RESUME

PERANCANGAN DAN

ANALISIS ALGORITMA

Disusun Oleh :

Hayatun Nupus

(21346011)

Lecturer:

Widya Darwin S.Pd,.M.Pd.T

PROGRAM STUDI S1 TEKNIK INFORMATIKA

FAKULTAS TEKNIK

UNIVERSITAS NEGERI PADANG

TAHUN 2023

DAFTAR ISI

A. Pengantar Analisis Algoritma

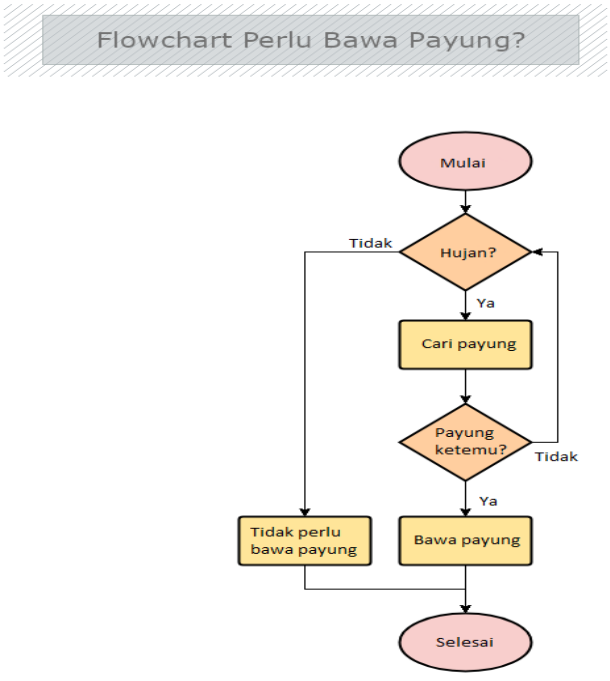
a. Sejarah dan Defenisi Algoritma

Algoritma berasal dari kata "algorismus" yang berasal dari nama seorang matematikawan Persia, Al-Khwarizmi. Al-Khwarizmi hidup pada abad ke-9 dan merupakan salah satu matematikawan dan ahli astronomi terkemuka pada masanya. Dia membuat kontribusi penting dalam matematika dan sains, termasuk penyebaran sistem angka Hindu-Arab dan pengembangan metode untuk menyelesaikan persamaan linear dan kuadrat.

Algoritma adalah urutan langkah-langkah terdefinisi yang diikuti untuk menyelesaikan masalah tertentu atau mencapai tujuan yang diinginkan. Algoritma digunakan dalam berbagai bidang, termasuk matematika, komputer, ilmu data, dan pemrograman.

Dalam bidang komputer, algoritma sangat diperlukan dalam menyelesaikan berbagai masalah pemrograman, terutama dalam komputasi numeris. Tanpa algoritma yang dirancang baik maka proses pemrograman akan menjadi salah, rusak, atau lambat dan tidak efisien.

b. Algoritma dalam kehidupan sehari-hari



c. Sifat-sifat dan Notasi Algoritma

- Finite: Algoritma harus berhenti setelah mengerjakan sejumlah langkah terbatas
- Definite: setiap langlah didefenisikan secara tepat, tidak boleh membingungkan (ambigu)
- Input: sebuah algoritma memiliki satu/lebih input sebelum dijalankan
- Output: algoritma memiliki satu/lebih output, yang biasanya bergantung kepada input
- Effective: setiap algoritma diharapkan memiliki sifat efektif (setiao langkah harus sederhana dan sehingga dapat dikerjakan dalam waktu yang masuk akal)

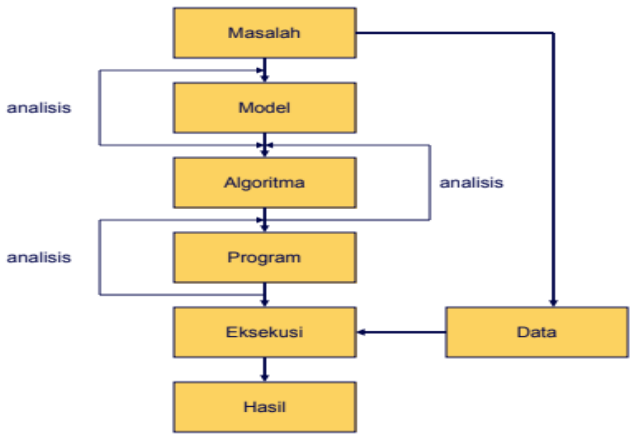
Notasi Algoritma

- Penulisan algoritma tidak tergantung dari spesifikasi bahasa pemrograman dan komputer yang mengeksekusinya
- Notasi algoritma bukan notasi bahasa pemrograman tetapi dapat diterjemahkan ke dalam berbagai bahasa pemrograman

d. Peranan dan Tahap Penyelesaian Masalah Algoritma

- Peran algoritma : fundamental (tidak ada algoritma tidak ada program)
- Algoritma + struktur data = program
  - Struktur data :teknik/cara penyusunan/ penyimpanan data dalam komputer
  - " memori seminim mungkin dan kecepatan eksekusi semaksimal mungkin"

Tahap penyelesaian masalah



e. Penulisan Algoritma

- **Pseudocode**

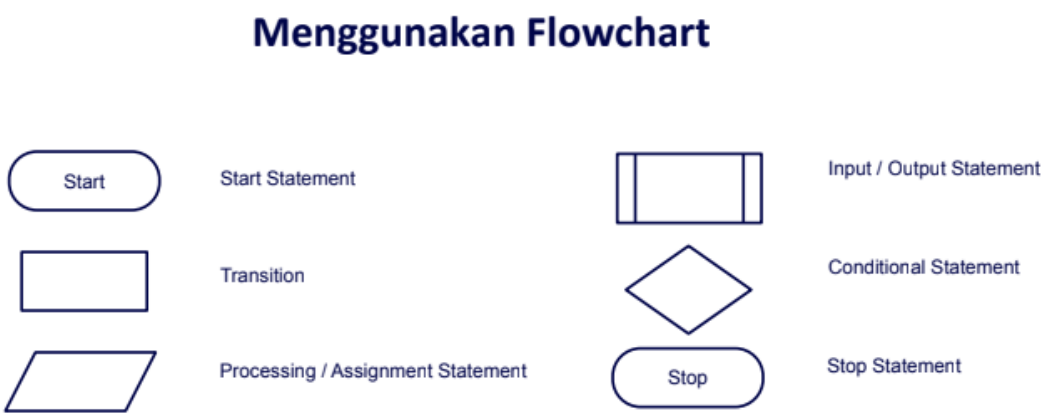
  Representasi informal dari algoritma yang menggunakan kombinasi antara bahasa manusia dan sintaks yang mirip dengan kode pemrograman. Pseudocode membantu dalam memberikan pemahaman yang lebih jelas tentang langkah-langkah yang harus diikuti dalam algoritma.

- **Diagram Alir**

  Representasi grafis dari algoritma yang menggunakan simbol-simbol dan panah untuk menggambarkan langkah-langkah dan hubungannya. Simbol-simbol yang umum digunakan dalam diagram alir antara lain persegi (untuk langkah-langkah), panah (untuk menghubungkan langkah-langkah), dan berlian (untuk pengambilan keputusan).

- **Bahasa pemrograman**

  Algoritma juga dapat ditulis dalam bahasa pemrograman yang spesifik. Ini dapat berguna jika algoritma akan diimplementasikan dalam suatu program komputer.



f. Design dan Analisi Algoritma

Design Algoritma

- **Pemahaman Masalah**

  Langkah pertama dalam design algoritma adalah memahami dengan baik permasalahan yang ingin dipecahkan. Identifikasi masukan, keluaran, dan batasan yang ada. Pahami tujuan utama dan kendala yang perlu dipertimbangkan dalam perancangan algoritma.

- **Identifikasi Pendekatan**

  Setelah memahami masalah, identifikasi pendekatan yang mungkin digunakan untuk menyelesaikannya. Ada berbagai pendekatan yang dapat digunakan, seperti pemecahan langsung (brute force), pendekatan berbasis aturan (greedy approach), pendekatan pemecahan masalah dengan pembagian dan penaklukan (divide and conquer), atau pendekatan berbasis graf (graph-based approach).

- **Pseudocode atau Diagram Alir**

  Tulis algoritma dalam bentuk pseudocode atau diagram alir. Gunakan bahasa yang mudah dipahami dan representasi visual yang jelas. Deskripsikan langkah-langkah secara terperinci dengan mengikuti logika dan urutan yang tepat. Jelaskan kondisi, pengulangan, dan pengambilan keputusan yang relevan.

- **Pemilihan Struktur Data dan Algoritma**

  Pilih struktur data yang tepat dan algoritma yang cocok untuk mencapai tujuan algoritma. Pertimbangkan kompleksitas waktu dan penggunaan memori dari masing-masing pilihan untuk memilih yang paling efisien.

- **Analisis Algoritma**

  Setelah merancang algoritma, lakukan analisis kinerja algoritma. Evaluasi kompleksitas waktu (time complexity) dan penggunaan memori (space complexity) algoritma tersebut. Identifikasi faktor-faktor yang dapat mempengaruhi kinerja algoritma dan pertimbangkan apakah perlu dilakukan optimasi.

Analisis Algoritma

- **Mengidentifikasi kompleksitas waktu dan ruang**

  Hitung jumlah langkah atau penggunaan memori algoritma dalam kaitannya dengan ukuran input yang diberikan.

- **Menggunakan notasi Big O**

  Gunakan notasi Big O untuk menggambarkan pertumbuhan algoritma seiring dengan peningkatan ukuran input.

- **Membandingkan algoritma**

  Bandingkan kinerja algoritma yang berbeda dalam situasi yang sama untuk menentukan algoritma mana yang lebih efisien.

- **Melakukan uji coba dan pengujian**

  Uji algoritma menggunakan berbagai kasus uji untuk mengamati kinerja algoritma dalam situasi nyata.

- **Melakukan optimasi (jika diperlukan)**

  Jika analisis mengungkapkan kekurangan dalam efisiensi algoritma, pertimbangkan untuk melakukan optimasi untuk meningkatkan kinerja.

B. Dasar-Dasar Efesiensi Algoritma

a. Pengertian dan Dasar Efisiensi Algortima

Efisiensi algoritma adalah kemampuan suatu algoritma untuk menjalankan tugasnya dengan waktu dan sumber daya yang optimal.

Dasar-dasar efisiensi algoritma:

1. Notasi O (Big O Notation)

Notasi O digunakan untuk menggambarkan kompleksitas waktu atau ruang dari sebuah algoritma. Notasi ini memberikan perkiraan seberapa cepat waktu eksekusi atau seberapa banyak ruang yang dibutuhkan oleh algoritma saat jumlah inputnya meningkat. Notasi O biasanya digunakan untuk menggambarkan kinerja algoritma dalam kasus terburuk (worst-case scenario). Contohnya, O(1), O(log n), O(n), O(n log n), O(n^2), dan O(2^n).

2. Waktu Eksekusi

Waktu eksekusi mengukur seberapa lama sebuah algoritma membutuhkan waktu untuk menyelesaikan tugasnya. Waktu eksekusi dapat diukur dalam satuan waktu seperti detik atau milidetik, atau dalam jumlah langkah-langkah operasi yang dieksekusi oleh algoritma.

3. Ruang Memori

Ruang memori mengukur seberapa banyak ruang yang diperlukan oleh algoritma untuk menyimpan dan memanipulasi data saat menjalankan tugasnya. Ruang memori dapat diukur dalam satuan seperti byte atau kilobyte.

4. Kompleksitas Waktu dan Ruang

Kompleksitas waktu mengukur seberapa efisien sebuah algoritma dalam hal waktu eksekusi, sedangkan kompleksitas ruang mengukur seberapa efisien algoritma dalam hal penggunaan ruang memori. Biasanya, algoritma dengan kompleksitas waktu atau ruang yang lebih rendah dianggap lebih efisien.

5. Pengoptimalan Algoritma

Pengoptimalan algoritma merupakan teknik yang digunakan untuk meningkatkan efisiensi sebuah algoritma. Hal ini dapat dilakukan dengan melakukan perubahan pada langkah-langkah algoritma, memanfaatkan struktur data yang lebih efisien, atau mengurangi operasi yang tidak perlu.

6. Analisis Kasus Terburuk (Worst-Case Analysis)

Dalam menganalisis efisiensi algoritma, penting untuk mempertimbangkan kasus terburuk (worst-case scenario), yaitu kondisi yang menghasilkan waktu eksekusi atau penggunaan ruang memori yang paling tinggi. Dengan memahami kasus terburuk, kita dapat menghindari kejadian di mana algoritma tidak berperforma dengan baik dalam situasi tertentu.

b. Pengukuran input untuk jangkauan operasi yang dilakukan

1. Ukuran Data

Ukuran data merujuk pada jumlah data yang digunakan sebagai masukan untuk operasi algoritma. Ini dapat diukur dalam berbagai unit seperti byte, kilobyte, megabyte, atau jumlah entitas data (misalnya, jumlah elemen dalam array). Ukuran data dapat mempengaruhi kompleksitas waktu dan ruang algoritma.

2. Panjang String

Jika operasi algoritma melibatkan pengolahan string, maka panjang string dapat dianggap sebagai ukuran input. Panjang string dapat diukur dalam jumlah karakter atau byte.

3. Jumlah Iterasi

Jumlah iterasi atau jumlah langkah yang diperlukan untuk menyelesaikan operasi algoritma juga dapat digunakan sebagai ukuran input. Ini umumnya relevan untuk algoritma berulang, seperti perulangan atau rekursi.

4. Ukuran Graf atau Struktur Data

Jika operasi algoritma melibatkan graf atau struktur data kompleks, ukuran graf atau struktur data tersebut dapat dijadikan ukuran input. Misalnya, dalam algoritma graf, ukuran input dapat dinyatakan dalam jumlah simpul dan jumlah tepi dalam graf.

5. Skala Masalah

Skala masalah mengacu pada tingkat kesulitan atau kompleksitas masalah yang ingin diselesaikan oleh operasi algoritma. Ini mungkin tidak berkaitan langsung dengan ukuran data, tetapi dapat mempengaruhi kompleksitas algoritma secara keseluruhan. Misalnya, dalam algoritma pemrosesan citra, ukuran citra mungkin sama, tetapi skala masalah dapat berbeda tergantung pada jenis operasi yang dilakukan.

c. Notasi analisis efesiensi algoritma

1. O(1) – Konstanta

Algoritma dengan kompleksitas O(1) memiliki waktu eksekusi konstan yang tidak bergantung pada ukuran masukan. Artinya,

algoritma ini memiliki kinerja yang tetap, terlepas dari ukuran masukan.

2. O(n) – Linier

Algoritma dengan kompleksitas O(n) memiliki waktu eksekusi yang tumbuh secara linier seiring dengan ukuran masukan. Artinya, waktu eksekusi algoritma meningkat secara proporsional dengan ukuran masukan. Contohnya adalah perulangan tunggal yang melintasi semua elemen dalam array.

3. O(n^2) – Kuadratik

Algoritma dengan kompleksitas O(n^2) memiliki waktu eksekusi yang tumbuh secara kuadratik seiring dengan ukuran masukan. Artinya, waktu eksekusi algoritma meningkat secara kuadratik dengan ukuran masukan. Contohnya adalah perulangan bersarang yang melibatkan dua tingkat perulangan.

4. O(log n) – Logaritmik

Algoritma dengan kompleksitas O(log n) memiliki waktu eksekusi yang tumbuh secara logaritmik seiring dengan ukuran masukan. Artinya, waktu eksekusi algoritma meningkat secara logaritmik saat ukuran masukan meningkat. Contohnya adalah algoritma pencarian biner di mana setiap langkah operasinya membagi ukuran masukan menjadi setengahnya.

5. O(n log n) - N log N

Algoritma dengan kompleksitas O(n log n) memiliki waktu eksekusi yang tumbuh dengan perkalian dari ukuran masukan dan logaritma ukuran masukan. Ini adalah kompleksitas yang umum ditemukan dalam algoritma pengurutan seperti Merge Sort dan Quick Sort.

d. Algoritma Rekursif dan Non Rekrusif

1. Algoritma Rekursif

Algoritma rekursif adalah algoritma yang memecahkan masalah dengan memanggil dirinya sendiri. Pada setiap iterasi, algoritma rekursif menguraikan masalah menjadi submasalah yang lebih kecil dan kemudian memanggil dirinya sendiri untuk memecahkan submasalah tersebut. Proses ini terus berlanjut hingga

mencapai kasus dasar atau kondisi terminasi yang menghentikan rekursi. Contoh algoritma rekursif termasuk algoritma pengurutan seperti Merge Sort atau algoritma pencarian seperti Pencarian Biner.

2. Algoritma Non-rekursif

Algoritma non-rekursif adalah algoritma yang memecahkan masalah tanpa menggunakan pemanggilan diri sendiri. Algoritma non-rekursif biasanya menggunakan perulangan dan struktur pengulangan seperti loop for, while, atau do-while untuk mengulangi operasi secara berulang hingga mencapai solusi. Algoritma non-rekursif umumnya lebih efisien secara waktu dan memori daripada algoritma rekursif. Contoh algoritma non-rekursif termasuk Bubble Sort, Insertion Sort, atau Linear Search.

C. Algoritma Brute-Force Exhaustive Search

a. Pengertian Brute-Force

Brute-force merupakan metode atau pendekatan dalam pemecahan masalah yang mencoba semua kemungkinan solusi secara langsung untuk mencari solusi yang benar. Pendekatan ini tidak menggunakan strategi atau optimasi yang kompleks, melainkan secara sistematis menjelajahi semua opsi yang mungkin untuk menemukan solusi yang diinginkan.

- Keuntungan dari pendekatan brute-force adalah dapat menjamin solusi yang benar, karena mencoba semua kemungkinan solusi secara eksplisit.
- Kelemahan utama dari pendekatan brute-force adalah efisiensi. Ketika masalah memiliki ukuran atau ruang solusi yang besar, pendekatan brute-force dapat menjadi sangat tidak efisien.

b. Algoritma selection sorts and buuble sort

1. Selection Sort:

- Algoritma Selection Sort bekerja dengan memilih elemen terkecil dari sisa array yang belum diurutkan dan menukar posisinya dengan elemen pertama dalam array yang belum diurutkan.
- Iterasi berlanjut hingga semua elemen dalam array terurut.

- Pada setiap iterasi, elemen terkecil ditemukan dengan membandingkan elemen-elemen yang belum diurutkan satu per satu.
- Setelah elemen terkecil ditemukan, elemen tersebut ditukar dengan elemen pertama dalam array yang belum diurutkan.
- Proses ini berlanjut sampai semua elemen terurut dan array menjadi terurut secara keseluruhan.

Analisis Selection Sort:

- Waktu: Algoritma Selection Sort memiliki kompleksitas waktu rata-rata O(n^2), di mana n adalah jumlah elemen dalam array.
- Waktu terbaik dan terburuk sama-sama O(n^2), karena dalam setiap iterasi, algoritma perlu membandingkan setiap elemen yang belum diurutkan dengan elemen lainnya.
- Ruang: Algoritma Selection Sort memiliki kompleksitas ruang O(1), karena hanya memerlukan sedikit ruang tambahan untuk menyimpan variabel bantu saat pertukaran elemen.

2. Bubble Sort:

- Algoritma Bubble Sort bekerja dengan membandingkan dua elemen bersebelahan dalam array dan menukar posisinya jika mereka tidak terurut.
- Iterasi berlanjut sepanjang array hingga tidak ada lagi pertukaran yang dilakukan pada satu iterasi.
- Pada setiap iterasi, elemen-elemen bersebelahan dibandingkan satu per satu, dan jika elemen berikutnya lebih kecil, mereka ditukar.
- Proses ini berlanjut sampai tidak ada lagi pertukaran yang dilakukan, menandakan bahwa array telah terurut secara keseluruhan.

Analisis Bubble Sort:

- Waktu: Algoritma Bubble Sort memiliki kompleksitas waktu rata-rata O(n^2), di mana n adalah jumlah elemen dalam array.
- Waktu terbaik adalah O(n) ketika array sudah terurut dan tidak ada pertukaran yang dilakukan pada iterasi tambahan.
- Waktu terburuk adalah O(n^2) ketika array diurutkan dalam urutan terbalik, sehingga setiap elemen harus ditukar pada setiap iterasi.

- Ruang: Algoritma Bubble Sort memiliki kompleksitas ruang O(1), karena hanya memerlukan sedikit ruang tambahan untuk menyimpan variabel bantu saat pertukaran elemen.

c. Algoritma sequential search menggunakan metode bruteforce

Algoritma sequential search adalah algoritma pencarian sederhana yang digunakan untuk mencari elemen tertentu dalam sebuah array atau struktur data linear.

Langkah-langkah:

1. Menerima input berupa array atau struktur data yang akan dicari elemennya, serta elemen yang ingin dicari (target).
2. Mulai dari elemen pertama dalam array, periksa setiap elemen satu per satu secara berurutan.
3. Bandingkan setiap elemen dengan elemen yang ingin dicari (target). Jika ada kecocokan, maka elemen tersebut ditemukan.
4. Jika tidak ada kecocokan pada elemen saat ini, lanjutkan ke elemen berikutnya dan ulangi langkah 3.
5. Jika semua elemen telah diperiksa dan tidak ada kecocokan, maka elemen tidak ditemukan dalam array atau struktur data.

d. Algoritma searching berpasangan menggunakan metode closest-pair and convexhull problem

Algoritma searching berpasangan menggunakan metode closest-pair dan convex hull problem adalah dua algoritma yang digunakan untuk mencari pasangan elemen dalam kumpulan data berdasarkan jarak terdekat atau bentuk poligon tertutup yang melingkupi kumpulan data tersebut.

Algoritma Closest-Pair digunakan untuk mencari pasangan dua elemen dengan jarak terdekat dalam kimpulan data. Algoritma convex Hull digunakan untuk mencari poligen tertutup terkecil yang melingkupi seluruh kumpulan data.

e. Algoritma exhaustive search travel solesman problem (TSO)

Algoritma exhaustive search (pencarian menyeluruh) digunakan dalam masalah Traveling Salesman Problem (TSP) untuk menentukan jalur terpendek yang melintasi semua titik dengan jarak minimal. TSP adalah masalah yang kompleks dalam

teori graf yang mencari jalur terpendek yang melibatkan mengunjungi semua titik dalam graf (misalnya, kota-kota) dengan kembali ke titik awal.

f. Algoritma depth-first search (DFS) and Breadth-first search (BFS)

1 Depth-First Search (DFS)

DFS adalah algoritma pencarian yang beroperasi dengan cara melintasi graf secara vertikal atau mengunjungi simpul secara mendalam sebelum menjelajahi simpul lain yang terhubung dengannya.

Algoritma DFS menggunakan pendekatan berbasis stack (tumpukan) untuk menjaga urutan simpul yang akan dikunjungi.

Langkah-langkah DFS:

- Pilih simpul awal dan tandai sebagai "dikunjungi".
- Jika ada tetangga yang belum dikunjungi, pilih salah satu dari mereka dan ulangi langkah ini untuk simpul tetangga tersebut.
- Jika tidak ada tetangga yang belum dikunjungi, kembali ke simpul sebelumnya dan lanjutkan ke tetangga lain yang belum dikunjungi.
- Teruskan langkah-langkah di atas hingga semua simpul terkunjungi atau sampai mencapai kondisi berhenti yang ditentukan.

2 **Breadth-First Search (BFS)**

BFS adalah algoritma pencarian yang beroperasi dengan cara melintasi graf secara horizontal atau mengunjungi simpul secara melebar sebelum menjelajahi simpul lain yang terhubung dengan simpul tersebut.

Algoritma BFS menggunakan pendekatan berbasis queue (antrian) untuk menjaga urutan simpul yang akan dikunjungi.
Langkah-langkah BFS:

- Pilih simpul awal dan tandai sebagai "dikunjungi".
- Tambahkan simpul awal ke dalam antrian.
- Selama antrian tidak kosong, ambil simpul pertama dari antrian dan periksa tetangga-tetangganya.
- Jika tetangga belum dikunjungi, tandai mereka sebagai "dikunjungi" dan tambahkan ke dalam antrian.
- Teruskan langkah-langkah di atas hingga antrian kosong atau sampai mencapai kondisi berhenti yang ditentukan.

D. Metode Decrease and Conquer

a. Pengertian metode Decrease and Conquer

Metode Decrease and Conquer (menurunkan dan menaklukkan) adalah salah satu pendekatan dalam desain algoritma yang digunakan untuk memecahkan masalah dengan cara memperkecil ukuran masalah awal menjadi masalah yang lebih kecil dan kemudian menyelesaikan masalah yang lebih kecil tersebut secara rekursif atau dengan pendekatan lainnya.

b. Algoritma dengan varian dari metode decrease and conquer dalam melakukan pengurutan elemen acak

Salah satu varian dari metode Decrease and Conquer yang digunakan dalam pengurutan elemen acak adalah algoritma Quicksort. Quicksort adalah algoritma pemisahan (partitioning) dan penaklukkan (conquer) yang berbasis pembanding (comparison-based) untuk mengurutkan elemen dalam sebuah array.

Berikut adalah langkah-langkah umum dalam algoritma Quicksort:

- Pilih elemen pivot: Pilih salah satu elemen dalam array sebagai pivot. Pivot ini akan digunakan untuk membagi array menjadi dua bagian.
- Partitioning: Susun ulang elemen-elemen array sehingga elemen-elemen yang lebih kecil daripada pivot berada di sebelah kiri pivot, dan elemen-elemen yang lebih besar berada di sebelah kanan pivot. Pada tahap ini, pivot akan berada pada posisi yang tepat dalam array yang terurut.
- Rekursif: Terapkan langkah-langkah Quicksort secara rekursif pada kedua bagian array yang dihasilkan dari tahap partitioning, yaitu bagian sebelah kiri pivot dan bagian sebelah kanan pivot. Teruskan proses rekursif hingga seluruh array terurut.
- Combine: Karena Quicksort melakukan pemisahan dan penggabungan pada tahap partitioning, tidak diperlukan tahap penggabungan khusus.

c. Algoritma dalam pengurutan berdasarkan topologi (Topologi Sorting)

Algoritma dalam pengurutan berdasarkan topologi (Topological Sorting) digunakan untuk mengurutkan simpul-simpul dalam sebuah graf berarah (directed graph) sedemikian rupa sehingga setiap simpul hanya muncul setelah semua simpul yang menjadi pendahulunya telah muncul. Algoritma ini berguna dalam situasi di mana terdapat ketergantungan antara elemen-elemen dalam graf, dan pengurutan topologi dapat memberikan urutan yang konsisten.

E. Metode Devide and Conquer

a. Metode evide and conquer

Metode Divide and Conquer (Membagi dan Menaklukkan) adalah pendekatan dalam desain algoritma yang melibatkan memecah masalah besar menjadi submasalah yang lebih kecil, menyelesaikan submasalah secara independen, dan kemudian menggabungkan solusi submasalah untuk mencapai solusi akhir.

Beberapa metode terkenal yang menggunakan pendekatan Divide and Conquer:

- **Merge Sort**
  Merge Sort adalah algoritma pengurutan yang menggunakan metode Divide and Conquer.

- **Quick Sort**
  Quick Sort adalah algoritma pengurutan yang juga menggunakan metode Divide and Conquer.

- **Binary Search**
  Binary Search adalah algoritma pencarian yang menggunakan metode Divide and Conquer.

- **Strassen's Algorithm**
  Strassen's Algorithm adalah algoritma yang digunakan untuk perkalian matriks yang menggunakan metode Divide and Conquer.

  Algoritma ini membagi matriks asli menjadi empat submatriks yang lebih kecil dan menggunakan rumus rekursif untuk mengalikan submatriks tersebut.
  Strassen's Algorithm mengurangi kompleksitas perkalian matriks menjadi O(n^log2(7)), yang lebih efisien daripada algoritma perkalian matriks konvensional.

b. Algortima pengurutan dengan metode merge sort

Algoritma pengurutan dengan metode Merge Sort menggunakan pendekatan Divide and Conquer. Ini adalah salah satu algoritma pengurutan yang efisien dan stabil.
Langkah-langkah untuk melakukan pengurutan menggunakan Merge Sort:

a. Divide (Membagi)
b. Conquer (Menaklukkan)
c. Combine (Menggabungkan)
d. Ulangi dan Gabungkan

Berikut adalah pseudocode untuk algoritma Merge Sort:

```
procedure mergeSort(arr)
    if length(arr) ≤ 1, return arr
    mid = length(arr) / 2
    left = mergeSort(arr[1...mid])
    right = mergeSort(arr[mid+1...length(arr)])
    return merge(left, right)

procedure merge(left, right)
    result = []
    while left ≠ empty and right ≠ empty
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    append remaining elements of left to result
    append remaining elements of right to result
    return result
```

c. Algoritma pencarian dengan metode search binary tree

Algoritma pencarian dengan metode Binary Search Tree (pohon pencarian biner) adalah pendekatan efisien untuk mencari elemen tertentu dalam struktur data pohon biner yang terurut.

Berikut adalah pseudocode untuk algoritma pencarian Binary Search Tree:

```
procedure binarySearch(root, target)
    if root = NULL or root.data = target
        return root
    else if target < root.data
        return binarySearch(root.left, target)
    else
        return binarySearch(root.right, target)
```

F. Algoritma dengan metode Transform and Conquer

a. Metode transform and conquer

Metode Transform and Conquer (Transformasi dan Penaklukkan) adalah pendekatan dalam desain algoritma yang melibatkan transformasi masalah menjadi bentuk yang lebih mudah atau lebih terstruktur, diikuti oleh penyelesaian masalah yang diubah menggunakan teknik penaklukkan atau algoritma yang sesuai.

Berikut adalah beberapa metode terkenal yang menggunakan pendekatan Transform and Conquer:

a) Sorting by Counting

Metode Sorting by Counting digunakan untuk mengurutkan elemen-elemen dalam kisaran tertentu (range) dengan kompleksitas waktu O(n).

b) Discrete Fourier Transform (DFT)

Metode Discrete Fourier Transform digunakan untuk mengubah sinyal waktu diskrit menjadi sinyal frekuensi diskrit.

c) Karatsuba Algorithm

Karatsuba Algorithm digunakan untuk perkalian bilangan besar dengan kompleksitas waktu O(n^log2(3)).

d) Fast Fourier Transform (FFT)

Metode Fast Fourier Transform adalah algoritma efisien untuk mengubah sinyal waktu diskrit menjadi sinyal frekuensi diskrit.

b. Algoritma pencarian dengan varian metode transform and conquer

Search adalah algoritma pencarian yang digunakan untuk mencari elemen dalam himpunan data terurut dengan menggunakan interpolasi linier. Algoritma ini dapat efisien jika data terdistribusi secara merata.

Berikut adalah langkah-langkah untuk melakukan pencarian menggunakan Interpolation Search:

1. Tentukan batas bawah dan batas atas dari himpunan data yang terurut.
2. Hitung perkiraan posisi elemen yang dicari dengan menggunakan rumus interpolasi linier:

- pos = low + ((value - arr[low]) * (high - low)) / (arr[high] - arr[low]) Di sini, value adalah elemen yang dicari, arr[low] dan

arr[high] adalah elemen pada indeks batas bawah dan batas atas, dan pos adalah perkiraan posisi elemen dalam himpunan data.

3. Periksa elemen pada posisi yang dihitung:

- Jika elemen pada posisi tersebut sama dengan elemen yang dicari, pencarian selesai dan elemen ditemukan.
- Jika elemen pada posisi tersebut lebih kecil daripada elemen yang dicari, cari di subarray kanan.
- Jika elemen pada posisi tersebut lebih besar daripada elemen yang dicari, cari di subarray kiri.

4. Ulangi langkah-langkah 2 dan 3 hingga elemen ditemukan atau batas bawah melebihi batas atas.

G. Space and time trade-offs

a Algoritma sorting bt counting

Algoritma sorting by counting (pengurutan dengan menghitung) adalah metode pengurutan yang efisien untuk mengurutkan elemen dalam kisaran tertentu (range) dengan kompleksitas waktu O(n), di mana n adalah jumlah elemen yang akan diurutkan. Algoritma ini bekerja dengan menghitung jumlah kemunculan setiap elemen dalam kisaran dan kemudian membangun array terurut berdasarkan informasi tersebut.

Berikut adalah pseudocode untuk algoritma Sorting by Counting:

```
procedure countingSort(arr)
    min = minimum value in arr
    max = maximum value in arr
    count = array of size (max - min + 1) initialized with 0

    for i = 0 to length(arr) - 1
        count[arr[i] - min] += 1

    for i = 1 to length(count) - 1
        count[i] += count[i - 1]

    sorted = array of same size as arr
    for i = length(arr) - 1 downto 0
        sorted[count[arr[i] - min] - 1] = arr[i]
        count[arr[i] - min] -= 1

    return sorted
```

b Algoritma dengan input string menggunakan Horspool

Untuk merancang algoritma pencarian string menggunakan metode Horspool, menggunakan pendekatan Shift Table yang digunakan untuk menentukan jumlah pergeseran saat mencocokkan pola dalam teks.

Berikut adalah pseudocode untuk algoritma pencarian string menggunakan metode Horspool:

```
procedure horspoolSearch(text, pattern)
    n = length(text)
    m = length(pattern)
    shiftTable = createShiftTable(pattern)
    i = m - 1

    while i < n
        k = 0
        while k < m and pattern[m - 1 - k] = text[i - k]
            k += 1
        if k = m
            return i - m + 1
        else
            i += shiftTable[text[i]]

    return -1


function createShiftTable(pattern)
    shiftTable = map of characters to integers
    m = length(pattern)

    for i = 0 to m - 2
        shiftTable[pattern[i]] = m - 1 - i

    return shiftTable
```

c. Pengertian Hasing

Hashing adalah proses mengubah data atau informasi menjadi nilai atau kunci yang lebih kecil melalui fungsi hash. Fungsi hash mengambil data sebagai input dan mengembalikan nilai hash yang merupakan representasi numerik tetap dengan panjang tetap. Nilai hash ini digunakan untuk mengidentifikasi atau merepresentasikan data dengan cara yang efisien.

d. B-trees dalam merancang algoritma

Penerapan B-trees (pohon B) dalam merancang algoritma biasanya terkait dengan operasi penyimpanan dan pencarian data dalam struktur data yang efisien. B-trees adalah struktur data yang digunakan untuk menyimpan data dalam urutan terurut dan memungkinkan operasi pencarian, penyisipan, dan penghapusan dengan kompleksitas waktu logaritmik.

Berikut adalah beberapa penerapan umum B-trees dalam merancang algoritma:

- **Pohon Indeks (Indexing)**
  B-trees sering digunakan sebagai struktur data pohon indeks dalam basis data. Pada pohon indeks, setiap simpul pohon mengandung kunci dan pointer ke blok data terkait. Pohon indeks mempercepat operasi pencarian dalam basis data dengan mengurangi jumlah akses ke disk, karena hanya perlu membaca blok data yang relevan.

- **Sistem File**
  B-trees sering digunakan dalam implementasi sistem file untuk menyimpan dan mengorganisir data. Dalam sistem file, B-trees digunakan untuk memetakan alamat blok data ke lokasi fisik pada media penyimpanan. Ini memungkinkan pencarian efisien dan akses berurutan ke blok data yang berdekatan.

- **Basis Data**
  B-trees sangat berguna dalam basis data untuk menyimpan dan mengakses data yang diurutkan. B-trees dapat digunakan dalam indeks pada tabel basis data untuk pencarian cepat berdasarkan kunci atau kolom tertentu. Mereka memungkinkan operasi seperti pencarian berdasarkan rentang nilai (range queries) dan penggabungan indeks yang efisien.

- **Sistem Operasi**
  B-trees digunakan dalam sistem operasi untuk menyimpan struktur data seperti file sistem indeks, direktori, dan struktur file lainnya. Dalam sistem operasi, B-trees memungkinkan pencarian berdasarkan nama file atau atribut lainnya secara efisien, dan juga mendukung operasi penyisipan dan penghapusan file dengan kompleksitas logaritmik.

- **Database Terdistribusi**
  B-trees juga diterapkan dalam sistem database terdistribusi untuk mengorganisir dan mengakses data secara efisien di beberapa node atau server. Dalam lingkungan terdistribusi, B-trees memungkinkan replikasi data, distribusi beban, dan operasi pencarian yang efisien melintasi banyak node.

H. Metode Dynamic Programming

a. Teknik knapsack problem dan fungsi memori

Knapsack Problem adalah masalah optimisasi di mana kita harus memilih item dari himpunan item yang diberikan untuk dimasukkan ke dalam knapsack (tas) dengan kapasitas tertentu, sedemikian rupa sehingga nilai total item yang dimasukkan maksimal.

b. Algoritma dengan binary search trees

Untuk merancang algoritma dengan pengoptimalan Binary Search Trees (BST), dapat menggunakan algoritma yang dikenal sebagai "Self-Balancing Binary Search Trees" atau BST yang seimbang secara otomatis. Salah satu implementasi yang populer dari BST yang seimbang adalah Red-Black Tree.

c. Konsep warshall's algorithm dan floyd algorithm

Warshall's Algorithm dan Floyd Algorithm adalah dua algoritma yang digunakan dalam teori graf untuk menyelesaikan masalah perutean terpendek dalam graf berbobot. Meskipun keduanya digunakan untuk mencari jalur terpendek antara dua titik dalam graf, mereka memiliki perbedaan dalam pendekatan dan kompleksitas waktu.

- **Warshall's Algorithm**
  Warshall's Algorithm digunakan untuk mencari semua jalur terpendek antara setiap pasangan simpul dalam graf berbobot.
  Algoritma ini menggunakan matriks ketetanggaan graf (matriks adjacency) sebagai input.

- **Floyd Algorithm**
  Floyd Algorithm juga digunakan untuk mencari jalur terpendek antara setiap pasangan simpul dalam graf berbobot.
  Algoritma ini menggunakan matriks jarak graf (matriks distance) sebagai input.

  Perbedaan utama antara kedua algoritma ini adalah pendekatan yang digunakan. Warshall's Algorithm menggunakan pendekatan matematis dengan matriks ketetanggaan, sedangkan Floyd Algorithm menggunakan pendekatan iteratif dengan matriks jarak. Kompleksitas waktu Warshall's Algorithm adalah O(n^3), sedangkan kompleksitas waktu Floyd Algorithm adalah O(n^3) juga.

I. Greedy Teachnique

a. Teknik Greedy Technique

Algoritma dengan menerapkan teknik Greedy (Gananci) adalah pendekatan yang memilih langkah terbaik pada setiap langkah dalam harapan bahwa ini akan menghasilkan solusi optimal secara keseluruhan. Algoritma Greedy tidak melihat gambaran keseluruhan atau mempertimbangkan konsekuensi jangka panjang, tetapi hanya fokus pada pengambilan keputusan lokal yang paling menguntungkan pada saat itu.

b. Prim's algoritma dalam perancangan

Algoritma Prim's adalah algoritma Greedy yang digunakan untuk mencari MST (Minimum Spanning Tree) dalam sebuah graf berbobot. MST adalah subset dari graf yang terdiri dari semua simpul, tetapi hanya mencakup subset yang membentuk pohon dengan bobot minimum. Algoritma Prim's berfungsi dengan cara memilih simpul awal secara acak, kemudian secara iteratif menambahkan simpul dengan bobot terkecil yang terhubung ke MST yang sedang dibangun.

Berikut adalah langkah-langkah dalam merancang algoritma dengan menggunakan Prim's Algorithm:

- Tentukan input
- Inisialisasi MST
- Pilih simpul awal
- Tandai simpul awal
- Iterasi
- Evaluasi MST

Algoritma Prim's bekerja dengan memilih sisi-sisi dengan bobot terkecil pada setiap langkah, sehingga memastikan bahwa MST yang dibangun memiliki bobot minimum.

c. Kruskal's algoritma

Algoritma Kruskal adalah algoritma Greedy yang digunakan untuk mencari MST (Minimum Spanning Tree) dalam sebuah graf berbobot. MST adalah subset dari graf yang terdiri dari semua simpul, tetapi hanya mencakup subset yang membentuk pohon dengan bobot minimum. Algoritma Kruskal bekerja dengan cara mengurutkan sisi-sisi graf berdasarkan bobotnya secara menaik, kemudian secara iteratif menambahkan sisi dengan bobot terkecil ke MST jika tidak membentuk siklus.

Berikut adalah langkah-langkah dalam menerapkan algoritma Kruskal:

- Tentukan input
- Inisialisasi MST
- Urutkan sisi-sisi
- Inisialisasi himpunan disjoint
- Iterasi
- Evaluasi MST

Algoritma Kruskal bekerja dengan memilih sisi-sisi dengan bobot terkecil yang tidak membentuk siklus dalam MST yang sedang dibangun.

d. Dijikstra's algoritma

Algoritma Dijkstra adalah algoritma Greedy yang digunakan untuk mencari jalur terpendek antara dua simpul dalam sebuah graf berbobot. Algoritma ini berfungsi dengan cara memilih simpul dengan jarak terpendek yang belum dikunjungi pada setiap iterasi.

Berikut adalah langkah-langkah dalam merancang algoritma dengan menggunakan algoritma Dijkstra:

- Tentukan input
- Inisialisasi jarak
- Inisialisasi simpul yang dikunjungi
- Iterasi
- Rekonstruksi jalur terpendek
- Evaluasi hasil

Algoritma Dijkstra bekerja dengan memilih simpul dengan jarak terpendek yang belum dikunjungi pada setiap iterasi, sehingga memastikan bahwa jarak yang tercatat pada setiap simpul adalah jarak terpendek yang diketahui saat ini. Algoritma ini cocok untuk graf berbobot positif dan tidak mengandung siklus negatif.

e. Huffmann tress dalam perancangan algoritma

Pohon Huffman (Huffman tree) adalah sebuah struktur data yang digunakan dalam kompresi data. Algoritma Huffman digunakan untuk menghasilkan kode biner yang optimal untuk setiap karakter dalam suatu himpunan karakter, dengan meminimalkan ukuran total dari representasi kode biner tersebut. Dalam perancangan algoritma, Pohon Huffman sangat penting untuk mengimplementasikan kompresi data dengan efisien.

Berikut adalah langkah-langkah dalam memahami Huffman Trees dalam perancangan algoritma:

- Tentukan input
- Membangun pohon huffman

- Menghasilkan kode biner
- Evaluasi hasil

Pohon Huffman memanfaatkan konsep probabilitas kemunculan karakter atau simbol untuk menghasilkan representasi kode biner yang optimal. Karakter-karakter yang lebih sering muncul memiliki representasi kode biner yang lebih pendek, sementara karakter-karakter yang jarang muncul memiliki representasi kode biner yang lebih panjang.

J. Iterative improvement dalam merancang algoritma

a. Metode simplex method

Metode Simpleks (Simplex Method) adalah sebuah algoritma yang digunakan untuk mencari solusi optimal dari permasalahan optimisasi linear. Metode ini berguna untuk menemukan nilai maksimum atau minimum dari fungsi objektif linier dalam batasan-batasan linear yang diberikan.

Berikut adalah langkah-langkah dalam menerapkan metode Simpleks:

- Tentukan input
- Konversu ke bentuk standar
- Inisialisasi tabel simpleks
- Iterasi
- Evaluasi hasil

b. Algoritma flow problem

Algoritma yang digunakan untuk memecahkan masalah Maksimal Flow (Maximum Flow) dalam teori jaringan adalah Algoritma Edmonds-Karp. Algoritma ini berdasarkan pada algoritma Ford-Fulkerson dengan menggunakan pendekatan Breadth-First Search (BFS) untuk mencari jalur peningkatan kapasitas (augmenting path) yang memaksimalkan aliran di dalam jaringan.

Berikut adalah langkah-langkah dalam merancang algoritma dengan memperhatikan Maksimal Flow Problem:

- Tentukan input
- Inisialisasi aliran awal
- Lakukan literasi
- Evaluasi hasil

Algoritma Edmonds-Karp menggunakan algoritma BFS untuk mencari jalur peningkatan kapasitas dengan kompleksitas waktu yang efisien. Dengan melakukan iterasi yang berulang, algoritma ini secara

bertahap meningkatkan aliran dalam jaringan hingga mencapai aliran maksimum. Algoritma ini dapat digunakan untuk menyelesaikan berbagai masalah seperti aliran di jaringan, transportasi, dan alokasi sumber daya.

c. Algoritma maximum matching in bipartite graphs

Algoritma yang digunakan untuk menemukan Maximum Matching dalam Bipartite Graphs adalah Algoritma Hopcroft-Karp. Algoritma ini berfungsi untuk mencari pasangan simpul yang saling terhubung dengan maksimum dalam graf bipartit.

Berikut adalah langkah-langkah dalam merancang algoritma dengan memperhatikan Maximum Matching dalam Bipartite Graphs:

- Tentukan input
- Inisialisasi matching awal
- Lakukan iterasi
- Evaluasi hasil

Algoritma Hopcroft-Karp menggunakan pendekatan Breadth-First Search (BFS) untuk mencari jalur peningkatan dan secara bertahap memperbarui matching dalam graf bipartit. Dengan melakukan iterasi yang berulang, algoritma ini secara bertahap membangun matching yang saling berhubungan dengan maksimum. Algoritma ini memiliki kompleksitas waktu yang efisien dan dapat diterapkan dalam berbagai masalah yang melibatkan pencocokan maksimum, seperti pencarian pasangan ideal dalam masalah pernikahan stabil.

d. Algoritma the stable marriage problem

Algoritma Gale-Shapley adalah algoritma yang digunakan untuk memecahkan masalah The Stable Marriage Problem (masalah pernikahan stabil). Masalah ini melibatkan pencocokan stabil antara dua himpunan orang dengan preferensi masing-masing.

Berikut adalah langkah-langkah dalam merancang algoritma dengan memperhatikan The Stable Marriage Problem:

- Tentukan input
- Inisialisasi pencocokan awal
- Lakukan iterasi
- Evaluasi hasil

K. Limitations of algorithm power dalam merancang algoritma

a. Lower-bound arguments dalam algoritma

Lower bound arguments (argumen batas bawah) digunakan dalam merancang algoritma untuk membuktikan bahwa tidak ada algoritma yang lebih efisien (dalam hal kompleksitas waktu) untuk menyelesaikan suatu masalah tertentu. Dengan menggunakan lower bound arguments, kita dapat menunjukkan bahwa setidaknya ada batasan minimal untuk seberapa efisien algoritma tersebut dapat bekerja.

b. Decision tree algoritma

Penerapan Decision Tree dalam Algoritma Pengurutan: Decision Tree dapat digunakan dalam pengurutan dengan mempertimbangkan atribut-atribut yang relevan untuk memutuskan urutan elemen-elemen.

Elemen atribut yang tepat:

- Tentukan input
- Tentukan atribut-atribut yang relevan
- Konstruksi decision tree
- Traversing decision tree

Penerapan Decision Tree dalam algoritma pengurutan dan pencarian memungkinkan kita untuk membuat keputusan berdasarkan atribut-atribut yang relevan dan mempercepat proses pengurutan atau pencarian. Dengan memilih atribut yang tepat, Decision Tree dapat membantu dalam mengoptimalkan algoritma dan mempercepat waktu eksekusi.

c. Algortima dengan p,np and NP-Complete Problems

1. Algoritma P

- P adalah kelas masalah yang dapat diselesaikan dalam waktu polinomial oleh sebuah algoritma deterministik.
- Contoh algoritma P adalah algoritma pengurutan seperti Merge Sort atau Quick Sort, yang memiliki kompleksitas waktu O(n log n) dalam kasus terburuk.

2. Algoritma NP

- NP adalah kelas masalah di mana solusi yang diajukan dapat diverifikasi dalam waktu polinomial oleh sebuah algoritma deterministik.
- Contoh algoritma NP adalah algoritma pencarian dengan backtracking, seperti algoritma pencarian depth-first search (DFS) yang digunakan dalam mencari jalur tertentu dalam graf.

3. **Algoritma NP-complete**
   - NP-complete adalah kelas masalah yang merupakan subset dari masalah NP, dan setiap masalah NP-complete dapat direduksi ke masalah lain dalam kelas tersebut.
   - Contoh algoritma NP-complete adalah algoritma untuk permasalahan Ransum (Subset Sum problem), yang melibatkan mencari subset elemen dalam himpunan yang jumlahnya sama dengan nilai tertentu.

   Dalam merancang algoritma dengan menerapkan P, NP, dan NP-complete problems, penting untuk mempertimbangkan kompleksitas waktu yang terlibat. Algoritma dalam kelas P diinginkan karena dapat diselesaikan dalam waktu polinomial yang efisien. Namun, masalah NP dan NP-complete lebih sulit dan biasanya memerlukan pendekatan heuristik atau algoritma aproksimasi untuk mencapai solusi yang memadai dalam waktu yang wajar.

d. Konsep challenges of numerical algorithms

Algoritma numerik menghadapi beberapa tantangan karena sifatnya yang melibatkan komputasi numerik dan aproksimasi.

Beberapa tantangan umum dalam algoritma numerik meliputi:

- **Akurasi dan Presisi**
  Algoritma numerik sering melibatkan aproksimasi dan kesalahan pembulatan. Menjaga akurasi dan presisi selama perhitungan dapat menjadi tantangan, terutama ketika berurusan dengan angka besar atau perhitungan iteratif. Pertimbangan yang cermat terhadap representasi numerik, analisis kesalahan, dan presisi yang tepat sangat penting.

- **Stabilitas dan Stabilitas Numerik**
  Stabilitas numerik merujuk pada kemampuan suatu algoritma untuk menghasilkan hasil yang akurat dalam kehadiran gangguan kecil atau kesalahan pada input. Algoritma yang tidak stabil dapat memperbesar kesalahan, menghasilkan hasil yang tidak benar atau tidak dapat diandalkan. Memastikan stabilitas numerik membutuhkan analisis stabilitas algoritma dan menerapkan teknik seperti analisis propagasi kesalahan, kondisioning, dan metode numerik yang tangguh.

- **Kompleksitas Komputasi**
  Algoritma numerik sering membutuhkan sumber daya komputasi, memori, dan waktu yang signifikan. Mengelola kompleksitas komputasi menjadi penting, terutama untuk masalah dalam skala

besar. Teknik seperti optimisasi algoritma, komputasi paralel, dan pemanfaatan properti masalah khusus dapat membantu meningkatkan efisiensi.

- **Konvergensi dan Tingkat Konvergensi**
  Algoritma numerik sering melibatkan proses iteratif untuk mencapai konvergensi menuju solusi. Tingkat konvergensi menentukan seberapa cepat algoritma mendekati solusi yang diinginkan. Konvergensi yang lambat dapat menghasilkan waktu komputasi yang lebih lama atau bahkan kegagalan konvergensi. Menganalisis sifat konvergensi algoritma dan menggunakan teknik percepatan konvergensi menjadi penting.

- **Skalabilitas**
  Skalabilitas algoritma numerik mengacu pada kemampuannya untuk menangani ukuran masalah yang semakin besar dengan efisien. Memperbesar masalah mungkin memerlukan penyesuaian algoritma atau menerapkan teknik komputasi paralel untuk memanfaatkan sumber daya komputasi secara efektif.

- **Keandalan**
  Algoritma numerik harus dapat mengatasi berbagai masukan, termasuk kasus ekstrem atau yang sulit dikondisikan. Mereka harus dapat menangani berbagai jenis input tanpa menghasilkan hasil yang salah atau tidak dapat diandalkan. Keandalan dicapai melalui penanganan kesalahan, validasi data, dan penggunaan teknik seperti regularisasi atau preconditioning.

- **Interpretasi dan Komunikasi Hasil**
  Algoritma numerik sering menghasilkan output yang kompleks yang perlu diinterpretasikan dan dikomunikasikan secara efektif kepada pemangku kepentingan. Visualisasi yang tepat, analisis data

L. Coping With the Limitations of algorithms power dalam algoritma

a. Konsep backtracking dalam algoritma

Backtracking adalah teknik yang digunakan dalam perancangan algoritma untuk mencari solusi secara sistematis melalui pencarian berulang di dalam ruang solusi. Konsep backtracking memungkinkan pemrosesan secara efisien dalam masalah yang melibatkan pemilihan dari sekumpulan pilihan yang tersedia, di mana solusi yang valid harus memenuhi sejumlah batasan atau kriteria tertentu.

Proses backtracking melibatkan langkah-langkah berikut:

- Pemilihan: Pilih salah satu opsi dari kumpulan opsi yang tersedia pada saat itu. Keputusan ini akan membentuk langkah awal dalam pencarian solusi.
- Evaluasi: Evaluasi solusi yang dihasilkan dengan menguji apakah solusi tersebut memenuhi semua kriteria atau batasan yang diberikan. Jika solusi tidak memenuhi batasan tertentu, maka dicoba opsi lain.
- Pemrosesan: Jika solusi yang dievaluasi memenuhi semua batasan, maka solusi tersebut dapat diproses atau dicatat sebagai solusi yang valid.
- Pemanggilan rekursif: Setelah langkah awal dipilih dan dievaluasi, langkah-langkah berikutnya dalam pencarian solusi diproses secara rekursif. Setiap langkah berikutnya akan mengikuti langkah-langkah yang sama seperti langkah 1 hingga 3.
- Pembatalan: Jika pada suatu titik pencarian tidak menemukan solusi yang memenuhi batasan tertentu, maka langkah-langkah yang telah diambil dapat dibatalkan, dan pencarian dilanjutkan dengan mencoba opsi yang lain.
- Berhenti: Proses pencarian berhenti ketika semua kemungkinan opsi telah dijelajahi atau ketika solusi yang memenuhi semua kriteria ditemukan.

Keunggulan dari menggunakan teknik backtracking adalah kemampuannya untuk memotong cabang pencarian yang tidak memungkinkan, mengurangi ruang pencarian solusi yang perlu dieksplorasi. Namun, perancangan algoritma dengan menggunakan backtracking harus memperhatikan efisiensi dan kompleksitas waktu yang mungkin meningkat seiring dengan pertumbuhan ukuran masalah.

Contoh penerapan backtracking adalah algoritma N-Queens, yang mencari solusi untuk menempatkan N ratu di papan catur N x N tanpa saling serang. Dengan menggunakan teknik backtracking, algoritma dapat mencari solusi secara sistematis dengan memilih langkah awal, mengevaluasi solusi, dan memproses langkah-langkah berikutnya hingga solusi yang valid ditemukan.

b. Konsep branch and bound algoritma

Branch and Bound adalah teknik yang digunakan dalam perancangan algoritma untuk mencari solusi optimal dalam ruang pencarian yang besar. Konsep ini menggabungkan pemisahan masalah menjadi submasalah yang lebih kecil (branching) dengan penggunaan batasan dan pemangkasan yang cerdas (bounding) untuk menghindari eksplorasi yang tidak perlu di dalam ruang pencarian.

Proses Branch and Bound melibatkan langkah-langkah berikut:

- **Pemisahan (Branching)**

  Pada langkah awal, masalah besar dibagi menjadi submasalah yang lebih kecil. Setiap submasalah menghasilkan percabangan atau cabang baru dalam ruang pencarian. Setiap cabang mewakili satu kemungkinan solusi.

- **Evaluasi Batasan (Bounding)**

  Pada setiap cabang, dilakukan evaluasi batasan untuk memutuskan apakah cabang tersebut perlu dieksplorasi lebih lanjut. Batasan ini digunakan untuk memperoleh batas atas atau batas bawah pada nilai solusi yang mungkin dicapai dalam cabang tersebut. Jika batasan menunjukkan bahwa cabang tidak mungkin menghasilkan solusi optimal, cabang tersebut dapat dipangkas atau dilewatkan.

- **Penyusunan Prioritas (Priority Queue)**

  Dalam proses pencarian, cabang-cabang yang belum dieksplorasi ditempatkan dalam antrian berprioritas berdasarkan batasan yang dihasilkan. Antrian ini membantu dalam mengatur urutan eksplorasi dan memprioritaskan cabang-cabang yang memiliki potensi lebih besar untuk menghasilkan solusi optimal.

- **Penyimpanan Solusi Sementara (Solution Pool)**

  Selama proses pencarian, solusi sementara yang ditemukan dan memenuhi batasan disimpan dalam pool solusi. Pool solusi digunakan untuk membandingkan solusi yang ditemukan dan memperbarui solusi optimal terbaik jika ditemukan solusi yang lebih baik.

- **Pemanggilan Rekursif**

  Setiap cabang yang memenuhi batasan dieksplorasi secara rekursif dengan menerapkan langkah-langkah 1 hingga 4 pada submasalah yang lebih kecil.

- **Berhenti**

  Proses pencarian berhenti ketika semua cabang telah ditempati atau ketika tidak ada lagi cabang yang perlu dieksplorasi.

Keunggulan dari menggunakan teknik Branch and Bound adalah kemampuannya untuk menghindari eksplorasi yang tidak perlu dan mempersempit ruang pencarian. Dengan menggunakan batasan yang cerdas, algoritma dapat memangkas cabang-cabang yang tidak memiliki potensi untuk menghasilkan solusi optimal, sehingga menghemat waktu dan sumber daya komputasi.

c. Konsep algoritma untuk persamaan linear

Algoritma untuk menyelesaikan persamaan linear adalah proses komputasional untuk mencari solusi dari sistem persamaan linear. Persamaan linear adalah bentuk persamaan matematika yang melibatkan variabel-variabel dengan pangkat eksponen 1, seperti:

a1x1 + a2x2 + ... + anxn = b

Konsep algoritma yang umum digunakan untuk menyelesaikan persamaan linear:

- **Metode Eliminasi Gauss**

  Metode ini melibatkan penggunaan operasi elemen dasar pada matriks untuk mengubah sistem persamaan linear menjadi bentuk matriks eselon tereduksi atau bentuk matriks segitiga atas. Dalam proses ini, persamaan-persamaan linear dikurangi atau ditambahkan untuk menghilangkan variabel-variabel yang tidak diinginkan.

- **Metode Eliminasi Gauss-Jordan**

  Metode ini melanjutkan dari metode Eliminasi Gauss dengan tujuan untuk mendapatkan bentuk matriks eselon tereduksi. Setelah matriks eselon tercapai, metode ini melibatkan penggunaan operasi elemen dasar lainnya untuk menghilangkan semua entri di atas dan di bawah setiap pivot. Pada akhirnya, solusi persamaan linear ditemukan dalam bentuk yang lebih sederhana.

- **Metode Matriks Balikan**

  Metode ini melibatkan penggunaan matriks balikan untuk menyelesaikan persamaan linear. Pertama, matriks koefisien sistem persamaan linear diinvers. Kemudian, solusi persamaan linear ditemukan dengan mengalikan matriks balikan dengan vektor konstanta.

- **Metode Dekomposisi LU**

  Metode ini melibatkan dekomposisi matriks koefisien sistem persamaan linear menjadi faktor-faktor LU, di mana L adalah

matriks segitiga bawah dan U adalah matriks segitiga atas. Setelah dekomposisi, persamaan linear diselesaikan dalam beberapa langkah dengan menggabungkan dekomposisi matriks dan substitusi mundur.

- **Metode Jacobi dan Gauss-Seidel**

  Metode ini adalah metode iteratif yang digunakan untuk menyelesaikan sistem persamaan linear. Dalam metode Jacobi, variabel-variabel sistem persamaan linear diperbarui secara simultan pada setiap iterasi, sedangkan dalam metode Gauss-Seidel, variabel-variabel diperbarui secara berurutan. Proses iterasi berlanjut hingga solusi konvergen ke solusi yang diinginkan.