat besar.
- f(n) adalah fungsi yang menggambarkan jumlah operasi atau langkah yang diperlukan oleh algoritma dalam kaitannya dengan ukuran input n.

3. Kasus Terburuk, Kasus Terbaik, dan Kasus Rata-rata:

- Kasus terburuk (worst-case) adalah situasi di mana algoritma membutuhkan waktu eksekusi maksimum untuk input tertentu. Ini memberikan batas atas untuk kinerja algoritma dalam kondisi terburuk.
- Kasus terbaik (best-case) adalah situasi di mana algoritma membutuhkan waktu eksekusi minimum untuk input tertentu. Namun, kasus terbaik seringkali tidak memberikan gambaran lengkap tentang kinerja algoritma.
- Kasus rata-rata (average-case) mengacu pada perkiraan waktu eksekusi rata-rata algoritma untuk semua kemungkinan input yang mungkin. Ini memperhitungkan probabilitas masing-masing input.

Penggunaan kasus terburuk, terbaik, dan rata-rata dalam analisis efisiensi algoritma:

- Kasus terburuk memberikan batas atas yang penting untuk memastikan algoritma berjalan dengan baik dalam semua kondisi.

- Kasus terbaik memberikan informasi tentang kinerja algoritma dalam situasi terbaik dan dapat membantu dalam mengidentifikasi keadaan khusus yang memberikan kinerja yang optimal.
- Kasus rata-rata memberikan perkiraan kinerja yang lebih realistis seiring dengan distribusi kemungkinan input yang mungkin dihadapi oleh algoritma.

Dengan memahami waktu eksekusi, order of growth, dan perbedaan antara kasus terburuk, terbaik, dan rata-rata, kita dapat menganalisis kinerja algoritma dengan lebih baik dan membuat keputusan yang tepat dalam pemilihan algoritma yang efisien untuk masalah yang spesifik.

C. Efisiensi

1. Pengertian Efisiensi: Efisiensi dalam konteks analisis algoritma mengacu pada kemampuan algoritma untuk menyelesaikan tugas dengan menggunakan sumber daya yang minimum. Efisiensi terkait dengan kinerja algoritma, seperti waktu eksekusi dan penggunaan ruang memori. Dalam pengembangan perangkat lunak, efisiensi merupakan faktor penting karena dapat mempengaruhi performa, responsivitas, dan penggunaan sumber daya secara keseluruhan.

Mengapa efisiensi penting dalam pengembangan perangkat lunak:

- Efisiensi mempengaruhi pengalaman pengguna. Algoritma yang lebih efisien dapat memberikan respon yang lebih cepat dan pengalaman pengguna yang lebih baik.
- Efisiensi berhubungan dengan skalabilitas. Algoritma yang efisien mampu menangani input yang semakin besar dengan tetap menjaga kinerja yang baik.
- Efisiensi berdampak pada penggunaan sumber daya. Algoritma yang tidak efisien dapat menghabiskan sumber daya seperti memori, daya komputasi, dan bandwidth secara berlebihan.

2. Faktor-faktor yang Mempengaruhi Efisiensi: Beberapa faktor yang dapat mempengaruhi efisiensi algoritma meliputi:

- Struktur Data: Pemilihan struktur data yang tepat dapat meningkatkan efisiensi algoritma. Misalnya, menggunakan struktur data yang efisien seperti array atau pohon dapat mengurangi waktu akses dan pencarian data.
- Penggunaan Looping: Perulangan yang berlebihan atau tidak perlu dapat mengurangi efisiensi algoritma. Optimalisasi perulangan dapat meningkatkan efisiensi dengan mengurangi jumlah operasi yang tidak diperlukan.
- Kompleksitas Algoritma: Algoritma dengan kompleksitas tinggi cenderung memiliki kinerja yang lebih buruk. Memilih algoritma dengan kompleksitas yang lebih rendah dapat meningkatkan efisiensi.

Contoh faktor-faktor yang perlu dipertimbangkan dalam menganalisis efisiensi algoritma meliputi ukuran input, kebutuhan pemrosesan data, penggunaan struktur data yang tepat, kompleksitas algoritma, dan penanganan kasus terburuk.

3. Strategi untuk Meningkatkan Efisiensi: Ada beberapa strategi yang dapat digunakan untuk meningkatkan efisiensi algoritma:

- Algoritma yang Lebih Efisien: Pemilihan algoritma yang lebih efisien berdasarkan analisis kinerja dan kompleksitas dapat memberikan perbaikan signifikan dalam efisiensi.
- Optimalisasi Kode: Memperbaiki kode algoritma dengan menghilangkan duplikasi, mengurangi operasi yang tidak perlu, dan menggunakan struktur data yang efisien dapat meningkatkan efisiensi.
- Penyusunan Kembali (Reordering): Mengubah urutan operasi atau perulangan dalam algoritma untuk meminimalkan jumlah operasi atau memanfaatkan caching dapat meningkatkan efisiensi.
- Penggunaan Caching: Menggunakan caching untuk menyimpan hasil perhitungan sebelumnya dan menghindari perhitungan berulang dapat mengurangi waktu eksekusi.
- Paralelisasi: Membagi algoritma menjadi bagian-bagian yang dapat dieksekusi secara paralel pada sistem multi-core atau distribusi komputasi dapat meningkatkan efisiensi.

Contoh penerapan strategi-strategi tersebut dalam perancangan algoritma meliputi mengganti algoritma dengan kompleksitas yang lebih rendah, mengoptimalkan perulangan dengan membatasi iterasi yang tidak perlu, menggunakan struktur data yang tepat seperti hashtable atau heap, dan menggunakan teknik memoisasi untuk menghindari perhitungan berulang.

D. Studi Kasus

Studi kasus pada algoritma pencarian biner:

- Analisis Framework pada Algoritma Tertentu:
    - Mengukur ukuran input: Ukuran input dalam kasus pencarian biner bisa diukur dalam jumlah elemen yang ada dalam array terurut.
    - Waktu eksekusi: Menghitung jumlah langkah yang diperlukan untuk menemukan elemen target menggunakan pencarian biner.
    - Efisiensi: Menganalisis kinerja algoritma berdasarkan waktu eksekusi dan kompleksitas.
- Langkah-langkah dalam menganalisis waktu eksekusi dan efisiensi algoritma:
    - Mengidentifikasi langkah-langkah algoritma dan operasi yang dilakukan.
    - Mengukur kompleksitas algoritma dalam notasi Big O untuk menentukan urutan pertumbuhan algoritma.
    - Menganalisis kasus terburuk, terbaik, dan rata-rata untuk memahami kinerja algoritma dalam berbagai situasi input.

- Membandingkan dengan algoritma lain atau strategi lain untuk mencari solusi yang lebih efisien.

Dengan melakukan analisis framework pada algoritma tertentu, kita dapat memahami waktu eksekusi, efisiensi, dan kinerja algoritma secara lebih mendalam. Hal ini membantu kita dalam pengembangan perangkat lunak yang efisien dan efektif.

BAB III
BRUTE FORCE DAN EXHAUSTIVE SEARCH

A. Selection Sort and Bubble Sort

Selection Sort: Selection Sort adalah algoritma pengurutan sederhana yang bekerja dengan membagi array menjadi dua bagian: bagian yang sudah terurut dan bagian yang belum terurut. Pada setiap langkah iterasi, algoritma memilih elemen terkecil dari bagian belum terurut dan menukar posisinya dengan elemen pertama dari bagian terurut. Proses ini terus diulang sampai seluruh array terurut.

Bubble Sort: Bubble Sort adalah algoritma pengurutan yang bekerja dengan membandingkan setiap pasangan elemen berturut-turut dalam array dan menukar posisinya jika tidak terurut. Selama proses ini, elemen-elemen yang lebih besar secara bertahap "menggelembung" ke ujung array. Algoritma berulang kali melintasi array sampai tidak ada lagi pertukaran yang perlu dilakukan, menandakan bahwa array sudah terurut.

Perbandingan:

- Selection Sort memiliki kompleksitas waktu O(n^2), di mana n adalah jumlah elemen dalam array, sedangkan Bubble Sort juga memiliki kompleksitas waktu O(n^2). Keduanya merupakan algoritma dengan kompleksitas waktu yang relatif tinggi.
- Selection Sort memiliki jumlah pertukaran yang lebih sedikit dibandingkan dengan Bubble Sort, sehingga pada kondisi tertentu (misalnya ketika akses ke media penyimpanan terbatas), Selection Sort mungkin lebih efisien daripada Bubble Sort.
- Bubble Sort dapat melakukan banyak pertukaran pada setiap iterasi, bahkan ketika array hampir terurut, sedangkan Selection Sort tetap melakukan pertukaran satu per satu pada setiap langkah iterasi.

B. Sequential Search and Brute-Force String Maching

Sequential Search: Sequential Search adalah algoritma pencarian sederhana yang bekerja dengan memeriksa setiap elemen dalam urutan tertentu sampai elemen yang dicari ditemukan atau sampai akhir dari urutan. Algoritma ini tidak memerlukan syarat atau asumsi khusus terhadap data yang dicari.

Brute-Force String Matching: Brute-Force String Matching adalah algoritma pencarian pola dalam sebuah teks dengan pendekatan brute-force. Algoritma ini bekerja dengan memeriksa setiap posisi dalam teks dan membandingkannya dengan pola yang ingin dicari. Jika ada kecocokan, maka pola ditemukan.

Perbandingan:

- Sequential Search adalah algoritma pencarian yang sederhana dan mudah dipahami. Namun, pada kasus terburuknya, kompleksitas waktu algoritma ini adalah O(n), di mana n adalah ukuran urutan yang dicari.

- Brute-Force String Matching memiliki kompleksitas waktu O(mn), di mana m adalah ukuran pola dan n adalah ukuran teks. Algoritma ini lebih efisien untuk mencari pola dalam teks daripada Sequential Search, tetapi kompleksitasnya masih relatif tinggi.
- Brute-Force String Matching lebih efektif dalam mencari pola dalam teks yang panjang, karena dapat melakukan "skip" dalam perbandingan berikutnya ketika ada ketidakcocokan parsial dengan pola.

C. Closest-Pair and Convex-Hull Problems

Closest-Pair Problem: Closest-Pair Problem adalah masalah untuk mencari pasangan titik dalam himpunan titik yang memiliki jarak terkecil di antara semua pasangan titik. Algoritma untuk menyelesaikan masalah ini melibatkan pemrosesan dan pembandingan berpasangan yang intensif.

Convex-Hull Problem: Convex-Hull Problem adalah masalah untuk menemukan himpunan terkecil dari titik yang membungkus semua titik dalam himpunan tersebut. Himpunan tersebut membentuk suatu poligon convex. Algoritma untuk menyelesaikan masalah ini mencari titik-titik yang berada di batas luar poligon convex.

Perbandingan:

- Closest-Pair Problem dan Convex-Hull Problem adalah masalah komputasional yang melibatkan pengolahan geometri dan pembandingan titik-titik.
- Solusi untuk kedua masalah ini melibatkan penggunaan algoritma yang memerlukan kompleksitas waktu yang lebih tinggi.
- Algoritma untuk Closest-Pair Problem sering kali menggunakan teknik Divide and Conquer, sementara algoritma untuk Convex-Hull Problem sering kali menggunakan teknik Incremental atau Graham Scan.

D. Exhaustive Search

Exhaustive Search adalah pendekatan dalam algoritma yang mencoba semua kemungkinan solusi secara sistematis untuk mencari solusi yang optimal atau memenuhi kriteria tertentu. Algoritma ini melakukan pencarian secara eksploratif melalui semua kombinasi solusi yang mungkin, sehingga dikenal juga sebagai algoritma brute-force.

Kelebihan Exhaustive Search:

- Exhaustive Search dapat menemukan solusi optimal atau solusi terbaik yang ada dalam ruang pencarian yang diberikan.
- Algoritma ini mudah dipahami dan diimplementasikan, karena tidak memerlukan asumsi atau teknik khusus.
- Exhaustive Search dapat digunakan untuk masalah yang memiliki ruang pencarian yang terbatas atau ukuran input yang kecil.

Kekurangan Exhaustive Search:

- Algoritma Exhaustive Search memiliki kompleksitas waktu yang tinggi, terutama ketika jumlah kombinasi yang mungkin sangat besar.
- Pada masalah dengan ruang pencarian yang besar, algoritma Exhaustive Search mungkin tidak efisien dan memerlukan waktu yang lama untuk menemukan solusi.

E. Depth-First Search and Breath-First Search

Depth-First Search (DFS): Depth-First Search adalah algoritma pencarian yang bekerja dengan menjelajahi semua simpul dalam suatu graf secara mendalam sebelum berpindah ke simpul lain. Algoritma ini menggunakan pendekatan rekursif atau menggunakan tumpukan (stack) untuk melacak simpul yang akan dieksplorasi selanjutnya.

Breadth-First Search (BFS): Breadth-First Search adalah algoritma pencarian yang bekerja dengan menjelajahi semua simpul dalam suatu graf secara melebar atau secara level. Algoritma ini menggunakan pendekatan antrian (queue) untuk melacak simpul yang akan dieksplorasi selanjutnya.

Perbandingan:

- DFS dan BFS adalah algoritma pencarian yang digunakan dalam pemrosesan graf.
- DFS melintasi graf dengan cara mendalam, sementara BFS melintasi graf dengan cara melebar.
- DFS cenderung memeriksa cabang-cabang dalam graf lebih dalam sebelum berpindah ke cabang lain, sedangkan BFS menjelajahi semua simpul dalam tingkat yang sama terlebih dahulu sebelum berpindah ke tingkat berikutnya.
- DFS menggunakan tumpukan (stack) untuk melacak simpul yang akan dieksplorasi selanjutnya, sedangkan BFS menggunakan antrian (queue).
- DFS biasanya digunakan untuk mencari jalur atau menggambarkan struktur graf, sementara BFS biasanya digunakan untuk mencari jalur terpendek atau menemukan semua simpul yang dapat dicapai dalam jumlah langkah terbatas.

Pemahaman tentang perbedaan dan kegunaan masing-masing algoritma tersebut dapat membantu dalam pemilihan dan implementasi yang tepat saat menyelesaikan masalah yang berkaitan dengan pencarian, pengurutan, atau pemrosesan graf.

BAB IV
DECREASE AND CONQUER

A. Three Major Variants of Decrease-and-Conquer

Decrease-and-Conquer adalah salah satu pendekatan dalam desain algoritma yang berfokus pada pengurangan ukuran masalah secara terus-menerus hingga masalah menjadi lebih kecil dan lebih mudah untuk dipecahkan. Terdapat tiga varian utama dari pendekatan Decrease-and-Conquer, yaitu:

1. Decrease by a Constant Factor: Dalam varian ini, ukuran masalah dikurangi dengan faktor tetap pada setiap iterasi. Misalnya, jika ukuran awal masalah adalah n, maka pada setiap langkah iterasi ukurannya berkurang menjadi n/c, di mana c adalah konstanta. Algoritma yang menggunakan varian ini biasanya memiliki kompleksitas waktu O(log n), karena ukuran masalah dikurangi dengan faktor logaritmik pada setiap langkah iterasi.

2. Decrease by a Constant Amount: Pada varian ini, ukuran masalah dikurangi dengan jumlah tetap pada setiap iterasi. Misalnya, jika ukuran awal masalah adalah n, maka pada setiap langkah iterasi ukurannya berkurang menjadi n - c, di mana c adalah konstanta. Algoritma yang menggunakan varian ini biasanya memiliki kompleksitas waktu O(n), karena ukuran masalah berkurang sebesar konstanta pada setiap langkah iterasi.

3. Decrease by a Variable Factor: Dalam varian ini, ukuran masalah dikurangi dengan faktor yang bervariasi pada setiap iterasi. Faktor pengurangan dapat tergantung pada sifat masalah itu sendiri. Misalnya, jika ukuran masalah adalah n, maka pada langkah pertama ukurannya bisa berkurang menjadi n/c1, kemudian pada langkah kedua berkurang menjadi n/c2, dan seterusnya. Algoritma yang menggunakan varian ini biasanya memiliki kompleksitas waktu yang bervariasi tergantung pada faktor pengurangan yang digunakan.

B. Sort

Sorting (pengurutan) adalah proses mengatur elemen-elemen dalam suatu koleksi data menjadi urutan yang teratur, misalnya secara menaik atau menurun. Pengurutan data sangat penting dalam pemrosesan data dan digunakan dalam berbagai aplikasi seperti pengindeksan, pencarian, dan analisis data. Terdapat berbagai algoritma pengurutan yang berbeda, di antaranya:

1. Bubble Sort: Bubble Sort adalah algoritma pengurutan sederhana yang membandingkan pasangan elemen berturut-turut dan menukar posisinya jika tidak terurut. Algoritma ini berulang kali melintasi koleksi data sampai tidak ada lagi pertukaran yang perlu dilakukan.

2. Selection Sort: Selection Sort adalah algoritma pengurutan yang bekerja dengan membagi koleksi data menjadi dua bagian: bagian yang sudah terurut dan bagian yang belum terurut. Pada setiap langkah iterasi, algoritma memilih elemen terkecil dari bagian belum terurut dan menukar posisinya dengan elemen pertama dari bagian terurut.

3. Insertion Sort: Insertion Sort adalah algoritma pengurutan yang memposisikan setiap elemen pada tempat yang tepat dalam urutan yang sedang dibangun. Algoritma ini secara iteratif memilih elemen berikutnya dari koleksi data dan memasukkannya ke dalam urutan yang sedang dibangun.

C. Topological Sorting

Topological Sorting adalah proses pengurutan simpul-simpul dalam suatu graf berarah sehingga setiap simpul hanya muncul setelah simpul-simpul yang menjadi pendahulunya telah muncul. Topological Sorting umumnya digunakan dalam pemodelan dan analisis masalah yang melibatkan ketergantungan antara tugas-tugas atau kegiatan-kegiatan.

Algoritma yang sering digunakan untuk Topological Sorting adalah algoritma DFS (Depth-First Search) atau algoritma BFS (Breadth-First Search). Algoritma ini melibatkan penjelajahan graf secara rekursif atau secara melebar untuk menentukan urutan topologi. Dalam hasil pengurutan, simpul yang tidak memiliki ketergantungan atau simpul-simpul yang menjadi sumber (source) akan berada di bagian paling awal, sedangkan simpul yang menjadi tujuan (sink) akan berada di bagian paling akhir. Topological Sorting sering digunakan dalam pemodelan jaringan, perencanaan proyek, pemrograman dinamis, dan masalah-masalah optimisasi lainnya.

BAB V
DEVIDE AND CONQUER

A. Mergesort

Mergesort adalah salah satu algoritma pengurutan yang mengimplementasikan pendekatan Divide and Conquer. Algoritma ini bekerja dengan membagi koleksi data menjadi dua bagian yang lebih kecil, mengurutkan masing-masing bagian secara terpisah, dan menggabungkan kembali bagian-bagian tersebut dalam urutan yang benar.

Langkah-langkah dalam algoritma Mergesort:

1. Divide: Memisahkan koleksi data menjadi dua bagian yang seimbang.
2. Conquer: Mengurutkan masing-masing bagian secara rekursif menggunakan algoritma Mergesort.
3. Combine: Menggabungkan kembali dua bagian yang terurut menjadi satu urutan yang lengkap.

Kelebihan Mergesort:

- Mergesort memiliki kompleksitas waktu yang konsisten, yaitu O(n log n), di mana n adalah jumlah elemen dalam koleksi data.
- Algoritma ini stabil, artinya jika ada elemen dengan kunci yang sama, urutan relatif mereka akan tetap sama setelah pengurutan.
- Mergesort efisien dalam mengurutkan koleksi data yang besar atau terdistribusi secara acak.

B. Quicksort

Quicksort adalah algoritma pengurutan lainnya yang menggunakan pendekatan Divide and Conquer. Algoritma ini bekerja dengan memilih elemen tertentu sebagai "pivot", membagi koleksi data menjadi dua bagian berdasarkan pivot, dan mengurutkan kedua bagian secara terpisah.

Langkah-langkah dalam algoritma Quicksort:

1. Divide: Memilih pivot dari koleksi data.
2. Partition: Memindahkan elemen-elemen koleksi data sehingga elemen-elemen yang lebih kecil dari pivot berada di sebelah kiri pivot, dan elemen-elemen yang lebih besar berada di sebelah kanan pivot.
3. Conquer: Mengurutkan masing-masing bagian di sebelah kiri dan kanan pivot secara rekursif menggunakan algoritma Quicksort.

Kelebihan Quicksort:

- Quicksort memiliki kompleksitas waktu yang baik dengan kasus rata-rata O(n log n), di mana n adalah jumlah elemen dalam koleksi data.

- Algoritma ini bersifat in-place, yang berarti tidak memerlukan alokasi memori tambahan.
- Quicksort efisien dalam mengurutkan koleksi data yang besar atau terdistribusi secara acak.

C. Binary Tree Traversals and Related Properties

Pohon biner adalah struktur data hirarkis yang terdiri dari simpul-simpul yang terhubung melalui tepi-tepi. Binary Tree Traversals adalah proses melintasi atau mengunjungi simpul-simpul dalam pohon biner secara teratur. Terdapat tiga metode utama dalam Binary Tree Traversals, yaitu:

1. Inorder Traversal: Inorder Traversal melintasi simpul-simpul dalam urutan kiri-akar-kanan. Artinya, simpul kiri dikunjungi terlebih dahulu, kemudian simpul akar, dan terakhir simpul kanan.
2. Preorder Traversal: Preorder Traversal melintasi simpul-simpul dalam urutan akar-kiri-kanan. Artinya, simpul akar dikunjungi terlebih dahulu, kemudian simpul kiri, dan terakhir simpul kanan.
3. Postorder Traversal: Postorder Traversal melintasi simpul-simpul dalam urutan kiri-kanan-akar. Artinya, simpul kiri dikunjungi terlebih dahulu, kemudian simpul kanan, dan terakhir simpul akar.

Sifat-sifat terkait pohon biner termasuk tinggi pohon, kedalaman simpul, dan struktur keseimbangan seperti AVL Tree dan Red-Black Tree. Tinggi pohon adalah panjangnya jalur terpanjang dari akar ke daun terjauh. Kedalaman simpul adalah panjangnya jalur dari akar ke simpul tersebut. Struktur keseimbangan adalah pengaturan simpul-simpul dalam pohon biner untuk mencapai efisiensi operasi seperti pencarian, penghapusan, dan penyisipan yang seimbang secara waktu.

BAB VI
TRANSFORM AND CONQUER

A. Instance Simplification

Instance Simplification adalah salah satu teknik dalam desain algoritma yang digunakan untuk mengurangi kompleksitas suatu masalah dengan menyederhanakan instansinya. Teknik ini bertujuan untuk mengubah instansi masalah yang sulit atau kompleks menjadi instansi masalah yang lebih sederhana, sehingga memudahkan proses analisis dan pemecahan masalah.

Langkah-langkah dalam Instance Simplification:

1. Identifikasi kompleksitas: Analisis kompleksitas masalah awal dan identifikasi elemen-elemen atau faktor-faktor yang menyebabkan kompleksitas tersebut.
2. Sederhanakan instansi: Identifikasi bagian-bagian masalah yang tidak relevan atau tidak berpengaruh signifikan terhadap solusi.
3. Modifikasi instansi: Lakukan perubahan pada instansi masalah, misalnya dengan mengubah aturan atau batasan masalah menjadi lebih sederhana tanpa menghilangkan esensi permasalahan.
4. Evaluasi kompleksitas: Analisis kembali kompleksitas masalah yang telah disederhanakan dan perbandingkan dengan kompleksitas masalah awal.

Instance Simplification membantu dalam menganalisis dan memecahkan masalah dengan cara memfokuskan pada inti permasalahan dan mengurangi kompleksitas yang tidak perlu. Teknik ini dapat meningkatkan efisiensi algoritma dan mempermudah pemahaman terhadap masalah yang dihadapi.

B. Representation Change

Representation Change adalah teknik dalam desain algoritma yang melibatkan perubahan representasi atau struktur data yang digunakan untuk memodelkan masalah. Teknik ini bertujuan untuk mengubah representasi masalah menjadi bentuk yang lebih sesuai atau lebih efisien untuk dipecahkan.

Langkah-langkah dalam Representation Change:

1. Analisis representasi awal: Tinjau representasi atau struktur data yang digunakan untuk memodelkan masalah dan identifikasi kelemahan atau ketidakcocokan yang ada.
2. Identifikasi kebutuhan baru: Tentukan representasi atau struktur data baru yang lebih sesuai atau lebih efisien dalam memodelkan masalah.
3. Transformasi data: Lakukan perubahan atau transformasi pada data awal untuk memindahkan atau mengonversi ke representasi baru.
4. Evaluasi efisiensi: Analisis kembali efisiensi algoritma setelah perubahan representasi dan perbandingkan dengan representasi awal.

Representation Change membantu dalam meningkatkan efisiensi algoritma dengan memilih atau mengubah representasi yang lebih cocok atau lebih efisien untuk memodelkan masalah. Dengan representasi yang tepat, algoritma dapat bekerja lebih efisien dan menghasilkan solusi yang lebih optimal.

C. Problem Reduction

Problem Reduction adalah teknik dalam desain algoritma yang melibatkan pengurangan kompleksitas masalah dengan menghubungkannya dengan masalah yang sudah dikenal atau sudah memiliki algoritma solusi yang efisien. Teknik ini memungkinkan kita untuk menggunakan solusi yang sudah ada untuk masalah serupa atau lebih kompleks.

Langkah-langkah dalam Problem Reduction:

1. Identifikasi masalah terkait: Cari masalah yang sudah dikenal atau memiliki algoritma solusi yang efisien dan memiliki kesamaan dengan masalah yang sedang dihadapi.
2. Definisikan reduksi: Tentukan bagaimana masalah yang sedang dihadapi dapat direduksi menjadi masalah yang sudah dikenal atau memiliki solusi yang efisien.
3. Terapkan solusi yang ada: Gunakan algoritma solusi yang sudah ada untuk memecahkan masalah yang sudah direduksi.
4. Evaluasi solusi: Evaluasi efektivitas dan efisiensi solusi yang diterapkan pada masalah yang direduksi.

Problem Reduction membantu dalam memecahkan masalah yang sulit atau kompleks dengan memanfaatkan solusi yang sudah ada untuk masalah serupa yang lebih sederhana atau sudah memiliki algoritma solusi yang efisien. Teknik ini mempercepat proses pemecahan masalah dan mengurangi kerumitan yang terlibat dalam merancang algoritma baru.

BAB VII
SPACE AND TIME TRADE-OFFS

A. Sorting by Counting

Sorting by Counting adalah algoritma pengurutan yang efisien untuk mengurutkan elemen-elemen dengan rentang nilai yang terbatas. Algoritma ini bekerja dengan menghitung jumlah kemunculan setiap elemen dalam rentang nilai, kemudian membangun urutan terurut berdasarkan informasi tersebut.

Langkah-langkah dalam Sorting by Counting:

1. Menentukan rentang nilai: Identifikasi rentang nilai elemen yang akan diurutkan.
2. Menghitung kemunculan: Hitung jumlah kemunculan setiap nilai dalam rentang tersebut.
3. Menghasilkan urutan terurut: Berdasarkan jumlah kemunculan, bangun urutan terurut dari elemen-elemen yang ada.

Kelebihan Sorting by Counting:

- Algoritma ini efisien ketika rentang nilai elemen yang akan diurutkan terbatas.
- Sorting by Counting memiliki kompleksitas waktu O(n + k), di mana n adalah jumlah elemen dan k adalah rentang nilai.

B. Input Enhancement in String Matching

Input Enhancement adalah teknik yang digunakan dalam algoritma pencocokan string untuk meningkatkan efisiensi dan akurasi pencarian pola dalam teks. Teknik ini melibatkan penggunaan informasi tambahan atau struktur data yang disiapkan sebelum pencarian untuk mempercepat proses pencocokan.

Contoh teknik Input Enhancement dalam String Matching:

1. Praproses Pola (Preprocessing): Melakukan praproses pada pola pencarian sebelum memulai pencocokan. Contohnya, menggunakan algoritma seperti Knuth-Morris-Pratt (KMP) atau Boyer-Moore untuk menghasilkan informasi tambahan yang memungkinkan loncatan yang efisien dalam pencocokan.

Kelebihan Input Enhancement:

- Meningkatkan efisiensi pencarian pola dalam teks dengan memanfaatkan informasi tambahan atau struktur data yang telah disiapkan sebelumnya.
- Meminimalkan jumlah perbandingan yang harus dilakukan saat mencari pola dalam teks.

C. Hashing

Hashing adalah teknik dalam desain algoritma yang digunakan untuk mengonversi data menjadi bilangan acak yang unik, yang disebut hash code atau nilai hash. Teknik ini memetakan data ke dalam struktur data yang disebut hash table atau tabel hash, yang memungkinkan pencarian dan pengambilan data secara efisien.

Langkah-langkah dalam Hashing:

1. Menghitung nilai hash: Menggunakan fungsi hash untuk mengubah data menjadi nilai hash.
2. Penyimpanan dalam hash table: Menyimpan data dalam hash table berdasarkan nilai hash yang dihasilkan.
3. Pencarian dan pengambilan: Menggunakan nilai hash untuk mencari dan mengambil data dengan cepat dari hash table.

Kelebihan Hashing:

- Hashing memungkinkan pencarian dan pengambilan data dengan kompleksitas waktu konstan (O(1)), terlepas dari jumlah data yang ada.
- Teknik ini sangat efisien untuk operasi pencarian dan pengambilan data yang sering dilakukan.

Namun, perlu dicatat bahwa dalam hashing terdapat kemungkinan terjadinya collision (tabrakan) ketika dua data memiliki nilai hash yang sama. Untuk mengatasi collision, teknik seperti chaining atau open addressing dapat digunakan.

BAB VIII
DYNAMIC PROGRAMMING

A. Three Basic

Three Basic adalah istilah yang merujuk pada tiga algoritma dasar yang sering digunakan dalam analisis algoritma. Ketiga algoritma ini mencakup:

1. Sequential Search: Sequential Search adalah algoritma pencarian linear yang digunakan untuk mencari elemen tertentu dalam suatu himpunan data dengan memeriksa setiap elemen satu per satu secara berurutan. Algoritma ini sederhana namun memiliki kompleksitas waktu O(n), di mana n adalah jumlah elemen dalam himpunan data.

2. Binary Search: Binary Search adalah algoritma pencarian efisien yang digunakan pada himpunan data terurut. Algoritma ini membagi himpunan data menjadi dua bagian pada setiap langkahnya dan membandingkan elemen yang dicari dengan elemen tengah himpunan data. Dengan demikian, setengah himpunan data yang tidak mungkin berisi elemen yang dicari dapat dieliminasi. Algoritma ini memiliki kompleksitas waktu O(log n), di mana n adalah jumlah elemen dalam himpunan data.

3. Bubble Sort: Bubble Sort adalah algoritma pengurutan sederhana yang bekerja dengan membandingkan pasangan elemen berturut-turut dalam himpunan data dan menukar posisinya jika urutan mereka salah. Algoritma ini berulang kali melewati himpunan data hingga tidak ada lagi perubahan yang dilakukan. Bubble Sort memiliki kompleksitas waktu O(n^2), di mana n adalah jumlah elemen dalam himpunan data.

B. The Knapsack Problem and Memory Functions

The Knapsack Problem adalah masalah optimisasi kombinatorial yang melibatkan pemilihan sejumlah item dengan bobot dan nilai tertentu untuk dimasukkan ke dalam "knapsack" (tas ransel) dengan kapasitas tertentu. Tujuan dari masalah ini adalah untuk memaksimalkan nilai total item yang dimasukkan ke dalam knapsack tanpa melampaui kapasitasnya.

Dalam pemecahan The Knapsack Problem, Memory Functions digunakan untuk menyimpan informasi yang relevan saat mencari solusi optimal. Memory Functions memungkinkan penggunaan pendekatan pemrograman dinamis, di mana hasil perhitungan yang sudah dilakukan dapat digunakan kembali untuk menghindari pengulangan yang tidak perlu.

Contoh Memory Functions yang sering digunakan dalam pemecahan The Knapsack Problem adalah:

1. Value Function (Fungsi Nilai): Fungsi ini menyimpan nilai maksimum yang dapat dicapai pada setiap langkah rekursif atau iteratif dalam mencari solusi optimal.

2. Weight Function (Fungsi Bobot): Fungsi ini menyimpan bobot maksimum yang dapat dicapai pada setiap langkah rekursif atau iteratif dalam mencari solusi optimal.

3. Item Selection Function (Fungsi Pemilihan Item): Fungsi ini menyimpan informasi tentang item mana yang dipilih pada setiap langkah rekursif atau iteratif dalam mencari solusi optimal.

Dengan menggunakan Memory Functions, pemecahan The Knapsack Problem dapat dilakukan secara efisien dengan menghindari perhitungan yang redundan.

C. Warshall's and Floyd's Algorithms

Warshall's Algorithm dan Floyd's Algorithm adalah dua algoritma yang digunakan dalam teori graf untuk mencari jalur terpendek atau mencari hubungan antara semua pasangan simpul dalam graf.

1. Warshall's Algorithm: Warshall's Algorithm digunakan untuk mencari jalur terpendek antara semua pasangan simpul dalam graf berarah dengan bobot. Algoritma ini menggunakan pendekatan pemrograman dinamis untuk memperbarui matriks jarak secara iteratif. Warshall's Algorithm memiliki kompleksitas waktu O(n^3), di mana n adalah jumlah simpul dalam graf.
2. Floyd's Algorithm: Floyd's Algorithm, juga dikenal sebagai Algoritma Floyd-Warshall, digunakan untuk mencari jalur terpendek antara semua pasangan simpul dalam graf berbobot positif maupun negatif. Algoritma ini juga menggunakan pendekatan pemrograman dinamis dengan memperbarui matriks jarak secara iteratif. Floyd's Algorithm memiliki kompleksitas waktu O(n^3), di mana n adalah jumlah simpul dalam graf.

Kedua algoritma ini sangat berguna dalam memecahkan masalah jarak terpendek, seperti mencari rute terpendek dalam jaringan transportasi atau mencari lintasan terpendek dalam graf jalan.

BAB IX
GREEDY TECHNIQUE

A. Prim's Algorithm

Prim's Algorithm adalah algoritma yang digunakan untuk mencari Minimum Spanning Tree (MST) dalam sebuah graf berbobot. MST adalah himpunan subset simpul-simpul dalam graf yang terhubung secara terpadu dan memiliki bobot total minimum. Algoritma Prim bekerja dengan memilih simpul awal secara acak, kemudian secara bertahap menambahkan simpul-simpul lain yang memiliki bobot terkecil dan terhubung dengan MST yang ada.

Langkah-langkah dalam Prim's Algorithm:

1. Pilih simpul awal secara acak dan tambahkan ke MST.
2. Ulangi langkah berikut hingga semua simpul terhubung:
    - Cari sisi dengan bobot terkecil yang menghubungkan simpul dalam MST dengan simpul di luar MST.
    - Tambahkan simpul yang terhubung ke MST.
3. MST akan terbentuk ketika semua simpul terhubung.

Kelebihan dari Prim's Algorithm:

- Algoritma ini menghasilkan MST yang optimal dengan kompleksitas waktu O(V^2), di mana V adalah jumlah simpul dalam graf.

B. Kruskal's Algorithm

Kruskal's Algorithm juga digunakan untuk mencari Minimum Spanning Tree (MST) dalam sebuah graf berbobot. Algoritma Kruskal bekerja dengan mengurutkan sisi-sisi graf berdasarkan bobotnya secara tidak menurun, kemudian secara bertahap menambahkan sisi-sisi tersebut ke MST jika tidak membentuk siklus.

Langkah-langkah dalam Kruskal's Algorithm:

1. Urutkan sisi-sisi graf berdasarkan bobotnya secara tidak menurun.
2. Ulangi langkah berikut hingga MST terbentuk atau semua sisi telah diproses:
    - Ambil sisi dengan bobot terkecil yang tidak membentuk siklus dengan MST yang ada.
    - Tambahkan sisi tersebut ke MST.
3. MST akan terbentuk ketika semua simpul terhubung.

Kelebihan dari Kruskal's Algorithm:

- Algoritma ini menghasilkan MST yang optimal dengan kompleksitas waktu O(E log E), di mana E adalah jumlah sisi dalam graf.

C. Dijkstra's Algorithm

Dijkstra's Algorithm digunakan untuk mencari jalur terpendek antara satu simpul asal dengan semua simpul lain dalam graf berbobot. Algoritma ini bekerja dengan mempertimbangkan bobot lintasan terpendek yang diketahui pada setiap langkah dan secara bertahap memperbarui lintasan terpendek untuk setiap simpul.

Langkah-langkah dalam Dijkstra's Algorithm:

1. Tetapkan bobot awal dari simpul asal ke 0 dan bobot awal dari simpul lainnya menjadi tak terbatas.
2. Ulangi langkah berikut hingga semua simpul telah dikunjungi:
    - Pilih simpul dengan bobot terkecil yang belum dikunjungi.
    - Perbarui bobot lintasan terpendek untuk semua simpul yang terhubung dengan simpul tersebut.
3. Lintasan terpendek dari simpul asal ke semua simpul lainnya akan terbentuk.

Kelebihan dari Dijkstra's Algorithm:

- Algoritma ini menghasilkan jalur terpendek dari simpul asal ke semua simpul lainnya dengan kompleksitas waktu O(V^2), di mana V adalah jumlah simpul dalam graf.

D. Huffman Trees and Codes

Huffman Trees and Codes adalah metode kompresi data yang efisien yang menggunakan pohon Huffman untuk mengurangi ukuran data. Algoritma Huffman berdasarkan pada prinsip bahwa karakter atau simbol yang paling sering muncul dalam data harus diwakili dengan kode yang lebih pendek, sedangkan karakter yang jarang muncul diwakili dengan kode yang lebih panjang.

Langkah-langkah dalam Huffman Trees and Codes:

1. Hitung frekuensi kemunculan setiap karakter dalam data.
2. Membangun pohon Huffman:
    - Buat simpul untuk setiap karakter dengan frekuensi kemunculan dan tambahkan ke daftar prioritas.
    - Ambil dua simpul dengan frekuensi terkecil dari daftar prioritas dan gabungkan mereka menjadi satu simpul dengan bobot yang merupakan jumlah frekuensi kedua simpul tersebut.
    - Tambahkan simpul baru ke daftar prioritas.
    - Ulangi langkah sebelumnya hingga hanya ada satu simpul di dalam daftar prioritas.
3. Assign kode biner untuk setiap karakter berdasarkan jalur dari akar pohon Huffman ke simpul karakter.

4. Kompresi data dengan menggantikan setiap karakter dalam data dengan kode Huffman yang sesuai.
5. Untuk mendekompresi data, gunakan pohon Huffman untuk menerjemahkan kode Huffman kembali menjadi karakter asli.

Kelebihan dari Huffman Trees and Codes:

- Algoritma Huffman menghasilkan kompresi data yang optimal, di mana karakter yang sering muncul direpresentasikan dengan kode yang lebih pendek dan karakter yang jarang muncul direpresentasikan dengan kode yang lebih panjang.

BAB X
ITERATIVE IMPROVEMENT

A. Metode Simpleks

Metode Simpleks adalah algoritma yang digunakan untuk menyelesaikan masalah pemrograman linier, yang melibatkan memaksimalkan atau meminimalkan fungsi tujuan linier dengan batasan linier. Metode ini bekerja dengan cara memperbaiki solusi yang layak secara iteratif hingga mencapai solusi optimal. Metode ini mengeksplorasi sudut-sudut yang memenuhi batasan untuk menemukan solusi optimal.

Langkah-langkah dalam Metode Simpleks:

1. Mengubah masalah pemrograman linier ke dalam bentuk standar dengan memperkenalkan variabel kelonggaran dan menulis semua ketimpangan sebagai persamaan.
2. Menyiapkan tabel simpleks awal, yang mencakup fungsi tujuan dan batasan-batasan.
3. Memilih kolom pivot berdasarkan koefisien paling negatif dalam baris fungsi tujuan.
4. Memilih baris pivot dengan mencari rasio minimum nonnegatif antara sisi kanan dengan kolom pivot yang sesuai.
5. Melakukan operasi baris untuk membuat elemen pivot menjadi 1 dan elemen lain dalam kolom pivot menjadi 0.
6. Memperbarui tabel simpleks dan mengulangi langkah-langkah 3-6 hingga mencapai solusi optimal.

Kelebihan Metode Simpleks:

- Metode Simpleks efisien dalam menyelesaikan masalah pemrograman linier dengan jumlah variabel dan batasan yang besar.
- Metode ini menjamin konvergensi ke solusi optimal.

B. Masalah Aliran Maksimum (Maximum-Flow Problem)

Masalah Aliran Maksimum adalah masalah di mana kita ingin menentukan aliran maksimum yang dapat mengalir dari satu simpul ke simpul lain dalam graf terarah. Setiap sisi dalam graf memiliki kapasitas yang menentukan jumlah maksimum aliran yang dapat melewati sisi tersebut. Tujuan dari masalah ini adalah untuk menemukan aliran maksimum dari sumber ke tujuan yang memenuhi batasan kapasitas sisi.

Algoritma Ford-Fulkerson adalah algoritma yang umum digunakan untuk memecahkan masalah aliran maksimum. Algoritma ini bekerja dengan menemukan jalur meningkat dalam graf dan meningkatkan aliran di sepanjang jalur tersebut hingga mencapai aliran maksimum.

Keuntungan Masalah Aliran Maksimum:

- Solusi dari masalah aliran maksimum dapat diterapkan dalam berbagai konteks, seperti pengaturan lalu lintas, pengiriman data, dan perencanaan produksi.

C. Maximum Matching in Bipartite Graphs

Penjajaran Maksimum dalam Graf Bipartit adalah masalah mencari himpunan sisi terbesar di dalam graf bipartit di mana setiap simpul di salah satu partisi terhubung dengan paling banyak satu simpul di partisi lain. Tujuan dari masalah ini adalah untuk menemukan pasangan terbesar antara dua partisi dalam graf bipartit.

Algoritma yang sering digunakan untuk menyelesaikan masalah penjajaran maksimum adalah Algoritma Augmenting Path atau Algoritma Berbasis Meningkat. Algoritma ini bekerja dengan menemukan jalur peningkatan dalam graf bipartit dan memperbarui penjajaran maksimum dengan menambahkan pasangan baru dalam jalur tersebut.

Keuntungan Penjajaran Maksimum dalam Graf Bipartit:

- Solusi dari masalah ini dapat diterapkan dalam berbagai konteks, seperti penugasan pekerjaan, penugasan kursus, atau pasangan dalam jaringan sosial.

D. The Stable Marriage Problem

Algoritma Gale-Shapley adalah algoritma yang sering digunakan untuk memecahkan masalah pernikahan stabil. Algoritma ini bekerja dengan mempertimbangkan preferensi setiap individu dan secara iteratif mencocokkan mereka berdasarkan preferensi mereka hingga mencapai penjajaran stabil.

Keuntungan The Stable Marriage Problem:

- Masalah ini mencerminkan situasi kehidupan nyata, seperti pencocokan rumah sakit dengan mahasiswa kedokteran atau pencocokan pasangan hidup yang cocok berdasarkan preferensi mereka.

BAB XI
LIMITATIONS OF ALGORITHM POWER

A. Lower-Bound Arguments

Lower-Bound Arguments adalah teknik yang digunakan dalam analisis algoritma untuk menentukan batas bawah (lower bound) dari kompleksitas waktu yang diperlukan untuk memecahkan suatu masalah. Teknik ini bertujuan untuk menunjukkan bahwa tidak mungkin ada algoritma yang lebih efisien daripada yang sudah ada untuk memecahkan masalah tersebut.

Dalam lower-bound arguments, umumnya digunakan metode reduksi untuk membuktikan bahwa suatu masalah tertentu memiliki kompleksitas waktu yang minimal. Dengan kata lain, jika kita dapat menunjukkan bahwa masalah lain yang telah terbukti sulit dikurangi ke masalah yang sedang kita analisis, maka kita dapat menyimpulkan bahwa masalah tersebut memiliki batas bawah yang sulit dipecahkan.

B. Decision Tree

Decision Tree (Pohon Keputusan) adalah struktur data yang digunakan untuk memodelkan pengambilan keputusan berdasarkan serangkaian aturan atau kondisi. Setiap simpul dalam decision tree mewakili suatu keputusan atau kondisi, sedangkan setiap cabang yang keluar dari simpul mewakili kemungkinan hasil atau perubahan keadaan.

Decision tree dapat digunakan dalam berbagai masalah, seperti klasifikasi data, prediksi, atau pengambilan keputusan. Algoritma pembangunan decision tree mencoba untuk memilih aturan yang paling informatif dan membagi data ke dalam kelompok yang paling homogen.

C. P, NP, dan NP-Complete Problems

P, NP, dan NP-Complete adalah kelas masalah yang didefinisikan dalam teori kompleksitas komputasional.

- Kelas P (Polynomial Time) adalah kelas masalah yang dapat diselesaikan dalam waktu polinomial oleh sebuah algoritma deterministik. Ini berarti algoritma tersebut memiliki kompleksitas waktu yang dapat dibatasi oleh fungsi polinomial dari ukuran input.
- Kelas NP (Nondeterministic Polynomial Time) adalah kelas masalah yang memiliki sertifikat verifikasi yang dapat diverifikasi dalam waktu polinomial oleh sebuah algoritma deterministik. Dalam hal ini, algoritma verifikasi mengonfirmasi kebenaran solusi yang diajukan oleh algoritma lain dalam waktu yang efisien.
- Masalah NP-Complete adalah subset dari masalah dalam kelas NP yang dianggap paling sulit. Jika ada algoritma yang dapat menyelesaikan masalah NP-Complete dalam waktu polinomial, maka itu berarti semua masalah dalam kelas NP dapat diselesaikan dalam waktu polinomial, yang akan membuktikan bahwa P = NP.

D. Tantangan Algoritma Numerik

Algoritma Numerik mengacu pada pengembangan algoritma untuk menyelesaikan masalah matematika yang melibatkan operasi numerik. Tantangan utama dalam pengembangan algoritma numerik adalah memastikan keakuratan dan kestabilan hasil numerik yang dihasilkan.

Beberapa tantangan yang dihadapi oleh algoritma numerik meliputi:

- Kesalahan pembulatan: Operasi aritmatika pada komputer menggunakan representasi bilangan yang terbatas, sehingga dapat menghasilkan kesalahan pembulatan yang dapat menumpuk seiring dengan jumlah operasi yang dilakukan.
- Ketidakstabilan numerik: Beberapa algoritma numerik dapat menjadi tidak stabil, artinya hasilnya sangat sensitif terhadap perubahan kecil pada data masukan.
- Efisiensi komputasi: Algoritma numerik seringkali harus memproses volume data yang besar dengan waktu eksekusi yang wajar, sehingga efisiensi komputasi menjadi faktor penting.

Dalam pengembangan algoritma numerik, penting untuk mempertimbangkan trade-off antara akurasi dan efisiensi, serta memastikan bahwa algoritma yang digunakan mampu menangani masalah numerik dengan baik dalam berbagai situasi.

BAB XII
COPING WITH THE LIMITATIONS OF ALGORITHM POWER

A. Backtracking

Backtracking adalah salah satu teknik algoritma yang digunakan untuk mencari solusi secara sistematis dengan menguji semua kemungkinan langkah-langkah yang mungkin. Algoritma backtracking berusaha untuk menemukan solusi dengan melakukan pencarian secara rekursif, dan jika suatu langkah tidak mengarah pada solusi yang valid, maka algoritma akan mundur (backtrack) dan mencoba langkah alternatif.

Keuntungan dari algoritma backtracking adalah kemampuannya untuk menemukan solusi secara eksploratif tanpa harus mempertimbangkan semua kemungkinan langkah di awal. Namun, kelemahan dari algoritma ini adalah kompleksitas waktu yang dapat menjadi sangat tinggi jika ruang pencarian yang dihasilkan sangat besar.

B. Branch and Bound

Branch and Bound adalah teknik algoritma yang digunakan untuk menyelesaikan masalah optimasi dengan cara membagi masalah menjadi submasalah yang lebih kecil (branching) dan membatasi pencarian pada submasalah yang memiliki potensi untuk menghasilkan solusi optimal (bounding).

Algoritma branch and bound bekerja dengan cara membangun pohon pencarian, di mana setiap simpul mewakili suatu submasalah. Pada setiap langkah, algoritma mengambil keputusan untuk membagi masalah menjadi submasalah yang lebih kecil, kemudian memperkirakan batas bawah (lower bound) dari solusi optimal untuk setiap submasalah. Jika batas bawah dari suatu submasalah lebih buruk daripada solusi yang telah ditemukan sejauh ini, maka submasalah tersebut dapat diabaikan (pruning).

Keuntungan dari algoritma branch and bound adalah kemampuannya untuk membatasi ruang pencarian dengan memperkirakan batas bawah solusi optimal. Hal ini dapat menghemat waktu dan sumber daya yang diperlukan untuk mencari solusi secara eksploratif. Namun, kelemahan dari algoritma ini adalah tergantung pada kualitas estimasi batas bawah yang digunakan.

C. Algoritma untuk Memecahkan Masalah Nonlinear

Algoritma untuk memecahkan masalah nonlinear berfokus pada pencarian solusi dari persamaan atau fungsi yang memiliki relasi nonlinier. Algoritma ini berbeda dengan algoritma untuk masalah linear yang memiliki relasi linier.

Ada beberapa pendekatan yang umum digunakan dalam algoritma untuk memecahkan masalah nonlinear, di antaranya:

- Metode Newton-Raphson: Metode iteratif yang menggunakan turunan dari persamaan untuk memperbaiki perkiraan solusi.
- Metode Gradien: Metode yang mengoptimalkan fungsi dengan mengikuti gradien atau turunan parsialnya.

- Metode Dekomposisi: Pendekatan yang membagi masalah menjadi submasalah yang lebih kecil untuk mempermudah pencarian solusi.

Setiap algoritma memiliki kelebihan dan kelemahan tergantung pada karakteristik masalah yang dihadapi. Pemilihan algoritma yang tepat untuk masalah nonlinear sangat penting untuk mencapai solusi yang akurat dan efisien.