BUKU AJAR

RANCANGAN ANALISA ALGORITMA

Oleh

MUHAMMAD ARHAMI, S. Si., M. Kom

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNOLOGI INFORMASI DAN KOMPUTER
POLITEKNIK NEGERI LHOKSEUMAWE
2020

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB I
MENGAPA ALGORITMA ?

a) Deskripsi Singkat

Algoritma merupakan langkah atau cara untuk menyelesaikan suatu masalah yang ada dalam kehidupan sehari-hari. Program komputer telah menggunakan algoritma-berbagai algoritma yang berbeda untuk memudahkan dalam mengintepretasi masalah agar lebih mudah diselesaikandan digunakan oleh penggunanya. Implementasikan algoritma ke dalam program dan mengerti tentang algoritma-algoritma yang digunakan akan membantu kita mempersingkat waktu dalam memecahkan suatu masalah. Analisis algoritma merupakan salah satu dasar ilmu komputer yang sangat penting dalam mengkaji penerapan algoritma dari sisi kompleksitas efektifitas dan efisiensi serta kompleksitasnya ruang dan waktu dari suatu algoritma yang telah dirancang. Bab ini akan membahas tentang apa itu algoritma, algoritma dalam perspektif, struktur dasar algoritma, bidang-bidang yang menggunakan algoritma, pengertian rancangan algoritma dan analisa algoritma.

b) Relevansi

Proses pembelajaran dilakukan dengan metode ceramah, diskusi, tanya jawab dan praktikum sehingga materi yang disampaikan kepada mahasiswa dapat dipahami dan mampu dijelaskan oleh mahasiswa, serta mahasiswa mempu membuat suatu algoritma dan mengimplementasikannya ke dalam program. Selain itu tugas mandiri juga akan diberikan untuk penguatan pemahaman dan kemampuan mahasiswa.

c) Capaian Pembelajaran

1. Mahasiswa dapat menjelaskan apa itu algoritma, struktur dasar algoritma, efisiensi algoritma
2. Mahasiswa mampu menjelaskan konsep algoritma dan hubungan dengan program
3. Mahasiswa mampu membuat dan mengimplementasikan algoritma ke dalam program.
4. Mahasiswa mampu menjelaskan pentingnya rancangan dan analisis algoritma

1.1 PENDAHULUAN

Jika anda ingin menjadi seorang professional di bidang komputer, maka ada dua alasan yang membuat anda harus mempelajari dan memahami berbagai hal terkait algoritma yaitu alasan praktikal dan teoritik. Berangkat dari algoritma maka suatu sistem yang akan dikomputerisasikan dari berbagai bidang akan mudah diterjemahkan oleh programmer ke dalam berbagai bahasa pemrograman berdasarkan disain yang ditampilkan. Untuk itu algoritma harus dipelajari dan dipahami.

Komputer pada dasarnya adalah mesin yang tidak bisa apa-apa. Kita harus memberikan perintah untuk dapat berbicara (berkomunikasi) dengan komputer, dengan cara memberikan serangkaian instruksi kepada komputer agar komputer dapat memecahkan masalah. Langkah-langkah yang kita lakukan dalam memberikan instruksi untuk memecahkan masalah dinamakan pemrograman komputer.

Pengertian Algoritma adalah suatu prosedur yang tepat untuk memecahkan masalah dengan menggunakan bantuan komputer serta menggunakan suatu bahasa pemrogaman tertentu seperti bahasa C, C++, C#, Pascal, Visual Basic, Java, prolog dan masih banyak lagi bahasa yang lain. Pranata (2002:8) dalam kehidupan sehari-hari, sebenarnya kita juga menggunakan algoritma untuk melaksanakan sesuatu. Sebagai contoh, ketika kita menulis surat, maka kita perlu melakukan beberapa langkah sebagai berikut:

1. Mempersiapkan kertas dan amplop.
2. Mempersiapkan alat tulis, seperti pena atau pensil.
3. Mulai menulis.
4. Memasukkan kertas ke dalam amplop.
5. Pergi ke kantor pos untuk mengeposkan surat tersebut.

Definisi Algoritma menurut Cormen dkk (20019) menyatakan bahwa, “suatu algoritma adalah setiap prosedur komputasi terdefinisi dengan baik yang mengambil beberapa nilai, atau serangkaian nilai, sebagai input dan menghasilkan beberapa nilai, atau set nilai sebagai output, sehingga dengan kata lain dapat dijelaskan bahwa :

a. Algoritma adalah urutan langkah-langkah berhingga untuk memecahkan masalah logika atau matematika

b. Algoritma adalah logika, metode dan tahapan (urutan) sistematis yang digunakan untuk memecahkan suatu permasalahan.
c. Algoritma adalah urutan langkah-langkah logis penyelesaian masalah yang disusun secara sistematis dan logis.
d. Algoritma adalah urutan logis pengambilan keputusan untuk pemecahan masalah.

Kata Logis merupakan kata kunci dalam Algoritma. Langkah-langkah dalam Algoritma harus logis dan harus dapat ditentukan bernilai salah atau benar. Pembuatan algoritma harus selalu dikaitkan dengan:

a. Kebenaran algoritma
b. Kompleksitas (lama dan jumlah waktu proses dan penggunaan memori)

Kriteria Algoritma yang baik:

1. Tepat, benar, sederhana, standar dan efektif
2. Logis, terstruktur dan sistematis
3. Semua operasi terdefinisi
4. Semua proses harus berakhir setelah sejumlah langkah dilakukan
5. Ditulis dengan bahasa yang standar dengan format pemrograman agar mudah untuk diimplementasikan dan tidak menimbulkan arti ganda.

Jika kita mempelajari algoritma maka ada beberpa hal yang bisa didapat dan bisa membantu kita diantaranya :

1. Untuk mengenali pola umum, atau strategi pemecahan masalah
2. Untuk dapat menganalisis algoritma untuk efisiensi waktu dan ruang
3. Untuk memperkuat matematika anda, pemrograman, dan keterampilan memecahkan masalah.
4. Untuk membangun "repertoar" algoritmik blok bangunan
5. Karena tidak ada optimasi yang paling baik daripada mengganti algoritma yang buruk dengan yang bagus.
6. Karena itu menyenangkan dan mencerahkan
7. Karena di era digitalisasi ini banyak perusahaan dan industry mempekerjakan oring-orang yang peduli algoritma, seperti GOJEK, TOKOPEDIA, Buka Lapak, dan lain sebaginya

1.2 ALGORITMA dalam PERSPEKTIF

Algoritma memiliki sejarah panjang dan istilah tersebut tentunya mempunyai sejarah sendiri. Pada ke-9, seorang ilmuwan, astronom, dan ahli matematika Persia, Abdullah Muhammad bin Musa al-Khwarizmi, yang sering disebut sebagai "Bapak Aljabar", memiliki kaitan yang sangat erat dengan istilah "Algoritma". Pada abad ke-12 salah satu bukunya yang berjudul "Algorithmi de numero Indorum". Pada awalnya kata algoritma adalah istilah yang merujuk kepada aturan-aturan aritmetis untuk menyelesaikan persoalan dengan menggunakan bilangan numerik arab, namun seiring dengan perkembangan ilmu pengetahuan pada abad ke-18, istilah ini berkembang menjadi **algoritma**, yang mencakup semua prosedur atau urutan langkah yang jelas dan diperlukan untuk menyelesaikan suatu permasalahan.

Algoritma saat ini bisa kita temukan di mana-mana. Banyak algoritma yang telah dikembangkan untuk memudahkan kehidupan kita sehari-hari. Hal ini dimulai dari menghitung algoritma dan hari ini kecerdasan buatan, data mining, macine learning, big data internet og things telah menjadikan algoritma berperan penting dalam menyelesaiakan perbagai permasalahan yang bentukan algoritmanya dapat berbasisi dari data. Google merupakan salah satu produk aktual dari pengembangan ini, pengetahuan berbasis data dan jaringan saraf akan menjadi hal yang penting untuk masa yang akan dating.

Saat ini kita berada pada zaman Revolusi Industri 4.0 dimana zaman data dan informasi menjadi hal yang paling penting dalam menggerakkan berbagai perubahan. Analisa data merupakan suatu cara yang dilakukan agar data dapat berfungsi dan bermakna dan dengan bantuan teknik komputasi yang kuat, maka seorang analis dapat memberikan sesuatu hal yang baru atau sesuatu yang berarti dari data yang berjumlah esar atau big data dalam berbagai bentuk seperti : bentuk: teks, angka, gambar, video, dan audio.

Jika ada klien datang dengan membawa data, maka ahli statistik dapat membantu mengungkapkan informasi tersembunyi melalui analisis eksplorasi, pengelompokan seperti jenis informasi dan mengekstraksi fitur utama melalui penggunaan pengambilan sampel sistematis dan teknik desain eksperimental sehingga

mereka yakin dan dapat memastikan data tersebut reliable atau dapat diandalkan dan akan mendukung keputusan dengan tingkat kepercayaan yang tinggi.

Selnjutnya Algoritma dari sisi computer dapat dilihat sebagai suatu alat bantu yang dapat memudahkan para programmer dalam menerjemahkannya ke dalam bentuk program dengan pelbagai basis bahasa pemrograman, Komputer berjalan pada program atau perangkat lunak dalam menyelesaikan permasalahan. Perangkat lunak merupakan salah satu contoh nyata dari suatu algoritma. Ada banyak algoritme yang memberi tahu cara mengurutkan dan menyusun ulang jutaan angka secara efisien, algoritme yang memberi tahu cara membuat peringkat situs web dalam hasil pencarian, suara manusia dikonversi menjadi teks komputer, banyak algoritma yang menyelesaikan permasalahan-permasalahan dalam AI dalam bentuksuatu game, dan masih banyak lagi. Semua yang disebutkan dapat diselesaikan jika ada algoritmanya dan algoritma-algoritma tersebut selanjutnya dikonvert ke dalam bahasa pemrograman sesuai dengan bahasa-bahasa pemrograman yang diinginkan.

Pemahaman tentang algoritma sangat penting, karena dalam perkembangan bahasa pemrograman, seiring perjalanan waktu semua bahasa pemrograman dapat berubah, berganti dan dimodifikasi seperti C++, JAVA, Visual basic, pyton, perl dan lain sebagainya akan mengalami perubahan dan digantikan oleh bahasa-bahasa pemrograman yang lebih efisien, namun Algoritma yang merupakan logika untuk menyelesaikan permasalahan tanpa harus bergantung kepada bahasa pemrograman yang ada. Jika kita mengetahui tentang cara menyelesaikan permasalahan artinya kita telah memahami bagaimana struktur beserta dengan elemen-elemen dari masalah yang ada, dan selanjutnya ketika menterjemahkannya ke dalam bahasa pemrograman dalam bentuk kode-kode program maka sintak-sintak bahasa pemrograman tertentu tersebut harus kita miliki.

Sebagai contoh ilustrasi yang sederhana dari suatu algoritma ketika cuacanya dingin maka gunakan jaket :

```
IF (cuaca dingin) {
        Pakai Jaket.
} else {
        Jangan pakai jaket.
}
```

Ilustrasi sederhana tersebut menunjukkan bahwa siapapun dapat menjadi subjek dan menjalankannya melalui suatu fungsi yang memungkinkan untuk memutuskan pilihan memilih atau tidak, da nada masih banyak algoritma-algoritma lainnya yang dapat diilusttasikan seperti itu diantaranya: Jika saya terlambat kerja, maka saya akan mendapat masalah. Jika saya mempercepat ke stasiun kereta, maka saya akan mendapatkan tiket.

Algoritma fungsi jaket diatas jika hendak dituliskan dalam bahasa pemrograman PHP misalnya maka menjadi :

```
if ($suhu == 15) {
    pakai jaket
}
```

Kekuatan algoritma menjadi terwujud ketika Anda menerapkannya pada sejumlah besar data atau data dalam ukuran bigdata. Sebagai contoh algoritma "pertemanan" dalam aplikasi facebooks. Ini tentunya suatu hal yang luar biasa karena membutuhkan data orang-orang yang dikenal. Ketika suatu saat kita ingin membuat pertemanan dengan yang lain agar kita dapat mengenal dan dikenal maka kita harus memeriksa orang-orang yang dikenal oleh teman kita atau teman dari teman kita agar terjalin pertemanan di Facebooks dan kita juga dapat menyarankan yang kita kenal kepada orang lain atau orang lain dapat menyarankan kepada kita. Permasalahan tersebut dapat diselesaikan dengan menggunakan algoritma diantaranya adalah dengan menambahkan atau menghapus kandidat potensial kita, sehingga hasil yang didapatkan lebih akurat.

Algoritma boleh jadi bukan sesuatu yang asing bagi sesorang programmer atau seseorang yang akan menyelesaikan suatu permasalahan dan didalamnya melibatkan komputasi dan membutuhkan langkah-langkah yang terstruktur dan runut. Ketrampilan membuat algoritma diperlukan dalam mengembangkan berbagai hal yang berkaitan dengan komputer.

Algoritma dapat digambarkan dalam serangkaian instruksi, sering disebut sebagai "proses," yang harus diikuti ketika memecahkan masalah tertentu. Meskipun secara teknis tidak dibatasi oleh definisi, kata algoritma hampir selalu terkait dengan komputer, karena algoritma yang diproses komputer dapat mengatasi masalah terkait komputasi jauh lebih cepat daripada yang dilakukan oleh manusia, Algoritma yang

dibuat juga melalui proses desain, analisis, dan penyempurnaan. Desain dan analisis algoritma membutuhkan latar belakang matematika yang kuat sehingga menghasilkan algoritma yang tangguh dan kuat karena algoritma juga berisi komputasi, seperti yang disampaikan oleh Francis Sullivan bahwa Algoritma yang hebat adalah seperti puisi perhitungan. Sama seperti bait-baitnya, mereka bisa singkat, kiasan, padat, dan bahkan misterius. Tapi begitu dibuka, mereka memberikan cahaya baru yang cemerlang dalam beberapa aspek komputasi.

Secara sederhana, algoritma pada dasarnya hanyalah seperangkat instruksi yang diperlukan untuk menyelesaikan tugas. Jauh sebelum munculnya era komputer modern, orang menetapkan rutinitas yang telah ditentukan untuk melihat bagaimana proses-proses yang mereka akan kerjakan dalam menyelesaikan tugas sehari-hari. Mereka, sering menuliskan daftar langkah yang harus dilakukan untuk mencapai tujuan penting, mengurangi risiko melupakan sesuatu yang penting. Hal-hal yang keseharian itu dilakukan sebenarnya secara tidak langsung bahwa mereka telah menuliskan sebuah algoritma dalam aktifitas sehari-hari.

Analogi yang yang sama seorang desainer mengambil pendekatan yang mirip dengan pengembangan algoritma untuk tujuan komputasi: pertama, mereka melihat masalah, kemudian, mereka menguraikan langkah-langkah yang akan diperlukan untuk menyelesaikannya. Akhirnya, mereka mengembangkan serangkaian operasi matematika untuk mencapai langkah-langkah tersebut, dan menghasilkan suatu penyelesaian untuk dibawa ke dalam program dengan bahasa pemrograman tertentu.

Algoritma perlu diketahui oleh setiap orang yang berada dalam bidang computer, dan mengapa algoritma diperlukan dan wajib dipelajari ?, W. Yang, G. Valian, V.V. Williams, J. Su (2015) menyebutkan bahwa :

a. Algoritma adalah dasar untuk semua area komputer: Algoritma adalah tulang punggung ilmu komputer. Di mana pun ilmu komputer mencapai, ada sebuah algoritma, dan paradigma desain algoritmik algoritme klasik yang kami bahas kembali terjadi di seluruh area CS. Sebagai contohdalam sistem operasi dan kompilasi memanfaatkan algoritma penjadwalan dan struktur data yang efisien, Jaringan secara krusial menggunakan algoritma jalur terpendek, Machine Learning memanfaatkan algoritma geometrik cepat dan pencarian kesamaan, Cryptography memanfaatkan algoritma bilangan cepat dan aljabar, dan Biologi

Komputasi memanfaatkan algoritma yang beroperasi pada string - dan sering menggunakan paradigma pemrograman dinamis. Algoritma dan perspektif komputasi ("lensa komputasi") juga telah banyak diterapkan di bidang lain, seperti fisika (misalnya komputasi kuantum), ekonomi (misalnya teori permainan algoritmik), dan biologi (misalnya untuk mempelajari evolusi, sebagai e fi siensi yang mengejutkan) algoritma yang mencari ruang genotipe).

b. Algoritma berguna: Banyak kemajuan yang telah terjadi dalam teknologi / industri disebabkan oleh perkembangan ganda perangkat keras yang ditingkatkan (ala "Hukum Moore" —sebuah prediksi yang dibuat pada tahun 1965 oleh salah seorang pendiri Intel bahwa kepadatan transistor pada sirkuit terintegrasi akan berlipat ganda setiap atau dua tahun), dan meningkatkan algoritma. Bahkan, semakin cepat komputer, semakin besar perbedaan antara apa yang dapat dicapai dengan algoritma yang cepat vs apa yang dapat dicapai dengan algoritma lambat .... Industri perlu terus mengembangkan algoritma baru untuk masalah besok, dan Anda dapat membantu menyumbangnya.
c. Algoritma itu menyenangkan : Desain dan analisis algoritma memerlukan kombinasi kreativitas dan ketepatan matematis. Ini adalah seni dan sains, dan semoga setidaknya beberapa dari Anda akan menyukai kombinasi ini. Salah satu alasan lain mengapa ini sangat menyenangkan adalah bahwa kejutan algoritmik berlimpah. Mudah-mudahan kelas ini akan membuat Anda memikirkan kembali apa yang menurut Anda mungkin secara algoritmik, dan membuat Anda terus bertanya "apakah ada algoritma yang lebih baik untuk tugas ini?". Bagian yang menyenangkan adalah bahwa Algoritma masih merupakan area yang masih muda, dan masih ada banyak misteri, dan banyak masalah yang (kami curiga) masih belum tahu algoritma terbaik. Inilah yang membuat penelitian dalam

Algoritma begitu menyenangkan dan mengasyikkan, dan semoga sebagian dari Anda akan memutuskan untuk melanjutkan ke arah ini dan menuliskan algoritma yang efisieien. Jika ditanyakan mengapa anda harus menulis algoritma yang efieien? Maka jawabannya adalah karena :

1. Anda akan menjadi pengembang perangkat lunak yang jauh lebih baik (dan mendapatkan pekerjaan / penghasilan yang lebih baik).

2. Menghabiskan lebih sedikit waktu untuk debugging, mengoptimalkan dan menulis ulang kode.
3. Perangkat lunak Anda akan berjalan lebih cepat dengan perangkat keras yang sama (lebih murah untuk skala).
4. Program Anda mungkin digunakan untuk membantu penemuan yang menyelamatkan nyawa (mungkin?).

1.3 KAIDAH ALGORITMA DALAM KOMPUTASI

Secara informal, suatu algoritma adalah prosedur komputasi yang terdefinisi dengan baik yang mengambil beberapa nilai, atau set nilai, sebagai input dan menghasilkan sejumlah nilai, atau set nilai, sebagai output. Algoritme dengan demikian adalah urutan langkah komputasi yang mengubah input menjadi output.

Algoritma juga dapat dilihat sebagai alat untuk memecahkan masalah komputasi yang ditentukan dengan baik. Pernyataan masalah menentukan secara umum hubungan input / output yang diinginkan. Algoritma ini menjelaskan prosedur komputasi tertentu untuk mencapai hubungan input / output itu.

Sebagai contoh, seseorang mungkin perlu mengurutkan urutan angka ke dalam urutan nondecreasing. Masalah ini sering muncul dalam praktik dan memberikan lahan subur untuk memperkenalkan banyak teknik desain standar dan alat analisis. Inilah cara kami secara resmi mendefinisikan masalah penyortiran:

Input: Urutan n angka $\langle a_1, a_2, ..., a_n \rangle$.

Output: Suatu permutasi (penataan ulang) dari urutan input

Misalnya, mengingat urutan input $\langle 31, 41, 59, 26, 41, 58 \rangle$, algoritma penyortiran kembali sebagai output urutan $\langle 26, 31, 41, 41, 58, 59$ 59. Urutan input seperti itu disebut turunan dari masalah penyortiran. Secara umum, instance dari masalah terdiri dari input (memenuhi batasan apa pun yang dikenakan dalam pernyataan masalah) yang diperlukan untuk menghitung solusi untuk masalah tersebut.

Penyortiran adalah operasi dasar dalam ilmu komputer (banyak program menggunakannya sebagai langkah menengah), dan sebagai hasilnya sejumlah besar algoritma penyortiran yang baik telah dikembangkan. Algoritma mana yang terbaik untuk aplikasi yang diberikan tergantung pada-di antara faktor-faktor lain-jumlah item

yang akan disortir, sejauh mana item sudah agak diurutkan, kemungkinan pembatasan pada nilai item, dan jenis perangkat penyimpanan yang akan digunakan : memori utama, disk, atau kaset.

Suatu algoritma dikatakan benar jika, untuk setiap *instance* (contoh) input, ia berhenti dengan output yang benar. Kami mengatakan bahwa algoritma yang benar memecahkan masalah komputasi yang diberikan. Algoritma yang salah mungkin tidak berhenti sama sekali pada beberapa contoh input, atau mungkin berhenti dengan jawaban selain yang diinginkan. Bertentangan dengan apa yang mungkin diharapkan seseorang, algoritma yang salah terkadang dapat berguna, jika tingkat kesalahannya dapat dikontrol.

Algoritma dapat spesifikasikan dalam bahasa Inggris, sebagai program komputer, atau bahkan sebagai desain perangkat keras. Satu-satunya persyaratan adalah bahwa spesifikasi harus memberikan deskripsi yang tepat tentang prosedur komputasi yang harus diikuti.

1.4 STRUKTUR DASAR ALGORITMA

Secara umum diketahui bahwa jika ingin menyelesaikan menyelesaikan beberapa masalah komputasi maka pertama yang dilakukan adalah mendefinisikan serangkaian langkah yang perlu diikuti untuk menyelesaikan masalah tersebut. Langkah-langkah ini secara kolektif dikenal sebagai algoritma.

Untuk memecahkan berbagai masalah, baik dengan bantuan komputer maupun tanpa bantuan komputer, dibutuhkan algoritma yang terdiri dari serangkaian AKSI. Rangkaian aksi ini bisa dilaksanakan dalam 3 struktur dasar, yaitu:

1. Aksi dilaksanakan secara berturut-turut atau beruntun (*sequence*)
2. Aksi yang akan dipilih atau dilaksanakan ditentukan oleh kondisi tertentu (*selection*)
3. Satu atau serangkaian aksi dilaksanakn secara berulang-ulang selama kondisi tertentu masih terpenuhi (*iteration/loop*). Dengan demikian, struktur perulangan (*iteration/loop*) pasti mengandung struktur pemilihan (*selection*).

Seringkali, untuk memecahkan sebuah masalah dibutuhkan ketiga struktur ini secara sekaligus. Misalnya, dalam contoh 1 berikut adalah masalah menghilangkan rasa haus, terdapat rincian seperti di bawah ini:

Contoh 1.

Input: Sebuah gelas kosong berukuran 200 ml dan botol berukuran 1000 ml penuh air

Algoritma (dalam notasi alami):

1. Angkat botol
2. Tuangkan air ke dalam gelas
3. Periksa kondisi air dalam gelas. Jika gelas belum penuh, lanjutkan ke langkah 4. Jika gelas sudah penuh, lanjutkan ke langkah 5.
4. Periksa kondisi air dalam botol. Jika botol belum kosong, kembali ke langkah 2. Jika botol sudah kosong, lanjutkan ke langkah 5.
5. Letakkan botol air.
6. Periksa air dalam gelas. Jika gelas kosong, lanjutkan ke langkah 11. Jika gelas berisi air, lanjutkan ke langkah 7
7. Angkat gelas.
8. Minum air.
9. Periksa kondisi rasa haus. Jika rasa haus belum hilang, lanjutkan ke langkah 10. Jika rasa haus sudah hilang, lanjutkan ke langkah 11.
10. Periksa kondisi air dalam gelas. Jika gelas belum kosong, kembali ke langkah 8. Jika gelas sudah kosong, kembali ke langkah 1.
11. Letakkan gelas.

Output: Hilangnya rasa haus.

Contoh 2.

Misalkan ada dua bilangan bulat "a" dan "b" dan ingin dicari jumlah dari dua angka itu. Bagaimana kita bisa menyelesaikan ini? Salah satu solusi yang mungkin untuk masalah tersebut adalah:

1. Ambil dua bilangan bulat sebagai input
2. buat variabel "jumlah" untuk menyimpan jumlah dua bilangan bulat
3. letakkan penjumlahan dari kedua variabel itu dalam variabel "penjumlahan"

4. kembalikan variabel "jumlah"

```
int findSum(int a, int b)
{
      int sum; // create a sum variable
            sum = a + b; // save the sum of the sums a and b
      return sum; // return to the sum variable
}
```

Beradasarkan contoh 1 dan contoh 2 diatas maka didapat tiga hal dalam menyelesaikan masalah melalui komputasi ini yaitu adanya input, algoritma dan output, seperti ilustrasi pada gmbar 2.1 berikut ini:

Gambar 1.1 Komponen algoritma

Gambar 2.1 dan kasus di atas dapat dijelaskan dengan mengikuti alur rancangan algoritma tersebut yaitu :

1. **Input**: Input adalah sesuatu yang Anda butuhkan untuk menulis suatu algoritma dan mengubahnya menjadi output yang diinginkan. Sama seperti di mesin di mana Anda memberikan beberapa produk mentah dan mesin mengubah produk mentah menjadi beberapa produk yang diinginkan. Dalam contoh kita, inputnya adalah dua angka yaitu "a" dan "b". Sebelum menulis suatu algoritma, Anda harus menemukan tipe data input, distribusi atau kisaran input dan detail relevan lainnya yang terkait dengannya. Jadi, analisis kritis masukan Anda sebelum menulis solusinya.
2. **Algoritma**: Suatu algoritma adalah langkah-langkah yang didefinisikan dengan baik oleh prosedur langkah yang mengambil beberapa nilai atau set nilai sebagai input dan menghasilkan beberapa nilai atau set nilai sebagai output. Dalam contoh di atas, kami memiliki tiga langkah untuk menemukan jumlah dua angka. Jadi, ketiga langkah tersebut secara kolektif disebut algoritma untuk menemukan jumlah dari dua angka.

3. **Keluaran:** Keluaran adalah hasil yang diinginkan dalam masalah. Sebagai contoh, jika kita menemukan jumlah dua bilangan bulat a dan b maka untuk setiap nilai a dan b itu harus menghasilkan jumlah yang benar sebagai output.

Penyajian Algoritma:

1. Teknik tulisan;
    a. English Structure
    b. Pseudocode
2. Gambar;
    a. Metode structure chart,
    b. Hierarchy plus input-process-output
    c. Flowchart
    d. Nassi Schneiderman chart

English Structure:

a. Menggunakan bahasa manusia
b. Menggambarkan suatu algoritma yang akan dikomunikasikan kepada pemakai sistem

Pseudocode

a. Pseudo= imitasi atau mirip / menyerupai
b. Code=program
c. Kode yang mirip dengan kode pemrograman yang sebenarnya
d. Menggambarkan Algoritma yang akan dikomunikasikan kepada programmer
e. Lebih rinci dari English Structure (mis: dalam menyatakan tipe data yang digunakan)

Structure Chart (bagan terstruktur)

a. Digunakan untuk mendefinisikan dan mengilustrasikan organisasi dari system secara berjenjang
b. Berbentuk modul dan submodul
c. Menunjukkan hubungan elemen data dan elemen control serta hubungan antar modul

Aturan Penulisan Teks Algoritma

Setiap algoritma akan selalu terdiri dari :

a. Judul (header)
b. Deklarasi (kamus)
c. Deskripsi Algoritma

Judul Algoritma ➔ Algoritma NAMA ALGORITMA

{Penjelasan tentang algoritma, berisi uraian singkat cara kerja program, kondisi awal dan akhir dari program} ➔ spesifikasi algoritma

Catatan, dalam menulis nama-nama dalam algoritma harus mempunyai makna yang mencerminkan proses, sifat atau identitas lainnya yang melekat dengan suatu proses, tipe, konstanta, variabel, sub-program dan lain-lainnya. Nama-nama yang bermakna disebut mnemonic.

Deklarasi

(Semua nama yang dipakai, meliputi nama file, nama variable, nama konstanta, nama prosedur serta nama fungsi }

Deskripsi

(Semua langkah/aksi algoritma }

Contoh:

a. Kepala algoritma: Algoritma Luas_Lingkaran { Menghitung luas lingkaran dengan ukuran jari-jari tertentu .Algoritma menerima masukan jejari lingkaran, menghitung luasnya, dan menyajikan hasilnya ke piranti keluaran }
b. Deklarasi algoritma:

Deklarasi { nama konstanta }

const PHI = 3.14; { Nilai phi = 22/7 }

{ nama peubah } var R : real; { input jejari lingkaran bilangan riil } l_Lingkaran : real; { luas lingkaran bilangan riil }

{ nama sub program } procedure TUKAR (input/output A:integer, input/output B:integer)

{Mempertukankan nilai A dan B.Parameter A dan B sudah terdefinisi nilainya.Setelah pertukaran, A berisi nilai B dan B berisi nilai A }

Deskripsi algoritma:

{ Baca data jejari lingkaran R.Jika R <= 0 tulis pesan data salah, selain itu hitung luas ingkaran. Tampilkan luas lingkaran. }

baca(R);

jika R <= 0 then tulis ("Data salah !") selain itu l_Lingkaran = PHI x R x R; tulis(l_Lingkaran);

1.5 PERMASALAHAN dalam BIDANG KOMPUTASI

Penyortiran tidak berarti satu-satunya masalah komputasi yang algoritma telah dikembangkan. (Anda mungkin menduga sebanyak ketika Anda melihat ukuran buku ini.) Aplikasi praktis dari algoritma ada di mana-mana dan termasuk contoh-contoh berikut:

a. Proyek Genom Manusia memiliki tujuan mengidentifikasi semua 100.000 gen dalam DNA manusia, menentukan urutan 3 miliar pasangan basa kimia yang membentuk DNA manusia, menyimpan informasi ini dalam basis data, dan mengembangkan alat untuk analisis data. Setiap langkah ini membutuhkan algoritma yang canggih. Sementara solusi untuk berbagai masalah yang terlibat berada di luar cakupan buku ini, ide-ide dari banyak bab dalam buku ini digunakan dalam solusi dari masalah biologis ini, dengan demikian memungkinkan para ilmuwan untuk menyelesaikan tugas sambil menggunakan sumber daya secara efisien. Penghematannya dalam waktu, baik manusia dan mesin, dan dalam uang, karena lebih banyak informasi dapat diambil dari teknik laboratorium.
b. Internet memungkinkan orang di seluruh dunia untuk dengan cepat mengakses dan mengambil sejumlah besar informasi. Untuk melakukannya, algoritma pintar digunakan untuk mengelola dan memanipulasi volume data yang besar ini. Contoh masalah yang harus dipecahkan termasuk menemukan rute yang baik di mana data akan melakukan perjalanan

c. Perdagangan elektronik memungkinkan barang dan jasa dinegosiasikan dan dipertukarkan secara elektronik. Kemampuan untuk menjaga informasi seperti nomor kartu kredit, kata sandi, dan laporan bank tetap penting jika perdagangan elektronik ingin digunakan secara luas. Kriptografi kunci publik dan tanda tangan digital adalah beberapa teknologi inti yang digunakan dan didasarkan pada algoritma numerik dan teori bilangan.

d. Dalam pembuatan dan pengaturan komersial lainnya, seringkali penting untuk mengalokasikan sumber daya yang langka dengan cara yang paling menguntungkan. Sebuah perusahaan minyak mungkin ingin tahu di mana menempatkan sumurnya untuk memaksimalkan keuntungan yang diharapkan. Seorang calon untuk kepresidenan Amerika Serikat mungkin ingin menentukan di mana harus mengeluarkan uang untuk membeli iklan kampanye untuk memaksimalkan peluang memenangkan pemilihan. Sebuah maskapai penerbangan mungkin ingin menugaskan kru untuk penerbangan dengan cara semurah mungkin, memastikan bahwa setiap penerbangan tercakup dan bahwa peraturan pemerintah tentang penjadwalan kru terpenuhi. Penyedia layanan Internet mungkin ingin menentukan di mana menempatkan sumber daya tambahan untuk melayani pelanggannya secara lebih efektif. Semua ini adalah contoh masalah yang dapat dipecahkan dengan menggunakan pemrograman linier,

Sementara beberapa detail dari contoh-contoh ini berada di luar cakupan buku ini, kami memberikan teknik-teknik mendasar yang berlaku untuk masalah-masalah dan bidang-bidang masalah ini. Kami juga menunjukkan cara mengatasi banyak masalah nyata dalam buku ini, termasuk yang berikut:

a. Kami diberi peta jalan tempat jarak antara setiap pasangan persimpangan yang berdekatan ditandai, dan tujuan kami adalah menentukan rute terpendek dari satu persimpangan ke persimpangan lainnya. Jumlah rute yang mungkin bisa sangat besar, bahkan jika kita melarang rute yang melintasi diri mereka sendiri. Bagaimana kita memilih rute mana yang paling pendek? Di sini, kami memodelkan peta jalan (yang merupakan model dari jalan sebenarnya) sebagai grafik (yang akan kita temui pada Bab 10 dan Lampiran B), dan kami ingin menemukan jalur terpendek dari satu titik ke titik lainnya dalam grafik.

b. Kami diberi urutan 〈A1, A2, ..., An〉 dari n matriks, dan kami ingin menentukan produk mereka A1 A2 An. Karena perkalian matriks adalah asosiatif, ada beberapa perintah perkalian hukum. Sebagai contoh, jika n = 4, kita dapat melakukan perkalian matriks seolah-olah produk dipasangkan dalam salah satu dari perintah berikut: (A1 (A2 (A3A4)))), (A1 ((A2A3) A4)),, ((A1A2) (A3A4)), ((A1 (A2A3)) A4), atau (((A1A2) A3) A4). Jika semua matriks ini berbentuk bujur sangkar (dan karenanya memiliki ukuran yang sama), urutan perkalian tidak akan mempengaruhi berapa lama perkalian matriks. Namun, jika matriks ini memiliki ukuran yang berbeda (namun ukurannya kompatibel untuk perkalian matriks), maka urutan perkalian dapat membuat perbedaan yang sangat besar. Jumlah pesanan multiplikasi yang mungkin adalah eksponensial dalam n, dan dengan demikian mencoba semua pesanan yang mungkin membutuhkan waktu yang sangat lama. Kita akan melihat di Bab 15 cara menggunakan teknik umum yang dikenal sebagai pemrograman dinamis untuk menyelesaikan masalah ini jauh lebih efisien.

c. Kita diberi sumbu persamaan ≡ b (mod n), di mana a, b, dan n adalah bilangan bulat, dan kami ingin menemukan semua bilangan bulat x, modulo n, yang memenuhi persamaan tersebut. Mungkin ada nol, satu, atau lebih dari satu solusi semacam itu. Kita cukup mencoba x = 0, 1, ..., n - 1 secara berurutan.

d. Kami diberi n poin di pesawat, dan kami ingin menemukan lambung cembung dari titik-titik ini. Lambung cembung adalah poligon cembung terkecil yang mengandung titik-titik. Secara intuitif, kita dapat menganggap setiap titik diwakili oleh paku yang mencuat dari papan. Lambung cembung akan diwakili oleh karet gelang ketat yang mengelilingi semua paku. Setiap paku yang dililit karet gelang adalah simpul cembung cembung. Salah satu dari himpunan bagian 2n dari titik-titik tersebut mungkin merupakan simpul dari cembung cembung. Mengetahui titik mana yang merupakan simpul dari cembung cembung juga tidak cukup, karena kita juga perlu mengetahui urutan kemunculannya. Ada banyak pilihan, oleh karena itu, untuk simpul cembung cembung.

Daftar ini jauh dari lengkap (seperti yang Anda duga dari tumpukan buku ini), tetapi tunjukkan dua karakteristik yang umum untuk banyak algoritma yang menarik.

a. Ada banyak solusi kandidat, yang sebagian besar bukan yang kita inginkan. Menemukan satu yang kita inginkan dapat memberikan tantangan yang cukup besar.
b. Ada aplikasi praktis. Dari masalah dalam daftar di atas, jalur terpendek memberikan contoh termudah. Perusahaan transportasi, seperti perusahaan angkutan truk atau kereta api, memiliki kepentingan finansial dalam menemukan jalur terpendek melalui jalan atau jaringan kereta api karena mengambil jalur yang lebih pendek menghasilkan biaya tenaga kerja dan bahan bakar yang lebih rendah. Atau simpul perutean di Internet mungkin perlu menemukan jalur terpendek melalui jaringan untuk merutekan pesan dengan cepat.

1.6 PERANCANGAN ALGORITMA

Algoritma merupakan seperangkat instruksi atau logika yang terbatas, yang ditulis secara berurutan, untuk menyelesaikan tugas yang telah ditentukan sebelumnya. Algoritma bukanlah kode atau program yang lengkap, itu hanya logika inti (solusi) dari suatu masalah, yang dapat dinyatakan sebagai deskripsi tingkat tinggi secara informal sebagai pseudocode atau menggunakan diagram alur. Untuk menghasilkan sebuah algoritma maka algoritma harus diketahui bagaimana cara membuatnya.

Mengapa perlu mempelajari algoritma? Jawabannya adalah, jika Anda akan menjadi seorang profesional di bidang komputer maka ada alasan praktis dan teoretis yang membuat anda harus mempelajari algoritma. Jika dilihat dari sudut pandang praktis, anda harus mengetahui serangkaian standar algoritma yang penting dari berbagai bidang komputasi; selain itu, anda juga harus mampu mendisain algoritma baru dan menganalisis efisiensinya, sedangkan dari sudut pandang teoretis, algoritma telah mendapat pengakuan sebagai landasan bagi ilmu komputer.

David Harel, dalam bukunya yang berjudul *Algorithmics: the Spirit of Computing* menuliskan bahwa : Algoritma lebih dari cabang ilmu komputer. Ini adalah inti dari ilmu komputer, dan, dalam semua kelayakannya, dapat dikatakan bahwa relevan dengan sebagian besar ilmu pengetahuan, bisnis, dan teknologi. [Har92, p. 6]

Berdasarkan yang disampaikan oleh David Harel, tidak menutup kemungkinan bahwa walaupun anda bukan seorang mahasiswa komputer namun, ada alasan yang

menarik untuk mempelajari algoritma. Pengakuan harus diberikan bahwa program komputer tidak akan ada tanpa algoritma. Dan dengan aplikasi komputer menjadi unsur utama yang paling penting bagi seseorang untuk menjadi profesional dan pribadi, mempelajari algoritma menjadi kebutuhan bagi semakin banyak orang. Alasan lain untuk mempelajari algoritma adalah kegunaannya dalam mengembangkan keterampilan analitik. Lagi pula, algoritma dapat dilihat sebagai jenis solusi khusus untuk masalah utama—bukan hanya jawaban tetapi juga prosedur yang ditentukan untuk mendapatkan jawaban. Oleh karena itu, teknik khusus yang dirancang untuk teknik dapat diinterpretasikan sebagai strategi penyelesaian masalah yang dapat berguna terlepas dari apakah komputer dilibatkan. Tentu saja, presisi secara inheren dipaksakan oleh pemikiran algoritmik membatasi berbagai masalah yang dapat diselesaikan dengan suatu algoritma. Anda tidak akan menemukan, misalnya, algoritma untuk menjalani kehidupan yang bahagia atau menjadi kaya dan terkenal jika anda tidak mau membuat dan mendisainnya.

Algoritma yang ada bukanlah lahir dengan serta merta namun ada prosesnya. Setiap Algoritma harus memenuhi dimensi berikut:

a. ***Input***- Harus ada 0 atau lebih input yang dipasok secara eksternal ke algoritma.
b. ***Output***- Harus ada minimal 1 output yang diperoleh.
c. ***Definiteness***- Setiap langkah dari algoritma harus jelas dan didefinisikan dengan baik sehingga suatu algoritma yang dibuat tidak ambigu..
d. ***Finiteness*** - Algoritme harus memiliki jumlah langkah yang terbatas.
e. ***Correctness***- Setiap langkah dari algoritma harus menghasilkan output yang benar.

Algoritma harus dirancang/didisain agar dapat memberikan kontribusi dalam menyelesaikan permasalahan. Prinsip dasar desain algoritma telah pernah diungkapkan oleh ahli matematika George_Polya: yang menyatakan bahwa "Jika ada masalah yang tidak bisa Anda pecahkan, maka ada masalah yang lebih mudah yang Anda bisa pecahkan: karenanya temukanlah."

Sebagai contoh, misalkan kita ingin menemukan elemen maksimum dari array n ints, maka masalah tersebut dapat diselesaikan dengan mengamati bahwa elemen maksimum adalah (a) elemen terakhir, atau (b) maksimum elemen n-1 pertama, tergantung mana yang lebih besar. Dan untuk mendapatkan maksimum dengan (b)

maka merupakan versi yang lebih mudah dari masalah aslinya. Berikut contoh kodingnya:

```
int
max_element(int a[], int n)
{
    int prefix_max;

    assert(n > 0);

    if(n == 1) {
        return a[0];
    } else {
        prefix_max = max_element(a, n-1);
        if(prefix_max < a[n-1]) {
            return a[n-1];
        } else {
            return prefix_max;
        }
    }
}
```

Perhatikan bahwa kita memerlukan case khusus untuk array 1-elemen, karena awalan kosong array tersebut tidak memiliki elemen maksimum. Kami juga menyatakan bahwa array berisi setidaknya satu elemen, hanya untuk menghindari kerusakan.

Satu masalah dengan algoritma ini (setidaknya ketika coding dalam C) adalah bahwa rekursi bisa menjadi sangat dalam. Untungnya, ada cara mudah untuk mengubah rekursi menjadi loop. Idenya adalah bahwa alih-alih mengembalikan nilai dari panggilan rekursif, kita memasukkannya ke dalam variabel yang akan digunakan pada pass berikutnya melalui loop. Hasilnya adalah

```
int
max_element(int a[], int n)
{
    int i;    /* mengganti n-1 dari versi recursive */
    int prefix_max;

    assert(n > 0);
    prefix_max = a[0];  /* ini adalah i == 0 case */

    for(i = 1; i < n; i++) {
        if(prefix_max < a[i]) {
            prefix_max = a[i];  /* kembali ke  a[n-1] */
        }
        /* else case menjadi prefix_max = prefix_max, a noop */
```

```
    }
    /* akhirnya kita mempunyai nilai return untuk bilangan real */
    return prefix_max;
}
```

Desain algoritma sering membutuhkan kreativitas dan pengetahuan khusus terhadap masalah yang akan dicari solusinya, tetapi ada beberapa teknik umum tertentu yang muncul berulang kali. Berikut ini adalah beberapa teknik umum dalam penyelesaian masalah yang klasifikasinya diadaptasi dari buku yang ditulis oleh Levitin:

a. BruteForce: Coba semua solusi yang mungkin sampai Anda menemukan yang tepat.
b. Divide and Conquer: Pisahkan masalah menjadi dua atau lebih subproblem, selesaikan subproblem secara rekursif, lalu gabungkan solusinya.
c. Decrease and Conquer: Kurangi masalah menjadi satu masalah yang lebih kecil, selesaikan masalah itu secara rekursif, dan kemudian gunakan solusi itu untuk memecahkan masalah yang asli.
d. Transform and Conquer: diantara dua hal ini yaitu (a) mengubah input ke bentuk yang membuat masalah mudah dipecahkan, atau (b) mentransformasikan input menjadi input ke masalah lain yang solusinya memecahkan masalah aslinya.
e. UseSpace: Memecahkan masalah menggunakan beberapa struktur data tambahan.
f. Dynamic Programming: Buat tabel solusi untuk subproblem yang semakin besar, di mana setiap entri baru dalam tabel dihitung menggunakan entri sebelumnya dalam tabel.
g. GreedyMethod: Jalankan masalah Anda selangkah demi selangkah, catat solusi tunggal terbaik di setiap langkah. Semoga dengan tulus bahwa ini tidak akan menuntun Anda untuk membuat pilihan yang tampaknya baik awal dengan konsekuensi buruk nanti.

Beberapa pendekatan tersebut bekerja lebih baik daripada yang lain --- itu adalah peran analisa Algorithm (dan percobaan dengan komputer nyata) untuk mencari tahu mana yang mungkin benar dan efisien dalam praktiknya. Tetapi terlepas dari semua itu anda dapat mencoba berbagai kemungkinan untuk solusi bagi masalah yang diberikan. Berikut ini adalah dua contohnya :

Contoh 1: Mencari maksimum

Meskipun klasifikasi ini tidak sepenuhnya terdefinisi dengan baik, dan agak arbitrer untuk beberapa algoritma, itu memang memberikan daftar hal-hal yang berguna untuk mencoba memecahkan masalah. Berikut adalah beberapa contoh penerapan pendekatan yang berbeda untuk masalah sederhana, masalah menemukan maksimum array bilangan bulat.

a. BruteForce: Untuk indeks i, uji apakah A [i] lebih besar dari atau sama dengan setiap elemen dalam array. Ketika Anda menemukan A [i] seperti itu, kembalikan. Untuk algoritma ini, T (n) = n * Theta (n) = Theta (n2) jika diterapkan dengan cara yang paling alami.
b. Divide and Conquer: Jika A hanya memiliki satu elemen, kembalikan. Kalau tidak, biarkan m1 menjadi maksimum A [1] .. A [n / 2]. Biarkan m2 menjadi maksimum A [n / 2 + 1] .. A [n]. Kembalikan yang lebih besar dari m1 dan m2. Waktu berjalan diberikan oleh T (n) = 2T (n / 2) + Theta (1) = Theta (n).
c. Decrease and Conquer: Jika A hanya memiliki satu elemen, kembalikan. Kalau tidak, biarkan m menjadi maksimum A [2] .. A [n]. Kembalikan yang lebih besar dari A [0] dan m. Sekarang waktu berjalan diberikan oleh T (n) = T (n-1) + Theta (1) = Sigmai = 1 ke n Theta (1) = Theta (n).
d. Transform And Conquer: Urutkan array, lalu kembalikan A [n]. Menggunakan semacam berbasis perbandingan yang optimal, ini membutuhkan waktu Theta (n log n) + Theta (1) = Theta (n log n). Keuntungan dari pendekatan ini adalah Anda mungkin tidak perlu membuat kode semacam itu.
e. UseSpace: Masukkan semua elemen ke dalam pohon pencarian biner seimbang, lalu kembalikan elemen paling kanan. Biaya adalah Theta (n log n) untuk melakukan n penyisipan, ditambah Theta (log n) untuk menemukan elemen paling kanan, untuk total Theta (n log n). Penyortiran setara dan mungkin lebih mudah.
f. Dynamic Programming: Buat array B bantu dengan indeks 1 ke n. Set B [1] = A [1]. Saat saya beralih dari 2 ke n, atur B [i] ke yang lebih besar dari B [i-1] dan A [i]. Kembali B [n]. Biaya: Theta (n).

g. GreedyMethod: Biarkan maks = A [1]. Untuk setiap elemen A [i] dalam A [2..n], jika A [i]> maks, atur maks = A [i]. Kembalikan nilai akhir maks. Biaya: Theta (n); Algoritma ini cukup identik dengan yang sebelumnya.

Contoh 2 : Penyortiran

Masalah penyortiran meminta, diberikan array n elemen dalam urutan acak, untuk array yang berisi n elemen yang sama dalam urutan nondecreasing, yaitu dengan A [i] <= A [i +1] untuk semua i. Kita dapat menerapkan masing-masing teknik di atas untuk masalah ini dan mendapatkan algoritma pengurutan (meskipun beberapa tidak terlalu bagus).

a. BruteForce: Untuk setiap n! permutasi dari input, uji apakah diurutkan dengan memeriksa A [i] <= A [i + 1] untuk semua i. Biaya jika diterapkan secara naif: n! * Theta (n) = Theta (n * n!). Algoritma ini dikenal sebagai monkeysort deterministik atau bogosort deterministik. Ini juga memiliki varian acak, di mana generasi semua n! permutasi digantikan oleh pengocokan. Varian acak lebih mudah untuk dikodekan dan berjalan tentang faktor dua lebih cepat dari varian deterministik, tetapi tidak menjamin penghentian jika pengocokan secara konsisten tidak beruntung.
b. DivideAndConquer: Sortir A [1..floor (n / 2)] dan A [lantai (n / 2 + 1) .. n] secara terpisah, kemudian gabungkan hasilnya (yang membutuhkan waktu Theta (n) dan Theta (n) tambahan ruang jika diimplementasikan dengan cara yang paling mudah). Biaya: T (n) = 2T (n / 2) + Theta (n) = Theta (n log n) oleh Teorema Master. Algoritma ini dikenal sebagai MergeSort, dan merupakan salah satu algoritma pengurutan tujuan umum tercepat. Penggabungan dapat dihindari dengan hati-hati memisahkan array menjadi elemen kurang dari dan elemen lebih besar dari beberapa pivot, lalu mengurutkan dua tumpukan yang dihasilkan; ini memberi QuickSort. Kinerja QuickSort seringkali lebih cepat daripada MergeSort dalam praktiknya, tetapi kinerja terburuknya (ketika pivot dipilih dengan buruk) sama buruknya dengan hasil dari:
c. DecreaseAndConquer: Hapus A [n], urutkan sisanya, lalu masukkan A [n] di tempat yang sesuai. Algoritma ini disebut InsertionSort. Langkah penyisipan

akhir membutuhkan menemukan tempat yang tepat (yang dapat dilakukan dengan cukup cepat jika ada yang pintar) tetapi kemudian naik ke elemen n-1 untuk memberikan ruang bagi A [n]. Total biaya diberikan oleh T (n) = T (n-1) + Theta (n) = T (n2).

d. Transform And Conquer: Saya tidak mengetahui adanya pendekatan TransformAndConquer umum yang baik untuk menyortir (ada beberapa yang buruk), tetapi dalam beberapa kasus seseorang dapat mengubah masalah penyortiran yang tampaknya umum (misalnya menyortir string) menjadi masalah penyortiran khusus yang memungkinkan solusi lebih cepat (misalnya menyortir bilangan bulat kecil).
e. Use Space: Masukkan elemen ke dalam pohon pencarian biner seimbang, lalu bacakan dari kiri ke kanan. Versi lain: masukkan ke tumpukan. Keduanya membutuhkan waktu Theta (n log n), tetapi lebih rumit untuk diterapkan daripada MergeSort atau QuickSort kecuali Anda memiliki kode BST atau heap yang sudah ada.
f. DynamicProgramming: (Insertion Sort dapat dilihat sebagai contohnya.)
g. GreedyMethod: Temukan elemen terkecil, tandai sebagai digunakan, dan output. Ulangi sampai tidak ada elemen yang tersisa. Hasilnya adalah SelectionSort, yang berjalan dalam waktu Theta (n2) yang terhormat tapi suboptimal.

1.7 ANALISA ALGORITMA

Analisa algoritma secara teoritik merupakan suatu cara untuk memperkirakan kompleksitas suatu algoritma dalam arti asimptotik, yaitu memperkirakan fungsi kompleksitas untuk input besar yang kadang diluar perkiraan dan kadang tidak berdasarkan akal sehat. Istilah "analisis algoritma" diciptakan oleh Donald Knuth.

Seorang programer atau sistem analisis paling tidak harus memiliki dasar untuk menganalisis algoritma. Analisis algoritma sangat membantu di dalam meningkatkan efisiensi program. Kecanggihan suatu program bukan dilihat dari tampilan program, tetapi berdasarkan efisiensi algoritma yang terdapat didalam program tersebut. Analisis algoritma telah menjadi bagian penting dari teori kompleksitas komputasi, yang

menyediakan estimasi teoritis untuk sumber daya yang diperlukan dari suatu algoritma dalam menyelesaikan masalah komputasi tertentu. Sebagian besar algoritma dirancang untuk bekerja dengan input yang panjangnya diluar perkiraan.

Analisis algoritma juga merupakan penentuan jumlah waktu dan sumber daya ruang yang diperlukan dalam mengesekusi dan menjalankannya. Algoritma dikatakan efisien dan cepat, jika membutuhkan waktu lebih sedikit untuk mengeksekusi dan menghabiskan lebih sedikit ruang memori. Kinerja suatu algoritma diukur berdasarkan sifat-sifat berikut:

a. **Kompleksitas Waktu**: dimana berfungsi untuk mengetahui berapa banyak waktu yang diperlukan untuk menjalankan suatu algoritma untuk data yang ada dan bagaimana ia akan tumbuh berdasarkan ukuran data (atau dalam beberapa kasus faktor-faktor lain seperti jumlah digit dan lain-lain).
b. **Kompleksitas ruang**: dimana terkait dengan memori yang sangat sangat bagus sehingga harus diketahui berapa banyak ruang kosong yang dibutuhkan untuk algoritma ini dan seperti waktu juga harus mampu melacak pertumbuhannya.

Efisiensi suatu algoritma tergantung pada jumlah waktu, penyimpanan dan sumber daya lain yang diperlukan untuk menjalankan algoritma. Efisiensi diukur dengan bantuan **notasi asimptotik**. Penggunaan notasi asimtotik seperti O-besar (Big O), Θ (.), Ω (.), o (.) dan ω (.) dalam ilmu komputer dan pengembangan perangkat lunak biasanya untuk menggambarkan efisiensi algoritma melalui kompleksitas waktu dan ruangnya. Khususnya ketika menggunakan notasi Big-O maka dapat menggambarkan efisiensi algoritma yang berhubungan dengan input: n, biasanya sebagai n mendekati tak terhingga. Saat memeriksa algoritma, biasanya yang diinginkan adalah kompleksitas waktu dan ruang yang lebih rendah. Kompleksitas waktu O(1) menunjukkan waktu yang konstan. Melalui perbandingan dan analisis algoritma maka kita dapat membuat aplikasi yang kinerjanya lebih efisien. Studi tentang perubahan kinerja algoritma dengan perubahan ukuran input dapat dikenal sebagai **analisis asimptotik**

Buku Ajar ini akan membahas kebutuhan untuk analisis algoritma dan bagaimana memilih algoritma yang lebih baik untuk masalah tertentu karena satu masalah komputasi dapat diselesaikan dengan algoritma yang berbeda.

Algoritma seringkali sangat berbeda satu sama lain, meskipun tujuan dari algoritma ini adalah sama. Misalnya, ditahui bahwa satu set angka dapat diurutkan menggunakan algoritma yang berbeda. Jumlah perbandingan yang dilakukan oleh satu algoritma dapat berbeda dengan yang lain untuk input yang sama. Oleh karena itu, kompleksitas waktu dari algoritma tersebut mungkin berbeda. Pada saat yang sama, kita perlu menghitung ruang memori yang dibutuhkan oleh masing-masing algoritma.

Analisis algoritma adalah proses menganalisis kemampuan pemecahan masalah dari algoritma dalam hal waktu dan ukuran yang diperlukan (ukuran memori untuk penyimpanan saat implementasi). Namun, perhatian utama dari analisis algoritma adalah waktu atau kinerja yang diperlukan. Secara umum, analisis performa yang dilakukan ada 4 jenis yaitu :

a. **Worst-case**- Jumlah maksimum langkah yang diambil pada ukuran instance apapun.
b. **Best-case** - Jumlah minimum langkah yang diambil pada setiap instance ukuran.
c. **Average case** - Jumlah rata-rata langkah yang diambil pada ukuran apa pun.
d. **Amortisasi**-Urutan operasi yang diterapkan pada input ukuran rata-rata seiring waktu.

Untuk mengatasi masalah, perlu dipertimbangkan waktu dan juga kompleksitas ruang karena program dapat berjalan pada sistem di mana memori terbatas tetapi ruang yang memadai tersedia atau mungkin sebaliknya. Sebagai contoh, jika dibandingkan bubble sort dan merge sort, berdasarkan hasil analisa dapat diketahui bahwa Bubble sort tidak membutuhkan memori tambahan, tetapi merge sort membutuhkan ruang tambahan. Meskipun kompleksitas waktu dari jenis bubble sort lebih tinggi dibandingkan dengan merge sort, namun boleh jadi dimungkinkan perlu menerapkan jenis bubble sort jika suatu program perlu dijalankan pada suatu lingkungan tertentu, di mana memori sangat terbatas.

1.8 RANGKUMAN

1. Langkah pertama menuju pemahaman tentang mengapa studi dan pengetahuan tentang algoritma sangat penting adalah mendefinisikan dengan tepat apa yang dimaksudkan dengan suatu algoritma.
2. Menurut buku teks algoritma populer, Pengantar Algoritma (Edisi Kedua oleh Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein),

"suatu algoritma adalah setiap prosedur komputasi terdefinisi dengan baik yang mengambil beberapa nilai, atau serangkaian nilai, sebagai input dan menghasilkan beberapa nilai, atau set nilai sebagai output. "

3. Algoritma yang terdiri dari serangkaian AKSI. Rangkaian aksi ini bisa dilaksanakan dalam 3 struktur dasar, yaitu:
   a. Aksi dilaksanakan secara berturut-turut atau beruntun (*sequence*)
   b. Aksi yang akan dipilih atau dilaksanakan ditentukan oleh kondisi tertentu (*selection*)
   c. Satu atau serangkaian aksi dilaksanakn secara berulang-ulang selama kondisi tertentu masih terpenuhi (*iteration/loop*). Dengan demikian, struktur perulangan (*iteration/loop*) pasti mengandung struktur pemilihan (*selection*).
4. Algoritma seringkali sangat berbeda satu sama lain, meskipun tujuan dari algoritma ini adalah sama. Misalnya, ditahui bahwa satu set angka dapat diurutkan menggunakan algoritma yang berbeda.
5. Jumlah perbandingan yang dilakukan oleh satu algoritma dapat berbeda dengan yang lain untuk input yang sama. Oleh karena itu, kompleksitas waktu dari algoritma tersebut mungkin berbeda. Pada saat yang sama, kita perlu menghitung ruang memori yang dibutuhkan oleh masing-masing algoritma
6. Dimensi dari analisa algoritma adalah correctness dan efisiensi.

1.9 TEST FORMATIF

a. Jelaskan apa yang dimaksud dengan Algoritma
b. Tuliskan dan jelaskan struktur dasar algoritma
c. Sebutkan dan jelaskan factor-faktor utama algoritma
d. Tuliskan Algoritma untuk mencari nilai maksimum dan minimum
e. Jelaskan mengapa analisa algoritma harus *correctness* dan efisiensi
f. Tuliskan Algoritma mengambil uang di ATM.
g. Buatlah Algoritma dan program yang memiliki fungsi:
   a. Membaca *input* berupa skor (angka bilangan bulat) dari seorang mahasiswa
   b. Menampilkan nilai (dalam huruf) dari mahasiswa tsb.

   Adapun syarat-syarat pemberian nilai adalah sebagai berikut:
   - Jika skor > 90 nilai: **A**

- Jika 86 <= skor <= 90 nilai: **A-**
- Jika 81 <= skor <= 85 nilai: **B+**
- Jika 76 <= skor <= 80 nilai: **B**
- Jika 71 <= skor <= 75 nilai: **B-**
- Jika 66 <= skor <= 70 nilai: **C+**
- Jika 61 <= skor <= 65 nilai: **C**
- Jika 56 <= skor <= 60 nilai: **C-**
- Jika 46 <= skor <= 55 nilai: **D**
- Jika skor <= 45 nilai: **E**

h. Buatlah algoritma dan program untuk menentukan R dari rumus: $\frac{1}{R} = \frac{1}{R1} + \frac{1}{R2} + \frac{1}{\frac{1}{R3}}$, dan jika diketahui R1= $\frac{2}{5}$ x dua angka terakhir NIM anda , R2=$\frac{3}{7}$ dan R3 = 3

i. Buatlah algoritma dan program untuk menghitung total pendapatan bulanan seorang karyawan dengan ketentuan sebagai berikut:

- Tunjangan istri/suami = 10% dari gaji pokok
- Tunjangan anak = 5% dari gaji pokok untuk setiap anak
- THR = Rp 5000 kali masa kerja (tahun)
- (-) Pajak = 15% dari gaji pokok, tunjangan istri & anak
- Bantuan transport = Rp 3000 kali masuk kerja (hari)
- (-) Polis asuransi = Rp 20000

tanda (-) artinya mengurangi pendapatan.

j. Buatlah program untuk algoritma berikut :

```
ALGORITMA FloorAkar (n)
// Menghitung floor dari akar kuadrat n dengan aritmetika dasar
// Input : bilangan integer positif n
// Output : bilangan integer hasil dari floor akar kuadrat n

      while i*i < n do
i = i + 1         // mencari nilai yang mendekati akar n
      if n mod i = 0 then
     print i            // jika i secara bulat adalah akar n
     else
```

```
        print i – 1 // jika i di-floor
      end.
```

k. Buatlah program untuk algoritma berikut :

```
ALGORITMA mencari_jarak_terdekat(A[n])
// input : Array A[0.....n-1] bilangan
// output : jarak terpendek dari dua elemen
dmin ← ∞
for i ← 0 to n-1 do
    for j ← i+1 to n-2 do
      if |A[i] = A[j]|
          return dmin ← 0
    else |A[i]-A[j]| < dmin
            dmin ← |A[i]-A[j]|
return dmin
```

BAB II
ANALISIS KOMPLEKSITAS ALGORITMA

a) Deskripsi Singkat

Penyelesaian suatu masalah dengan algoritma terkadang memiliki banya cara dan kita harus benar-benar memahami masalah tersebut. Kita perlu belajar bagaimana membandingkan kinerja berbagai algoritma dan memilih yang terbaik untuk menyelesaikan masalah tertentu. Saat menganalisis suatu algoritma, sebagian kita biasanya mempertimbangkan kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu dari suatu algoritma mengkuantifikasi jumlah waktu yang dibutuhkan oleh suatu algoritma untuk dijalankan sebagai fungsi dari panjang input. Demikian pula, kompleksitas ruang dari suatu algoritma mengkuantifikasi jumlah ruang atau memori yang diambil oleh suatu algoritma untuk dijalankan sebagai fungsi dari panjang input. Kompleksitas waktu dan ruang tergantung pada banyak hal seperti perangkat keras, sistem operasi, prosesor, dll. Namun, kami tidak mempertimbangkan faktor-faktor ini saat menganalisis algoritma. Kami hanya akan mempertimbangkan waktu eksekusi suatu algoritma. Bab ini akan membahas tentang notasi-notasi yang ada dalam kompleksitas waktu, bagaimana kompleksitas waktu dihitung dan cara menemukan komplesitas waktu dari suatu algoritma.

b) Relevansi

Keterkaitan materi dengan proses pembelajaran untuk materi yang ada pada Bab II ini dapat dilihat dari metode ceramah, diskusi, tanya jawab dan praktikum. Pemahaman mahasiswa melalui metode ceramah dan diskusi serta Tanya jawaba dapat memberikan pemahaman dan kemampuan mahasiswa untuk menentukan kompleksitas waktu dan ruang dari algoritma. Selain itu tugas mandiri juga dapat diberikan dan untuk mengimplementasikan pemahaman mahasiswa terkait materi maka praktikum dilaksanakan sebagai unjuk kerja keahlian mahasiswa dalam pemrograman.

c) Capaian Pembelajaran

1. Mahasiswa mampu menentukan kompleksitas waktu algoritma,
2. Mahasisw mampu menentukan kompleksitas ruang algoritma

3. Mahasiswa mampu menentukan kompleksitas waktu *asymptotic* dari sebuah algoritma
4. Mahasiswa mampu menghitung kompleksitas waktu asimptotik dari algoritma rekursif

2.1 PENDAHULUAN

Setiap hari banyak masalah yang kita temukan baik masalah untuk diri kita sendiri maupun masalah yang terkait dengan masyarakat. Masalahnya dapat beragam seperti masalah sosial, budaya, politik, hukum, pendidikan, transportasi dan lain sebagainya. Masalah-masalah tersebut biasanya dapat ditemukan solusinya baik solusi dalam rumusan biasa maupun solusi dalam rumusan komputasi. Solusi yang ditemukan terkadang lebih dari satu solusi. Beberapa solusi mungkin lebih efisien dibandingkan dengan solusi yang lain dan beberapa solusi mungkin kurang efisien. Secara umum, pastinya yang digunakan adalah solusi yang paling efisien.

Sebagai contoh, saat kita pergi dari rumah ke kantor atau sekolah atau perguruan tinggi, maka kemungkinan ada "n" jumlah jalur, karena ada n jalur tentunya kita akan memilih hanya satu jalur untuk pergi ke tujuan kita yaitu jalur terpendek dan dengan biaya yang murah.

Analogi tersebut dapat kita terapkan dalam kasus masalah komputasi atau pemecahan masalah melalui komputer. Jika kita memiliki satu masalah komputasi maka kita dapat merancang berbagai solusi sebagai penyelesaiannya, misalkan kita merancangnya dalam bentuk Algoritma dan nantinya kita akan memilih algoritma yang paling efisien dari algoritma yang dikembangkan tersebut.

Berdasarkan hal tersebut maka Gagasan kritis untuk dipikirkan adalah

a. Apa itu masalah komputasi?
b. Bagaimana kita mengekstrak detail yang relevan dan mengubah masalah kehidupan nyata menjadi masalah komputasi?

Jawaban dari kedua gagasan tersebut adalah dengan merancang algoritma-algoritma dan kemudian menganalisa melalui perhitungan kompleksitasnya baik itu kompleksitas waktu maupun kompleksitas ruang.

Kompleksitas Waktu merupakan cara untuk merepresentasikan jumlah waktu yang diperlukan oleh program untuk dijalankan hingga selesai. Hal ini merupakan

praktek yang baik untuk mencoba menjaga waktu yang diperlukan berada pada waktu minimum, sehingga algoritma mampu dimungkinkan menyelesaikan eksekusi dalam waktu minimum. Kompleksitas waktu dari algoritma paling umum diekspresikan menggunakan notasi O besar atau ditulis (Big-O). Big-O merupakan salah satu notasi asimptotik yang sering digunakan untuk merepresentasikan kompleksitas waktu selain omega (Ω) dan theta (θ).

Kompleksitas waktu bisa didapatkan dengan "menghitung" jumlah operasi yang dilakukan oleh kode pada program yang dibuat. Kompleksitas waktu ini didefinisikan sebagai fungsi dari ukuran input n menggunakan notasi Big-O. n menunjukkan ukuran input, sedangkan O adalah fungsi laju pertumbuhan skenario terburuk.

Penggunaan notasi Big-O adalah untuk mengklasifikasikan algoritma berdasarkan waktu berjalan atau ruang (memori yang digunakan) saat input bertambah. Fungsi O adalah tingkat pertumbuhan dalam fungsi dalam ukuran input n.

Sebagai ilustri awal untuk masalah apa pun yang ditentukan, mungkin ada sejumlah N solusi. Ini berlaku secara umum. Jika kita memiliki masalah dan kita membahas masalah dengan semua teman atau orang lain, maka mereka semua akan menyarankan kepada kita solusi yang berbeda, dan kita yang harus memutuskan solusi mana yang terbaik berdasarkan keadaan-keadaan yang ada.

Demikian juga untuk masalah apa pun yang harus diselesaikan dengan menggunakan program, boleh jadi ada sejumlah solusi yang tak terbatas. Salah satu solusi untuk masalah ini adalah, menjalankan loop untuk n kali, dimulai dengan angka n dan menambahkan n padanya, setiap kali.

Berikut contoh sederhana untuk memahami hal tersebut dimana ada dua algoritma yang berbeda dalam menemukan kuadrat angka (untuk beberapa waktu, lupakan kuadrat dari angka apa pun n adalah n * n):

```
/*
    Perhitungan untuk akar dari n
*/
for i=1 to n
    do n = n + n
return n
```

Atau, kita cukup menggunakan operator matematika * untuk menemukan kuadrat.

```
/*
     Perhitungan untuk akar dari n
*/
return n*n
```

Berdasarkan dua algoritma sederhana di atas, dapat dilihat bagaimana satu masalah dapat memiliki banyak solusi. Sementara solusi pertama membutuhkan loop yang akan dieksekusi untuk n beberapa kali, solusi kedua menggunakan operator matematika * untuk mengembalikan hasilnya dalam satu baris. Jika diamati maka ada dua pilihan, mana pendekatan yang lebih baik yang akan dipilih, jawabannya adalah yang kedua, emangapa memilih yang kedua karena tidak memakan waktu lama dalam perhitungannya.

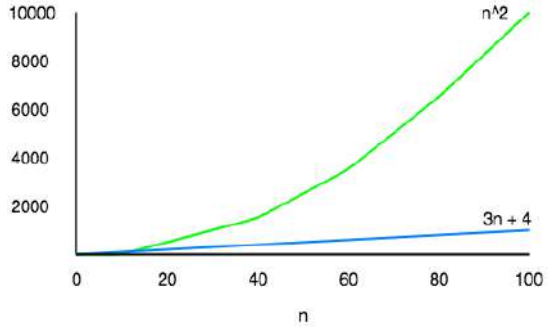
2.2 NOTASI *ASYMPTOTIC* untuk KOMPLEKSITAS WAKTU

Bab 2.1 telah membahas sepintas tentang notasi kompleksitas *asymptotik* yaitu Big-O, Ω dan θ. Notasi-notasi tersebut biasanya digunakan untuk merepresentasikan kompleksitas algoritma. Penggunaan notasi-notasi tersebut adalah bagian dari menganalisis kompleksitas algoritma dalam kaitannya dengan waktu dan ruang, karena kita tidak pernah bisa memberikan angka yang tepat untuk menentukan waktu yang dibutuhkan dan ruang yang dibutuhkan oleh algoritma, sehingga untuk menyatakan tingkatannya maka dapat menggunakan beberapa notasi standa yang dikenal sebagai Notasi *Asymptotic*.

Algoritma apapun yang dianalisa, biasanya akan didapatkan formula yang mewakili jumlah waktu yang diperlukan untuk eksekusi atau waktu yang dibutuhkan oleh komputer dalam menjalankan baris kode algoritmanya, jumlah akses memori, jumlah perbandingan, variabel sementara yang menempati memori spasi, dan lain sebagainya. Formula ini sering berisi detail yang tidak penting dan tidak benar-benar memberi tahu informasi apa pun tentang waktu eksekusi.

Contohnya, jika beberapa algoritma memiliki kompleksitas waktu $T(n)= (n^2+ 3n + 4)$, yang merupakan persamaan kuadratik. Untuk menentukan derajat n yang paling tinggi maka dapat dilihat bahwa 3n + 4 akan menjadi bagian tidak signifikan jika dibandingkan dengan bagian n^2 karena deraja yang dimiliki oleh 3n + 4 adalah 1. Ilustrasi ini dapat dilihat pada Gambar 2.1.

Gambar 2.1 menunjukkan bahwa untuk n = 1000, maka n^2 akan menjadi 10.00000 sedangkan 3n + 4 akan menjadi 3004, terlihat ketimpangan yang jauh sehingga tidak signifikan jika dinbandingkan dengan n^2, dan juga ketika dibandingkan waktu eksekusi dua algoritma, koefisien konstantanya dalam orde tinggi juga diabaikan.



Gambar 2.1 Grafik Perbandingan n^2 dengan 3n + 4

Suatu Algoritma yang membutuhkan waktu $200n^2$ akan lebih cepat daripada algoritma lain yang membutuhkan waktu n^3, untuk nilai n lebih besar dari 200, selain itu berapapun besarnya nilai konstanta dapat diabaikan, karenanya penggunaan perilaku asimptotik dari pertumbuhan fungsi dapat digunakan sebagai salah satu reperesentasi kompleksitas algoritma.

Sebelum mempelajari lebih lanjut tentang tiga notasi asimptotik ini, maka sebaiknya para pembaca perlu mengetahui dan mempelajari tentang Best case, Average case, and Worst case dari suatu algoritma.

Suatu algoritma dapat memiliki waktu berbeda untuk input berbeda. Mungkin diperlukan 1 detik untuk beberapa input dan 10 detik untuk beberapa input lainnya. Sebagai contoh: Jika kita memiliki satu array bernama "arr" dan integer "k", maka kita perlu mencari apakah integer "k" itu ada di array "arr" atau tidak? Jika bilangan bulat ada di sana, maka kembalikan 1 dan 0 untuk kembalian lainnya. Cobalah untuk membuat algoritma untuk pertanyaan ini.

Berdasarkan pertanyaan di atas maka informasi berikut dapat diambil yaitu:

Input: Nilai inputnya adalah array integer dengan ukuran "n" dan ada satu integer "k" yang perlu dicari dalam array itu.

Output: Jika elemen "k" ditemukan dalam array, maka yang harus dikembalikan 1, kalau tidak yang dikembalikan 0.

Selanjutnya, satu solusi yang mungkin untuk masalah di atas dapat berupa pencarian linier, yaitu kita akan melintasi setiap elemen array dan membandingkan elemen tersebut dengan "k". Jika sama dengan "k" maka kembalikan 1, jika tidak, teruslah membandingkan untuk lebih banyak elemen dalam array dan jika Anda mencapai pada akhir array dan Anda tidak menemukan elemen apa pun, maka *return* 0.

```
/*
* @type of arr: integer array
* @type of n: integer (size of integer array)
* @type of k: integer (integer to be searched)
*/
int searchK (int arr [], int n, int k)
{
     // for-loop untuk beralih dengan setiap elemen dalam array
     untuk (int i = 0; i <n; ++ i)
     {
         // periksa apakah elemen ith sama dengan "k" atau tidak
         if (arr [i] == k)
             return 1; // kembalikan 1, jika Anda menemukan "k"
     }
     return 0; // kembalikan 0, jika Anda tidak menemukan "k"
}
/*
* [Penjelasan]
* i = 0 ------------> akan dieksekusi 1 kali
* i < n ------------> akan dieksekusi n+1 kali
* i++ --------------> akan dieksekusi n kali
* if(arr[i] == k) --> akan dieksekusi n kali
* return 1 ---------> akan dieksekusi 1 kali(jikaf "k" ada dalam
array)
* return 0 ---------> will be executed once(jikaf "k" ada dalam
array)
*/
```

Setiap pernyataan dalam kode membutuhkan waktu konstan, katakanlah "C", di mana "C" adalah konstanta. Jadi, setiap kali integer dideklarasikan integer maka diperlukan waktu konstan ketika nilai diubah dari beberapa integer atau variabel lain maka dibutuhkan waktu konstan, ketika dibandingkan dua variabel maka dibutuhkan waktu konstan. Jadi, jika sebuah pernyataan mengambil jumlah waktu "C" dan dieksekusi "N" kali, maka dibutuhkan waktu C * N. Sekarang, pikirkan input berikut untuk algoritma di atas yang baru saja dituliskan:

CATATAN:

Di sini diasumsikan bahwa bahwa setiap pernyataan membutuhkan waktu 1 detik untuk dieksekusi, maka :

1. Jika array input adalah [1, 2, 3, 4, 5] dan Anda ingin mengetahui apakah "1" ada dalam array atau tidak, maka kondisi jika kode akan dieksekusi 1 kali dan ia akan menemukan bahwa elemen 1 ada di dalam array. Jadi, jika-kondisi akan memakan waktu 1 detik di sini.

2. Jika array input adalah [1, 2, 3, 4, 5] dan Anda ingin mengetahui apakah "3" ada di dalam array atau tidak, maka jika kondisi kode akan dieksekusi 3 kali dan ia akan menemukan bahwa elemen 3 ada di dalam array. Jadi, jika-kondisi akan memakan waktu 3 detik di sini.
3. Jika array input adalah [1, 2, 3, 4, 5] dan Anda ingin mengetahui apakah "6" ada di dalam array atau tidak, maka jika kondisi kode akan dieksekusi 5 kali dan ia akan menemukan bahwa elemen 6 tidak ada dalam array dan algoritma akan mengembalikan 0 dalam hal. Jadi, jika-kondisi akan memakan waktu 5 detik di sini.

Seperti yang dilihat bahwa untuk larik input yang sama, maka ada waktu berbeda untuk nilai "k" yang berbeda. Jadi, ini dapat dibagi menjadi tiga kasus:

1. ***Best case***: Ini adalah batas bawah pada waktu berjalan suatu algoritma. Kita harus mengetahui kasus yang menyebabkan jumlah minimum operasi yang harus dijalankan. Dalam contoh di atas, array-nya adalah [1, 2, 3, 4, 5] dan ditemukan apakah "1" ada dalam array atau tidak. Jadi di sini, setelah hanya satu perbandingan, akan didapatkan bahwa elemen tersebut ada dalam array. Jadi, ini adalah kasus terbaik dari algoritma tersebut.
2. ***Average case***: waktu berjalan (*running time*) dihitung untuk semua input yang mungkin, jumlah semua nilai yang dihitung, dan bagi jumlah dengan jumlah total input. Kita harus mengetahui (atau memprediksi) distribusi kasusnya.
3. ***Worst case***: Ini adalah batas atas waktu berjalan suatu algoritma. Kita harus tahu kasus yang menyebabkan jumlah operasi maksimum dieksekusi. Dalam contoh yang diberikan ***Worst case*** dapat jika array yang diberikan adalah [1, 2, 3, 4, 5] dan dicoba untuk menemukan apakah elemen "6" ada dalam array atau tidak. Di sini, jika-kondisi loop akan dieksekusi 5 kali dan kemudian algoritma akan memberikan "0" sebagai output.

Selanjutnya untuk notasi kompleksitas waktu dalam buku ajar ini menggunakan tiga jenis notasi asimtotik untuk mewakili pertumbuhan algoritma sebarang, karena input meningkat, notasi-notasi tersebut adalah :

1. **Notasi Big-O**

   Notasi Big-O mendefinisikan batas atas dari setiap algoritma dimana algoritma yang ada tidak dapat mengambil lebih banyak waktu dari waktu ini, atau dengan

kata lain, dapat dikatakan bahwa notasi Big-O menunjukkan waktu maksimum yang diambil oleh suatu algoritma atau kompleksitas waktu terburuk dari suatu algoritma. Jadi, notasi Big-O merupakan notasi yang paling banyak digunakan untuk kompleksitas waktu suatu algoritma. Jadi, jika diberikan suatu fungsi g(n), maka representasi Big-O dari g (n) dapat dituliskan sebagai O(g(n)) atau dibaca "big-oh dari fungsi g(n)" dan hubungannya dapat dituliskan sebagai berikut :

$O(g(n)) = \{f(n)$: ada konstanta positif c dan n_0 sedemikian sehingga

$0 \leq f(n) \leq c*g(n)$ untuk semua $n \geq n_0\}$

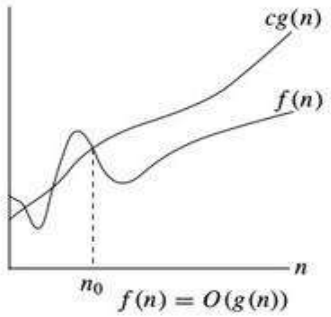
Ekspresi di atas dapat dibaca sebagai Big-O dari g(n) didefinisikan sebagai himpunan fungsi *f(n)* yang ada beberapa konstanta c dan n_0 sehingga *f(n)* lebih besar dari atau sama dengan 0 dan *f(n*) lebih kecil dari atau sama dengan *c*g (n)* untuk semua n lebih besar dari atau sama dengan n_0.

Contoh :

jika $f(n) = 2n^2 + 3n + 1$ dan $g(n) = n^2$

maka untuk c = 6 dan $n_0 = 1$, maka dapat dikatakan bahwa $f(n) = O(n^2)$

Big-O dapat digambarkan dalam bentuk grafik seperti gambar 2.2.



Gambar 2.2 Grafik Notasi Big-O

2. Notasi Ω

Notasi den menunjukkan batas bawah suatu algoritma yaitu waktu yang diambil oleh suatu algoritma tidak dapat tumbuh lebih cepat dari waktu ini. Dengan kata lain, Anda dapat mengatakan bahwa notasi den menunjukkan waktu minimum yang diambil oleh algoritma untuk berbagai input atau kompleksitas waktu kasus terbaik dari suatu algoritma. Jadi, jika suatu fungsi adalah g (n), maka representasi omega ditampilkan sebagai Ω (g (n)) dan relasinya ditunjukkan sebagai:

Ω(g(n))={f (n): ada konstanta positif c dan n_0 sedemikian sehingga

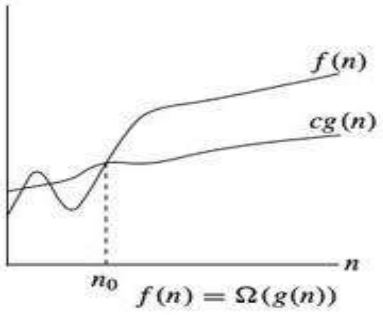
*0 ≤ c*g (n) ≤ f (n)* untuk semua $n \geq n_0$}

Ekspresi di atas dapat dibaca sebagai omega dari g(n) didefinisikan sebagai himpunan semua fungsi f(n) yang ada beberapa konstanta c dan n0 sehingga c*g(n) kurang dari atau sama dengan f(n), untuk semua n lebih besar dari atau sama dengan n0.

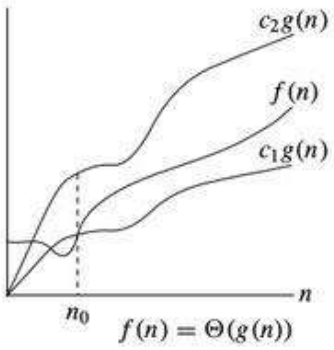
Jika $f(n) = 2n^2 + 3n + 1$ dan $g(n) = n^2$

maka untuk c = 2 dan $n_0 = 1$, dapat dikatakan bahwa $f(n) = \Omega(n^2)$

Big- Ω dapat digambarkan dalam bentuk grafik seperti gambar 2.3.



Gambar 2.3 Grafik Notasi Big- Ω



Gambar 2.4 Grafik Notasi Big- **θ**

3. Notasi θ

Notasi Θ digunakan untuk menemukan batas rata-rata suatu algoritma, yaitu, menentukan batas atas dan batas bawah, dan algoritma Anda akan terletak di antara level-level ini. Jadi, jika suatu fungsi adalah g (n), maka representasi theta ditampilkan sebagai Θ (g (n)) dan relasinya ditunjukkan sebagai:

Θ (g (n)) = {f (n): ada konstanta positif c_1, c_2 dan n_0 sedemikian sehingga

$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ untuk semua $n \geq n_0$}

Ekspresi di atas dapat dibaca sebagai theta dari g (n) didefinisikan sebagai himpunan semua fungsi f(n) yang ada beberapa konstanta positif c_1, c_2, dan n_0 sehingga c_1*g(n) kurang dari atau sama dengan f (n) dan f (n) kurang dari atau sama dengan c_2*g(n) untuk semua n yang lebih besar dari atau sama dengan n_0.

Sebagai contoh:

jika $f(n) = 2n^2 + 3n + 1$ dan $g(n) = n^2$

maka untuk $c_1 = 2$, $c_2 = 6$, dan $n_0 = 1$, dapat dikatakan bahwa $f(n) = \Theta(n^2)$

Big- **θ** dapat digambarkan dalam bentuk grafik seperti gambar 2.4.

2.3 MENGHITUNG KOMPLEKSITAS WAKTU

Kadangkala kita menemukan beberapa algoritma untuk masalah tertentu yang memerlukan pertimbangan untuk memilih algoritma terbaik yang sesuai dengan keinginan dan harapan. Oleh karenanya kita perlu membandingkan beberapa algoritma dan memilih algoritma terbaik. Pemilihan algoritma terbaik biasanya dilakukan melalui tahapan analisis algoritmanya.

Tujuan analisis algoritma adalah untuk membandingkan algoritma dengan beberapa faktor seperti *running time*, memori, upaya pengembangan, dll. Faktor ideal yang akan dipilih untuk tujuan perbandingan adalah *running time* dari algoritma yang merupakan fungsi dari ukuran input, n. Karena *running time* merupakan fungsi dari ukuran input, dan tidak tergantung pada waktu eksekusi mesin, gaya pemrograman dll.

Kompleksitas waktu (atau *running time*) merupakan perkiraan waktu yang dibutuhkan suatu algoritma untuk berjalan. Namun, kita tidak mengukur kompleksitas waktu dalam detik, tetapi sebagai fungsi input. Jika dipikirkan mengapa tidak menggunakan waktu dalam detik ? hal ini agak terasa aneh dan membingungkan, namun demikian penjelasana berikut ini akan menuntun anda untuk memahami perhitungan kompleksitas waktu melalui .penjelaskannya dengan cara yang paling sederhana.

Mengapa kompleksitas waktu dinyatakan sebagai fungsi input? Baiklah, katakanlah Anda ingin mengurutkan susunan angka. Jika elemen sudah diurutkan, program akan melakukan operasi lebih sedikit. Sebaliknya, jika item dalam urutan terbalik, itu akan membutuhkan lebih banyak waktu untuk menyelesaikannya. Jadi,

waktu yang diperlukan suatu program untuk mengeksekusi secara langsung berkaitan dengan ukuran input dan bagaimana elemen-elemen tersebut diatur.

> Kompleksitas waktu bukan tentang menentukan waktu berapa lama algoritma tersebut membutuhkan. Sebaliknya, berapa banyak operasi yang dijalankan. Jumlah instruksi yang dijalankan oleh suatu program dipengaruhi oleh ukuran input dan bagaimana elemen-elemennya diatur (Adrian Meija)

Sekarang metrik yang paling umum untuk menghitung kompleksitas waktu adalah notasi O Besar. Ini menghilangkan semua faktor konstan sehingga waktu berjalan dapat diperkirakan dalam kaitannya dengan N, ketika N mendekati tak terhingga. Secara umum Anda bisa memikirkannya seperti ini:

```
statement;
```

Di atas kami memiliki satu pernyataan. Kompleksitas Waktunya akan Konstan. Waktu berjalan pernyataan tidak akan berubah sehubungan dengan N.

```
for(i=0; i < N; i++)
{
    statement;
}
```

Kompleksitas waktu untuk algoritma di atas adalah Linear. Waktu berjalan dari loop berbanding lurus dengan N. Saat N berlipat ganda, demikian juga waktu berjalan.

```
for(i=0; i < N; i++)
{
    for(j=0; j < N;j++)
    {
    statement;
    }
}
```

Kali ini, kompleksitas waktu untuk kode di atas adalah kuadratik. Waktu berjalan dari dua loop sebanding dengan kuadrat N. Ketika N berlipat ganda, waktu berjalan meningkat sebesar N * N.

```
while(low <= high)
{
    mid = (low + high) / 2;
    if (target < list[mid])
        high = mid - 1;
    else if (target > list[mid])
        low = mid + 1;
```

```
    else break;
}
```

ini adalah algoritma untuk memecah satu set angka menjadi dua bagian, untuk mencari bidang tertentu (kita akan mempelajari ini secara rinci nanti). Sekarang, algoritma ini akan memiliki Kompleksitas Waktu Logaritmik. Waktu berjalan dari algoritma sebanding dengan berapa kali N dapat dibagi dengan 2 (N adalah tinggi-rendah di sini). Ini karena algoritma membagi area kerja menjadi dua dengan setiap iterasi.

```
void quicksort(int list[], int left, int right)
{
    int pivot = partition(list, left, right);
    quicksort(list, left, pivot - 1);
    quicksort(list, pivot + 1, right);
}
```

Dengan memajukan algoritma sebelumnya, di atas kami memiliki sedikit logika Quick Sort (kami akan mempelajari ini secara terperinci nanti). Sekarang di Quick Sort, kami membagi daftar menjadi dua bagian setiap kali, tetapi kami mengulangi iterasi N kali (di mana N adalah ukuran daftar). Karenanya kompleksitas waktu adalah N * log (N). Waktu berjalan terdiri dari N loop (iteratif atau rekursif) yang bersifat logaritmik, sehingga algoritma ini merupakan kombinasi linear dan logaritmik.

Catatan:

Secara umum, melakukan sesuatu dengan setiap item dalam satu dimensi adalah linier, melakukan sesuatu dengan setiap item dalam dua dimensi adalah kuadratik, dan membagi area kerja menjadi setengahnya adalah logaritmik.

Jenis Notasi untuk Kompleksitas Waktu

Sekarang kita akan membahas dan memahami berbagai notasi yang digunakan untuk Kompleksitas Waktu.

a. Big Oh menunjukkan "lebih sedikit atau sama dengan" iterasi <expression>.
b. Big Omega menunjukkan "lebih dari atau sama dengan" iterasi <expression>.
c. Big Theta menunjukkan "sama dengan" iterasi <expression>.
d. Little Oh menunjukkan "lebih sedikit dari" iterasi <expression>.
e. Little Omega menunjukkan iterasi "lebih dari" <ekspresi>.

O (ekspresi) adalah himpunan fungsi yang tumbuh lebih lambat dari atau pada tingkat yang sama dengan ekspresi. Ini menunjukkan maksimum yang dibutuhkan oleh

suatu algoritma untuk semua nilai input. Ini mewakili kasus terburuk dari kompleksitas waktu suatu algoritma.

Omega (ekspresi) adalah serangkaian fungsi yang tumbuh lebih cepat dari atau pada tingkat yang sama dengan ekspresi. Ini menunjukkan waktu minimum yang diperlukan oleh suatu algoritma untuk semua nilai input. Ini mewakili kasus terbaik dari kompleksitas waktu suatu algoritma.

Teta (ekspresi) terdiri dari semua fungsi yang terletak pada O (ekspresi) dan Omega (ekspresi). Ini menunjukkan batas rata-rata suatu algoritma. Ini mewakili kasus rata-rata kompleksitas waktu suatu algoritma.

Misalkan kita telah menghitung bahwa suatu algoritma mengambil operasi f(n), di mana,

```
f. f(n) = 3*n^2 + 2*n + 4.   // n^2 means square of n
```

Karena polinomial ini tumbuh pada kecepatan yang sama dengan n^2, maka Anda dapat mengatakan bahwa fungsi f terletak pada himpunan Theta (n^2). (Itu juga terletak pada himpunan O (n^2) dan Omega (n^2) untuk alasan yang sama.) Penjelasan paling sederhana adalah, karena Theta menunjukkan sama dengan ungkapan. Oleh karena itu, ketika f(n) tumbuh oleh faktor n^2, kompleksitas waktu dapat digambarkan sebagai Theta (n^2).

Berikut ini adalah beberapa contoh menentukan kompleksitas waktu suatu program (atau algoritma) tertentu :

Contoh 1.

```
Compute the maximum element in the array a.
Algorithm max(a):
    max ← a[0]
    for i = 1 to len(a)-1
        if a[i] > max
            max ← a[i]
    return max
```

Untuk menghitung kompleksitas dari algoritma diatas jawabannya tergantung pada faktor-faktor seperti input, bahasa pemrograman dan runtime, keterampilan pengkodean, kompiler, sistem operasi, dan perangkat keras. Namun demikian biasanya diasumsikan bahwa kompleksitas suatu algoritma tergantung pada algoritma dan

inputnya. Hal ini dapat dicapai dengan memilih operasi elementer, yang dilakukan algoritma secara berulang kali, dan menentukan kompleksitas waktu T(n) sebagai jumlah operasi yang dilakukan algoritma dengan panjang array n.

Untuk algoritma di atas kita dapat memilih perbandingan a [i]> maks sebagai operasi dasar.

a. Bisa dilihat bahwa running time dari algoritma adalah baik, karena perbandingan mendominasi semua operasi lain dalam algoritma khusus ini.
b. Selain itu, waktu untuk melakukan perbandingan adalah konstan: tidak tergantung pada ukuran a.

Kompleksitas waktu, diukur dalam jumlah perbandingan, kemudian menjadi T(n) = n-1. Secara umum, operasi dasar harus memiliki dua sifat yaitu:

c. Tidak akan ada operasi lain yang dilakukan lebih sering seiring ukuran input bertambah.
d. Waktu untuk mengeksekusi operasi elementer harus konstan: itu tidak boleh bertambah ketika ukuran input bertambah. Ini dikenal sebagai biaya satuan.

Contoh 2

```
main(){
 int a=10,b=20,sum; //konstanta waktu c1
 sum = a + b; // konstanta waktu c2
}
```

Kompleksitas waktu dari program di atas = O (1)

Bagaimana mendapatkan O (1) ?. Pertama, hitung total waktu setiap pernyataan dalam program (atau algoritma). Waktu yang diambil untuk setiap baris. Mengapa waktu konstan? Karena pernyataan yang akan dieksekusi tidak tergantung pada ukuran input. Jadi dibutuhkan waktu eksekusi konstan yang tergantung pada mesin. Karena tidak tergantung pada ukuran input, maka dapat dianggap sebagai waktu konstan yang dinyatakan dengan c. Jadi total waktu yang diambil oleh program ini adalah Total Waktu = c1 + c2, karenanya Kompleksitas Waktu = O(1), Apa yang dimaksud dengan Notasi Big-O? Notasi ini memberikan batas atas dari fungsi yang diberikan. misalnya, jika $f(n) = n^3 + 25n^2 + n + 10$, batas atas f (n) adalah n^3. Itu berarti n^3 memberikan

tingkat pertumbuhan maksimum untuk f (n) pada nilai n yang lebih besar. Oleh karena itu f (n) = O (n^3). Contoh lainnya adalah :

Contoh 3

```
for(i=1; i<=n; i++)
       sum = sum + 2; // konstanta waktu c
```

Kode di atas adalah untuk loop. Kita tahu bahwa body for for akan dieksekusi n kali. Karena tubuh loop hanya terdiri dari satu pernyataan dan karena hanya membutuhkan waktu konstan c untuk mengeksekusi total waktu yang diambil oleh for loop, Total waktu = nc, di sini f (n) = nc, oleh karena itu Kompleksitas Waktu = O (n)

Contoh 4

```
for(i=1; i<=n; i++)
       i = i * 2; // konstanta waktu c
```

Di sini kita dapat melihat bahwa nilai i adalah dua kali lipat pada setiap iterasi. i.e Pada iterasi pertama i = 2, iterasi kedua i = 4, iterasi ketiga i = 8. Asumsikan bahwa pada iterasi ke-k, nilai i mencapai n. Yaitu, 2^k = n, dengan mengalikan log di kedua sisi didapatkan:

log (2^k) = log n

k log 2 = log n

k = log n

Oleh karena itu kompleksitas waktu = O (log n)

Contoh 5:

```
for(i=1; i<=n; i++){
   for(j=1; i<=n; i++)
      sum = sum + 2; // konstanta waktu c
}
```

Pada contoh 4 bisa dilihat loop bersarang. Loop dalam berulang n kali dan loop luar berulang kali n. Oleh karena itu, total waktu = c*n*n = c*n^2 = O (n^2)

Contoh 6:

```
sum = a + b; // konstanta waktu c
for(i=1; i<=n; i++)
       i = i * 2; // konstanta waktu d
for(i=1; i<=n; i++){
```

```
   for(j=1; i<=n; i++)
       sum = sum + 2; // konstanta waktu e
}
```
Total waktu = c + dn + en^2 = $O(n^2)$

Contoh 7

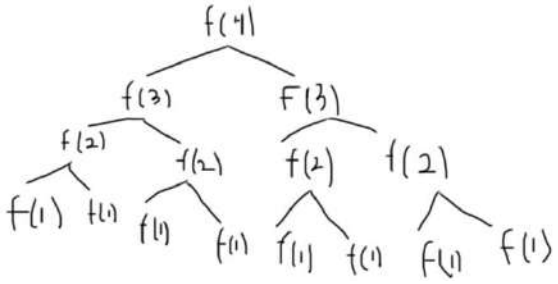
```
if(i==1) // konstanta waktu untuk kondisi eksekusi c
  return false; // konstanta waktu d
else{
  for(j=1; i<=n; i++)
       sum = sum + 2; // constant time, e
}
```
Total waktu = c + d + (f + e) n = O(n)

Contoh 8.

```
if(i==1){ //constant time for executing condition,c
   for(i=1; i<=n; i++)
     for(j=1; i<=n; i++)
       sum = sum + 2; // constant time, d
}else{
   for(j=1; i<=n; i++)
       sum = sum + 2; // constant time, e
}
```
Total waktu = c + dn^2 + e n = $O(n^2)$

Contoh 9

```
int func(int n) {
    if (n <= 1) {
        return 1;
    }    return func(n-1) + func(n-1);
}
```
Misalkan dipanggil dengan func(4) , maka kita bisa gambarkan pohon rekursifnya seperti ini :



Gambar 2.5 Pohon Rekursif untuk nilai Func (4)

Lalu polanya dicari seperti pada table 2.1.

Tabel 2.1 Hasil Pola Perhitungan Contoh 8.

Level	Node Count	Or
0	1	2^0
1	2	2^0
2	4	2^0
3	8	2^0

Jadi, fungsi diatas memiliki kompleksitas waktu sebesar **O(2^n).**

2.4 KOMPLEKSITAS RUANG SUATU ALGORITMA

Kompleksitas Ruang dari suatu algoritma menunjukkan total ruang yang digunakan atau dibutuhkan oleh algoritma untuk kerjanya, untuk berbagai ukuran input. Sebagai contoh:

Contoh 10.

```
vector<int> myVec(n);
for(int i = 0; i < n; i++)
      cin >> myVec[i];
```

Pada contoh 8 tersebut, diketahui vektor ukuran n, sehingga kompleksitas ruang dari kode di atas adalah dalam urutan "n" yaitu jika n akan meningkat, kebutuhan ruang juga akan meningkat. Selain itu ketika variable dibuat, maka diperlukan ruang untuk menjalankan algoritma tersebut. Semua ruang yang dibutuhkan untuk algoritma secara kolektif disebut Kompleksitas Ruang dari algoritma atau dengan kata lain ada keperluan memori untuk menyelesaikannya.

Untuk setiap algoritma memori dapat digunakan untuk:

a. Variabel (Ini termasuk nilai konstan, nilai sementara)
b. Instruksi Program
c. Eksekusi

Kompleksitas ruang juga berkaitan dengan jumlah memori yang digunakan oleh algoritma (termasuk nilai input untuk algoritma) untuk mengeksekusi dan menghasilkan hasilnya. Kompleksitas ruang adalah jumlah dari *Auxiliary Space* **(**ruang bantu**)** dan ruang input.

Kompleksitas Ruang = Ruang Bantu + Ruang input

Auxiliary Space adalah ruang ekstra atau ruang sementara yang digunakan oleh algoritma selama eksekusi itu.

Saat mengeksekusi, algoritma menggunakan ruang memori karena tiga alasan:

a. Ruang instruksi

   Ini adalah jumlah memori yang digunakan untuk menyimpan versi instruksi yang dikompilasi.

b. *Enviroment Stack* (Tumpukan Lingkungan)

   Kadang-kadang suatu algoritma (fungsi) dapat disebut di dalam algoritma lain (fungsi). Dalam situasi seperti itu, variabel saat ini didorong ke tumpukan sistem, di mana mereka menunggu eksekusi lebih lanjut dan kemudian panggilan ke algoritma dalam (fungsi) dibuat.

   Sebagai contoh, Jika fungsi A () memanggil fungsi B () di dalamnya, maka semua variabel ke fungsi A () akan disimpan pada sistem stack sementara, sementara fungsi B () dipanggil dan dieksekusi di dalam fungsi A ().

c. Ruang Data

   Jumlah ruang yang digunakan oleh variabel dan konstanta.

   Tetapi kadangkala ketika menghitung Kompleksitas Ruang dari algoritma apa pun, biasanya hanya dipertimbangkan Ruang Data sedangkan Ruang Instruksi dan Stack Lingkungan biasanya diabaikan.

Selanjutnya untuk menghitung kompleksitas ruang, perlu diketahui bahwa nilai memori yang digunakan oleh berbagai jenis tipe data dari variabel, yang umumnya bervariasi untuk sistem operasi yang berbeda, tetapi metode untuk menghitung kompleksitas ruang tetap sama.

Tabel 2.2 Besar ukuran untuk type data

Type	Size
bool, char, unsigned char, signed char, __int8	1 byte
int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

Berikut ini beberapa contoh bagaimana menghitung kompleksitas ruang :

Contoh 11.

```
{
    int z = a + b + c;
    return(z);
}
```

Ekspresi pada contoh 8 dapat dilihat bahwa variabel a, b, c dan z semuanya memiliki tipe integer, sehingga variable-variabel tersebut akan mengambil masing-masing 4 byte, sehingga total kebutuhan memori adalah (4 (4) + 4) = 20 byte, tambahan 4 byte ini adalah untuk nilai pengembalian. Dan karena persyaratan ruang ini ditetapkan untuk contoh di atas, maka disebut Kompleksitas Ruang Konstan.

Contoh 12.

```
// n adalah panjang array a[]
int sum(int a[], int n)
{
        int x = 0;              // 4 bytes untuk x
        for(int i = 0; i < n; i++)    // 4 bytes untuk i
        {
            x  = x + a[i];
        }
        return(x);
}
```

Pada contoh 10, dari kodenya ada 4*n byte ruang diperlukan untuk elemen array []. 4 byte masing-masing untuk x, n, i dan nilai kembali. Oleh karena itu total kebutuhan memori adalah (4n + 12), yang meningkat secara linier dengan peningkatan nilai input n, maka itu disebut sebagai *Linear Space Complexity*.

Demikian juga dalam kompleksitas ruang juga ada kompleksitas kuadratik dan kompleks lainnya, seiring meningkatnya kompleksitas suatu algoritma, namun harus selalu fokus pada penulisan kode algoritma sedemikian rupa sehingga dapat dijaga kompleksitas ruang minimum.

CATATAN:

Pada pemrograman normal, penggunaan ruang yang diizinkan adalah 256MB untuk masalah tertentu. Jadi, jika ada array maka tidak dapat dibuat dengan ukuran lebih dari 10 ^ 8 karena hanya akan diizinkan menggunakan 256MB. Selain itu, array juga tidak

dapat dibuat dalam ukuran lebih dari 10 ^ 6 dalam suatu fungsi karena ruang maksimum yang dialokasikan untuk fungsi adalah 4 MB. Jadi, untuk menggunakan array berukuran lebih besar, maka dapat dilakukan dengan membuat array global.

Berikut ini adalah beberapa contoh lainnya untuk perhitungan kompleksitas suatu algoritma dimana *running time* tergantung kepada beberapa faktor akan tetapi perhitungannya didasarkan pada ukuran inputannya, sehingga diperlukan untuk melakukan perhitungan ***rate of growth of time***.

Asumsi yang diambil adalah :

a. Semua operasi secara aritmatika atau logika dihitung 1 unit waktu
b. Semua statement "return" juga dihitung 1 unit waktu

Contoh 13 : ***Konstanta Waktu***

```
int sumOfNumber(a,b)
{
        int c = a+b;        → 1 + 1 = 2 unit waktu
        return c;           →       1 unit waktu
}
```

Total biaya = 2 + 1
= 3 unit waktu

Contoh 14 : ***Waktu Linear***

Diketahui Algoritma

```
int sumOfNumber(a[ ], n)                    Cost        JumlahWaktu
{
        int sum = 0;                  →      1              1
        for (int i = 0; i < n; i++)   →      2            n + 1
           {
                Sum =  sum +a[i];     →      2              n
           }
        return sum;                   →      1              1
}
```

Total biaya = 1 + 2 (n+1) + 2 (n) + 1
= 1 + 2n + 2 + 2n + 1
= 4n + 4

Penjelasan :

1) T(n) dari sum = 3 unit = konstanta = $O(1)$ → θ
2) T(n) dari sum of Array = $4n + n$= linear = $O(n)$
3) T(n) dari perkalian Matriks = $an^2 + bn + c$ → quadratik
   = $O(n^2)$

- Notasi untuk Asimtotik dapat dituliskan sebagai O, Ω, dan θ

Contoh 15 :

Diketahui Algoritma:

```
for (i = 0; i < n; i++)
{
    for (j = 0; j < i; j++)
    {
        Statement;
    }
}
```

Setelah ditracer diperoleh :

i	j	n of time
0	0	0
1	0	1
	1	
2	0	2
	1	
	2	
3	0	3
	1	
	2	
⋮	3	⋮
⋮		⋮
N		n

berdasarkan hasil tracer diperoleh :

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} \qquad \rightarrow 1 + 2 + 3 + \cdots + n = f(n)$$

$$f(n) = \frac{n^2+n}{2}$$

Sehingga kompleksitasnya adalah $O(n^2)$.

Contoh 16 :

Diketahui algoritma :

```
P=0
for (i = 1; p <= n; i++)
{
    p = p + i;
}
```

i	p
1	0 + 1 = 1
2	1 + 2 = 3
3	1 + 2 + 3 = 6
4	1 + 2 + 3 + 4 = 10
⋮	⋮
K	1 + 2 + 3 + 4 + ⋯ + k

Asumsi : $p > n$

$$p = \frac{k\,(k+1)}{2}$$

Sehingga

$$\frac{k\,(k+1)}{2} > n$$

$$k^2 > n$$

$$k > \sqrt{n}$$

$$\therefore O(\sqrt{n})$$

Contoh 17:

Diketahui Algoritma

```
for (i = 1; p <= n; i++)
{
    Statement;
}
```

Asumsi : $i \geq n$

$$i = 2^k$$

Sehingga

$$2^k >= n$$

Tracer :

i
1
1 x 2 = 2
2 x 2 = 2^2
2^2 x 2 = 2^3
⋮
2^k

$2^k = n$

$k =^2 \log n$

$\therefore O\ (^2\log\ n)$

Dapat ditulis juga :

$i = 1 \times 2 \times 2 \times 2 \times \cdots = n$

$2^k = n$

Jika dilihat perbedaannya pada perulangan :

```
for (i = 1; i <= n; i++)
{
        Statement;
}
```

untuk perulangan disamping dapat ditulis $i = 1 + 1 + 1 + 1 + 1 + \cdots = n$ sehingga k = n

$\therefore O(n)$

Contoh 18 :

Diketahui Algoritma

```
for (i = n; i >= 1; i=i/2)
{
        Statement;
}
```

Asumsi bahwa i < 1

$\frac{n}{2^k} < 1$

$\frac{n}{2^k} = 1$

$n = 2^k \quad \rightarrow k =^2 \log n$

$\therefore O\ (^2\log\ n)$

Tracer :

$\frac{i}{n}$
$\frac{n}{2}$
$\frac{n}{2^2}$
$\frac{n}{2^3}$
⋮
$\frac{n}{2^k}$

Contoh 19 :

Diketahui Algoritma

```
for (i = 0; i * i < n; i++)
{
        Statement;
}
```

$i \times i < n$

Asumsi : $i \times i >= n$

$i^2 = n$

maka $i = \sqrt{n}$

Sehingga kompleksitasnya adalah $\therefore O(\sqrt{n})$

Contoh 20:

Diketahui Algoritma

```
p = 0;
for (i = 1; i < n; i = i * 2)
{
    p++;                          → log n = p
    for (j = 1; j < p ; j = j * 2)
    {
        Statement;                → log p
    }
}
```

$\therefore O(\log \log n)$

Contoh 21 :

Diketahui Algoritma

```
for (i = 0; i < n; i++)           → n
{
    for (j = 1; j < n ; j = j * 2) → n * log n
    {
        Statement;                → n * log n
    }
}
```

2n log n + n

Jadi kompleksitasnya adalah : $\therefore O(n \log n)$

Kesimpulan :

for (i = 0; i < n; i++)	→	$O(n)$
for (i = 0; i < n; i+2)	→	$\frac{n}{2} O(n)$
for (i = n; i < n; i– –)	→	$O(n)$
for (i = 1; i < n; i= i * 2)	→	$O\ (^2\log\ n)$
for (i = 1; i < n; i= i * 3)	→	$O\ (^2\log\ n)$
for (i = n; i < n; i= $i/_2$)	→	$O\ (^2\log\ n)$

Kaidah umum menentukan running time dari ***suatu algoritma***.

a) Loop

Contoh 21.

Diketahui Algoritma

```
for (i = 1; i <= n; i++)
{
    a = a + b;  //konstanta
}
```

Jadi, running time = $c \times n = c.n = O(n)$.

b) Nested Loop

Contoh.

Diketahui Algoritma

```
for (i = 1; i <= n; i++)                    → n
{                                                  n × n = n²
    for (j = 1; j <= n; j++)                → n
    {
        a = a + b;      //constan time → c      → c.n²
    }
}
```

Sehingga, running timenya = $c.n^2 = O(n^2)$.

c) IF – Else Statement

```
IF(conf)
{
        ----------
        ----------   } → O(n)
        ----------
}
Else                                        O (n)+O(n²)
{                                           Worst Case
        ----------
        ----------   } → O(n²)              ∴ O(n²)
        ----------
}
```

Sehingga, *running timenya* $= O(n^2)$.

d) Logaritmic Complexity

```
for (i = 1; i < n; i++)
{
        ----------
        ----------        → log n = O(log n)
        ----------
        i = i * 2;
}
```

Dapat diartikan $i = i \times 2$ adalah $1, 2, 4, 8, 16, \cdots$

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$$

Sehingga kompleksitasnya adalah $O(n^2)$

e) Consecutive Statement

Diketahui Algoritma

```
a = a + b;                  → C1 →        C1
for (i = 1; i <= n; i++)    → n
{                                         C2.n
      x = x + y;            → C2
}
for (j = 1; j <= n; j++)    → n
{                                         C3.n
        c = c + d;          → C3
}
```

Sehingga running timenya adalah :

$$C_1 + C_2.n + C_3.n = C_1 + (C_2 + C_3)n = C_1 + C'_2.n$$

Sehingga running timenya adalah $O(n)$

2.5 ANALISA ALGORITMA REKURSIF

Algoritma rekursif adalah algoritma yang menyebut dirinya dengan nilai input "lebih kecil (atau lebih sederhana)", dan yang memperoleh hasil untuk input saat ini dengan menerapkan operasi sederhana ke nilai yang dikembalikan untuk input yang lebih kecil (atau lebih sederhana).

Secara umum, program komputer rekursif membutuhkan lebih banyak memori dan komputasi dibandingkan dengan algoritma iteratif, tetapi mereka lebih sederhana dan untuk banyak kasus menjadi cara berpikir alami tentang penyelesaian masalah tersebut.

Rencana Umum untuk Analisis algoritma Rekursif

1. Tentukan parameter n yang menunjukkan ukuran input
2. Identifikasi operasi dasar algoritma
3. Menentukan kasus terburuk, rata-rata, dan terbaik untuk input ukuran n
4. Atur relasi perulangan, dengan inisial kondisi, untuk berapa kali dasar operasi dijalankan
5. Selesaikan perulangannya, atau setidaknya pastikanurutan pertumbuhan solusi (lihat Lampiran B buku Levitin : 2002).

2.6 RANGKUMAN

1. Kompleksitas waktu adalah jumlah operasi yang dilakukan algoritma untuk menyelesaikan tugasnya sehubungan dengan ukuran input (mengingat bahwa setiap operasi membutuhkan jumlah waktu yang sama). Algoritma yang melakukan tugas dalam jumlah operasi terkecil dianggap yang paling efisien.
2. Kompleksitas waktu menyatakan berapa lama suatu algoritma berjalan ketika runtime berdasarkan input yang diberikan, sedangkan kompleksitas ruang menyatakan berapa banyak ruang dalam memori yang dibutuhkan suatu algoritma ketika beroperasi.
3. Ukuran Input: Ukuran input didefinisikan sebagai jumlah total elemen yang ada dalam input. Untuk masalah yang diberikan, kami mengkarakterisasi ukuran input dan tepat. Contoh t:
    a. Masalah penyortiran: Jumlah total item yang akan disortir
    b. Masalah Grafik: Jumlah total simpul dan tepi
    c. Masalah Numerik: Jumlah total bit yang diperlukan untuk merepresentasikan angka
4. Waktu yang digunakan oleh suatu algoritma juga tergantung pada kecepatan komputasi sistem yang digunakan, tetapi faktor-faktor eksternal diabaikan dan hanya diperhatikan berapa kali pernyataan tertentu dieksekusi sehubungan

dengan ukuran inputnya. Contoh, untuk mengeksekusi satu pernyataan, waktu yang diambil adalah 1 detik, lalu berapa waktu yang diperlukan untuk mengeksekusi n pernyataan, Ini akan memakan waktu n detik.

5. Ada tiga notasi asimptotik yang digunakan untuk merepresentasikan kompleksitas waktu suatu algoritma yaitu:
   a. Notasi Θ (theta)
   b. Notasi Big-O dan
   c. Notasi Ω

2.7 TEST FORMATIF

1. Jelaskan apa yang dimaksud dengan kompleksitas waktu dan ruang
2. Sebutkan dan jelaskan tentang notasi asymptotic
3. Dapatkan kompleksitas waktu dari loop berikut ini dimana n merupakan inputannya

```
i <-- n;
while(i > 1) {
    j = i; //%% Tidak dimulai dari 0
    while (j < n) {
        k <-- 0;
        while (k < n) {
           k = k + 2;
        }
        j <-- j * 2;
    }
    i <-- i / 2;
}
```

4. Buktikan bahwa Big O *transitif relation* yaitu: jika *f(n) ≤ O (g(n)) dan g(n) ≤ O (h (n))*, maka*, f (n) ≤ O (h (n)).*
5. Tentukan kompleksitas waktu dari :

```
a. int count = 0;
   for (int i = 0; i < N; i++)
      for (int j = 0; j < i; j++)
            count++;
b. int count = 0;
   for (int i = N; i > 0; i /= 2)
      for (int j = 0; j < i; j++)
            count++;
```

6. Tentukan Kompleksitas untuk pseudo code berikut :

```
a.  x <- 0
       for x <- 0 to n:
```

```
          for y <- 0 to n:
             a=c+d
```

b.
```
  x <- 0
        for x <- 0 to n:
          for y <- x to n-3:
             a=c+d
```

c.
```
  x <- 0
        for x <- 2 to n+2:
          for y <- x-1 to n:
            a=c+d
```

BAB III
ALGORITMA BRUTE FORCE

a) Deskripsi Singkat

Algoritma brute force sendiri memiliki cara berpikir yang sederhana dalam memecahkan masalah. Inti dari pemecahan masalah adalah mencoba seluruh kemungkinan solusi yang ada. Semakin besar skala permasalahan, maka jumlah kemungkin solusi juga akan bertambah. Pertambahan ini bergantung pada jenis permasalah, ada yang mengalamai peningkatan secara linear, logaritma, hingga eksponensial. Banyak algoritma-algoritma rumit yang sebenarnya merupakan modifikasi dan pengembangan dari algoritma brute force. Algoritma brute force sendiri seringkali digunakan sebagai pembanding saat dilakukan uji coba kualitas suatu algoritma. Terdapat juga masalah yang hanya dapat diselekasikan menggunakan algoritma brute force seperti permasalahan mencarai elemen maksimum di dalam senarai.

b) Relevansi

Proses pembelajaran dilakukan dengan metode ceramah, tanya jawab, duskusi dan praktikum. Berkaitan dengan materi pada Bab 3 ini, kepada mahasiswa akan dijelaskan dengan detail materinya untuk kemudian algoritmanya diimplementasikan ke program melalui praktikum menggunakan bahasa pemrograman C, C++, Java atau python atau bahasa pemrograman yang dikuasi mahasiswa. Kesempatan bertanya juga diberikan dan tugas mandiri sebagai pengayaan materi-materi pada bab 3 ini.

c) Capaian Pembelajaran

1. Mahasiswa mampu menyelesaikan menyelesaikan masalah dengan strategi brute force
2. Mahasiswa mampu melakukan proses tracer terhadap program yang telah dibuat

3.1 PENDAHULUAN

Algoritma Brute Force mengacu pada gaya pemrograman yang tidak termasuk cara pintas untuk meningkatkan kinerja, tetapi sebaliknya mengandalkan kekuatan

komputasi belaka untuk mencoba semua kemungkinan sampai solusi untuk suatu masalah ditemukan.

Levitin dalam bukunya menggunakan istilah brute force sebagai cara menjelaskan desain algoritma apa pun di mana daya komputasi digunakan sebagai pengganti kepintaran programmer. Brute Force biasanya meliputi pembongkaran spesifikasi ke dalam suatu algoritma, baik dengan mengimplementasikan spesifikasi secara langsung (seperti dalam masalah perkalian matriks), atau dengan pencarian semua ruang kemungkinan keluaran atau solusi untuk menemukan satu yang memenuhi spesifikasi. Tujuan dari desain algoritma brute force tidak selalu untuk mendapatkan algoritma terbaik, tetapi juga untuk mendapatkan beberapa algoritma untuk menyelesaikan masalah yang dapat diimplementasikan secara cepat. Banyak algoritma seperti itu bekerja dengan *exhaustive search* (pencarian lengkap) atau dikenal juga sebagai *generate and test*. Idenya adalah bahwa jika kita dapat menghitung beberapa himpunanyang berisi output yang benar, maka kita dapat menggunakan spesifikasi untuk memutuskan kapan kita menemukannya. Pola dasar terlihat seperti ini:

```
for x in possible outputs do:
  if specification(input, x) = OK then:
    return x
```

Jika diperhatikan pola diatas begitu sederhana dimana apa pun yang dikembalikan prosedur ini akan memenuhi spesifikasi. Apa yang kadang-kadang lebih sulit adalah melihat bagaimana menghasilkan himpunan output yang mungkin, atau berapa waktu berjalan algoritma jika ukuran himpunan ini sulit untuk dijelaskan. Kita akan melihat beberapa contohnya di bawah ini.

Algoritma *exhaustive search* cenderung lambat, terutama jika himpunan output yang mungkin dalam ukuran besar. Kadang-kadang kita dapat menggunakan solusi naïve exhaustive search sebagai titik awal untuk algoritma yang lebih baik dengan memikirkan cara membatasi sekumpulan solusi untuk memasukkan lebih sedikit solusi yang kurang sesuai. Contoh klasik adalah masalah salesman keliling (TSP). Misalkan seorang salesman perlu mengunjungi 10 kota di seluruh negeri. Bagaimana cara menentukan urutan kota mana yang harus dikunjungi sehingga jarak total yang ditempuh diminimalkan? Solusi brute force hanyalah menghitung jarak total untuk setiap rute yang mungkin dan kemudian memilih yang terpendek. Ini tidak terlalu

efisien karena dimungkinkan untuk menghilangkan banyak rute yang mungkin melalui algoritma pintar. Contoh lain: 5 digit kata sandi, dalam skenario terburuk akan membutuhkan 105 percobaan untuk memecahkan. Kompleksitas waktu dari brute force adalah O (n * m). Jadi, jika kita mencari string karakter ‘n’ dalam string karakter ‘m’ menggunakan brute force, itu akan membawa kita n * m mencoba.

Algoritma brute force merupakan algoritma yang biasanya digunakan untuk pemecahan masalah dengan skala kecil hingga menengah. Algioritma brute force tidak efisien jika digunakan untuk memecahkan masalah dengan domain skala besar. Algoritma brute force sendiri memiliki cara berpikir yang sederhana dalam memecahkan masalah. Inti dari pemecahan masalah adalah mencoba seluruh kemungkinan solusi yang ada. Semakin besar skala permasalahan, maka jumlah kemungkin solusi juga akan bertambah. Pertambahan ini bergantung pada jenis permasalah, ada yang mengalamai peningkatan secara linear, logaritma, hingga eksponensial. Banyak algoritma-algoritma rumit yang sebenarnya merupakan modifikasi dan pengembangan dari algoritma brute force. Algoritma brute force sendiri seringkali digunakan sebagai pembanding saat dilakukan uji coba kualitas suatu algoritma. Terdapat juga masalah yang hanya dapat diselekasikan menggunakan algoritma brute force seperti permasalahan mencarai elemen maksimum di dalam senarai. Levitin (2003) mendefinisikan Brute force sebagai suatu pendekatan yang langsung/lempeng untuk menyelesaikan suatu masalah, biasanya secara langusng sesuai dengan pernyataan masalah dan definisi dari konsep yang ada. Kata ”force” mempengaruhi definisi strategi yaitu suatu mesin/robot/komputer bukan sesutu yang berbau pintar(*intelligent*). ” Just do it!” dapat merupakan cara lain untuk mendeskripsikan pengertian dari brute force. dan tentu saja strategi brute-force merupakan strategi yang paling mudah diaplikasikan.

Munir (2004) menjelaskan bahwa Brute Force adalah sebuah pendekatan langsung (*straight forward*) untuk memecahkan suatu masalah, yang biasanya didasarkan pada pernyataan masalah (problem statement) dan definisi konsep yang dilibatkan. Pada dasarnya algoritma Brute Force adalah alur penyelesaian suatu permasalahan dengan cara berpikir yang sederhana dan tidak membutuhkan suatu permikiran yang lama. Sebenarnya, algoritma Brute Force merupakan algoritma yang muncul karena pada dasarnya alur pikir manusia adalah Brute Force (langsung/to the

point). Beberapa karakteristik dari algoritma Brute Force dapat dijelaskan sebagai berikut.

a. Membutuhkan jumlah langkah yang banyak dalam menyelesaikan suatu permasalahan sehingga jika diterapkan menjadi suatu algoritma program aplikasi akan membutuhkan banyak memori.
b. Digunakan sebagai dasar dalam menemukan suatu solusi yang lebih efektif.
c. Banyak dipilih dalam penyelesaian sebuah permasalahan yang sederhana karena kemudahan cara berpikirnya.
d. Pada banyak kasus, algoritma ini banyak dipilih karena hampir dapat dipastikan dapat menyelesaikan banyak persoalan yang ada.
e. Digunakan sebagai dasar bagi perbandingan keefektifan sebuah algoritma.

Kelebihan dari algoritma Brute Force adalah sebagai berikut.

a. Metode Brute Force dapat digunakan untuk memecahkan hampir sebagian besar masalah (wide applicability).
b. Metode Brute Force sederhana dan mudah dimengerti.
c. Metode Brute Force menghasilkan algoritma yang layak untuk beberapa masalah penting, seperti pencarian, pengurutan, pencocokan string, perkalian matriks.
d. Metode Brute Force menghasilkan algoritma baku (standard) untuk tugas-tugas komputasi seperti penjumlahan/perkalian n buah bilangan, menentukan elemen minimum atau maksimum di dalam tabel (list).

Selain itu, terdapat beberapa kelemahan dari algoritma Brute Force:

a. Metode Brute Force jarang menghasilkan algoritma yang efektif.
b. Beberapa algoritma Brute Force lambat sehingga tidak dapat diterima.
c. Tidak sekontruktif/sekreatif teknik pemecahan masalah lainnya

3.2 CONTOH-CONTOH ALGORITMA BRUTE FORCE

Berikut ini adalah beberapa contoh algoritma brute force

Contoh 1

Mencari elemen terbesar (terkecil)

Persoalan: Diberikan sebuah senarai yang beranggotakan *n* buah bilangan bulat (a_1, a_2, …, a_n). Carilah elemen terbesar di dalam senarai tersebut. *brute force*: bandingkan setiap elemen senarai untuk menemukan elemen terbesar

Algoritma:

```
maks←a1
for k←2 to n do
  if ak > maks then
    maks←ak
  endif
endfor
```

Implementasi dengan menggunakan Bahasa Program Java diperoleh :

```
int maks = [0];
For (k=1;k,n;k++)
{
      if(a[k]>maks)
      {
            Maks=a[k];
      }
}
```

Trace

Jika diambil contoh untuk mengurutkan 4 bilangan : 0, 9, 0, 6 maka ada n=4 dan maks=a[0]=0. Selanjutnya hasil tracer programnya dapat dilihat pada tabel 3.1.

Tabel 3.1 Hasil Tracer Mencari elemen terbesar

k	k<n	a[k]>maks	maks=a[k]	k++
1	1<4	a[1]>maks 9>0 T	maks=a[1] maks=9	2
2	2<4	a[2]>maks 0>9 F	-	3
3	3<4	a[3]>maks 6>9 F	-	4
4	4<4	-	-	-

Contoh 2.

Pencarian beruntun *(Sequential Search)*

Persoalan: Diberikan senarai yang berisi *n* buah bilangan bulat (a_1, a_2, …, a_n). Carilah nilai *x* di dalam senarai tersebut. Jika *x* ditemukan, maka keluarannya adalah indeks elemen senarai, jika *x* tidak ditemukan, maka keluarannya adalah -1.

Pencarian selesai jika *x* ditemukan atau elemen senarai sudah habis diperiksa.

Algoritma

```
k←1
while (k < n) and (a_k ≠ x) do
   k ← k + 1
endwhile
{ k = n or a_k = x }
if a_k = x then   { x ditemukan }
   idx ← k
else
   idx ← -1       { x tidak ditemukan }
endif
```

Implementasi dengan menggunakan Bahasa Program Java diperoleh :

```
int k = 0;
while (k<n && a[k]!=x)
{

      K+=1;
{
      if(a[k==x)
      {
            idx=k;
      }
      else
      {
            Idx=-1
      }
```

Trace

Jika diambil contoh untuk mengurutkan 4 bilangan : 0, 9, 0, 6 maka ada n=4 dan maks=a[0]=0. Selanjutnya hasil tracer programnya dapat dilihat pada tabel 3.2.

Tabel 3.2 Hasil Tracer Pencarian beruntun

k	**k<n && a[k]=x**	**k+=1**
	0<4 & a[0] ≠0	
0	0<4 & 0 ≠0	-
	T & F	

Contoh 3.

Menghitung a^n ($a > 0$, n adalah bilangan bulat tak-negatif)

Definisi:

$a^n = a \times a \times \ldots \times a$ (n kali) , jika $n > 0$

$= 1$, jika $n = 0$

Algoritma *brute force*: kalikan 1 dengan a sebanyak n kali

Algoritma

```
function pangkat(a : real, n : integer) → real
{ Menghitung a^n }
Deklarasi
  i : integer
  hasil : real
Algoritma:
    hasil ← 1
  for i ← 1 to n do
    hasil ← hasil * a
  end
  return hasil
```

Bahasa Program Java

```
hasil = 1;
for (int i=1;i<=n;i++)
{
      hasil=hasil*a
}
      return hasil;
```

Trace

Selanjutnya hasil tracer programnya untuk a=9 dan n=6 dapat dilihat pada tabel 3.3.

Tabel 3.3 Hasil Tracer a^n untuk a=9 dan n=6

i	i<=n	hasil *= a	i++
1	1<=6	hasil = 1 * 9 hasil = 9	2
2	2<=6	hasil = 9 * 9 hasil = 81	3

i	i<=n	hasil *= a	i++
3	3<=6	hasil = 81 * 9 hasil = 729	4
4	4<=6	hasil = 729 * 9 hasil = 6561	5
5	5<=6	hasil = 6561 * 9 hasil = 59049	6
6	6<=6	hasil = 59049 * 9 hasil = 531441	7
7	-	-	-

Hasil akhir yang didapat adalah untuk a=9 dan n=6 adalah : hasil = 531441

Contoh 4.

Menghitung *n*! (*n* bilangan bulat tak-negatif)

Definisi:

$$n! = 1 \times 2 \times 3 \times \ldots \times n \quad \text{, jika } n > 0$$

$$= 1 \quad \text{, jika } n = 0$$

Algoritma *brute force*: kalikan *n* buah bilangan, yaitu 1, 2, 3, …, *n*, bersama-sama

Algoritma

```
function faktorial(n : integer) → integer
{ Menghitung n! }
Deklarasi
  i : integer
  fak : real
Algoritma:
    fak ← 1
  for i ← 1 to n do
    fak ← fak * i
  end
  return fak
```

Bahasa Program Java

```
fak = 1
for (int i=1;i<=n;i++)
{
```

```
        fak=fak*a
    }
        return fak;
```

Trace

Selanjutnya hasil tracer programnya untuk fak = 1 dan n = 6 dapat dilihat pada tabel 3.4

Tabel 3.4 Hasil Tracer **factorial** untuk fak = 1 dan n = 6

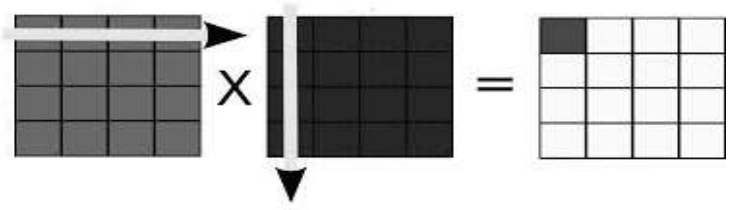
i	i<=n	fak = fak * i	i++
1	1<=6	fak = 1 * 1 fak = 1	2
2	2<=6	fak = 1 * 2 fak = 2	3
3	3<=6	fak = 2 * 3 fak = 6	4
4	4<=6	fak = 6 * 4 fak = 24	5
5	5<=6	fak = 24 * 5 fak = 120	6
6	6<=6	fak = 120 * 6 fak = 720	7
7	-	-	-

Contoh 5.

Mengalikan dua buah matriks, *A* dan *B*

Definisi: Misalkan $C = A \times B$ dan elemen-elemen matriks dinyatakan dalam c_{ij}, a_{ij}, dan b_{ij}

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$



Algoritma *brute force*: hitung setiap elemen hasil perkalian satu per satu, dengan cara mengalikan dua vektor yang panjangnya *n*.

Algoritma

```
for i←1 to n do
    for j←1 to n do
```

```
        C[i,j]←0    { inisialisasi penjumlah }
        for k ← 1 to n do
           C[i,j]←C[i,j] + A[i,k]*B[k,j]
        endfor
     endfor
  endfor
```

Contoh 6.

Tes Bilangan Prima

Persoalan: Diberikan sebuah bilangan bilangan bulat positif. Ujilah apakah bilangan tersebut merupakan bilangan prima atau bukan.

Definisi: bilangan prima adalah bilangan yang hanya habis dibagi oleh 1 dan dirinya sendiri.

Algoritma *brute force*: bagi *n* dengan 2 sampai $\sqrt{n}$. Jika semuanya tidak habis membagi *n*, maka *n* adalah bilangan prima.

Algoritma

```
if x < 2 then    { 1 bukan prima }
    return false
  else
    if x = 2 then  { 2 adalah prima, kasus khusus }
      return true
    else
      y←⌈√x⌉
      test←true
      while (test) and (y ≥ 2) do
        if x mod y = 0 then
          test←false
        else
          y←y - 1
        endif
      endwhile
      { not test or y < 2 }
```

```
      return test
   endif
endif
```

3.3 ALGORITMA BUBBLE SORT

Bubble Sort adalah algoritma sederhana yang digunakan untuk mengurutkan set elemen yang diberikan dalam bentuk array dengan jumlah elemen. Bubble Sort membandingkan semua elemen satu per satu dan mengurutkannya berdasarkan nilainya.

Jika array yang diberikan harus diurutkan dalam urutan menaik, maka bubble sort akan mulai dengan membandingkan elemen pertama dari array dengan elemen kedua, jika elemen pertama lebih besar dari elemen kedua, itu akan menukar kedua elemen, dan kemudian beralih untuk membandingkan elemen kedua dan ketiga, dan seterusnya. Jika kita memiliki total n elemen, maka kita perlu mengulangi proses ini untuk n-1 kali. Ini dikenal sebagai semacam gelembung, karena dengan setiap iterasi lengkap elemen terbesar dalam array yang diberikan, gelembung naik ke tempat terakhir atau indeks tertinggi, seperti gelembung air naik ke permukaan air. Penyortiran terjadi dengan melangkah melalui semua elemen satu per satu dan membandingkannya dengan elemen yang berdekatan dan menukar mereka jika diperlukan.

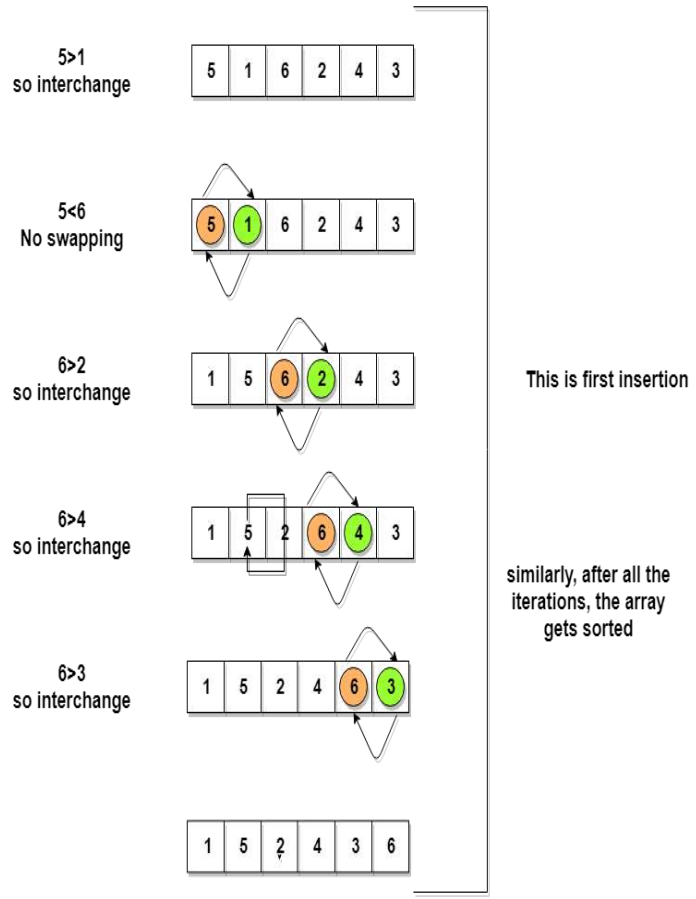
Beberapa Karakteristik Bubble Sort yaitu :

a. Nilai besar selalu disortir terlebih dahulu.
b. Hanya perlu satu iterasi untuk mendeteksi bahwa koleksi sudah diurutkan.
c. Kompleksitas waktu terbaik untuk Bubble Sort adalah O (n). Kompleksitas waktu rata-rata dan terburuk adalah O (n^2).
d. Kompleksitas ruang untuk Bubble Sort adalah O (1), karena hanya diperlukan satu ruang memori tambahan.

Bubble sort dapat dituliskan ke dalam algoritma atau langkah-langkah nya (untuk mengurutkan array yang diberikan dalam urutan menaik) yaitu:

1. Dimulai dengan elemen pertama (indeks = 0), bandingkan elemen saat ini dengan elemen array berikutnya.
2. Jika elemen saat ini lebih besar dari elemen array selanjutnya, tukar.
3. Jika elemen saat ini kurang dari elemen berikutnya, pindah ke elemen berikutnya. Ulangi Langkah 1.

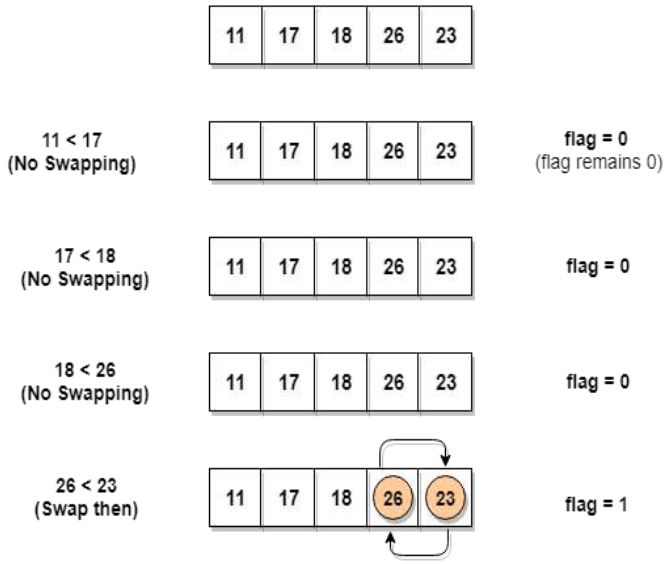
Mari kita pertimbangkan sebuah array dengan nilai {5, 1, 6, 2, 4, 3}, Di bawah, kami memiliki representasi bergambar tentang bagaimana bubble sort akan mengurutkan array yang diberikan.



Gambar 3.1 Proses algoritma bubble sort mengurutkan array

Gambar 3.1 menunjukkan bahwa setelah iterasi pertama, 6 ditempatkan pada indeks terakhir, yang merupakan posisi yang tepat untuk itu. Demikian pula setelah iterasi kedua, 5 akan berada di indeks terakhir kedua, dan seterusnya.

Meskipun logika pada gambar 3.1 akan mengurutkan array yang tidak disortir, tetap saja algoritma di atas tidak efisien karena sesuai dengan logika di atas, bagian luar untuk loop akan terus mengeksekusi untuk 6 iterasi bahkan jika array akan diurutkan setelah iterasi kedua. Jadi, berdasarkan hasil tersebut diperoleh dengan jelas pengoptimalan algoritma Bubble sort tersebut



Gambar 3.2 Representasi bubble sort yang telah dioptimalkan mengurutkan array

Untuk mengoptimalkan algoritma **Bubble** sort, pada iterasi tertentu, tidak ada swapping yang terjadi, itu berarti array telah diurutkan dan kita dapat melompat keluar dari for loop, alih-alih mengeksekusi semua iterasi.

Jika diketahui sebuah array dengan nilai {11, 17, 18, 26, 23} berikut ini merupakan representasi bergambar tentang bagaimana buble sort yang dioptimalkan akan mengurutkan array yang diberikan.

Gambar 3.2 menunjukkan bahwa pada iterasi pertama, swapping terjadi, maka nilai flag menjadi 1, sebagai hasilnya, eksekusi memasuki loop for lagi. Tetapi dalam iterasi kedua, tidak ada swapping yang akan terjadi, maka nilai flag akan tetap 0, dan eksekusi akan keluar dari loop.

Kode program untuk algoritma bubble sort dapat ditulis:

```
#include <stdio.h>

void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
```

```
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}

int main()
{
    int arr[100], i, n, step, temp;
        printf("Masukkann jumlah elemen yang akan diurutkan: ");
    scanf("%d", &n);
        for(i = 0; i < n; i++)
    {
        printf("Masukkan elemen ke. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    bubbleSort(arr, n);

    return 0;
}
```

Koding lainnya untuk bubble sort seperti berikut ini

```
#include <stdio.h>

void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {

            int flag = 0;
            if( arr[j] > arr[j+1])
            {

                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;

                flag = 1;
```

```
            }
        }

        if(!flag)
        {
            break;
        }
    }
        printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}
int main()
{
    int arr[100], i, n, step, temp;

    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);

    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    bubbleSort(arr, n);

    return 0;
}
```

Dalam kode di atas, dalam fungsi bubble Sort, jika untuk satu siklus lengkap iterasi (untuk loop), tidak ada swapping yang terjadi, maka flag akan tetap 0 dan kemudian kita akan keluar dari loop for nya, karena array sudah disortir.

Berikutnya untuk menentukan kompleksitasnya, perbandingan n-1 Dalam Bubble Sort akan dilakukan di pass pertama, n-2 di pass kedua, n-3 di pass ketiga dan seterusnya. Jadi jumlah total perbandingan akan menjadi,

Output

```
(n-1) + (n-2) + (n-3) + ..... + 3 + 2 + 1
Sum = n(n-1)/2
i.e O(n2)
```

Karenanya kompleksitas waktu Bubble Sort adalah O (n^2).

Keuntungan utama Bubble Sort adalah kesederhanaan algoritma.

Kompleksitas ruang untuk Bubble Sort adalah O (1), karena hanya diperlukan satu ruang memori tambahan, mis. Untuk variabel temp. Kompleksitas waktu kasus terbaik adalah O (n), yaitu ketika daftar sudah diurutkan. Berikut ini adalah kompleksitas Waktu dan Ruang untuk algoritma Bubble Sort yaitu :

- Kompleksitas Waktu Kasus Terburuk [Big-O]: O (n^2)
- Kompleksitas Waktu Kasus Terbaik [Big-omega]: O (n)
- Kompleksitas Waktu Rata-Rata [Big-theta]: O (n^2)
- Kompleksitas Ruang: O (1)
-

3.5 ALGORITMA SELECTION SORT

Selection Sort secara konseptual adalah algoritma penyortiran yang paling sederhana. Algoritma ini pertama-tama akan menemukan elemen terkecil dalam array dan menukarnya dengan elemen di posisi pertama, kemudian akan menemukan elemen terkecil kedua dan menukar dengan elemen di posisi kedua, dan akan terus melakukan ini sampai seluruh array diurutkan. Penyebutan selection sort karena algoritma ini akan berulang kali memilih elemen terkecil berikutnya dan menukarnya ke tempat yang tepat.

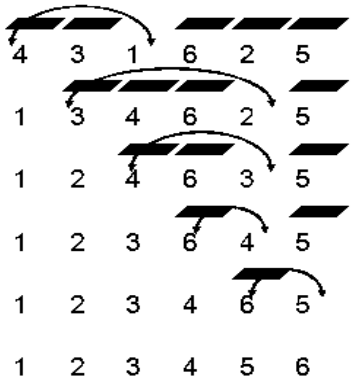
Bagaimana algoritma selection sort bekerja? berikut ini adalah langkah-langkah yang terlibat dalam pengurutan pilihan (untuk mengurutkan array yang diberikan dalam urutan menaik):

1. *Iterasi* ke –1:
    a. Cari elemen terkecil mulai di dalam *s*[1..*n*]
    b. Letakkan elemen terkecil pada posisi ke-1 (pertukaran)
2. *Iterasi* ke-2:
    a. Cari elemen terkecil mulai di dalam *s*[2..*n*]
    b. Letakkan elemen terkecil pada posisi 2 (pertukaran)
3. Perlakuan tersebut diulangi sampai hanya tersisa 1 elemen

    Semuanya ada *n* –1 kali *Iterasi*

Sebagai contoh jika array dengan nilai {4, 3, 1, 6, 2, 5}akan diurutkan dengan menggunakan selection sort maka hasilnya dapat dilihat representasi bergambar tentang bagaimana pemilihan akan mengurutkan array yang diberikan pada gambar 3.3.

Gambar 3.3 menunjukkan bahwa pada pass pertama, elemen terkecil adalah 1, sehingga akan ditempatkan di posisi pertama. Kemudian meninggalkan elemen pertama, elemen terkecil berikutnya akan dicari, dari elemen yang tersisa. Selanjutnya didapatkan 2 sebagai yang terkecil, sehingga akan ditempatkan di posisi kedua. Kemudian meninggalkan 1 dan 2 (karena mereka berada di posisi yang benar), selanjutnya akan dicari terkecil berikutnya dari sisa elemen dan meletakkannya di posisi ketiga dan seterusnya dilakukan sampai array diperoleh telah berurutan.



Gambar 3.3 Representasi Algoritma Selection Sort dari array {4, 3, 1, 6, 2, 5}

Untuk lebih jelasnya dari algoritma selection sort ini dapat diimplementasikan ke dalam bahasa pemrograman. Untuk kesempatan ini bahasa pemrograman yang digunakan adalah bahasa pemrograman C. Program selection sort tersebut telah dibagi menjadi fungsi-fungsi kecil, sehingga lebih mudah bagi yang membacanya untuk memahami bagian mana dan apa yang dilakukan.

```
// program implementasi Selection Sort
# include <stdio.h>

void swap(int arr[], int firstIndex, int secondIndex)
{
    int temp;
    temp = arr[firstIndex];
    arr[firstIndex] = arr[secondIndex];
    arr[secondIndex] = temp;
}
int indexOfMinimum(int arr[], int startIndex, int n)
{
    int minValue = arr[startIndex];
    int minIndex = startIndex;

    for(int i = minIndex + 1; i < n; i++) {
```

```
        if(arr[i] < minValue)
        {
            minIndex = i;
            minValue = arr[i];
        }
    }
    return minIndex;
}
void selectionSort(int arr[], int n)
{
    for(int i = 0; i < n; i++)
    {
        int index = indexOfMinimum(arr, i, n);
        swap(arr, i, index);
    }
}
void printArray(int arr[], int size)
{
    int i;
    for(i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main()
{
    int arr[] = {46, 52, 21, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Analisis Kompleksitas Selection Sort

Selection sort memerlukan dua bersarang (nested) loop "for" untuk menyelesaikan sendiri, satu loop "for" dalam *function selectionSort*, dan di dalam loop pertama dibuat panggilan ke *function* lainnya yaitu *indexOfMinimum*, yang memiliki loop kedua (dalam) loop "for", karenanya untuk ukuran input n yang diberikan, berikut ini akan menjadi kompleksitas waktu dan ruang bagi algoritma *selection sort*:

Kompleksitas Waktu Kasus Terburuk [Big-O]: O (n2)

Kompleksitas Waktu Kasus Terbaik [Big-omega]: O (n2)

Kompleksitas Waktu Rata-Rata [Big-theta]: O (n2)

Kompleksitas Ruang: O (1)

3.5 EXHAUSTIVE SEARCH

Exhaustive search (Pencarian lengkap) juga dikenal sebagai algoritma *backtracking*, meskipun tidak semua algoritma backtracking lengkap. Algoritma ini akan mencari setiap cara yang mungkin untuk mendapatkan solusi. Biasanya dikombinasikan dengan *pruning* (pemangkasan) untuk mengurangi jumlah item yang dicari. Ia juga dikenal sebagai Backtracking.

Nievergelt (2000) mendefinisikan Exhaustive search suatu Pencarian yang dijamin untuk menemukan semua negara yang memenuhi kendala yang diberikan. Pencarian yang lengkap tidak perlu mengunjungi semuanya dalam setiap contoh. Ini dapat menghilangkan subruang S' dari S berdasarkan argumen matematis yang menjamin bahwa S tidak dapat mengandung solusi apa pun. Namun, dalam konfigurasi terburuk, pencarian lengkap dipaksa untuk mengunjungi semua status S'.

Exhaustive search juga merupakan salah satu solusi brute force untuk masalah yang melibatkan pencarian elemen dengan sifat-sifat khusus, biasanya di antara objek kombinatorial seperti permutasi, kombinasi, atau himpunan bagian dari himpunan.
Terminologi lain yang terkait erat dengan *brute force* adalah *exhaustive search*. Baik *brute force* maupun *exhaustive search* sering dianggap dua istilah yang sama, padahal dari jenis masalah yang dipecahkan ada sedikit perbedaan.

Exhaustive search adalah teknik pencarian solusi secara *brute force* pada masalah yang melibatkan pencarian elemen dengan sifat khusus, biasanya di antara objek-objek kombinatorik seperti permutasi, kombinasi, atau himpunan bagian dari sebuah himpunan. Berdasarkan definisi ini, maka *exhaustive search* adalah *brute force* juga. Metode *exhaustive search* dapat dirumuskan langkah-langkahnya sebagai berikut:

1. Enumerasi (*list*) setiap solusi yang mungkin dengan cara yang sistematis.
2. Evaluasi setiap kemungkinan solusi satu per satu, mungkin saja beberapa kemungkinan solusi yang tidak layak dikeluarkan, dan simpan solusi terbaik yang ditemukan sampai sejauh ini (*the best solusi found so far*).
3. Bila pencarian berakhir, umumkan solusi terbaik (*the winner*)

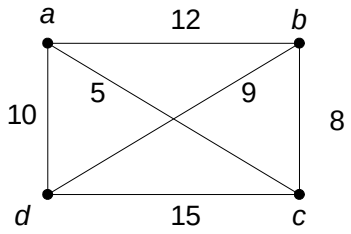
Jelaskah bahwa algoritma *exhaustive search* memeriksa secara sistematis setiap kemungkinan solusi satu per satu dalam pencarian solusinya. Meskipun algoritma *exhaustive* secara teoritis menghasilkan solusi, namun waktu atau sumberdaya yang dibutuhkan dalam pencarian solusinya sangat besar.

Beberapa literatur strategi algoritmik menyebutkan bahwa contoh masalah yang sering diasosiasikan dengan *exhaustive search* atau *brute force* adalah masalah *Travelling Salesperson Problem* (TSP). Masalah TSP sudah pernah dibahas dalam kuliah Matematika Diskrit pada pokok bahasan Graf. Untuk mengingat kembali masalah TSP ini, berikut diulang kembali deskripsi masalahnya.

TSP: diberikan *n* buah kota serta diketahui jarak antara setiap kota satu sama lain. Temukan perjalanan (*tour*) terpendek yang melalui setiap kota lainnya hanya sekali dan kembali lagi ke kota asal keberangkatan.

Jika setiap kota direpresentasikan dengan simpul dan jalan yang menghubungkan antar kota sebagai sisi, maka persoalan TSP ini dimodelkan dengan graf lengkap dengan *n* buah simpul. Bobot pada setiap sisi menyatakan jarak antar setiap kota. Persoalan TSP tidak lain adalah menemukan sirkuit Hamilton dengan bobot minimum.

Contoh 7: $n = 4$



Gambar 3.4 Graf 4 buah kota dengan panjang lintasannya

Algoritma *exhaustive search* untuk persoalan TSP ini adalah:

1. Enumerasikan (*list*) semua sirkuit Hamilton dari graf lengkap dengan n buah simpul.
2. Hitung (evaluasi) bobot setiap sirkuit Hamilton yang ditemukan pada langkah 1.
3. Pilih sirkuit Hamilton yang mempunyai bobot terkecil.

Misalkan simpul *a* adalah kota tempat dimulainya perjalanan (*starting city*). Enumerasikan semua sirkuit Hamilton dan hasilnya seperti ditunjukkan pada table 3.4.

Tabel 3.4 menunjukkan rute-rute perjalanan yang berawal dan berakhir pada simpul yang sama dalam hal ini simpul awalnya adalah a maka sudah pasti simpul akhirnya juga a. Untuk menentukan jumlah simpul dari *n* buah simpul semua rute perjalanan digunakan rumus permutasi dari $n - 1$ buah simpul. Permutasi dari $n - 1$

buah simpul adalah (n – 1)!. Untuk gambar 3.4 diperoleh jumlah rute lintasan sebanyak (4 – 1)! = 3! = 6 buah rute perjalanan.

Berdasarkan table 3.4 dari 4 kota dengan 6 buah kemungkinan rute perjalanan (atau sirkuit Hamilton) diperoleh Rute perjalananan terpendek adalah $a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$ atau $a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$ dengan bobot = 32.

Tabel 3.5 Hasil enumerasi semua lintasan sirkuit hamilton

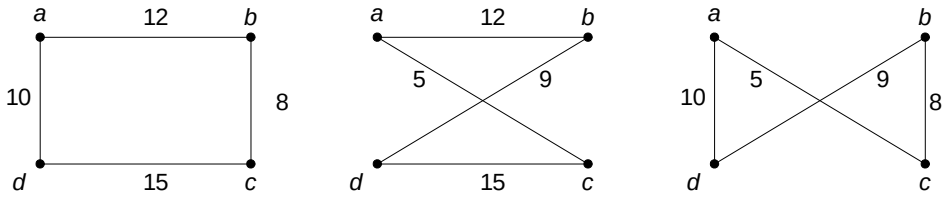
Rute perjalanan (*tour*)	Bobot
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	10+12+8+15 = 45
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	12+5+9+15 = 41
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	10+5+9+8 = 32
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	12+5+9+15 = 41
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	10+5+9+8 = 32
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	10+12+8+15 = 45

Jika persoalan TSP diselesaikan dengan metode *exhaustive search*, maka kita harus mengenumerasi sebanyak (n – 1)! buah sirkuit Hamilton, menghitung setiap bobotnya, dan memilih sirkuit Hamilton dengan bobot terkecil. Untuk menghitung bobot setiap sirkuit Hamilton dibutuhkan waktu $O(n)$, maka kompleksitas waktu algoritma *exhaustive search* untuk persoalan TSP sebanding dengan (n – 1)! dikali dengan waktu untuk menghitung bobot setiap sirkuit Hamilton. Dengan kata lain, kompleksitas waktu algoritma *exhaustive search* untuk persoalan TSP adalah $O(n \cdot n!)$.

Kita dapat menghemat jumlah komputasi dengan mengamati bahwa setengah dari rute perjalanan adalah hasil pencerminan dari setengah rute yang lain, yakni dengan mengubah arah rute perjalanan : 1 dan 6 ; 2 dan 4 serta 3 dan 5, maka dapat dihilangkan setengah dari jumlah permutasi (dari 6 menjadi 3). Ketiga buah sirkuit Hamilton yang dihasilkan adalah seperti gambar 3.5, dimana untuk graf dengan n buah simpul, kita hanya perlu mengevaluasi sirkuit Hamilton sebanyak (n – 1)!/2.

Jelaslah bahwa kebutuhan waktu algoritma *exhaustive search* adalah dalam orde ekponensial. Algoritma ini hanya bagus untuk ukuran masukan (n) yang kecil sebab

bebutuhan waktunya masih realistis. Untuk ukuran masukan yang besar, algoritma *exhaustive search* menjadi sangat tidak mangkus. Pada persoalan TSP misalnya, untuk jumlah simpul n = 20 akan terdapat (19!)/2 = 6×10^{16} sirkuit Hamilton yang harus dievaluasi satu per satu. Sayangnya, untuk persoalan TSP tidak ada algoritma lain yang lebih baik daripada algoritam *exhaustive search*. Jika anda dapat menemukan algoritma yang mangkus untuk TSP, anda akan menjadi terkenal dan kaya! Algoritma yang mangkus selalu mempunyai kompleksitas waktu dalam orde polinomial.



Gambar 3.5 hasil pencerminan dari setengah rute

Exhaustive search sering disebut-sebut di dalam bidang kriptografi, yaitu sebagai teknik yang digunakan penyerang untuk menemukan knci enkripsi dengan cara mencoba semua kemungkinan kunci. Serangan semacam ini dikenal dengan nama *exhaustive ke search attack* atau *brute force attack*.

Misalnya pada algoritma kriptografi DES (Data Encryption Standard), panjang kunci enkripsi adalah 64 bit (atau setara dengan 8 karakter). Dari 64 bit tersebut, hanya 56 bit yang digunakan (8 bit paritas lainnya tidak dipakai). Karena ada 56 posisi pengisian bit yang masing-masing memiliki dua kemungkinan nilai, 0 atau 1, maka jumlah kombinasi kunci yang harus dievaluasi oleh pihak lawan adalah sebanyak

(2)(2)(2)(2)(2)… (2)(2) (sebanyak 56 kali)= 2^{56} = 7.205.759.403.7927.936 buah.

Jika untuk percobaan dengan satu kunci memerlukan waktu 1 detik, maka untuk jumlah kunci sebanyak itu diperlukan waktu komputasi kurang lebih selama 228.4931.317 tahun!

Meskipun algoritma *exhaustive search* tidak mangkus, namun –sebagaimana ciri algoritma *brute force* pada umumnya– nilai plusnya terletak pada keberhasilannya yang selalu menemukan solusi (jika diberikan waktu yang cukup).

3.6 0/1 KNAPSAK PROBLEM

Masalah Knapsack adalah masalah yang terkenal dalam algoritma. Diberikan n items/objek yang memiliki properti seperti bobot $w_1,w_2,\cdots w_n$ dan keuntungan $p_1,p_2,\cdots,p_n$. Dengan kapasitas bobot Knapsack K. Permasalahan yang terjadi adalah bagaimana memilih memilih objek - objek yang dimasukkan ke dalam knapsack sedemikian sehingga memaksimumkan keuntungan. Total bobot objek yang dimasukkan ke dalam knapsack tidak boleh melebihi kapasitas knapsack K.

Persoalan 0/1 Knapsack dapat dipandang sebagai mencari himpunan bagian (subset) dari keseluruhan objek yang muat ke dalam knapsack dan memberikan total keuntungan terbesar, contoh penggunaan pada jasa pengangkutan barang.

Solusi persoalan dinyatakan sebagai vektor *n*-tupel:

$X = \{x_1, x_2, \ldots, x_n\}$

dengan,

$x_i = 1$ jika objek ke-*i* dimasukkan ke dalam *knapsack*,

$x_i = 0$ jika objek ke-*i* tidak dimasukkan.

Formulasi secara matematis dapat dibentuk menjadi:

Maksimumkan $$F = \sum_{i=1}^{n} p_i x_i$$

dengan kendala (*constraint*)

$$\sum_{i=1}^{n} w_i x_i \leq W$$

dengan, $x_i = 0$ atau 1, $i = 1, 2, \ldots, n$

Persoalan knapsack dapat diselesaikan salah satunya adalah dengan menggunakan Algoritma *exhaustive search*. Langkah-langkah yang dilakukan adalah :

1. Enumerasikan (*list*) semua himpunan bagian dari himpunan dengan n objek.
2. Hitung (evaluasi) total keuntungan dari setiap himpunan bagian dari langkah 1.
3. Pilih himpunan bagian yang memberikan total keuntungan terbesar.

Contoh 8:

Diketahui data dengan $n = 4$. dengan objek-objek tersebut diberi nomor 1, 2, 3, dan 4. Properti setiap objek *i* dan kapasitas *knapsack* adalah sebagai berikut

$w_1 = 2$; $p_1 = 20$

$w_2 = 5$; $p_1 = 30$

$w_3 = 10$; $p_1 = 50$

$w_4 = 5$; $p_1 = 10$

Kapasitas *knapsack* $W = 16$

Langkah-langkah pencarian solusi 0/1 *Knapsack* secara *exhaustive search* dirangkum dalam tabel 3.6:

Tabel 3.6 Hasil enumerasi himpunan bagian, bobot dan keuntungan

Himpunan Bagian	Total Bobot	Total keuntungan
{}	0	0
{1}	2	20
{2}	5	30
{3}	10	50
{4}	5	10
{1, 2}	7	50
{1, 3}	12	70
{1, 4}	7	30
{2, 3}	**15**	**80**
{2, 4}	10	40
{3, 4}	15	60
{1, 2, 3}	17	tidak layak
{1, 2, 4}	12	60
{1, 3, 4}	17	tidak layak
{2, 3, 4}	20	tidak layak
{1, 2, 3, 4}	22	tidak layak

Berdasarkan table 3.5 menunjukkan bahwa himpunan bagian objek yang memberikan keuntungan maksimum adalah {2, 3} dengan total keuntungan adalah 80 dan solusi persoalan 0/1 *Knapsack* di atas adalah $X = \{0, 1, 1, 0\}$

Contoh 9.

Misalkan A berisi 4 buah job ($n = 4$). Tenggat setiap *job* dan keuntungan masing-masing:

$(p_1, p_2, p_3, p_4) = (50, 10, 15, 30)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Tabel 3.7 Tenggat setiap job dan keuntungan

Job	Tenggat	Harus selesai pada pukul
1	2 jam	8.00
2	1 jam	7.00
3	2 jam	8.00
4	1 jam	7.00

Pemecahan Masalah dengan Algoritma Exhaustive Search

- Persoalan penjadwalan job dengan tenggat waktu dapat dipandang sebagai mencari himpunan bagian (*subset*) *job* yang layak dan memberikan total keuntungan terbesar.

Barisan *job*	Keuntungan (*F*)	Keterangan
{}	0	layak
{1}	50	layak
{2}	10	layak
{3}	15	layak
{4}	30	layak
{1, 2}	-	tidak layak
{1, 3}	65	layak
{1, 4}	-	tidak layak
{2, 1}	60	layak
{2, 3}	25	layak
{2, 4}	-	tidak layak
{3, 1}	65	layak
{3, 2}	-	tidak layak
{3, 4}	-	tidak layak
{4, 1}	**80**	**layak (Optimum!)**
{4, 2}	-	tidak layak
{4, 3}	45	layak

Solusi optimum: $J = \{4, 1\}$ dengan $F = 80$.

Algoritma *exhaustive search* untuk persoalan penjadwalan job dengan tenggat waktu mempunyai kompleksitas $O(n \cdot 2^n)$.

3.7 RANGKUMAN

1. Algoritma Brute Force mengacu pada gaya pemrograman yang tidak termasuk cara pintas untuk meningkatkan kinerja, tetapi sebaliknya mengandalkan

kekuatan komputasi belaka untuk mencoba semua kemungkinan sampai solusi untuk suatu masalah ditemukan.

2. Berapa banyak himpunan bagian dari sebuah himpunan dengan *n* elemen? Jawab adalah 2^n. Ini berarti, algoritma *exhaustive search* untuk persoalan 0/1 *Knapsack* mempunyai kompleksitas $O(2^n)$.
3. TSP dan 0/1 *Knapsack*, adalah contoh persoalan yang mempunyai kompleksitas algoritma eksponensial. Keduanya digolongkan sebagai persoalan NP (*Non-deterministic Polynomial*), karena tidak mungkin dapat ditemukan algoritma polinomial untuk memecahkannya
4. Bubble Sort adalah algoritma sederhana yang digunakan untuk mengurutkan set elemen yang diberikan dalam bentuk array dengan jumlah elemen. Bubble Sort membandingkan semua elemen satu per satu dan mengurutkannya berdasarkan nilainya.
5. *Exhaustive search* adalah teknik pencarian solusi secara *brute force* pada masalah yang melibatkan pencarian elemen dengan sifat khusus, biasanya di antara objek-objek kombinatorik seperti permutasi, kombinasi, atau himpunan bagian dari sebuah himpunan.

3.8 TES FORMATIF

1. Tuliskan program untuk Mengalikan dua buah matriks, *A* dan *B* dengan A dan B matriks sebarang dimana algoritmanya seperti pada sub bab 3.3
2. Tuliskan program untuk Tes Bilangan prima dengan inputan sebarang dan algoritmanya seperti pada subbab 3.3.
3. Metode yang paling lempang dalam memecahkan masalah pengurutan adalah algoritma pengurutan *bubble sort*. Algoritma *bubble sort* mengimplementasikan teknik *brute force* dengan jelas sekali. Tuliskan algoritma tersebut
4. Diberikan algoritma seperti di bawah ini

```
fungsi mistery( n: integer) -> integer
----------------------------------------
KAMUS:
       i: int
       Temp= int ----------------------------------------
Algoritma
       Temp=0;
   for( int i=1; i<=n; i++)
```

```
      Temp=Temp+ (i)/2
   endFor
return Temp
----------------------------------------
```

a. Jelaskan algoritma apakah di atas! Jika n=8, maka tentukan nilai dari fungsi mistery di atas!
b. Ubahlah algoritma di atas menjadi bentuk rekursif!
c. Jika setiap penjumlahan (+) dan pembagian (/) dihitung masing-masing 1 satuan waktu, maka tentukan relasi rekursi dari bentuk rekursif soal no (b)!
d. Tentukan waktu asimtotik dari relasi rekursi soal no (c) menggunakan metode coba-coba/ subsitusi!
e. Tentukan waktu asimtotik dari relasi rekursi soal no (c) menggunakan metode karakteristik!

5. Tinjau persoalan *Fractional Knapsack* dengan n = 6. Misalkan objek-objek tersebut kita beri nomor 1, 2, 3, 4, 5, dan 6. Properti setiap objek i dan kapasitas *knapsack* adalah sebagai berikut

   $w_1 = 10;\ p_1 = 45$
   $w_2 = 5;\ \ p_2 = 10$
   $w_3 = 2;\ \ p_3 = 25$
   $w_4 = 5;\ \ p_4 = 30$
   $w_5 = 8;\ \ p_5 = 16$
   $w_6 = 8;\ \ p_6 = 12$
   Kapasitas *knapsack* $W = 18$

   Selesaikan persoalan ini dengan algoritma *Brute Force* sehingga diperoleh keuntungan yang maksimum. Hitung juga berapa jumlah keuntungan yang dapat diperoleh.

6. Seorang turis ingin melakukan perjalanan yang menyenangkan di akhir pekan bersama teman-temannya. Dia akan pergi ke gunung untuk melihat berbagai keindahan alam, sehingga dia harus mempersiapakannya dengan baik. Dia memiliki ransel yang bagus untuk membawa barang-barang, namun ketika dia memperkirakan berapa muatan yang dapata ditampung di ranselnya dia mendapatkan bahwa maksimum hanya 4 kg di dalamnya, dan itu harus bertahan sepanjang hari. Dia membuat daftar apa yang ingin dia bawa untuk perjalanan

tetapi berat total semua barang terlalu banyak. Dia kemudian memutuskan untuk menambahkan kolom ke daftar awalnya yang merinci bobotnya dan nilai numerik yang menunjukkan betapa pentingnya barang itu untuk perjalanan.

Tabel 3.8 Daftar barang, bobot dan nilainya

item	weight (dag)	value
map	9	150
compass	13	35
water	153	200
sandwich	50	160
glucose	15	60
tin	68	45
banana	27	60
apple	39	40
cheese	23	30
Tea bottle	52	10
suntan cream	11	70
camera	32	30
T-shirt	24	15
trousers	48	10
umbrella	73	40
waterproof trousers	42	70
waterproof overclothes	43	75
note-case	22	80
sunglasses	7	20
towel	18	12
socks	4	50
book	30	10
knapsack	≤400 dag	?

Perlihatkan barang-barang mana yang dapat dibawa turis dalam ranselnya sehingga berat totalnya tidak melebihi 4 kg, dan nilai totalnya dimaksimalkan.

[dag = decagram = 10 gram]

7. Untuk menyerang para Dead Eaters, Harry Potter harus pergi ke Hogsmeade Village untuk membeli bahan-bahan pertempuran. Mengingat sisa uang Harry sebanyak 15 Galleons, maka bantulah Harry Potter memilih bahan pertempuran, dengan list barang-barang yang ada di Hogsmeade Village adalah sbb:

Tabel 3.9 Daftar nama senjata, harga dan kekuatan

No	Nama senjata	Harga	Kekuatan
1	Horcrux	10 Galleons	100
2	Jubah Ajaib	7 Galleons	80

3	The Marauder's Map	6 Galloens	50
4	Quidditch balls	12 Galloens	100

a. Tentukan definisi K, list pi dan wi untuk i = 1,2,3,4 pada cerita Harry Potter di atas!!

b. Carilah solusi 0/1 Knapsack menggunakan algoritma brute force Exhaustive Search!

8. Misalkan *u* dan *v* adalah dua buah bilangan bulat besar yang panjangnya *n* angka (*digit*):

$$u = u_1u_2\ldots u_n \quad \text{dan} \quad v = v_1v_2\ldots v_n$$

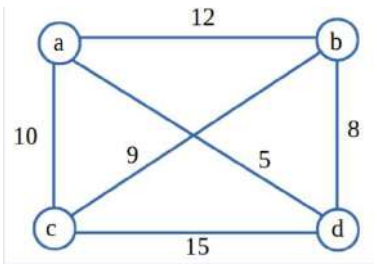
(keterangan: u_i dan v_i adalah angka $\in \{0, 1, 2, \ldots, 9\}$)

Kita ingin menjumlahkan dua buah bilangan bulat tersebut ($u + v$).

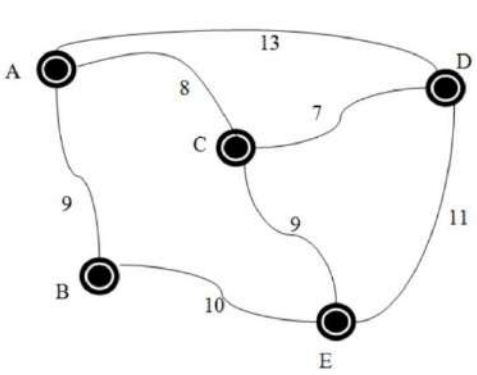
Bagaimana algoritma penjumlahan dengan algoritma *Brute Force*?

9. Tentukan rute dan jarak terpendek dari gambar graf berikut berikut. Pada TSP berikut:

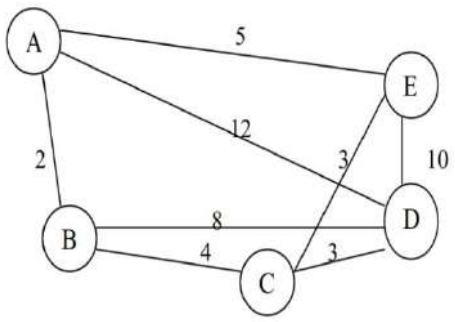
a.



b.



b.



BAB IV

ALGORITMA *DIVIDE AND CONQUER*

a) Deskripsi Singkat

Pendekatan divide and conquer mempunyai ciri khas yaitu membagi masalah menjadi sub-masalah yang lebih kecil, sub-masalah ini selanjutnya diselesaikan secara rekursif. Hasil dari setiap sub-masalah tidak disimpan untuk referensi di masa mendatang, Penggunaan pendekatan *divide and conquer* dilakukan ketika subproblem yang sama tidak diselesaikan beberapa kali

.

b) Relevansi

Keterkaitan materi dan proses pembelajaran dapat dilihat dari metode yang digunakan yang meliputi ceramah, tanya jawab, duskusi dan praktikum. Berkaitan dengan materi pada Bab 4 ini, kepada mahasiswa akan dijelaskan dengan detail materinya untuk kemudian algoritmanya diimplementasikan ke program melalui praktikum menggunakan bahasa pemrograman C, C++, Java atau python atau bahasa pemrograman yang dikuasi mahasiswa. Kesempatan bertanya dan berdiskusi juga diberikan dan tugas mandiri sebagai pengayaan materi-materi pada bab 3 ini.

c) Capaian Pembelajaran

1. Mahasiswa mampu menyelesaikan masalah dengan strategi *divide and conquer*

4.1 PENDAHULUAN

Divide and conquer merupakan salah satu teknik desain algoritma yang membagi masalah menjadi satu atau lebih banyak sub-masalah dengan ukuran yang kira-kira sama. *Divide and Conquer* dulunya adalah strategi militer yang dikenal dengan nama *divide ut imperes.* Sekarang strategi tersebut menjadi strategi fundamental di dalam ilmu komputer dengan nama *Divide and Conquer*.

Divide and Conquer Algorithms seperti halnya Pemrograman Greedy dan Pemrograman Dinamis, Divide and Conquer juga merupakan algoritma paradigm, yaitu

algoritma yang secara umum pendekatannya untuk mengkonstruksi solusi-solusi yang efisien dalam menyelesaikan berbagai permasalahan.

Untuk penyelesaian masalah dalam divede and conquer ini dapat menggunakan tiga tahapan atau langkah seperti berikut ini :

1. Divide: Pecahkan masalah yang diberikan ke dalam subproblem dengan tipe yang sama, dimana membagi masalah menjadi beberapa upa-masalah yang memiliki kemiripan dengan masalah semula namun berukuran lebih kecil (idealnya berukuran hampir sama),
2. Conquer: Selesaikan secara subproblem ini, dimana memecahkan (menyelesaikan) masing-masing upa-masalah (secara rekursif),
3. Combine: Gabungkan jawaban dengan tepat, dimana, mengabungkan solusi masing-masing upa-masalah sehingga membentuk solusi masalah semula

Obyek permasalahan yang dibagi adalah masukan (*input*) atau *instances* yang berukuran *n*: tabel (larik), matriks, eksponen, dan sebagainya, bergantung pada masalahnya.

Tiap-tiap upa-masalah mempunyai karakteristik yang sama (*the same type*) dengan karakteristik masalah asal, sehingga metode *Divide and Conquer* lebih natural diungkapkan dalam skema rekursif.

Berikut ini adalah beberapa algoritma standar yang merupakan algoritma Divide and Conquer.

a. Binary Search adalah algoritma pencarian. Di setiap langkah, algoritma membandingkan elemen input x dengan nilai elemen tengah dalam array. Jika nilainya cocok, kembalikan indeks tengah. Jika tidak, jika x kurang dari elemen tengah, maka algoritma berulang untuk sisi kiri elemen tengah, yang lain berulang untuk sisi kanan elemen tengah.
b. Quicksort adalah algoritma penyortiran. Algoritme mengambil elemen pivot, mengatur ulang elemen array sedemikian rupa sehingga semua elemen lebih kecil dari elemen pivot yang dipetik bergerak ke sisi kiri pivot, dan semua elemen yang lebih besar bergerak ke sisi kanan. Akhirnya, algoritma secara rekursif mengurutkan subarrays di kiri dan kanan elemen pivot.

c. Merge Sort juga merupakan algoritma penyortiran. Algoritma membagi array menjadi dua, membagi secara rekursif dan akhirnya menggabungkan dua bagian yang diurutkan.
d. Closest Pair of Points dimana masalahnya adalah menemukan pasangan poin terdekat dalam satu set poin dalam bidang x-y. Masalahnya dapat diselesaikan dalam waktu O (n^2) dengan menghitung jarak setiap pasangan poin dan membandingkan jarak untuk menemukan minimum. Algoritma Divide and Conquer menyelesaikan masalah dalam waktu O (nLogn).
e. Algoritma Strassen adalah algoritma yang efisien untuk melipatgandakan dua matriks. Metode sederhana untuk mengalikan dua matriks membutuhkan 3 loop bersarang dan O (n^ 3). Algoritma Strassen mengalikan dua matriks dalam waktu O (n ^ 2.8974).
f. Algoritma Cooley-Tukey Fast Fourier Transform (FFT) adalah algoritma yang paling umum untuk FFT. Ini adalah algoritma divide and conquer yang bekerja pada waktu O (nlogn).
g. Algoritma Karatsuba adalah algoritma multiplikasi pertama yang secara asimptotik lebih cepat daripada algoritma "grade school" kuadratik. Ini mengurangi penggandaan dua angka n-digit paling banyak ke n ^ 1,585 (yang merupakan perkiraan log 3 dalam basis 2) produk digit tunggal. Karena itu lebih cepat daripada algoritma klasik, yang membutuhkan n ^ 2 produk satu digit.

Berikut ini adalah beberapa contoh masalah yang diselesaikan dengan algoritma untuk divide and conquer.

Contoh 1.

```
procedure DIVIDE_and_CONQUER(input n : integer)
{ Menyelesaikan masalah dengan algoritma D-and-C.
  Masukan: masukan yang berukuran n
  Keluaran: solusi dari masalah semula
}
Deklarasi
     r, k : integer
Algoritma
  if n ≤ n₀ then  {ukuran masalah sudah cukup kecil }
```

```
    SOLVE upa-masalah yang berukuran n ini
  else
    Bagi menjadi r upa-masalah, masing-masing berukuran n/k
    for masing-masing dari r upa-masalah do
      DIVIDE_and_CONQUER(n/k)
    endfor
    COMBINE solusi dari r upa-masalah menjadi solusi masalah semula }
  endif
```

Jika pembagian selalu menghasilkan dua upa-masalah yang berukuran sama:

```
procedure DIVIDE_and_CONQUER(input n : integer)
{ Menyelesaikan masalah dengan algoritma D-and-C.
  Masukan: masukan yang berukuran n
  Keluaran: solusi dari masalah semula
}
Deklarasi
    r, k : integer

Algoritma
  if n ≤ n0 then {ukuran masalah sudah cukup kecil }
    SOLVE upa-masalah yang berukuran n ini
  else
    Bagi menjadi 2 upa-masalah, masing-masing berukuran n/2
    DIVIDE_and_CONQUER(upa-masalah pertama yang berukuran n/2)
    DIVIDE_and_CONQUER(upa-masalah kedua yang berukuran n/2)
    COMBINE solusi dari 2 upa-masalah
  Endif
```

Kompleksitas Waktu algoritma:

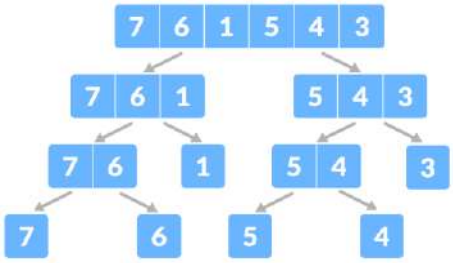
$$T(n) = \begin{cases} g(n) & ,n \le n_0 \\ 2T(n/2) + f(n) & ,n > n_0 \end{cases}$$

Contoh 2.

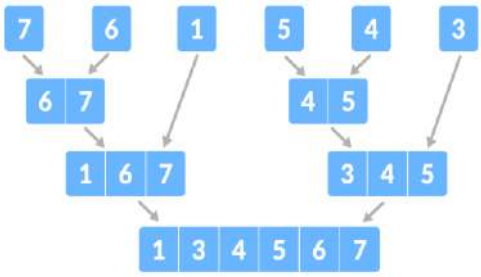
Jika diberikan array seperti berikut ini :

Dari data yang diberikan tersebut akan ditentukan urutan dari yang kecil ke yang terbesar, maka dengan menggunakan algoritma divide and conquer didapatkan :

Data array yang diketahui dibagi menjadi dua,



Sekali lagi, setiap sub bagian dibagi secara rekursif menjadi dua bagian sampai didapatkan elemen individual



Sekarang, masing-masing elemen digabungkan dengan cara diurutkan. Di sini, langkah-langkah *conquer* dan gabungan (*combine*) satu demi satu

Kompleksitas

Kompleksitas algoritma divide and conquer dihitung menggunakan teorema master.

T (n) = aT (n / b) + f (n),

dimana,

n = ukuran input

a = jumlah submasalah dalam rekursi

n/b = ukuran setiap sub-masalah. Semua submasalah diasumsikan memiliki ukuran yang sama.

f(n) = biaya pekerjaan yang dilakukan di luar panggilan rekursif, yang meliputi biaya membagi masalah dan biaya menggabungkan solusi

Contoh 3.

Persoalan Minimum dan Maksimum (MinMaks)

Persoalan: Misalkan diketahui tabel *A* yang berukuran *n* elemen sudah berisi nilai *integer*. Kita ingin menentukan nilai minimum dan nilai maksimum sekaligus di dalam tabel tersebut.

Penyelesaian dengan Algoritma Brute Force

```
procedure MinMaks1(input A : TabelInt, n : integer,
                   output min, maks : integer)
{ Mencari nilai minimum dan maksimum di dalam tabel A yang berukuran n
elemen, secara brute force.
Masukan: tabel A yang sudah terdefinisi elemen-elemennya
Keluaran: nilai maksimum dan nilai minimum tabel
}
Deklarasi
    i : integer

Algoritma:
    min← A1 { inisialisasi nilai minimum}
    maks←A1 { inisialisasi nilai maksimum }
    for i←2 to n do
      if Ai < min then
        min←Ai
      endif
      if Ai > maks then
        maks←Ai
      endif
   endfor
```

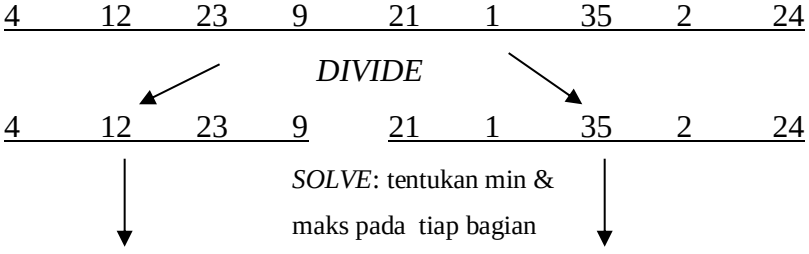
Jika $T(n)$ dihitung dari jumlah perbandingan elemen-elemen tabel:

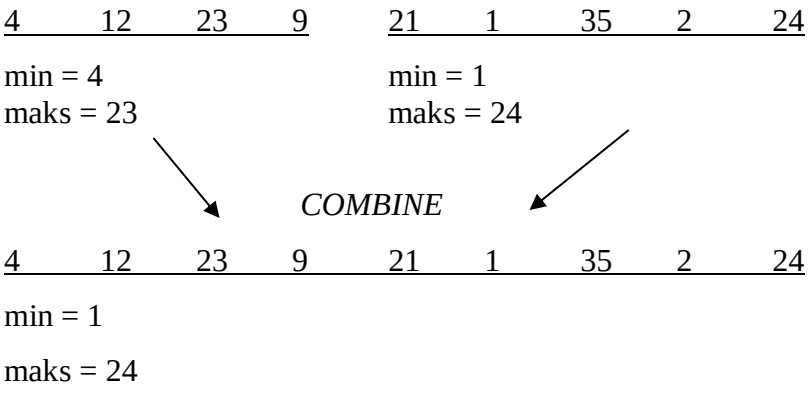
$$T(n) = (n-1) + (n-1)$$
$$= 2n - 2$$
$$= O(n)$$

Penyelesaian dengan Algoritma Divide and Conquer

4 12 23 9 21 1 35 2 24

Ide dasar algoritma secara *Divide and Conquer*:





Contoh 4.

Misalkan tabel *A* berisi elemen-elemen sebagai berikut:

Ukuran tabel hasil pembagian dapat dibuat cukup kecil sehingga mencari minimum dan maksimum dapat diselesaikan (SOLVE) secara lebih mudah. Dalam hal ini, ukuran kecil yang dipilih adalah 1 elemen atau 2 elemen.

Algoritma MinMaks:

1. Untuk kasus $n = 1$ atau $n = 2$,

   SOLVE: Jika $n = 1$, maka $min = maks = A_n$.

   Jika $n = 2$, maka bandingkan kedua elemen untuk menentukan *min* dan *maks*.

2. Untuk kasus $n > 2$,

   (a) DIVIDE: Bagi dua tabel *A* secara rekursif menjadi dua bagian yang berukuran sama, yaitu bagian kiri dan bagian kanan.

   (b) CONQUER: Terapkan algoritma *Divide and Conquer* untuk masing-masing bagian, dalam hal ini *min* dan *maks* dari tabel bagian kiri dinyatakan dalam peubah *min1* dan *maks1*, dan *min* dan *maks* dari tabel bagian kanan dinyatakan dalam peubah *min2* dan *maks2*.

   (c) COMBINE:

   Bandingkan *min1* dengan *min2* untuk menentukan *min* tabel *A*

   Bandingkan *maks1* dengan *maks2* untuk menentukan *maks* tabel *A*.

Contoh 5.

Tinjau kembali Contoh 2 di atas.

DIVIDE dan CONQUER-nya adalah: Bagi tabel menjadi dua bagian sempai berukuran 1 atau 2 elemen:

4 12 23 9 21 1 35 2 24

4 12 23 9 21 1 35 2 24

4 12 23 9 21 1 35 2 24

SOLVE dan COMBINE: Tentukan min dan maks masing-masing bagian tabel, lalu gabung:

4 12	23 9	21 1	35	2 24
min = 4	min = 9	min = 1	min = 35	min = 2
maks = 12	maks = 23	maks = 21	maks =35	maks = 24

4 12 23 9	21 1	35 2 24
min = 4	min = 1	min = 2
maks = 23	maks = 21	maks = 35

4 12 23 9	21 1 35 2 24
min = 4	min = 1
maks = 23	maks = 35

4 12 23 9 21 1 5 2 24

min = 1

maks = 35

Jadi, nilai minimum tabel = 1 dan nilai maksimum = 35.

Algoritma

```
procedure MinMaks2(input A : TabelInt, i, j : integer,
                   output min, maks : integer)
{ Mencari nilai maksimum dan minimum di dalam tabel A yang berukuran n
elemen secara Divide and Conquer.
Masukan: tabel A yang sudah terdefinisi elemen-elemennya
```

```
Keluaran: nilai maksimum dan nilai minimum tabel
}
Deklarasi
      min1, min2, maks1, maks2 : integer
Algoritma:
      if i=j then                    { 1 elemen  }
        min←Ai
        maks←Ai
      else
        if (i = j-1) then            { 2 elemen  }
           if Ai < Aj then
               maks←Aj
            min←Ai
           else
             maks←Ai
             min←Aj
           endif
        else                         { lebih dari 2 elemen }
            k←(i+j) div 2            { bagidua tabel pada posisi k }
            MinMaks2(A, i, k, min1, maks1)
            MinMaks2(A, k+1, j, min2, maks2)
            if min1 < min2 then
               min←min1
            else
               min←min2
            endif

            if maks1<maks2 then
               maks←maks2
            else
               maks←maks2
          endif
```

Listing program dalam bahasa C untuk Maksimum dan minimu *divide and conquer* menggunakan bahasa pemrograman C:

```
#include<stdio.h>
#include<stdio.h>
int max, min;
int a[100];
void maxmin(int i, int j)
{
 int max1, min1, mid;
 if(i==j)
 {
  max = min = a[i];
 }
 else
 {
  if(i == j-1)
  {
   if(a[i] <a[j])
   {
    max = a[j];
    min = a[i];
   }
   else
   {
    max = a[i];
    min = a[j];
   }
  }
  else
  {
   mid = (i+j)/2;
   maxmin(i, mid);
   max1 = max; min1 = min;
   maxmin(mid+1, j);
   if(max <max1)
    max = max1;
   if(min > min1)
    min = min1;
  }
 }
}
int main ()
{
 int i, num;
 printf ("\nEnter the total number of numbers : ");
 scanf ("%d",&num);
 printf ("Enter the numbers : \n");
 for (i=1;i<=num;i++)
  scanf ("%d",&a[i]);
```

```
max = a[0];
min = a[0];
maxmin(1, num);
printf ("Minimum element in an array : %d\n", min);
printf ("Maximum element in an array : %d\n", max);
return 0;}
```

Pada pemanggilan prosedur MinMaks2 pertama kali, $i = 1$ dan $j = n$ seperti di bawah ini:

MinMaks2(A,1,n,min,maks)

Kompleksitas waktu asimptotiknya adalah:

$$T(n) = \begin{cases} 0 & ,n = 1 \\ 1 & ,n = 2 \\ 2T(n/2) + 2 & ,n > 2 \end{cases}$$

Penyelesaian:

Asumsi: $n = 2^k$, dengan k bilangan bulat positif, maka

$T(n) = 2\mathrm{T}(n/2) + 2$

$= 2(2T(n/4) + 2) + 2 = 4T(\mathrm{n}/4) + 4 + 2$

$= 4T(2\mathrm{T}(n/8) + 2) + 4 + 2 = 8T(n/8) + 8 + 4 + 2$

$= ...$

$= 2^{k-1}\, T(2) + \sum_{i=1}^{k-1} 2^i$

$= 2^{k-1} \cdot 1 + 2^k - 2$

$= n/2 + n - 2$

$= 3n/2 - 2$

$= O(n)$

MinMaks1 secara *brute force* : $T(n) = 2\mathrm{n} - 2$

MinMaks2 secara *divide and conquer*: $T(n) = 3n/2 - 2$

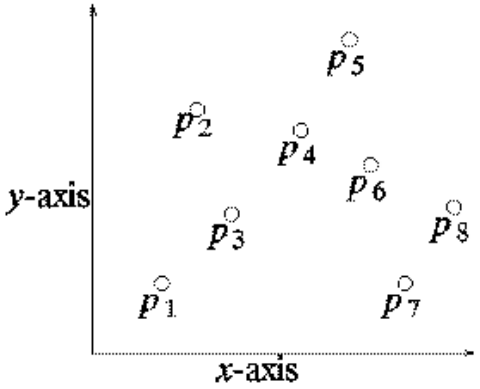
Perhatikan: $3n/2 - 2 < 2n - 2$ untuk $n \geq 2$.

Kesimpulan: algoritma MinMaks lebih mangkus dengan metdoe *Divide and Conquer*.

Contoh 6.

Mencari Pasangan Titik yang Jaraknya Terdekat (*Closest Pair*)

Persoalan: Diberikan himpunan titik, *P*, yang terdiri dari *n* buah titik, (x_i, y_i), pada bidang 2-D seperti yang ditunjukkan pada gambar 4.1. Tentukan jarak terdekat antara dua buah titik di dalam himpunan *P*. .



Gambar 4.1 titik, (x_i, y_i), pada bidang 2-D

Jarak dua buah titik $p_1 = (x_1, y_1)$ dan $p_2 = (x_2, y_2)$ dapat ditentukan dengan menggunakan rumus jarak Euclidean seperti berikut ini :

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Ada dua teknik untuk menyelesaikannya yaitu dengan menggunakan algoritma brute force dan algoritma divide and conquer.

a. Penyelesaian dengan Algoritma Brute Force

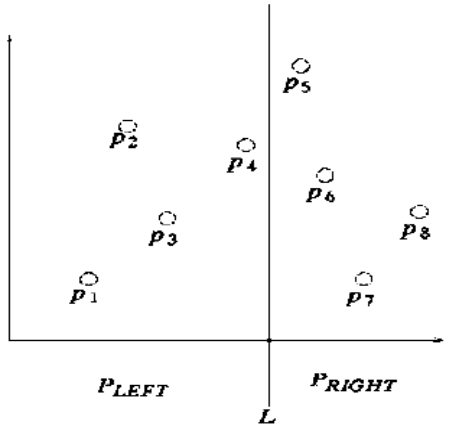
Algoritma *brute force* akan menghitung sebanyak $C(n, 2) = n(n - 1)/2$ pasangan titik dan memilih pasangan titik yang mempunyai jarak terkecil. Jelas kompleksitas algoritma adalah $O(n^2)$.

b. Penyelesaian dengan Algoritma *Divide and Conquer*

Asumsi: $n = 2^k$ dan titik-titik diurut berdasarkan absis (x).

Algoritma *Closest Pair*:

1. SOLVE: jika n = 2, maka jarak kedua titik dihitung langsung dengan rumus Euclidean.
2. DIVIDE: Bagi titik-titik itu ke dalam dua bagian, P_{left} dan P_{right}, setiap bagian mempunyai jumlah titik yang sama seperti pada gambar 4.2.



Gambar 4.2 titik, (x_i, y_i), pada bidang 2-D setelah dibagi dua

3. CONQUER: Secara rekursif, terapkan algoritma *D-and-C* pada masing-masing bagian.
4. Pasangan titik yang jaraknya terdekat ada tiga kemungkinan letaknya:
   (a) Pasangan titik terdekat terdapat di bagian P_{Left}.
   (b) Pasangan titik terdekat terdapat di bagian P_{Right}.
   (c) Pasangan titik terdekat dipisahkan oleh garis batas L, yaitu satu titik di P_{Left} dan satu titik di P_{Right}.

Jika kasusnya adalah (c), maka lakukan tahap COMBINE untuk mendapatkan jarak dua titik terdekat sebagai solusi persoalan semula.

```
procedure FindClosestPair2(input P: SetOfPoint, n : integer,
                           output delta : real)

{ Mencari jarak terdekat sepasang titik di dalam himpunan P. }

Deklarasi:
   DeltaLeft, DeltaRight : real

Algoritma:
   if n = 2 then
      delta ← jarak kedua titik dengan rumus Euclidean
   else
      P-Left← {p1, p2 ,..., pn/2 }
```

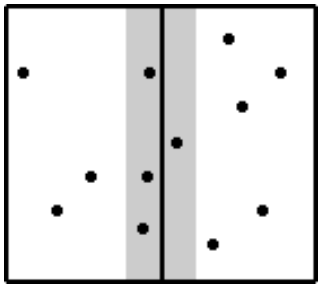
```
      P-Right ← {p_{n/2+1}, p_{n/2+2} ,..., p_n }
      FindClosestPair2(P-Left, n/2, DeltaLeft)
      FindClosestPair2(P-Right, n/2, DeltaRight)
      delta ← minimum(DeltaLeft, DeltaRight)
      {--*********************************************************--
}
      Tentukan apakah terdapat titik p_l di P-Left dan p_r di P-Right
      Dengan jarak(p_l, p_r) < delta. Jika ada, set delta dengan jarak
      terkecil tersebut.
      {--*********************************************************--
}
         endif
```

Jika terdapat pasangan titik p_l and p_r yang jaraknya lebih kecil dari *delta*, maka kasusnya adalah:

(i) Absis *x* dari p_l dan p_r berbeda paling banyak sebesar *delta*.

(ii) Ordinat *y* dari p_l dan p_r berbeda paling banyak sebesar *delta*.

Ini berarti p_l and p_r adalah sepasang titik yang berada di daerah sekitar garis vertikal *L* (lihat daerah yang diarsir pada gambar 4.3):

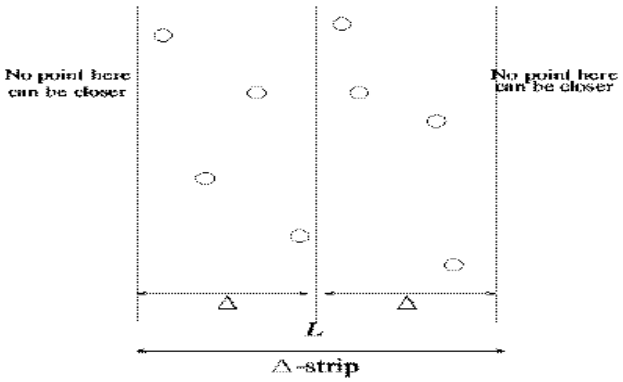


Gambar 4.3 Daerah arsiran titik yang berada di daerah sekitar garis vertikal *L*

Oleh karena itu, tahap COMBINE diimplementasikan sebagai berikut:

a. Temukan semua titik di P_{Left} yang memiliki absis *x* paling sedikit $x_{n/2} - delta$.

b. Temukan semua titik di P_{Right} yang memiliki absis *x* paling banyak $x_{n/2} + delta$.



Gambar 4.4 Daerah arsiran titik yang berada di daerah sekitar garis vertikal *L*

- Sebut semua titik-titik yang ditemukan pada langkah (i) dan (ii) tersebut sebagai himpunanP_{strip} yang berisi *s* buah titik. Urut titik-titik tersebut dalam urutan absis *y* yang menaik. Misalkan q_1, q_2 , ..., q_s menyatakan hasil pengurutan.
- Langkah COMBINE:

```
   for j←i+1 to s do
      exit when (|qi.x – qj.x | > Delta or  |qi.y – qj.y | > Delta
      if jarak (qi, qj) < Delta then
         Delta ← jarak(qi, qj) { dihitung dengan rumus Euclidean }
      endif
   endfor
endfor
```

- Kompleksitas algoritma menghitung jarak terdekat sepasang titik dengan algoritma *Divide and Conquer* :

$$T(n) = \begin{cases} 2T(n/2) + cn & , n > 2 \\ a & , n = 2 \end{cases}$$

Solusi dari persamaan di atas adalah $T(n) = O(n \log n)$.

Contoh 7.

Algoritma Pengurutan dengan Metode *Divide and Conquer*

A	4	12	3	9	1	21	5	2

Dua pendekatan (*approach*) pengurutan:

1. Mudah membagi, sulit menggabung (*easy split/hard join*)

   Tabel *A* dibagidua berdasarkan posisi elemen:

Divide:	*A*1	4	12	3	9	*A*2	1	21	5	2
Sort:	*A*1	3	4	9	12	*A*2	1	2	5	21
Combine:	*A*1	1	2	3	4	5	9	12	21	

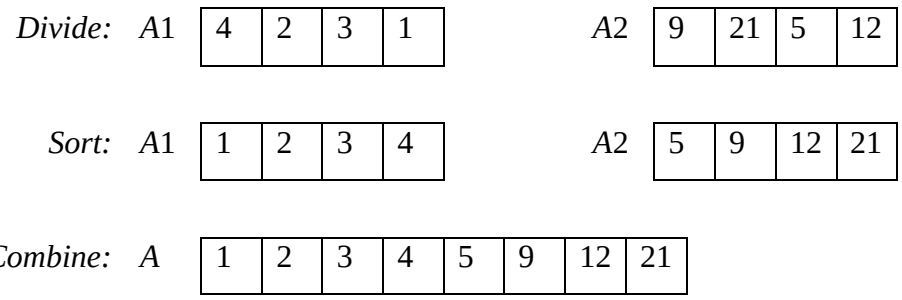
Algoritma pengurutan yang termasuk jenis ini:

a. urut-gabung (*Merge Sort*)

b. urut-sisip (*Insertion Sort*)

2. Sulit membagi, mudah menggabung *(hard split/easy join)*

Tabel *A* dibagidua berdasarkan nilai elemennya. Misalkan elemen-elemen *A*1 ≤ elemen-elemen *A*2.



Algoritma pengurutan yang termasuk jenis ini:

a. urut-cepat *(Quick Sort)*
b. urut-seleksi *(Selection Sort)*

Algoritmanya dapat ditulis seperti berikut ini:

```
procedure Sort(input/output A : TabelInt, input n : integer)
{ Mengurutkan tabel A dengan metode Divide and Conquer
  Masukan: Tabel A dengan n elemen
  Keluaran: Tabel A yang terurut
}
Algoritma:
   if Ukuran(A) > 1 then
      Bagi A menjadi dua bagian, A1 dan A2, masing-masing berukuran n1
      dan n2 (n = n1 + n2)
      Sort(A1, n1)   { urut bagian kiri yang berukuran n1 elemen }
      Sort(A2, n2)   { urut bagian kanan yang berukuran n2 elemen }
      Combine(A1, A2, A)  { gabung hasil pengurutan bagian kiri dan
                            bagian kanan }
   end
```

4.2 *MERGE SORT*

Merge sort dapat dikategorikan sebagai algoritma *divide-and-conquer* yang ide dsarnya diperoleh dari memecah daftar atau array dari beberapa bilangan yang diberikan menjadi beberapa sub-daftar sampai setiap sublist terdiri dari elemen tunggal dan

kemudian menggabungkan sublists dengan cara yang menghasilkan daftar yang diurutkan.

Ide:

a. Bagilah daftar yang tidak disortir ke dalam N sublists, masing-masing berisi elemen.
b. Ambil pasangan yang berdekatan dari dua daftar tunggal dan gabungkan mereka untuk membentuk daftar 2 elemen. N sekarang akan dikonversi menjadi daftar N/2 ukuran 2.
c. Ulangi proses ini sampai satu daftar tunggal yang diurutkan diperoleh.

Sementara membandingkan dua sublists untuk digabung, elemen pertama dari kedua daftar dipertimbangkan. Saat mengurutkan dalam urutan menaik, elemen yang nilainya lebih rendah menjadi elemen baru dari daftar yang diurutkan. Prosedur ini diulangi sampai kedua sublists yang lebih kecil kosong dan sublist gabungan yang baru terdiri dari semua elemen dari kedua sublists.

Algoritma:

1. Untuk kasus $n = 1$, maka tabel *A* sudah terurut dengan sendirinya (langkah SOLVE).
2. Untuk kasus $n > 1$, maka
   (a) DIVIDE: bagi tabel *A* menjadi dua bagian, bagian kiri dan bagian kanan, masing-masing bagian berukuran $n/2$ elemen.
   (b) CONQUER: Secara rekursif, terapkan algoritma *D-and-C* pada masing-masing bagian.
   (c) MERGE: gabung hasil pengurutan kedua bagian sehingga diperoleh tabel *A* yang terurut.

Karena yang digunakan divide-and-conquer untuk mengurutkan, maka perlu diputuskan seperti apa bentuk sub-masalah nantinya. Masalah lengkapnya adalah mengurutkan seluruh array. Katakanlah subproblem adalah menyortir subarray. Khususnya, akan diganggap subproblem sebagai pengurutan subarray mulai dari index p dan melalui index r. Akan lebih mudah jika notasi untuk subarray, jadi katakanlah array [p…r] menunjukkan subarray array ini. Untuk array n elemen, maka dapat dikatakan bahwa masalah aslinya adalah mengurutkan array [0…n-1].

Berikut ini adalah langkha merer sort menggunakan algoritma *Divide-and-Conquer*:

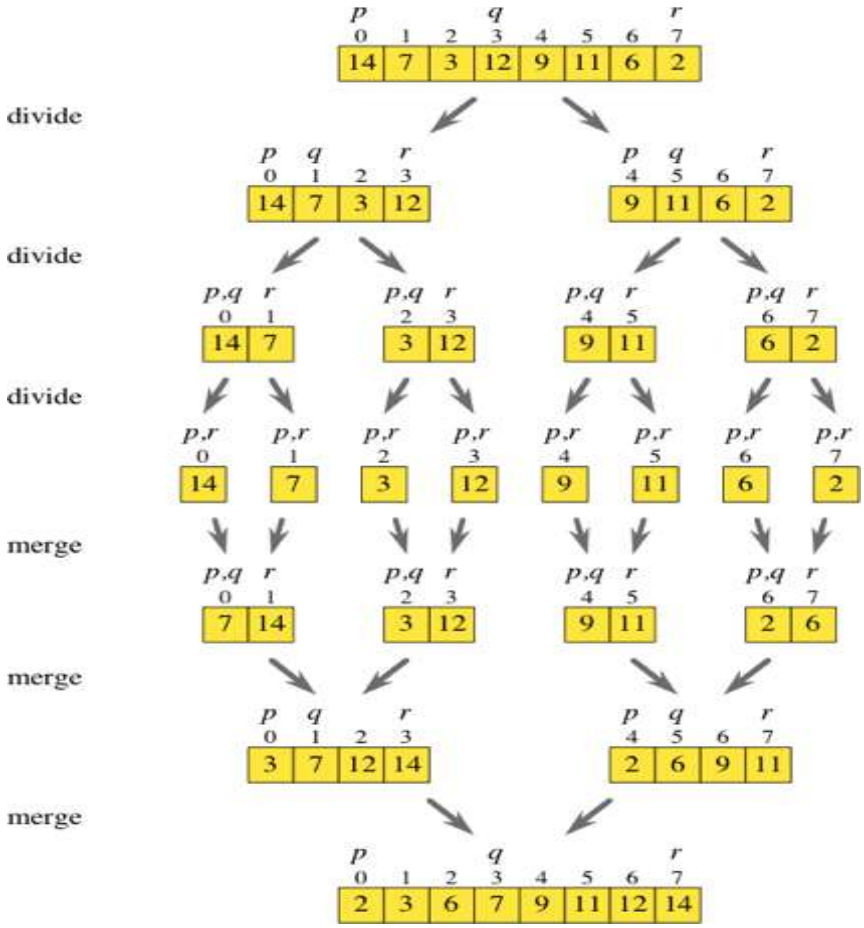
1. **Devide** : dengan menemukan nomor q dari posisi tengah antara p dan r. Lakukan langkah ini dengan cara yang sama ditemukan titik tengah dalam pencarian biner: tambahkan p dan r, bagi dengan 2, dan bulatkan.
2. **Conquer** : dengan menyortir secara rekursif subarrays di masing-masing dari dua subproblem yang telah dibuat padah langkah divide, yaitu, melalui pengurutan secara rekursif subarray array [p…q] dan pengurutan secara rekursif subarray array [q + 1...r].
3. **Combine** : dengan menggabungkan dua sub-array yang diurutkan kembali ke dalam bentuk tunggal elemen yang telah diurutkan dari subarray array [p..r].

Kita membutuhkan kasus dasar. Kasus dsarnyanya adalah subarray yang mengandung kurang dari dua elemen, yaitu, ketika $p \geq r$, karena subarray tanpa elemen atau hanya satu elemen sudah diurutkan sebelumnya. Sehingga algoritma *Divide-and-Conquer* digunakan ketika $p < r$.

Untuk lebih memahaminya berikut adalah contoh dari implementasi merger sort. Jika diketahui array [14, 7, 3, 12, 9, 11, 6, 2], maka subarray pertama sebenarnya adalah array penuh, array [0..7] (p = 0 dan r =, 7). Langkah-langkah penyelesaiannya adalah :

a. Untuk langkah divide, kita menghitung q = 3.
b. Langkah conquer dengan membagi kelompok urutan menjadi 2 subarrays yaitu array [0..3], yang terdiri dari [14, 7, 3, 12], dan array [4..7], yang berisi [9, 11, 6, 2]. Ketika kita kembali ke langkah conquer, masing-masing dari dua subarrays diurutkansehingga array [0..3] berisi [3, 7, 12, 14] dan array [4..7] berisi [2, 6, 9, 11], sehingga array penuh adalah [3, 7, 12, 14, 2, 6, 9, 11].
c. Akhirnya, langkah combine melalui menggabungkan dua sub-array yang diurutkan di tahap pertama dan tahap kedua, menghasilkan array akhir yang telah diurutkan [2, 3, 6, 7, 9, 11, 12, 14].

Bagaimana subarray dari array [0..3] diurutkan? Cara yang sama. Ini memiliki lebih dari dua elemen, dan jadi itu bukan kasus dasar. Dengan p 0 dan r = 3, hitung q = 1, susunan pengurutan secara rekursif [0..1] ([14, 7]) dan larik [2..3] ([3, 12]), menghasilkan array [0..3] yang berisi [7, 14, 3, 12], dan menggabungkan paruh pertama dengan paruh kedua, menghasilkan [3, 7, 12, 14].



Gambar 4.5 ilustrasi bagaimana algoritma merger sort bekerja

Bagaimana array subarray [0..1] diurutkan? Dengan 0 dan r = 1, hitung q=0, susunan array rekursif [0..0] ([14]) dan array [1] ..1] ([7]), menghasilkan array [0..1] masih mengandung [14, 7], dan menggabungkan paruh pertama dengan paruh kedua, menghasilkan [7, 14].

Subarrays Array [0..0] dan array [1..1] adalah kasus dasar, karena masing-masing berisi kurang dari dua elemen. Ilustrasi bagaimana algoritma merger sort bekerja dapat dilihat pada gambar 4.5 :

```
procedure MergeSort(input/output A : TabelInt, input i, j : integer
{ Mengurutkan tabel A[i..j] dengan algoritma Merge Sort
  Masukan: Tabel A dengan n elemen
  Keluaran: Tabel A yang terurut
}
Deklarasi:
   k : integer
Algoritma:
   if i < j then           { Ukuran(A)> 1}
     k←(i+j) div 2
     MergeSort(A, i, k)
```

```
    MergeSort(A, k+1, j)
    Merge(A, i, k, j)
  endif
```

Pseudo-code prosedur *Merge*:

```
procedure Merge(input/output A : TabelInt, input kiri,tengah,kanan : integer)
{ Menggabung tabel A[kiri..tengah] dan tabel A[tengah+1..kanan]
  menjadi tabel A[kiri..kanan] yang terurut menaik.
  Masukan: A[kiri..tengah] dan tabel A[tengah+1..kanan] yang sudah terurut
   menaik.
  Keluaran: A[kiri..kanan] yang terurut menaik.
}
Deklarasi
   B : TabelInt
   i, kidal1, kidal2 : integer
Algoritma:
   kidal1←kiri           { A[kiri .. tengah] }
   kidal2←tengah + 1     { A[tengah+1 .. kanan] }
   i←kiri
   while (kidal1 ≤ tengah) and (kidal2 ≤ kanan) do
       if Akidal1 ≤ Akidal2 then
           Bi←Akidal1
          kidal1←kidal1 + 1
      else
           Bi←Akidal2
          kidal2←kidal2 + 1
      endif
      i←i + 1
    endwhile
   { kidal1 > tengah or kidal2 > kanan }
   { salin sisa A bagian kiri ke B, jika ada }
   while (kidal1 ≤ tengah) do
       Bi←Akidal1
      kidal1←kidal1 + 1
      i←i + 1
    endwhile
```

```
{ kidal1 > tengah }
{ salin sisa A bagian kanan ke B, jika ada }
while (kidal2 ≤ kanan) do
   Bi←Akidal2
   kidal2←kidal2 + 1
   i←i + 1
endwhile
{ kidal2 > kanan }
{ salin kembali elemen-elemen tabel B ke A }
for i←kiri to kanan do
   Ai←Bi
 endfor
{ diperoleh tabel A yang terurut membesar }
```

Kompleksitas waktu:

Asumsi: $n = 2^k$

$T(n)$ = jumlah perbandingan pada pengurutan dua buah upatabel + jumlah perbandingan pada prosedur *Merge*

$$T(n) = \begin{cases} a & , n = 1 \\ 2T(n/2) + cn & , n > 1 \end{cases}$$

Penyelesaian persaman rekurens:

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/4) + cn/2) + cn = 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn = 8T(n/8) + 3cn \\ &= \ ... \\ &= 2^k \, T(n/2^k) + kcn \end{aligned}$$

Berhenti jika ukuran tabel terkecil, n = 1:

$$n/2^k = 1 \rightarrow k = {}^2\log n$$

sehingga

$$\begin{aligned} T(n) &= nT(1) + cn \, {}^2\log n \\ &= na + cn \ {}^2\log n \\ &= O(n \, {}^2\log n) \end{aligned}$$

Salah satu implementasi algoritma atau pseudocode dari merger sort ke dalam program dengan bahasa C seperti berikut ini :

```
#include <stdio.h>
// function to sort the subsection a[i .. j] of the array a[]
void merge_sort(int i, int j, int a[], int aux[]) {
    if (j <= i) {
        return;     // the subsection is empty or a single element
    }
    int mid = (i + j) / 2;
    // left sub-array is a[i .. mid]
    // right sub-array is a[mid + 1 .. j]
    merge_sort(i, mid, a, aux); // sort the left sub-array recursively
    merge_sort(mid + 1, j, a, aux); // sort the right sub-array
                                       recursively
    int pointer_left = i; // pointer_left points to the beginning of
                             the left sub-array
    int pointer_right = mid + 1;        // pointer_right points to the
                                   beginning of the right sub-array
    int k;  // k is the loop counter
    // we loop from i to j to fill each element of the final merged
        array
    for (k = i; k <= j; k++) {
        if (pointer_left == mid + 1)   // left pointer has reached the
                                          limit
            aux[k] = a[pointer_right];
            pointer_right++;
        } else if (pointer_right == j + 1) {   // right pointer has
                                                  reached the limit
            aux[k] = a[pointer_left];
            pointer_left++;
        } else if (a[pointer_left] < a[pointer_right]) {  // pointer
                                       left points to smaller element
            aux[k] = a[pointer_left];
            pointer_left++;
        } else {        // pointer right points to smaller element
            aux[k] = a[pointer_right];
            pointer_right++;
        }
    }
    for (k = i; k <= j; k++) {  // copy the elements from aux[] to a[]
        a[k] = aux[k];
    }
}
int main() {
  int a[100], aux[100], n, i, d, swap;
  printf("Enter number of elements in the array:\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (i = 0; i < n; i++)
```

```
    scanf("%d", &a[i]);
  merge_sort(0, n - 1, a, aux);
  printf("Printing the sorted array:\n");
  for (i = 0; i < n; i++)
     printf("%d\n", a[i]);
  return 0;
}
```

4.3 *INSERTION SORT*

Analogi kerja algoritma Insertion sort sama seperti kita mengurutkan kartu di tangan kita dalam permainan kartu. Jika diasumsikan bahwa kartu pertama sudah disortir maka selanjutnya kita memilih kartu yang tidak disortir. Jika kartu yang tidak disortir lebih besar dari kartu yang ada, selanjutnya kartu itu diletakkan di sebelah kanan sebaliknya, di sebelah kiri. Dengan cara yang sama, kartu yang tidak disortir lainnya diambil dan diletakkan di tempat yang tepat. Pendekatan serupa digunakan oleh jenis penyisipan..

Insertion Sort didasarkan pada gagasan bahwa satu elemen dari elemen input dikonsumsi di setiap iterasi untuk menemukan posisi yang benar yaitu, posisi di mana ia berada dalam array yang diurutkan atau dengan kata lain bahwa Insertion sort merupakanh algoritma penyortiran yang menempatkan elemen yang tidak disortir di tempat yang sesuai di setiap iterasi

Iterasi elemen input dengan menumbuhkan array diurutkan di setiap iterasi. Ini dilakukan dengan membandingkan elemen saat ini dengan nilai terbesar dalam array yang diurutkan. Jika elemen saat ini lebih besar, maka elemen tersebut akan berpindah tempatnya ke posisi elemen di tempatnya dan pindah ke elemen berikutnya yang lain ia menemukan posisi yang benar dalam array yang diurutkan dan memindahkannya ke posisi itu. Ini dilakukan dengan menggeser semua elemen, yang lebih besar dari elemen saat ini, dalam array yang diurutkan ke satu posisi di depan

Insertion sort adalah algoritma penyortiran di mana array diurutkan dengan mengambil satu elemen pada suatu waktu. Prinsip di balik penyisipan sortir adalah mengambil satu elemen, beralih melalui array yang diurutkan & temukan posisinya yang benar dalam array yang diurutkan. Langkah-langkah pengurutan elemen dengan menggunaka algoritma insertion sort adalah :

i. Jika elemen adalah yang pertama, sudah diurutkan.
ii. Pindah ke elemen berikutnya

iii. Bandingkan elemen saat ini dengan semua elemen dalam array yang diurutkan

iv. Jika elemen dalam array yang diurutkan lebih kecil dari elemen saat ini, beralih ke elemen berikutnya. Jika tidak, geser semua elemen yang lebih besar dalam array dengan satu posisi ke kanan

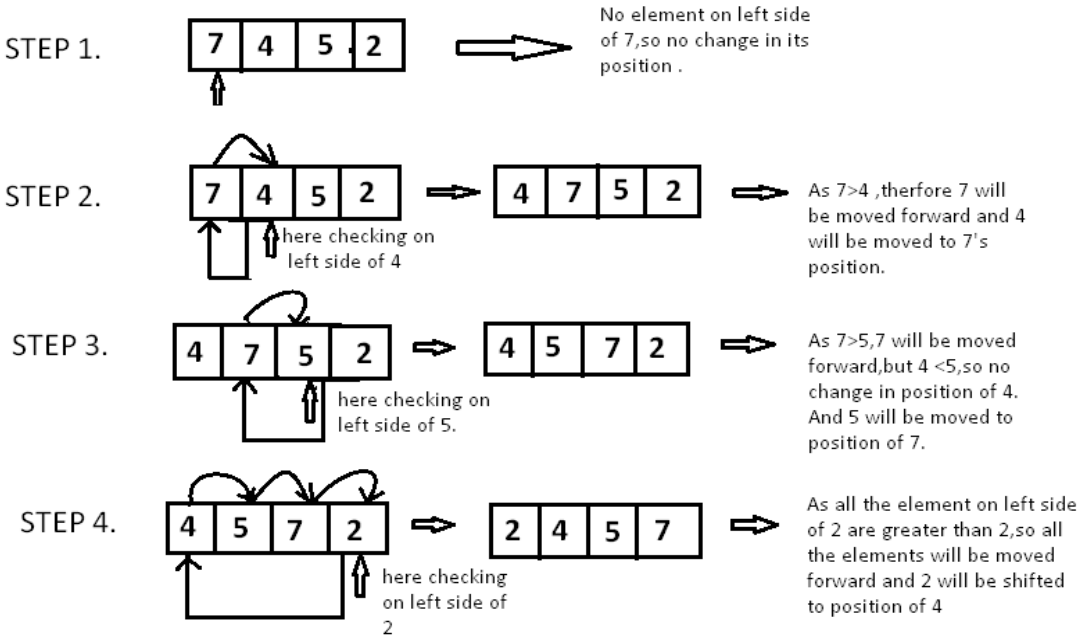
v. Masukkan nilai pada posisi yang benar

vi. Ulangi sampai daftar lengkap diurutkan

Dengan ilustrasi gambar dapat dilihat seperti berikut ini :

Misalkan diambil nilai yang hendak diurutkan adalah [7, 4, 5, 2] maka dapat dilihat proses pengurutan dengan insertion seperti pada gambar 4.6.

Gambar 4.6 dapat dijelaskan bahwa 7 adalah elemen pertama, tidak memiliki elemen lain untuk dibandingkan, maka ia tetap pada posisinya. Sekarang ketika bergerak ke posisi berikutnya ada bilangan 4, 7 adalah elemen terbesar dalam daftar yang disortir dan lebih besar dari 4. Jadi, 4 pindah ke posisi yang benar yaitu sebelumnya diposisi 7. Demikian pula dengan 5, karena 7 (elemen terbesar dalam daftar diurutkan) lebih besar dari 5, maka 5 akan dipindah ke posisi yang benar. Akhirnya untuk, semua elemen di sisi kiri (daftar yang disortir) 2 dipindahkan satu posisi ke depan karena semua lebih besar dari 2 dan kemudian ditempatkan di posisi pertama. Akhirnya, array yang diberikan akan menghasilkan array yang diurutkan.



Gambar 4.6 Ilustrasi proses insertion sort untuk mengurutkan [7, 4, 5, 7]
Sumber : https://www.hackerearth.com/

```
procedure InsertionSort(input/output A : TabelInt,
                        input i, j : integer)
{ Mengurutkan tabel A[i..j] dengan algoritma Insertion Sort.
  Masukan: Tabel A dengan n elemen
  Keluaran: Tabel A yang terurut
}
Deklarasi:
   k : integer

Algoritma:
   if i < j then          { Ukuran(A)> 1}
     k←i
     InsertionSort(A, i, k)
     InsertionSort(A, k+1, j)
     Merge(A, i, k, j)
   endif
```

Perbaikan:

```
procedure InsertionSort(input/output A : TabelInt,
                        input i, j : integer)
{ Mengurutkan tabel A[i..j] dengan algoritma Insertion Sort.
  Masukan: Tabel A dengan n elemen
  Keluaran: Tabel A yang terurut
}
Deklarasi:
   k : integer

Algoritma:
   if i < j then          { Ukuran(A)> 1}
     k←i
     Insertion(A, k+1, j)
     Merge(A, i, k, j)
   endif
```

Prosedur *Merge* dapat diganti dengan prosedur penyisipan sebuah elemen pada tabel yang sudah terurut (lihat algoritma *Insertion Sort* versi iteratif).

Contoh 9. Misalkan tabel *A* berisi elemen-elemen berikut:

4	12	23	9	21	1	5	2

DIVIDE, CONQUER, dan SOLVE::

4	12	3	9	1	21	5	2
4	12	3	9	1	21	5	2
4	12	3	9	1	21	5	2
4	12	3	9	1	21	5	2

	4	12	3	9	1	21	5	2
	4	12	3	9	1	21	5	2
	4	12	3	9	1	21	5	2
	4	12	3	9	1	21	5	2
	4	12	3	9	1	21	5	2
MERGE:	4	12	3	9	1	21	5	2
	3	4	12	9	1	21	5	2
	3	4	9	12	1	21	5	2
	1	3	4	9	12	21	5	2
	1	3	4	9	12	21	5	2
	1	3	4	5	9	12	21	2
	1	2	3	4	5	9	12	21

Kompleksitas waktu algoritma *Insertion Sort* dihitung berdasarkan jumlah perbandingan elemen-elemen tabel:

$T(n)$ = jumlah perbandingan pada pengurutan bagian kanan yang berukuran $n-1$ elemen + jumlah perbandingan elemen pada prosedur Gabung

$T(n)$ adalah relasi rekurens:

$$T(n) = \begin{cases} a & , n = 1 \\ T(n-1) + cn & , n > 1 \end{cases}$$

Penyelesaian persamaan rekurens:

$$\begin{aligned} T(n) &= cn + T(n-1) \\ &= cn + \{ c \cdot (n-1) + T(n-2) \} \\ &= cn + c(n-1) + \{ c \cdot (n-2) + T(n-3) \} \\ &= cn + c \cdot (n-1) + c \cdot (n-2) + \{c(n-3) + T(n-4) \} \\ &= ... \\ &= cn + c \cdot (n-1) + c(n-2) + c(n-3) + ... + c2 + T(1) \\ &= c\{ n + (n-1) + (n-2) + (n-3) + ... + 2 \} + a \\ &= c\{ (n-1)(n+2)/2 \} + a \\ &= cn^2/2 + cn/2 + (a - c) \\ &= O(n^2) \end{aligned}$$

Dibandingkan *Merge Sort* dengan kompleksitas $O(n \log n)$, *Insertion Sort* yang kompkleksitasnya $O(n^2)$ tidak lebih baik daripada *Merge Sort*.

Selanjutnya berikut ini adalah salah satu implementasi algoritma insertion sort dengan bahasa pemrograman C.

```
void insertion_sort ( int A[ ] , int n)
{
     for( int i = 0 ;i < n ; i++ ) {
    /*storing current element whose left side is checked for its
             correct position .*/

      int temp = A[ i ];
      int j = i;

       /* check whether the adjacent element in left side is greater or
            less than the current element. */

          while(  j > 0  && temp < A[ j -1]) {

           // moving the left side element to one position forward.
                A[ j ] = A[ j-1];
                j= j - 1;
           }
         // moving current element to its  correct position.
           A[ j ] = temp;
     }
}
```

atau

```
#include <stdio.h>
#define MAX 20

void insertion(int arr[],int n)
{
        int j,k,i;
        /*Insertion sort*/
        for(j=1;j<n;j++)
        {
                k=arr[j]; /*k is to be inserted at proper place*/
                for(i=j-1;i>=0 && k<arr[i];i--)
                {
                        arr[i+1]=arr[i];
                }
                arr[i+1]=k;
        }
}

int main()
{
        int arr[MAX],i,n;
```

```
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for (i = 0; i < n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d", &arr[i]);
        }

        printf("Unsorted list is :\n");
        for (i = 0; i < n; i++)
                printf("%d ", arr[i]);
        printf("\n");

        insertion(arr,n);

        printf("Sorted list is :\n");
        for (i = 0; i < n; i++)
                printf("%d ", arr[i]);
        printf("\n");

        return 0;
}/*End of main()*/
```

4.4 *STRASSEN'S MATRIX MULTIPLICATION ALGORITHM*

Perkalian Matriks adalah salah satu operasi paling mendasar dalam Machine Learning dan mengoptimalkannya adalah kunci untuk beberapa optimasi. Secara umum, mengalikan dua matriks ukuran N X N membutuhkan N^3 operasi.

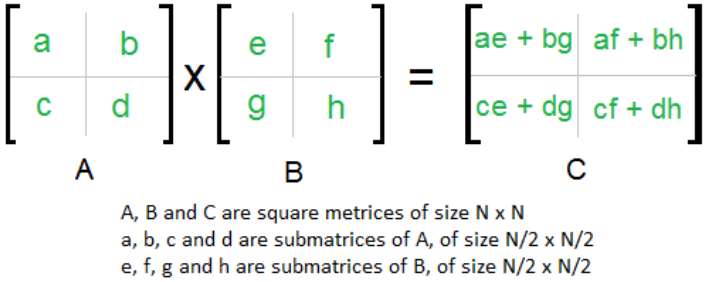
Volker Strassen pertama kali mempublikasikan algoritmanya pada tahun 1969. Ini adalah algoritma pertama yang membuktikan bahwa runtime untuk perkalian matriks tersebut $O(n^3)$ bukanlah yang optimal.dengan menggunakan algoritma *divide and conquer* maka kompleksitas keseluruhan untuk perkalian dua matriks berkurang.

Ilustrasi tersebut dapat kita lihat berikut ini : Jika diberikan dua matrikas A dan matriks B, maka dengan menggunakan metode naive maka algoritmanya diperoleh seperti berikut ini :

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
```

```
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

Kompleksitas dari algoritma diatas adalah $O(n^3)$.



Gambar 4.7 Hasil perkalian matriks dengan metode Strassen

Upaya untuk memperbaiki kompleksitas perkalian dua matriks dilakukan oleh Strassen dengan menggunakan algoritma *divide and conquer* dan langkah-langkahnya adalah :

a. Bagilah matriks A dan B dalam 4 sub-matriks ukuran N / 2 x N / 2 seperti yang ditunjukkan pada diagram di bawah ini.
b. Hitung nilai-nilai berikut secara rekursif. ae + bg, af + bh, ce + dg dan cf + dh.

Modelnya seperti pada gambar 4.5 berikut ini.

Dalam metode di atas, dilakukan 8 perkalian untuk matriks ukuran N / 2 x N / 2 dan 4 penambahan. Penambahan dua matriks membutuhkan waktu O (N2). Jadi kompleksitas waktu dapat ditulis : $T(N) = 8T(N/2) + O(N^2)$

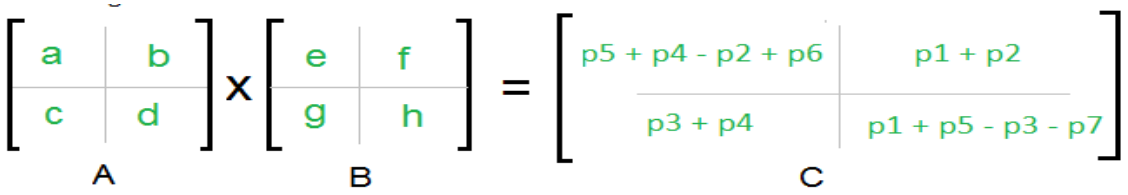
Dalam metode *divide and conquer* di atas, komponen utama untuk kompleksitas waktu tinggi adalah 8 panggilan rekursif. Gagasan metode Strassen adalah untuk mengurangi jumlah panggilan rekursif ke 7. Metode Strassen mirip dengan metode *divide and conquer* di atas dalam arti bahwa metode ini juga membagi matriks ke sub-matriks ukuran N / 2 x N / 2 sebagai ditunjukkan dalam diagram di atas, tetapi dalam metode Strassen, empat sub-matriks hasil dihitung menggunakan rumus berikut:

P1 = A(F −H) P2 = (A+B)H

P3 = (C +D)E P4 = D(G−E)

P5 = (A+D)(E +H) P6 = (B−D)(G+H) P7 = (A−C)(E +F)

Matriks A x B dapat dihitung menggunakan delapan perkalian di atas, berikut ini adalah nilai empat sub-matriks dengan notasi hasil C.



Gambar 4.8 nilai empat sub-matriks dengan notasi hasil C

Kompleksitas waktu untuk Metode Strassen's dengan penambahan dan perkalian dua matriks adalah $O(N^2)$ kali, sehingga kompleksitas waktu nya dapat ditulis $T(N) = 7T(N/2) + O(N^2)$.

Secara umum algoritma Strassen tidak disarankan untuk aplikasi praktis karena alasan berikut.

4. Konstanta yang digunakan dalam metode Strassen tinggi dan untuk aplikasi biasa metode Naif bekerja lebih baik.
5. Untuk matriks *sparse*, ada metode yang lebih baik yang dirancang khusus untuk mereka.
6. Sub matriks dalam rekursi membutuhkan ruang ekstra.
7. Karena ketepatan aritmatika komputer terbatas pada nilai-nilai noninteger, kesalahan yang lebih besar terakumulasi dalam algoritma Strassen daripada di Metode Naive (Cormen dkk).

Algoritma perkalian matriks Stressen dalam bentuk pseudocode dapat ditulis sebagai berikut :

Algorithm Strassen(n, a, b, d)

```
begin
        If n = threshold then compute
                C = a * b is a conventional matrix.
        Else
                Partition a into four sub matrices  a11, a12, a21, a22.
                Partition b into four sub matrices b11, b12, b21, b22.
                Strassen ( n/2, a11 + a22, b11 + b22, d1)
                Strassen ( n/2, a21 + a22, b11, d2)
                Strassen ( n/2, a11, b12 – b22, d3)
                Strassen ( n/2, a22, b21 – b11, d4)
                Strassen ( n/2, a11 + a12, b22, d5)
                Strassen (n/2, a21 – a11, b11 + b22, d6)
                Strassen (n/2, a12 – a22, b21 + b22, d7)
```

```
                C = d1+d4-d5+d7     d3+d5
                d2+d4           d1+d3-d2-d6

        end if

        return (C)
end.
```

Implementasi pseudo code tersebut dalam bahasa C adalah :

```
#include <stdio.h>
int main()
{
        int a[2][2],b[2][2],c[2][2],i,j;
        int m1,m2,m3,m4,m5,m6,m7;

        printf("Enter the 4 elements of first matrix: ");
        for(i=0;i<2;i++)
                for(j=0;j<2;j++)
                        scanf("%d",&a[i][j]);

        printf("Enter the 4 elements of second matrix: ");
        for(i=0;i<2;i++)
                for(j=0;j<2;j++)
                        scanf("%d",&b[i][j]);

        printf("\nThe first matrix is\n");
        for(i=0;i<2;i++)
        {
                printf("\n");
                for(j=0;j<2;j++)
                        printf("%d\t",a[i][j]);
        }
        printf("\nThe second matrix is\n");
        for(i=0;i<2;i++)
        {
                printf("\n");
                for(j=0;j<2;j++)
                        printf("%d\t",b[i][j]);
        }
        m1= (a[0][0] + a[1][1])*(b[0][0]+b[1][1]);
        m2= (a[1][0]+a[1][1])*b[0][0];
        m3= a[0][0]*(b[0][1]-b[1][1]);
        m4= a[1][1]*(b[1][0]-b[0][0]);
        m5= (a[0][0]+a[0][1])*b[1][1];
        m6= (a[1][0]-a[0][0])*(b[0][0]+b[0][1]);
        m7= (a[0][1]-a[1][1])*(b[1][0]+b[1][1]);

        c[0][0]=m1+m4-m5+m7;
        c[0][1]=m3+m5;
        c[1][0]=m2+m4;
```

```
        c[1][1]=m1-m2+m3+m6;

        printf("\nAfter multiplication using \n");
        for(i=0;i<2;i++)
        {
                printf("\n");
                for(j=0;j<2;j++)
                        printf("%d\t",c[i][j]);
        }
        return 0;}
```

4.5 RANGKUMAN

1. *Divide and conquer* merupakan salah satu teknik desain algoritma yang membagi masalah menjadi satu atau lebih banyak sub-masalah dengan ukuran yang kira-kira sama. *Divide and Conquer* dulunya adalah strategi militer yang dikenal dengan nama *divide ut imperes.* Sekarang strategi tersebut menjadi strategi fundamental di dalam ilmu komputer dengan nama *Divide and Conquer*.
2. Merge sort dapat dikategorikan sebagai algoritma *divide-and-conquer* yang ide dsarnya diperoleh dari memecah daftar atau array dari beberapa bilangan yang diberikan menjadi beberapa sub-daftar sampai setiap sublist terdiri dari elemen tunggal dan kemudian menggabungkan sublists dengan cara yang menghasilkan daftar yang diurutkan.
3. ***Insertion Sort*** didasarkan pada gagasan bahwa satu elemen dari elemen input dikonsumsi di setiap iterasi untuk menemukan posisi yang benar yaitu, posisi di mana ia berada dalam array yang diurutkan atau dengan kata lain bahwa Insertion sort merupakanh algoritma penyortiran yang menempatkan elemen yang tidak disortir di tempat yang sesuai di setiap iterasi

4.6 TEST FORMATIF

1. Anda telah diberi array A yang terdiri dari N integer. Semua elemen dalam array ini dijamin unik. Untuk setiap posisi i dalam array A, Anda perlu menemukan posisi A[i] yang harus adadidalamnya, jika array adalah array yang telah diurutkan. Anda perlu menemukan ini untuk setiap i dan mencetak solusi yang dihasilkan.

   Masukka :

Baris pertama berisi bilangan bulat tunggal N yang menunjukkan ukuran array A. Baris berikutnya berisi bilangan bulat yang dipisahkan dengan N yang menunjukkan elemen-elemen dari array A.

Keluaran:

Cetak N spasi yang dipisahkan bilangan bulat pada satu baris, di mana bilangan bulat I menunjukkan posisi jika array ini diurutkan.

Kendala:

$1 \leq N \leq 100$

$1 \leq A[i] \leq 100$

Tuliskan program untuk kasus tersebut.

2. Gunakan algoritma MERGERSORT untuk mengurutkan data berikut ini :

12	4	6	8	1	2	20

3. Buktikan bahwa Algorithm mergesort memiliki kompleksitas $\Theta(n)$.
4. (*Quick Sort*) Tuliskan langkah-langkah pengurutan 7 buah elemen di bawah ini dengan algoritma *Quick Sort* sehingga seluruh elemen terurut menaik. Elemen yang dijadikan *pivot* adalah elemen pertama:

   12, 6, 10, 25, 1, 7, 14
5. (*Merge Sort*) Algoritma pengurutan *Merge Sort* selalu membagi tabel (*array*) yang berukuran *n* menjadi dua bagian yang berukuran sama (*n*/2). Misalkan algoritma tersebut dimodifikasi sehingga membagi tabel menjadi empat bagian yang masing-masing berukuran *n*/4.

   Jika pada algoritma *Merge Sort* yang biasa jumlah operasi perbandingan elemen data adalah $T(n) = an + cn\ ^2\log(n)$, *a* dan *c* adalah konstanta, maka tentukan pula jumlah operasi perbandingan pada algoritma *Merge Sort* yang telah dimodifikasi.

   Apakah jumlah operasi perbandingan elemen-elemen yang terjadi lebih sedikit atau lebih banyak dibandingkan dengan algoritma Mergesort biasa?
6. (*Sum of Big Integers Problem*) Misalkan *u* dan *v* adalah dua buah bilangan bulat besar yang panjangnya *n* angka (*digit*):

   $u = u_1u_2\ldots u_n$ dan $v = v_1v_2\ldots v_n$

   (keterangan: u_i dan v_i adalah angka $\in \{0, 1, 2, \ldots, 9\}$)

Kita ingin menjumlahkan dua buah bilangan bulat tersebut ($u + v$).

(a) Bagaimana algoritma penjumlahan dengan algoritma *Brute Force*? Berapa kompleksitas algoritmanya?

(b) Jika penjumlahan dilakukan dengan algoritma *Divide and Conquer*, jelaskan caranya? Apakah kompleksitas algoritmanya lebih baik daripada algoritma *Brute Force*?

7. Misalkan pada algoritma mergesort tabel dibagi menjadi tiga bagian masing-masing berukuran n/3 elemen, urutkan masing-masing bagian, dan gabung hasil pengurutan tiga bagian tersebut.
   a. Gambarkan proses pengurutan dengan mergesortyang disebutkan di atas untuk data integerberikut: 12, 8, 15, 9, 10, 5, 2, 1, 14, 20, 13, 8, 11, 7, 4, 9
   b. Nyatakan kompleksitas waktu algoritma mergesort di atas dalam notasi rekurens, lalu selesaikan relasi tersebut untuk memperoleh notasi O-nya.
8. Di sebuah perguruan tinggi BHMN, seperti ITB, pemilihan rektor dilakukan melalui pemungutan suara oleh n orang anggota MWA (Majelis Wali Amanat). Tiap anggota MWA hanya memilih 1 orang kandidat dari beberapa orang kandidat rektor yang tersedia (bisa terdapat lebih dari 2 orang kandidat). Misalkan suara-suara (votes), yaitu nama kandidat yang dipilih oleh n orang anggota MWA, dinyatakan sebagai elemen-elemen tabel (contoh: B, A, A, B, C, B).Rancanglah algoritma divide and conquer untuk menentukan kandidat yang memperoleh suara paling banyak (mayoritas). (Petunjuk: asumsikan n genap, dan bagilah tabel menjadi dua bagain yang masing-masing berukuran n/2 elemen. Catatlah bahwa seorang kandidat tidak mendapat mayoritas suara tanpa memperoleh mayoritas suara di paling sedikit salah satu dari dua bagian tabel)

9. Aplikasikan algoritma Merge Sort dan Quick Sort pada kumpulan huruf pada tabel berikut ini menjadi terurut abjad.

S	E	M	A	N	G	A	T

   a. Gambarkan pohon pemanggilan rekursif untuk masing-masing algoritma tersebut. Untuk Quick Sort, pivot yang digunakan adalah elemen pertama tabel.

Pada setiap langkah perjelas posisi yang ditukar, dan posisi pada tabel di mana partisi dilakukan.

b. Untuk setiap algoritma, tentukan berapa kali pemanggilan rekursif dilakukan dalam rangka mendapatkan kumpulan huruf yang sudah terurut abjad.

10. Terdapat sebuah larik *A* yang berisi bilangan bulat positif dan negatif sebagai berikut. {2, -4, 1, 9, -6, 7, -3}

    a. Rancanglah pendekatan *Divide and Conquer* untuk menemukan bagian upa-larik yang sekuensnya memiliki jumlah maksimum (yang berlaku untuk semua larik dan tidak hanya pada contoh larik *A* di atas). Misal *i* adalah indeks awal dan j adalah indeks akhir suatu sekuens upa-larik *A*, maka dicari$S_{ij} = \sum_{k=i}^{j} A[k]$ bernilai maksimal. Contoh untuk larik *A* yang indeksnya dimulai dari 1, maka upa-larik yang menghasilkan sekuens nilai maksimal adalah *i*=3 dan *j*=6, yang jumlah elemennya adalah 11. Rancangan tidak perlu sampai *pseudo code,* cukup penjelasan langkah-langkah.

    b. Tentukan waktu yang diperlukan dari pendekatan yang anda usulkan dalam notasi *T*(*n*), dengan menentukan basis dan rekurensnya.

    c. Tentukan kompleksitas dari pendekatan yang anda usulkan dalam notasi Big O, dengan memanfaatkan Teorema Master (perjelas nilai *a*, *b*, dan *d* sesuai jawaban anda di butir

BAB V
ALGORITMA *GREEDY*

a) Deskripsi Singkat

Greedy adalah paradigma algoritmik yang membangun solusi sepotong demi sepotong, selalu memilih karya berikutnya yang menawarkan manfaat paling jelas dan langsung. Jadi masalah di mana memilih optimal secara lokal juga mengarah ke solusi global yang paling cocok untuk Greedy. Sebagai contoh, pertimbangkan Masalah Fractional Knapsack. Strategi optimal lokal adalah memilih item yang memiliki nilai maksimum vs rasio berat. Strategi ini juga mengarah ke solusi optimal global karena mengizinkan untuk mengambil pecahan dari suatu item. Bab ini berisikan tentang algoritma greedy, algoritma knapsack by greedy, Travelling Salesman Problem. Setelah membaca modul ini diharapkan mahasiswa dapat menerapkan algoritma greedy untuk menyelesaikan permasalahan-permasalahan nyata.

b) Relevansi

Relevansinya adalah melalui metode ceramah, tanya jawab, duskusi dan praktikum. Berkaitan dengan materi pada Bab 5 ini, kepada mahasiswa akan dijelaskan dengan detail materinya untuk kemudian algoritmanya diimplementasikan ke program melalui praktikum menggunakan bahasa pemrograman C, C++, Java atau python atau bahasa pemrograman yang dikuasi mahasiswa. Kesempatan bertanya juga diberikan dan tugas mandiri sebagai pengayaan materi-materi pada bab 5 ini.

c) Capaian Pembelajaran

1. Mahasiswa mampu menyelesaikan masalah-masalah dengan menggunakan algoritma Greedy
2. Mahasiswa dapat menghitung jarak terpendek dengan menggunakan minimum spanning tree berbasis algoritma greedy

5.1 PENDAHULUAN

Filosofi menjadi greedy (serakah) adalah “rabun dekat”. Selalu mencari yang terbaik berikutnya, selalu mengoptimalkan kehadiran, tanpa memperhatikan masa

depan, dan tidak pernah berubah pikiran tentang masa lalu. Paradigma *greedy* ini biasanya diterapkan untuk masalah optimasi. Algoritma *greedy* menjadi metode yang paling populer untuk memecahkan persoalan optimasi.

Algoritma greedy juga merupakan algoritma yang memecahkan masalah langkah per langkah, pada setiap langkah:

1. Mengambil pilihan yang terbaik yang dapat diperoleh pada saat itu tanpa memperhatikan konsekuensi ke depan (prinsip " take what you can get now!")
2. Berharap bahwa dengan memilih optimum lokal pada setiap langkah akan berakhir dengan optimum global.

Persoalan optimasi (*optimization problems*): persoalan yang menuntut pencarian solusi optimum. Persoalan optimasi ada dua macam yaitu maksimasi (*maximization*) dan minimasi (*minimization*)

Solusi optimum (terbaik) adalah solusi yang bernilai minimum atau maksimum dari sekumpulan alternatif solusi yang mungkin. Elemen persoalan optimasi:

1. kendala (*constraints*)
2. fungsi objektif(atau fungsi optiamsi)

Solusi yang memenuhi semua kendala disebut **solusi layak** (*feasible solution*). Solusi layak yang mengoptimumkan fungsi optimasi disebut **solusi optimum**.

Contoh masalah sehari-hari yang menggunakan prinsip *greedy*:

1. Memilih beberapa jenis investasi (penanaman modal)
2. Mencari jalur tersingkat dari Lhokseumawe ke Meulaboh
3. Memilih jurusan di Perguruan Tinggi
4. Bermain kartu remi

Algoritma *greedy* membentuk solusi langkah per langkah (*step by step*), terdapat banyak pilihan yang perlu dieksplorasi pada setiap langkah solusi. Oleh karena itu, pada setiap langkah harus dibuat keputusan yang terbaik dalam menentukan pilihan. Keputusan yang telah diambil pada suatu langkah tidak dapat diubah lagi pada langkah selanjutnya. Pendekatan yang digunakan di dalam algoritma *greedy* adalah membuat pilihan yang "tampaknya" memberikan perolehan terbaik, yaitu dengan membuat pilihan **optimum lokal** (*local optimum*) pada setiap langkah dengan harapan bahwa sisanya mengarah ke solusi **optimum global** (*global optimm*).

Pertanyaannya adalah bagaimana anda memutuskan pilihan mana yang optimal? Jawabannya adalah asumsikan bahwa anda memiliki fungsi objektif yang perlu dioptimalkan (baik dimaksimalkan atau diminimalkan) pada titik tertentu. Algoritma Greedy membuat pilihan "serakah" pada setiap langkah untuk memastikan bahwa fungsi objektif dioptimalkan. Algoritma Greedy hanya memiliki satu kesempatan untuk menghitung solusi optimal sehingga tidak pernah kembali dan membalikkan keputusan.

Algoritma greedy memiliki beberapa kelebihan dan kekurangan diantaranya :

a. Sangat mudah untuk membuat algoritma serakah (atau bahkan beberapa algoritma serakah) untuk suatu masalah.
b. Menganalisis *run time* untuk algoritma greedy umumnya akan lebih mudah daripada teknik lainnya (seperti *divide and conquer*). Untuk teknik *Divide and Conquer*, tidak jelas apakah tekniknya cepat atau lambat. Ini karena pada setiap tingkat rekursi ukuran semakin kecil dan jumlah sub-masalah meningkat.
c. Bagian yang sulit adalah untuk algoritma greedy Anda harus bekerja lebih keras untuk memahami masalah kebenaran. Bahkan dengan algoritma yang benar, sulit untuk membuktikan mengapa itu benar. Membuktikan bahwa algoritma serakah itu benar lebih merupakan seni daripada sains. Ini melibatkan banyak kreativitas.

Pertanyaan berikutnya adalah bagaimana cara anda membuat algoritma greedy, jawabannya adalah kita ambil analogi bahwa kita telah menjadi orang yang sangat sibuk di era milleneal ini. Jika kita punya waktu T untuk melakukan beberapa hal menarik dan ingin melakukan hal-hal maksimum seperti itu. Maka untuk menyelesaikan persoalan tersebut maka diasumsikan anda diberi array A dari bilangan bulat, di mana setiap elemen menunjukkan waktu yang dibutuhkan untuk menyelesaikannya. Anda ingin menghitung jumlah hal maksimum yang dapat Anda lakukan dalam waktu terbatas yang anda miliki. Masalah tersebut dapat dikategorikan ke dalam algoritma Greedy sederhana, dimana dalam setiap iterasi, anda harus dengan "rakus" memilih hal-hal yang akan menghabiskan waktu minimum untuk diselesaikan sambil mempertahankan dua variabel currentTime dan numberOfings. Untuk menyelesaikan perhitungan, Anda harus:

1. Sortir array A dalam urutan yang tidak menurun.
2. Pilih setiap item yang harus dilakukan satu-per-satu.

3. Tambahkan waktu yang diperlukan untuk menyelesaikan item agenda ke currentTime.
4. Tambahkan satu ke numberOfings.
5. Ulangi ini selama currentTime kurang dari atau sama dengan T.

Sebagai ilustrasinya dapat dilihat pada kasus berikut :

Contoh 1.

Misal : **A = {5, 3, 4, 2, 1}** dan **T = 6**, Setelah sorting, **A = {1, 2, 3, 4, 5}**

Setelah iterasi pertama:

- **currentTime** = 1
- **numberOfThings** = 1

Setelah iterasi kedua :

- **currentTime** is 1 + 2 = 3
- **numberOfThings** = 2

Setelah iterasi ke 3 :

- **currentTime** is 3 + 3 = 6
- **numberOfThings** = 3

Setelah iterasi keempat, **currentTime** adalah 6 + 4 = 10, dimana lebih besar dari T, oleh karenya jawabannya adalah 3.

Kasus ini sangat ringan dan segera setelah Anda membaca masalahnya, jelas bahwa anda dapat menerapkan algoritma Greedy untuk itu.

Jika diimplementasikan ke program maka dapat dilihat source kodingnya berikut ini :

```
#include <iostream>
    #include <algorithm>

    using namespace std;
    const int MAX = 105;
    int A[MAX];

    int main()
    {
        int T, N, numberOfThings = 0, currentTime = 0;
        cin >> N >> T;
        for(int i = 0;i < N;++i)
            cin >> A[i];
        sort(A, A + N);
        for(int i = 0;i < N;++i)
        {
            currentTime += A[i];
            if(currentTime > T)
```

```
                break;
            numberOfThings++;
        }
        cout << numberOfThings << endl;
        return 0;
}
```

Contoh 2.

(Masalah Penukaran uang):

Persoalan: Diberikan uang senilai *A*. Tukar *A* dengan koin-koin uang yang ada. Berapa jumlah minimum koin yang diperlukan untuk penukaran tersebut?

Misalkan tersedia koin-koin 1, 5, 10, dan 25

Uang senilai 32 dapat ditukar dengan cara berikut:

32 = 1 + 1 + … + 1 (32 koin)

32 = 5 + 5 + 5 + 5 + 10 + 1 + 1 (7 koin)

32 = 10 + 10 + 10 + 1 + 1 (5 koin)

… dan seterusnya

Minimum: 32 = 25 + 5 + 1 + 1) hanya 4 koin

Strategi *greedy* yang digunakan adalah: Pada setiap langkah, pilihlah koin dengan nilai sebesar mungkin dari himpunan koin yang tersisa dengan syarat (kendala) tidak melebihi nilai uang yang ditukarkan.

Tinjau masalah menukarkan uang 32 dengan koin 1, 5, 10, dan 25:

Langkah 1: pilih 1 buah koin 25 (Total = 25)

Langkah 2: pilih 1 buah koin 5 (Total = 25 + 5 = 30)

Langkah 3: pilih 2 buah koin 1 (Total = 25+5+1+1= 32)

Solusi: Jumlah koin minimum = 4 (solusi optimal!)

Pada setiap langkah di atas kita memperoleh optimum lokal, dan pada akhir algoritma kita memperoleh optimum global (yang pada contoh ini merupakan solusi optimum).

5.2 SKEMA UMUM ALGORITMA GREEDY

Algoritma greedy disusun oleh elemen-elemen berikut:

1. Himpunan kandidat : Berisi elemen-elemen pembentuk solusi.

2. Himpunan solusi : Berisi kandidat-kandidat yang terpilih sebagai solusi persoalan.
3. Fungsi seleksi (*selection function*) :Memilih kandidat yang paling memungkinkan mencapai solusi optimal. Kandidat yang sudah dipilih pada suatu langkah tidak pernah dipertimbangkan lagi pada langkah selanjutnya.
4. Fungsi kelayakan (*feasible*) : Memeriksa apakah suatu kandidat yang telah dipilih dapat memberikan solusi yang layak, yakni kandidat tersebut bersama-sama dengan himpunan solusi yang sudah terbentuk tidak melanggar kendala (*constraints*) yang ada. Kandidat yang layak dimasukkan ke dalam himpunan solusi, sedangkan kandidat yang tidak layak dibuang dan tidak pernah dipertimbangkan lagi.
5. Fungsi obyektif, yaitu fungsi yang memaksimumkan atau meminimumkan nilai solusi (misalnya panjang lintasan, keuntungan, dan lain-lain).

Contoh pada masalah penukaran uang, elemen-elemen algoritma *greedy*-nya adalah:

1. Himpunan kandidat: himpunan koin yang merepresentasikan nilai 1, 5, 10, 25, paling sedikit mengandung satu koin untuk setiap nilai.
2. Himpunan solusi: total nilai koin yang dipilih tepat sama jumlahnya dengan nilai uang yang ditukarkan.
3. Fungsi seleksi: pilihlah koin yang bernilai tertinggi dari himpunan kandidat yang tersisa.
4. Fungsi layak: memeriksa apakah nilai total dari himpunan koin yang dipilih tidak melebihi jumlah uang yang harus dibayar.
5. Fungsi obyektif: jumlah koin yang digunakan minimum.

Psudo-code algoritma *greedy* adalah sebagai berikut:

```
procedure greedy(input C: himpunan_kandidat;
                 output S : himpunan_solusi)
{ menentukan solusi optimum dari persoalan optimasi dengan algoritma greedy
  Masukan: himpunan kandidat C
  Keluaran: himpunan solusi S
}
Deklarasi
    x : kandidat;
Algoritma:
    S←{}                  { inisialisasi S dengan kosong }
    while (belum SOLUSI(S)) and (C ≠ {} ) do
       x←SELEKSI(C);      { pilih sebuah kandidat dari C}
       C← C - {x}       { elemen himpunan kandidat berkurang satu }
      if LAYAK(S ∪ {x}) then
```

```
            S←S ∪ {x}
        endif
    endwhile
{SOLUSI(S) sudah diperoleh or C = {} }
```

Akhir setiap lelaran, solusi yang terbentuk adalah optimum lokal dan pada akhir kalang while-do diperoleh optimum global, namun adakalanya optimum global merupakan solusi *sub-optimum* atau *pseudo-optimum*. Alasan:

1. algoritma *greedy* tidak beroperasi secara menyeluruh terhadap semua alternatif solusi yang ada (sebagaimana pada metode *exhaustive search)*.
2. pemilihan fungsi SELEKSI: Mungkin saja terdapat beberapa fungsi SELEKSI yang berbeda, sehingga kita harus memilih fungsi yang tepat jika kita ingin algoritma bekerja dengan benar dan menghasilkan solusi yang benar-benar optimum, karena itu, pada sebagian masalah algoritma *greedy* tidak selalu berhasil memberikan solusi yang benar-benar optimum.

Jika jawaban terbaik mutlak (benar-benar optimum) tidak diperlukan, maka algoritma *greedy* sering berguna untuk menghasilkan solusi yang menghampiri (*approximation*) optimum, daripada menggunakan algoritma yang lebih rumit untuk menghasilkan solusi yang eksak. Bila algoritma *greedy* optimum, maka keoptimalannya itu dapat dibuktikan secara matematis

Misalkan koin-koin dinyatakan dalam himpunan-ganda (*multiset*) $\{d_1, d_2, \ldots, d_n\}$. Solusi persoalan dinyatakan sebagai tupel $X = \{x_1, x_2, \ldots, x_n\}$, sedemikian sehingga $x_i = 1$ jika d_i dipilih, atau $x_i = 0$ jika d_i tidak dipilih. Misalkan uang yang akan ditukar dengan sejumlah koin adalah A.

Obyektif persoalan adalah

Minimisasi $F = \sum_{i=1}^{n} x_i$ (fungsi obyektif)

dengan kendala $\sum_{i=1}^{n} d_i x_i = A$

Algoritma Exhaustive Search

a. Karena setiap elemen $X = \{x_1, x_2, \ldots, x_n\}$ adalah 0 atau 1, maka terdapat 2^n kemungkinan nilai X.

b. Waktu yang dibutuhkan untuk mengevaluasi fungsi obyektif adalah $O(n)$, oleh karena itu kompleksitas algoritma *exhaustive search* untuk masalah ini adalah $O(n \cdot 2^n)$.

Pemecahan Masalah dengan Algoritma Greedy

1. Strategi *greedy* yang digunakan dalam memilih koin berikutnya:
2. Pada setiap langkah, pilihlah koin dengan nilai sebesar mungkin dari himpunan koin yang tersisa dengan syarat tidak melebihi nilai uang yang ditukarkan.
3. Agar pemilihan koin berikutnya optimal, maka perlu mengurutkan himpunan koin dalam urutan yang menurun (*noninceasing order*).
4. Jika himpunan koin sudah terurut menurun, maka kompleksitas algoritma *greedy* adalah $O(n)$. Jika waktu pengurutan diperhitungkan, maka kompleksitas algoritma *greedy* ditentukan dari kompleksitas algoritma pengurutan.
5. Apakah algoritma *greedy* untuk masalah penukaran uang ini selalu menghasilkan solusi optimum? Jawabannya: tidak selalu, bergantung pada koin mata uang yang digunakan.

Contoh 3.

(a) Koin: 5, 4, 3, dan 1

Uang yang ditukar = 7.

Solusi dengan algoritma *greedy*:

7 = 5 + 1 + 1 (3 koin) → tidak optimal

Solusi yang optimal: 7 = 4 + 3 (2 koin)

(b) Koin: 10, 7, 1

Uang yang ditukar: 15

Solusi dengan algoritma *greedy*:

15 = 10 + 1 + 1 + 1 + 1 + 1 (6 koin)

Solusi yang optimal: 15 = 7 + 7 + 1 (hanya 3 koin)

(c) Koin: 15, 10, dan 1

Uang yang ditukar: 20

Solusi dengan algoritma *greedy*: 20 = 15 + 1 + 1 + 1 + 1 + 1 (6 koin)

Solusi opgtimal: 20 = 10 + 10 (2 koin)

Untuk sistem mata uang dollar AS, euro Eropa, dan *crown* Swedia, algoritma *greedy* selalu memberikan solusi optimum. Misalnya untuk menukarkan \$6,39 dengan uang kertas (*bill*) dan koin sen (*cent*), kita dapat memilih:

a. Satu buah uang kertas senilai \$5
b. Satu buah uang kertas senilai \$1 (\$5 + \$1 = \$6))
c. Satu koin 25 sen ($5 + $1 + 25^c = \$6,25)
d. Satu koin 10 sen ($5 + $1 + 25^c + 10^c = \$6,35)
e. Empat koin 1 sen ($5 + $1 + 25^c + 10^c + 1^c + 1^c + 1^c + 1^c = \$6,39)

Bagaimana dengan mata uang rupiah Indonesia?

5.3 MINIMASI WAKTU dalam SISTEM (PENJADWALAN)

Persoalan: Sebuah *server* (dapat berupa *processor*, pompa, kasir di bank, dll) mempunai *n* pelanggan (*customer*, *client*) yang harus dilayani. Waktu pelayanan untuk setiap pelanggan sudah ditetapkan sebelumnya, yaitu pelanggan *i* membutuhkan waktu t_i. Kita ingin meminimumkan total waktu di dalam sistem,

$$T = \sum_{i=1}^{n} \text{(waktu di dalam sistem untuk pelanggan } i\text{)}$$

Karena jumlah pelanggan adalah tetap, meminimumkan waktu di dalam sistem ekivalen dengan meminimumkan waktu rata-rata.

Contoh 4. Misalkan kita mempunyai tiga pelanggan dengan

$t_1 = 5$, $t_2 = 10$, $t_3 = 3$,

maka enam urutan pelayanan yang mungkin adalah:

Urutan	T
1, 2, 3:	5 + (5 + 10) + (5 + 10 + 3) = 38
1, 3, 2:	5 + (5 + 3) + (5 + 3 + 10) = 31
2, 1, 3:	10 + (10 + 5) + (10 + 5 + 3) = 43
2, 3, 1:	10 + (10 + 3) + (10 + 3 + 5) = 41
3, 1, 2:	**3 + (3 + 5) + (3 + 5 + 10) = 29 ← (optimal)**
3, 2, 1:	3 + (3 + 10) + (3 + 10 + 5) = 34

Pemecahan Masalah dengan Algoritma Exhaustive Search

1. Urutan pelangan yang dilayani oleh *server* merupakan suatu permutasi
2. Jika ada *n* orang pelanggan, maka tedapat *n*! urutan pelanggan. Waktu yang dibutuhkan untuk mengevaluasi fungsi obyektif adalah *O*(*n*), oleh karena itu kompleksitas algoritma *exhaustive search* untuk masalah ini adalah O(*nn*!)

Pemecahan Masalah dengan Algoritma Greedy

Strategi *greedy* untuk memilih pelanggan berikutnya adalah:

1. Pada setiap langkah, masukkan pelanggan yang membutuhkan waktu pelayanan terkecil di antara pelanggan lain yang belum dilayani.
2. Agar proses pemilihan pelanggan berikutnya optimal, maka perlu mengurutkan waktu pelayanan seluruh pelanggan dalam urutan yang menaik. Jika waktu pengurutan tidak dihitung, maka kompleksitas algoritma *greedy* untuk masalah minimisasi waktu di dalam sistem adalah *O*(*n*).

```
procedure PenjadwalanPelanggan(input n:integer)
{ Mencetak informasi deretan pelanggan yang akan diproses oleh server
tunggal
  Masukan: n pelangan, setiap pelanggan dinomori 1, 2, …, n
  Keluaran: urutan pelanggan yang dilayani
}
Deklarasi
   i : integer

Algoritma:
  {pelanggan 1, 2, ..., n sudah diurut menaik berdasarkan t_i}
  for i←1 to n do
     write('Pelanggan ', i, ' dilayani!')
  endfor
```

Pemilihan srategi *greedy* untuk penjadwalan pelanggan akan selalu menghasilkan solusi optimum. Keoptimuman ini dinyatakan dengan Teorema 3.1 berikut:

Teorema 2.1. Jika $t_1 \le t_2 \le \ldots \le t_n$ maka pengurutan $i_j = j$, $1 \le j \le n$ meminimumkan

$$T = \sum_{k=1}^{n} \sum_{j=1}^{k} t_{i_j}$$

untuk semua kemungkinan permutasi i_j.

5.4 PENYELESAIAN PERSOALAN KNAPSACK dengan GREEDY

Knapsack dapat diartikan sebagai karung, kantung, atau buntilan. Karung digunakan untuk memuat sesuatu. Dan tentunya tidak semua objek dapat ditampung di dalam karung. Karung tersebut hanya dapat menyimpan beberapa objek dengan total ukurannya (weight) lebih kecil atau sama dengan ukuran kapasitas karung.

Setiap objek itupun tidak harus kita masukkan seluruhnya. Tetapi bisa juga sebagian saja. Kali ini kami akan menggunakan knapsack 0/1, yaitu suatu objek diambil seluruh bagiannya atau tidak sama sekali. Setiap objek mempunyai nilai keuntungan atau yang disebut dengan profit. Tentunya kita menginginkan profit yang maksimal. Belum tentu semakin banyak objek yang masuk, semakin menguntungkan. Bisa saja hal yang sebaliknya yang terjadi.

Untuk menangani masalah ini, tentulah banyak caranya. Tetapi manakah cara yang terbaik? Penilaian cara ini bukan hanya dari penilaian dari hasilnya optimal atau tidak saja. Banyak langkah yang dibutuhkan juga mempengaruhi penilaian suatu cara penyelesaian masalah itu.

Ada beberapa strategi algoritma yang benar-benar menghasilkan solusi optimal. Contohnya adalah Brute Force. Tetapi strategi ini terlalu naïf, dan tidak efisien karena terlalu langsung mengacu pada permasalahannya.

Persoalan knapsack 0/1 ini dibahas dalam modul ini menggunakan strategi greedy. Pendekatan yang digunakan di dalam algoritma greedy adalah membuat pilihan yang " tampaknya" memberikan perolehan terbaik, yaitu dengan membuat pilihan optimum lokal (local optimum) pada setiap langkah dengan harapan bahwa sisanya mengarah ke solusi optimum global (global optimum). Dengan algoritma greedy, persoalan knapsack dapat dipecahkan dengan memasukkan objek satu per satu ke dalam knapsack. Sekali objek dimasukkan ke dalam knapsack, objek tersebut tidak bisa dikeluarkan lagi.

(a) *0/1 Knapsack*

Diberikan n buah objek dan sebuah knapsack dengan kapasitas bobot W. Setiap objek memiliki properti bobot (weigth) wi dan keuntungan(profit) pi. Objektif persoalan adalah memilih memilih objekobjek yang dimasukkan ke dalam knapsack sedemikian sehingga memaksimumkan keuntungan. Total bobot objek yang dimasukkan ke dalam

knapsack tidak boleh melebihi kapasitas knapsack. Solusi persoalan dinyatakan sebagai vector n-tupel:

X = {x1, x2, … , xn}

xi = 1 jika objek ke-i dimasukkan ke dalam knapsack,

xi = 0 jika objek ke-i tidak dimasukkan.

Formulasi secara matematis:

maksimasi $F = \sum_{i=1}^{n} p_i x_i$

dengan kendala (*constraint*)

$$\sum_{i=1}^{n} w_i x_i \leq W$$

yang dalam hal ini, $x_i = 0$ atau 1, $i = 1, 2, \ldots, n$

Algoritma Exhaustive Search

Algoritma *exhaustive search* untuk persoalan 0/1 *Knapsack* mempunyai kompleksitas $O(n \cdot 2^n)$.

Algoritma Greedy

1. Masukkan objek satu per satu ke dalam *knapsack*. Sekali objek dimasukkan ke dalam *knapsack*, objek tersebut tidak bisa dikeluarkan lagi.
2. Terdapat beberapa strategi *greedy* yang heuristik yang dapat digunakan untuk memilih objek yang akan dimasukkan ke dalam *knapsack*:

Tujuan dari algoritma knapsack adalah untuk memberikan hasil keuntungan atau profit yang optimum. Dalam kehidupan nyata, permasalahan Knapsack ini mungkin bisa dimplementasaikan dalam bidang Peti Kemas, Penggudangan, Pengepakan Barang dan sebagainya.

1. *Greedy by profit*.

Pada setiap langkah, *knapsack* diisi dengan objek yang mempunyai keuntungan terbesar. Strategi ini mencoba memaksimumkan keuntungan dengan memilih objek yang paling menguntungkan terlebih dahulu.

2. *Greedy by weight.*.

Pada setiap langkah, *knapsack* diisi dengan objek yang mempunyai berat paling ringan. Strategi ini mencoba memaksimumkan keuntungan dengan memasukkan sebanyak mungkin objek ke dalam *knapsack*.

3. *Greedy by density*.

Pada setiap langkah, *knapsack* diisi dengan objek yang mempunyai densitas, p_i / w_i terbesar. Strategi ini mencoba memaksimumkan keuntungan dengan memilih objek yang mempunyai keuntungan per unit berat terbesar.

Pemilihan objek berdasarkan salah satu dari ketiga strategi di atas tidak menjamin akan memberikan solusi optimal. Bahkan ada kemungkinan ketiga stategi tersebut tidak memberikan solusi optimum. Contoh 4 berikut memberikan ilustrasi kedua kasus ini.

Contoh 5 : Tinjau persoalan 0/1 *Knapsack* lain dengan 6 objek:

$w_1 = 100$; $p_1 = 40$
$w_2 = 50$; $p_2 = 35$
$w_3 = 45$; $p_3 = 18$
$w_4 = 20$; $p_4 = 4$
$w_5 = 10$; $p_5 = 10$
$w_6 = 5$; $p_6 = 2$
Kapasitas *knapsack* $W = 100$

Tabel 5.1 Penyelesaian dengan algoritma greedy

Properti objek				*Greedy by*			Solusi Optimal
i	w_i	p_i	p_i/w_i	*profit*	*weight*	*density*	
1	100	40	0,4	1	0	0	0
2	50	35	0,7	0	0	1	1
3	45	18	0,4	0	1	0	1
4	20	4	0,2	0	1	1	0
5	10	10	1,0	0	1	1	0
6	5	2	0,4	0	1	1	0
Total bobot				100	80	85	100
Total keuntungan				40	34	51	55

Pada contoh ini, algoritma *greedy* dengan ketiga strategi pemilihan objek tidak berhasil memberikan solusi optimal. Solusi optimal permasalah ini adalah $X = (0, 1, 1, 0, 0, 0)$ dengan total keuntungan = 55.

Kesimpulan: Algoritma *greedy* tidak selalu berhasil menemukan solusi optimal untuk masalah 0/1 *Knapsack*.

(b) *Fractional Knapsack*

Serupa dengan persoalan 0/1 *Knapsack* di atas, tetapi $0 \le x_i \le 1$, untuk $i = 1, 2, \ldots, n$

maksimasi $F = \sum_{i=1}^{n} p_i x_i$

dengan kendala (*constraint*)

$$\sum_{i=1}^{n} w_i x_i \le W$$ yang dalam hal ini, $0 \le x_i \le 1$, $i = 1, 2, \ldots, n$

*Algoritma Exhaustive Search :*Oleh karena $0 \le x_i \le 1$, maka terdapat tidak berhinga nilai-nilai x_i. Persoalan *Fractional Knapsack* menjadi malar (*continuous*) sehingga tidak mungkin dipecahkan dengan algoritma *exhaustive search.*

Pemecahan Masalah dengan Algoritma Greedy Ketiga strategi *greedy* yang telah disebutkan di atas dapat digunakan untuk memilih objek yang akan dimasukkan ke dalam *knapsack*.

Contoh 6. Tinjau persoalan *fractional knapsack* dengan $n = 3$.

$w_1 = 18$; $p_1 = 25$
$w_2 = 15$; $p_1 = 24$
$w_3 = 10$; $p_1 = 15$
Kapasitas *knapsack* $W = 20$

Tabel 5.2 Solusi dengan algoritma *greedy*:

Properti objek				*Greedy by*		
i	w_i	p_i	p_i/w_i	*profit*	*weight*	*density*
1	18	25	1,4	1	0	0
2	15	24	1,6	2/15	2/3	1
3	10	15	1,5	0	1	1/2
Total bobot				20	20	20
Total keuntungan				28,2	31,0	31,5

1. Penyelesaian persoalan *knapsack* yang memakai strategi pemilihan objek berdasarkan p_i/w_i terbesar memberikan keuntungan yang maksimum (optimum).
2. Solusi optmal persoalan *knapsack* di atas adalah $X = (0, 1, 1/2)$ yang memberikan keuntungan maksimum 31,5.
3. Agar proses pemilihan objek berikutnya optimal, maka kita perlu mengurutkan objek terlebih dahulu berdasarkan p_i/w_i dalam urutan yang menurun, sehingga objek berikutnya yang dipilih adalah objek sesuai dalam urutan itu.
4. Algoritma persoalan *fractional knapsack*:
    a. Hitung harga p_i/w_i , $i = 1, 2, ..., n$
    b. Urutkan seluruh objek berdasarkan nilai p_i/w_i yang menurun
    c. Panggil SolveFractinonalKnapsack untuk mencari solusi optimum.

```
procedure SolveFractionalKnapsack(input p, w, : Tabel,
                                  W : real, n : integer,
                                  output x : Tabel, TotalUntung : real)
```

```
{ Penyelesaian persoalan fractional knapsack dengan algoritma greedy
yang menggunaka strategi pemilihan objek berdasarkan density (pi/wi)
Asumsi: Sebelum pemanggilan SolveFractionalKnapsack, harus dilakukan
prosedur
pendahuluan sebagai berikut :
  1. Hitung harga pi/wi , i = 1, 2, ..., n
  2. Urutkan seluruh objek berdasarkan nilai pi/wi yang menurun
  3. Baru kemudian panggil procedure SolveFractionalKnapsack ini
}
Deklarasi
    i         : integer;
    kapasitas : real;
    MasihMuatUtuh : boolean;
Algoritma:
   for i←1 to n do
      xi ← 0   { inisialisasi setiap fraksi objek i dengan 0 }
   endfor

   kapasitas←W  { kapasitas knapsack awal }
   TotalUntung←0
   i←1
   MasihMuatUtuh←true
   while (i ≤ n) and (MasihMuatUtuh) do
     if wi ≤ kapasitas then
        xi←1
        TotalUntung←TotalUntung + pi
        kapasitas←kapasitas - wi
        i←i+1;
     else
       MasihMuatUtuh←false
     endif
   endwhile
   { i > n or not MasihMuatUtuh }

   if i ≤ n then   { objek terakhir yang akan dimasukkan}
      xi ← kapasitas/wi
      TotalUntung←TotalUntungt + xi*pi
   endif
```

Kompleksitas waktu asimptotik algoritma di atas (dengan mangabaikan waktu pengurutan objek) adalah $O(n)$.

Untuk mengimplentasikannya ke dalam program, maka dengan bobot dan nilai n item, kita perlu meletakkan item ini dalam ransel dengan kapasitas W untuk mendapatkan nilai total maksimum dalam ransel.

Untuk Masalah 0-1 Knapsack, kita tidak diizinkan untuk memecah item. Kita mengambil seluruh barang atau tidak mengambilnya.

```
Input:
  Items as (value, weight) pairs
  arr[] = {{60, 10}, {100, 20}, {120, 30}}
  Knapsack Capacity, W = 50;
Output:
  Maximum possible value = 220
  by taking items of weight 20 and 30 kg
```

Untuk *Fractional Knapsack*, kita dapat memecahkan item untuk memaksimalkan nilai total knapsack. Analoginya ketika ada masalah di mana kita dapat memilah-milah atau memisahkan barang, juga disebut masalah *Fractional Knapsack*.

```
Input :
   Same as above
Output :
   Maximum possible value = 240
   By taking full items of 10 kg, 20 kg and
   2/3rd of last item of 30 kg
```

Jika kasus ini diselesaikan menggunakan brute-force dengan mencoba semua subset yang mungkin dari semua fraksi yang berbeda tetapi kelemahannya adalah akan memakan banyak waktu.

Solusi yang efisien adalah dengan menggunakan pendekatan Greedy. Ide dasar dari pendekatan greedy adalah untuk menghitung nilai rasio/berat untuk setiap item dan mengurutkan item berdasarkan rasio ini. Kemudian ambil item dengan rasio tertinggi dan tambahkan sampai kita tidak bisa menambahkan item berikutnya secara keseluruhan dan pada akhirnya tambahkan item berikutnya sebanyak yang kita bisa, yang nantinya akan menjadi solusi optimal untuk masalah ini.

Kode sederhana dengan fungsi perbandingan dapat ditulis sebagai berikut, silakan lihat fungsi sortir lebih dekat, argumen ketiga untuk fungsi sortir adalah fungsi perbandingan kami yang mengurutkan item berdasarkan rasio nilai / berat dalam urutan yang tidak menurun.

Setelah mengurutkan kita perlu mengulang item-item ini dan menambahkannya dalam ransel kita memenuhi kriteria yang disebutkan di atas.

Listing Program

Listing berikut dengan bahasa pemrograman C++

```
#include <bits/stdc++.h>

using namespace std;

// Structure for an item which stores weight and corresponding
// value of Item
struct Item
{
    int value, weight;

    // Constructor
    Item(int value, int weight) : value(value), weight(weight)
    {}
};

// Comparison function to sort Item according to val/weight ratio
bool cmp(struct Item a, struct Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

// Main greedy function to solve problem
double fractionalKnapsack(int W, struct Item arr[], int n)
{
    //    sorting Item on basis of ratio
    sort(arr, arr + n, cmp);

    //    Uncomment to see new order of Items with their ratio
    /*
    for (int i = 0; i < n; i++)
    {
        cout << arr[i].value << "  " << arr[i].weight << " : "
             << ((double)arr[i].value / arr[i].weight) << endl;
    }
    */

    int curWeight = 0;  // Current weight in knapsack
    double finalvalue = 0.0; // Result (value in Knapsack)

    // Looping through all Items
    for (int i = 0; i < n; i++)
    {
        // If adding Item won't overflow, add it completely
        if (curWeight + arr[i].weight <= W)
        {
            curWeight += arr[i].weight;
            finalvalue += arr[i].value;
        }

        // If we can't add current Item, add fractional part of it
```

```
        else
        {
            int remain = W - curWeight;
            finalvalue += arr[i].value * ((double) remain / arr[i].weight);
            break;
        }
    }

    // Returning final value
    return finalvalue;
}

// driver program to test above function
int main()
{
    int W = 50;   //    Weight of knapsack
    Item arr[] = {{60, 10}, {100, 20}, {120, 30}};

    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum value we can obtain = "
         << fractionalKnapsack(W, arr, n);
    return 0;
}
```

Output :

```
Maximum value in Knapsack = 240.
```

Algoritma *greedy* untuk persoalan *fractional knapsack* dengan strategi pemilihan objek berdasarkan p_i/w_i terbesar akan selalu memberikan solusi optimal. Hal ini dinyatakan dalam Teorema 3.2 berikut ini.

Teorema 5.2. Jika $p_1/w_1 \geq p_2/w_2 \geq ... \geq p_n/w_n$ maka algoritma greedy dengan strategi pemilihan objek berdasarkan p_i/w_i terbesar menghasilkan solusi yang optimum.

5.5 *TRAVELLING SALESPERSON PROBLEM* (TSP)

Penggambaran yang sangat sederhana dari istilah *Traveling Salesman Problem (TSP)* adalah seorang *salesman* keliling yang harus mengunjungi *n* kota dengan aturan sebagai berikut :

a. Ia harus mengunjungi setiap kota hanya sebanyak satu kali.
b. Ia harus meminimalisasi total jarak perjalanannya.
c. Pada akhirnya ia harus kembali ke kota asalnya.

Dengan demikian, apa yang telah ia lakukan tersebut akan kita sebut sebagai sebuah *tour*. Guna memudahkan permasalahan, pemetaan *n* kota tersebut akan digambarkan dengan sebuah *graph*, dimana jumlah *vertice* dan *edge*-nya terbatas

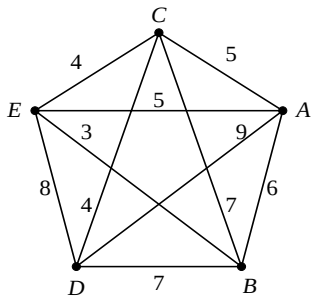
(sebuah *vertice* akan mewakili sebuah kota dan sebuah *edge* akan mewakili jarak antar dua kota yang dihubungkannya). Penanganan problem TSP ini ekuivalen dengan mencari sirkuit Hamiltonian terpendek.

Terdapat berbagai algoritma yang dapat diterapkan untuk menangani kasus TSP ini, mulai dari *exhaustive search* hingga *dynamic programming*. Akan tetapi saat ini yang akan digunakan adalah algoritma *Greedy*. Strategi *greedy* yang digunakan untuk memilih kota berikutnya yang akan dikunjungi adalah sebagai berikut :

”Pada setiap langkah, akan dipilih kota yang belum pernah dikunjungi, dimana kota tersebut memiliki jarak terdekat dari kota sebelumnya”, berdasarkan aturan tersebut dapat dilihat bahwa *greedy* tidak mempertimbangkan nilai *heuristic* (dalam hal ini bisa berupa jarak langsung antara dua kota). Strategi *greedy* untuk memilih kota selanjutnya:

Pada setiap langkah, pilih kota yang belum pernah dikunjungi yang mempunyai jarak terdekat.

Contoh 7. $n = 5$; perjalanan dimulai dari kota *E*.



Langkah 1: Dari *E* pergi ke *B* (Total jarak = 3)

Langkah 2: Dari *B* pergi ke *A* (Total jarak = 3 + 6 = 9)

Langkah 3: Dari *A* pergi ke *C* (Total jarak = 9 + 5 = 14)

Langkah 4: Dari *C* pergi ke *D* (Total jarak = 14 + 4 = 18)

Langkah 5: Dari *D* kembali lagi ke *E* (Total jarak = 18 + 8 = 26)

Perjalanan (sirkuit Hamilton) yang dihasilkan:

$$E \to B \to A \to C \to D \to E$$

Panjang = 3 + 6 + 5 + 4 + 8 = 26. (tidak optimal)

Solusi yang lebih baik:

$$E \to B \to D \to C \to A \to E$$

dengan panjang = 3 + 7 + 4 + 5 + 5 = 24. (optimal)

Kesimpulan: Masalah TSP tidak dapat diselesaikan dengan algoritma *greedy*, karena solusi yang diahasilkan tidak dijamin merupakan solusi optimal. Namun jika solusi hampiran dianggap mencukupi, maka solusi dengan algoritma greedy dapat dijadikan sebagai titik awal untuk menemukan sirkuit Hamilton yang minimum.

5.6 PENJADWALAN *JOB* DENGAN TENGGAT WAKTU

Persoalan: Ada *n* buah *job* yang akan dikerjakan oleh sebuah mesin. Tiap *job* diproses oleh mesin selama satu satuan waktu dan tenggat waktu (*deadline*) –yaitu batas waktu *job* tersebut harus sudah selesai diproses– tiap *job i* adalah $d_i \geq 0$ (d_i dalam satuan waktu yang sama dengan waktu proses mesin). *Job i* akan memberikan keuntungan sebesar p_i jika dan hanya jika *job* tersebut diselesaikan tidak melebihi tenggat waktunya.

Obyektif persoalan adalah memilih *job-job* yang akan dikerjakan oleh mesin sehingga keuntungan yang diperoleh dari pengerjaan itu maksimum. Secara matematis, fungsi obyektif persoalan ini dapat ditulis sebagai

$$\text{Maksimasi } F = \sum_{i \in J} p_i$$

Solusi yang layak (*feasible*) ialah himpunan *J* yang berisi urutan *job* yang akan diproses oleh sebuah mesin sedemikian sehingga setiap *job* di dalam *J* selesai dikerjakan sebelum tenggatnya. Solusi optimum ialah solusi layak yang memaksimumkan *F*.

Contoh 8. Misalkan A berisi 4 buah job ($n = 4$). Tenggat setiap *job* dan keuntungan masing-masing: $(p_1, p_2, p_3, p_4) = (50, 10, 15, 30)$; $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

Tabel 5.3 Job dan keuntungan

Job	Tenggat	Harus selesai pada pukul	Profit
1	2 jam	8.00	50
2	1 jam	7.00	10
3	2 jam	8.00	15
4	1 jam	7.00	30

Tabel 5.4 Hasil penyelesaian dengan greedy

Langkah	J	$F = \sum p_i$	Keterangan
0	{}	0	-
1	{1}	50	layak
2	{4,1}	50 + 30 = 80	layak
3	{4, 1, 3}	-	tidak layak
4	{4, 1, 2}	-	tidak layak

Solusi optimum: J = {4, 1} dengan F = 80.

Pemecahan Masalah dengan Algoritma Greedy

a. Strategi *greedy* untuk memilih *job* adalah p*ada setiap langkah, pilih job i dengan p_i yang terbesar untuk menaikkan nilai fungsi obyektif F.*
b. Agar proses pemilihan pelanggan berikutnya optimal, maka algoritma *greedy* untuk masalah ini memerlukan kita mengurutkan *job* berdasarkan nilai p_i yang menurun (dari besar ke kecil) terlebih dahulu.

```
procedure PenjadwalanJobDenganTenggat(input A : HimpunanJob,
                                      n : integer,
                                      output J: HimpunanJob)
{ Menentukan barisan job yang akan diproses oleh mesin.
  Masukan:  job-job di dalam himpunan A. Setiap job mempunyai
  properti di dan pi.
  Keluaran: himpunan job yang akan diproses oleh mesin dan
  memberikan total keuntungan terbesar.
}
Deklarasi
  i : integer

Algoritma:
  {job 1, 2, ..., n sudah diurut menurun berdasarkan pi}
  J←{1}          {job 1 selalu terpilih}
  for i←2 to n do
     if (semua job dalam J ∪ {i} layak) then
      J←J ∪ {i}
     endif
  endfor
```

5.7 SHORTEST JOB FIRST (SJF), PENJADWALAN CPU

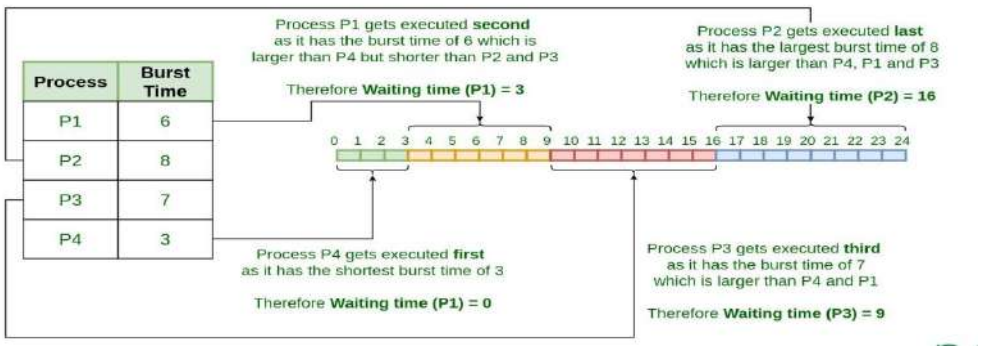
Shortest Job First (SJF) atau pekerjaan terpendek berikutnya, adalah kebijakan penjadwalan yang memilih proses menunggu dengan waktu eksekusi terkecil untuk dieksekusi berikutnya. SJN adalah algoritma non-preemptive.

Beberapa hal tentang algoritma SJF ini :

a. Pekerjaan terpendek pertama memiliki keuntungan memiliki waktu tunggu rata-rata minimum di antara semua algoritma penjadwalan.
b. Merupakan Algoritma Greedy.
c. dapat menyebabkan starvasi jika proses yang lebih pendek tetap bertahan. Masalah ini dapat dipecahkan dengan menggunakan konsep *ageing*.
d. Ini praktis tidak layak sebagai Sistem Operasi yang kemungkinan tidak tahu *burst time* dan karena itu mungkin tidak menyortirnya. Meskipun tidak mungkin untuk memprediksi waktu eksekusi, beberapa metode dapat digunakan untuk memperkirakan waktu eksekusi untuk suatu pekerjaan, seperti rata-rata tertimbang dari waktu eksekusi sebelumnya. SJF dapat digunakan di lingkungan khusus di mana perkiraan waktu berjalan yang akurat tersedia

Algoritma:

a. Urutkan semua proses sesuai dengan waktu kedatangan.
b. Kemudian pilih proses yang memiliki waktu kedatangan minimum dan waktu Burst minimum.
c. Setelah selesai proses membuat kumpulan proses yang setelah sampai penyelesaian proses sebelumnya dan pilih proses itu di antara kumpulan yang memiliki waktu Burst minimum.



Gambar 5.1 Proses SJF

Bagaimana cara menghitung waktu dalam SJF menggunakan program?

Menghitung waktu SJF dengan menggunakan program meliputi

a. Waktu Penyelesaian: Waktu di mana proses menyelesaikan pelaksanaannya.
b. Waktu Putar: Perbedaan waktu antara waktu selesai dan waktu kedatangan.

Turn Around Time = Completion Time – Arrival time

c. Waiting Time (W.T): Perbedaan Waktu antara waktu putar dan waktu burst.

   Waiting Time = Turn Around Time - Burst Time

Listing Program

Listing berikut dengan bahasa pemrograman C++

```
/ C++ program to implement Shortest Job first with Arrival Time
#include<iostream>
using namespace std;
int mat[10][6];

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void arrangeArrival(int num, int mat[][6])
{
    for(int i=0; i<num; i++)
    {
        for(int j=0; j<num-i-1; j++)
        {
            if(mat[j][1] > mat[j+1][1])
            {
                for(int k=0; k<5; k++)
                {
                    swap(mat[j][k], mat[j+1][k]);
                }
            }
        }
    }
}
void completionTime(int num, int mat[][6])
{
    int temp, val;
    mat[0][3] = mat[0][1] + mat[0][2];
    mat[0][5] = mat[0][3] - mat[0][1];
    mat[0][4] = mat[0][5] - mat[0][2];

    for(int i=1; i<num; i++)
    {
        temp = mat[i-1][3];
        int low = mat[i][2];
        for(int j=i; j<num; j++)
        {
            if(temp >= mat[j][1] && low >= mat[j][2])
            {
```

```
                low = mat[j][2];
                val = j;
            }
        }
        mat[val][3] = temp + mat[val][2];
        mat[val][5] = mat[val][3] - mat[val][1];
        mat[val][4] = mat[val][5] - mat[val][2];
        for(int k=0; k<6; k++)
        {
            swap(mat[val][k], mat[i][k]);
        }
    }
}

int main()
{
    int num, temp;

    cout<<"Enter number of Process: ";
    cin>>num;

    cout<<"...Enter the process ID...\n";
    for(int i=0; i<num; i++)
    {
        cout<<"...Process "<<i+1<<"...\n";
        cout<<"Enter Process Id: ";
        cin>>mat[i][0];
        cout<<"Enter Arrival Time: ";
        cin>>mat[i][1];
        cout<<"Enter Burst Time: ";
        cin>>mat[i][2];
    }

    cout<<"Before Arrange...\n";
    cout<<"Process ID\tArrival Time\tBurst Time\n";
    for(int i=0; i<num; i++)
    {
        cout<<mat[i][0]<<"\t\t"<<mat[i][1]<<"\t\t"<<mat[i][2]<<"\n";
    }

    arrangeArrival(num, mat);
    completionTime(num, mat);
    cout<<"Final Result...\n";
    cout<<"Process ID\tArrival Time\tBurst Time\tWaiting
Time\tTurnaround Time\n";
    for(int i=0; i<num; i++)
    {
        cout<<mat[i][0]<<"\t\t"<<mat[i][1]<<"\t\t"<<mat[i][2]<<"\t\t"<
<mat[i][4]<<"\t\t"<<mat[i][5]<<"\n";
    }
}
```

Output :

```
Process ID      Arrival Time    Burst Time
1               2               3
2               0               4
3               4               2
4               5               4
Final Result...
Process ID      Arrival Time    Burst Time      Waiting Time    Turnaround
Time
2               0               4               0               4
3               4               2               0               2
1               2               3               4               7
4               5               4               4               8
```

5.8 KASUS POLISI MENANGKAP PENCURI

Diberikan array ukuran n yang memiliki spesifikasi sebagai berikut:

a. Setiap elemen dalam array mengandung polisi atau pencuri.
b. Setiap polisi hanya dapat menangkap satu pencuri.
c. Seorang polisi tidak dapat menangkap seorang pencuri yang lebih dari unit K dari polisi.

Kita perlu menemukan jumlah maksimum pencuri yang bisa ditangkap.

```
Contoh :
Input : arr[] = {'P', 'T', 'T', 'P', 'T'},
            k = 1.
Output : 2.
Di sini maksimal 2 pencuri bisa ditangkap, pertama
polisi menangkap pencuri pertama dan polisi kedua
dapat menangkap pencuri kedua atau ketiga.
Input : arr[] = {'T', 'T', 'P', 'P', 'T', 'P'},
            k = 2.
Output : 3.

Input : arr[] = {'P', 'T', 'P', 'T', 'T', 'P'},
            k = 3.
Output : 3.
```

Pendekatan brute force merupakan pendekatan yang memeriksa semua kombinasi polisi dan pencuri yang layak dan mengembalikan ukuran maksimum yang ditetapkan di antara mereka. Kompleksitas waktunya eksponensial dan dapat dioptimalkan jika kita mengamati properti penting.

Untuk kasus ini solusi yang efisien adalah dengan menggunakan algoritma greedy. Tapi dengan properti greedy menggunakannya bisa jadi agak sedikit rumit. Kita dapat mencoba menggunakan: "Untuk setiap polisi dari kiri menangkap pencuri terdekat." Misalnya diketahui tiga yang diberikan tetapi gagal dua karena menghasilkan 2 yang tidak benar. Kita juga dapat mencoba: "Untuk setiap polisi dari kiri menangkap pencuri yang paling jauh". misalnya dua yang diberikan di atas tetapi gagal tiga karena menghasilkan 2 yang tidak benar.

Argumen simetris dapat diterapkan untuk menunjukkan bahwa pelintasan untuk ini dari sisi kanan array juga gagal. Kita dapat mengamatinya melalui berpikir irrespektf dari polisi dan fokus hanya pada penjatahan kerja:

a. Dapatkan indeks polisi dan pencuri terendah t. Buat jatah jika | p-t | <= k dan kenaikan ke p dan t berikutnya ditemukan.
b. Jika tidak, tambahkan min (p, t) ke p atau t berikutnya yang ditemukan.
c. Ulangi di atas dua langkah sampai p dan t berikutnya ditemukan.
d. Kembalikan jumlah jatah yang dibuat.

berikut ini adalah implementasi dari algoritma di atas. Ini menggunakan vektor untuk menyimpan indeks polisi dan pencuri dalam array dan memprosesnya.

Listing Program

Listing berikut dengan bahasa pemrograman C++

```
#include <iostream>
#include <vector>
#include <cmath>

using namespace std;

// Returns maximum number of thieves that can
// be caught.
int policeThief(char arr[], int n, int k)
{
    int res = 0;
    vector<int> thi;
```

```
    vector<int> pol;

    // store indices in the vector
    for (int i = 0; i < n; i++) {
        if (arr[i] == 'P')
            pol.push_back(i);
        else if (arr[i] == 'T')
            thi.push_back(i);
    }

    // track lowest current indices of
    // thief: thi[l], police: pol[r]
    int l = 0, r = 0;
    while (l < thi.size() && r < pol.size()) {

        // can be caught
        if (abs(thi[l] - pol[r]) <= k) {
            res++;
            l++;
            r++;
        }

        // increment the minimum index
        else if (thi[l] < pol[r])
            l++;
        else
            r++;
    }

    return res;
}

// Driver program
int main()
{
    int k, n;

    char arr1[] = { 'P', 'T', 'T', 'P', 'T' };
    k = 2;
    n = sizeof(arr1) / sizeof(arr1[0]);
    cout << "Maximum thieves caught: "
         << policeThief(arr1, n, k) << endl;

    char arr2[] = { 'T', 'T', 'P', 'P', 'T', 'P' };
    k = 2;
    n = sizeof(arr2) / sizeof(arr2[0]);
    cout << "Maximum thieves caught: "
         << policeThief(arr2, n, k) << endl;

    char arr3[] = { 'P', 'T', 'P', 'T', 'T', 'P' };
    k = 3;
```

```
    n = sizeof(arr3) / sizeof(arr3[0]);
    cout << "Maximum thieves caught: "
         << policeThief(arr3, n, k) << endl;

    return 0;
}
```

```
Output:
Maximum thieves caught: 2
Maximum thieves caught: 3
Maximum thieves caught: 3
Kompleksitas waktu dari pendekatan di atas dalam O (N) di mana N
adalah ukuran array
```

5.9 WADAH AIR

Diberikan n bilangan bulat non-negatif a_1, a_2, ..., a_n di mana masing-masing mewakili titik pada koordinat (i, a_i). 'N' garis vertikal digambar sedemikian rupa sehingga dua titik akhir dari garis i berada pada (i, a_i) dan (i, 0). Temukan dua garis, yang bersama-sama dengan sumbu x membentuk wadah, sehingga wadah tersebut mengandung paling banyak air.

Program harus mengembalikan bilangan bulat yang sesuai dengan area maksimum air yang dapat ditampung (area maksimum bukannya volume maksimum terdengar aneh tapi ini adalah bidang 2D yang sedang kami kerjakan untuk kesederhanaan).

Catatan:

Wadah tidak boleh dimiringkan

Contoh :

```
Input : [1, 5, 4, 3]
Output : 6
Penjelasan :
5 and 3 are distance 2 apart.
So the size of the base = 2.
Height of container = min(5, 3) = 3.
So total area = 3 * 2 = 6
Input : [3, 1, 2, 4, 5]
Output : 12
Explanation :
```

```
5 and 3 are distance 4 apart.
So the size of the base = 4.
Height of container = min(5, 3) = 3.
So total area = 4 * 3 = 12
```

Pendekatan:

a. **Catatan 1:** Ketika Anda mempertimbangkan a1 dan aN, maka area tersebut adalah (N-1) * min (a_1, a_N).
b. **Catatan 2**: Basis (N-1) adalah maksimum yang mungkin.

Dari catatan teresebut maka :

a. Ini menyiratkan bahwa jika ada solusi yang lebih baik, maka pasti akan memiliki Ketinggian lebih besar dari min (a_1, a_N).
   Base * Tinggi> (N-1) * mnt (a_1, a_N)
b. Kita tahu itu, Basis min (a_1, a_N) yang berarti bahwa kita dapat membuang min (a_1, a_N) dari perangkat kita dan mencari untuk menyelesaikan masalah ini lagi dari awal.
c. Jika $a_1 < a_N$, maka masalahnya berkurang untuk menyelesaikan hal yang sama untuk a_2, a_N.
d. Selain itu, mengurangi untuk menyelesaikan hal yang sama untuk a_1, a_{N-1}

Listing Program

Listing berikut dengan bahasa pemrograman C++

```
#include<iostream>
using namespace std;

int maxArea(int A[], int len)
{
    int l = 0;
    int r = len -1;
    int area = 0;

    while (l < r)
    {
        // Calculating the max area
        area = max(area, min(A[l],
                        A[r]) * (r - l));

            if (A[l] < A[r])
                l += 1;
```

```
            else
                r -= 1;
    }
    return area;
}

// Driver code
int main()
{
    int a[] = {1, 5, 4, 3};
    int b[] = {3, 1, 2, 4, 5};

    int len1 = sizeof(a) / sizeof(a[0]);
    cout << maxArea(a, len1);

    int len2 = sizeof(b) / sizeof(b[0]);
    cout << endl << maxArea(b, len2);
}
```

Output :

```
6

12
```

5.10 PENGKODEAN HUFFMAN (*HUFFMAN CODING*)

1. Pegantar

Pengodean Huffman adalah salah satu metode kompresi dasar, yang telah terbukti bermanfaat dalam standar kompresi gambar dan video. Saat menerapkan teknik penyandian Huffman pada Gambar, simbol sumber dapat berupa intensitas piksel Gambar, atau output dari fungsi pemetaan intensitas.

Kode Huffman sebagai algoritma kompresi data menggunakan teknik greedy untuk implementasinya. Algoritma didasarkan pada frekuensi karakter yang muncul dalam file.

Kita tahu bahwa file kita disimpan sebagai kode biner di komputer dan setiap karakter dari file tersebut diberi kode karakter biner dan biasanya, kode karakter ini memiliki panjang tetap untuk karakter yang berbeda.

Kode Awalan, berarti kode (urutan bit) ditetapkan sedemikian rupa sehingga kode yang ditetapkan untuk satu karakter bukan awalan kode yang ditugaskan untuk karakter lain. Ini adalah bagaimana Huffman Coding memastikan bahwa tidak ada ambiguitas ketika decoding bitstream yang dihasilkan.

Jika diketahui ada empat karakter a, b, c dan d, dan kode panjang variabel yang sesuai adalah 00, 01, 0 dan 1. Pengkodean ini mengarah pada ambiguitas karena kode yang ditugaskan ke c adalah awalan dari kode yang ditugaskan ke a dan b. Jika bit stream terkompresi adalah 0001, output de-kompresi mungkin "cccd" atau "ccb" atau "acd" atau "ab".

Ada dua bagian utama dalam Huffman Coding

1. Membangun Pohon Huffman dari karakter input.
2. Melintasi Pohon Huffman dan menetapkan kode ke karakter.

2. Langkah-langkah untuk membangun Pohon Huffman

Input adalah array karakter unik beserta frekuensi kemunculannya dan outputnya adalah Huffman Tree.

1. Buat simpul daun untuk setiap karakter unik dan buat tumpukan min dari semua simpul daun (Min Heap digunakan sebagai antrian prioritas. Nilai bidang frekuensi digunakan untuk membandingkan dua simpul dalam tumpukan minimum. Awalnya, karakter yang paling jarang muncul adalah root)
2. Ekstrak dua node dengan frekuensi minimum dari min heap.
3. Buat simpul internal baru dengan frekuensi yang sama dengan jumlah dari dua node frekuensi. Jadikan simpul yang diekstraksi pertama sebagai anak kirinya dan simpul yang diekstrak lainnya sebagai anak kanannya. Tambahkan simpul ini ke tumpukan min.
4. Ulangi langkah # 2 dan # 3 hingga heap hanya berisi satu node. Node yang tersisa adalah node root dan pohon sudah selesai.

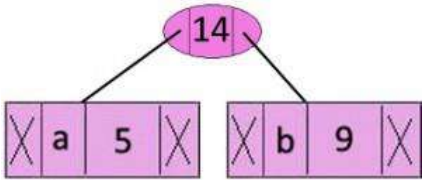
Berikut contoh untuk pemahaman penggunaan melalui algoritmanya:

character	Frequency
a	5
b	9
c	12
d	13

e	16
f	45

Langkah 1. Bangun heap min yang berisi 6 node di mana setiap node mewakili root dari pohon dengan node tunggal.

Langkah 2 Ekstrak dua node frekuensi minimum dari min heap. Tambahkan simpul internal baru dengan frekuensi 5 + 9 = 14 .:

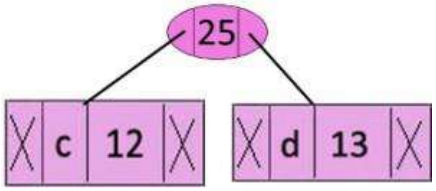


Gambar 5.2 Minimum heap 5 simpul

Sekarang min heap berisi 5 simpul di mana 4 simpul adalah akar pohon dengan masing-masing elemen tunggal, dan satu simpul tumpukan adalah akar pohon dengan 3 elemen

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

Langkah 3: Ekstrak dua node frekuensi minimum dari heap. Tambahkan simpul internal baru dengan frekuensi 12 + 13 = 25



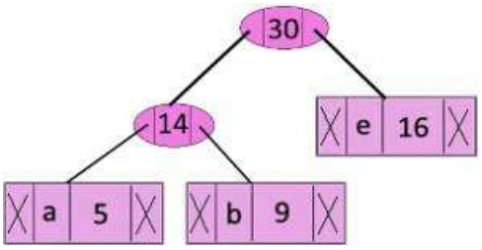
Gambar 5.3 Minimum heap 4 simpul

Sekarang min heap berisi 4 simpul di mana 2 simpul adalah akar pohon dengan masing-masing elemen, dan dua simpul tumpukan adalah akar pohon dengan lebih dari satu simpul.

character	Frequency
Internal Node	14
e	16

```
Internal Node          25
       f               45
```

Langkah 4: Ekstrak dua node frekuensi minimum. Tambahkan simpul internal baru dengan frekuensi 14 + 16 = 30

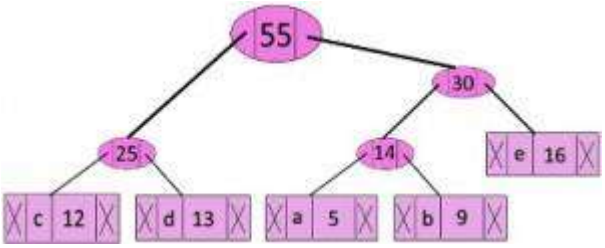


Gambar 5.4 Minimum heap 3 simpul

Sekarang min heap berisi 3 node.

```
character          Frequency
Internal Node         25
Internal Node         30
      f               45
```

Langkah 5: Ekstrak dua node frekuensi minimum. Tambahkan simpul internal baru dengan frekuensi 25 + 30 = 55

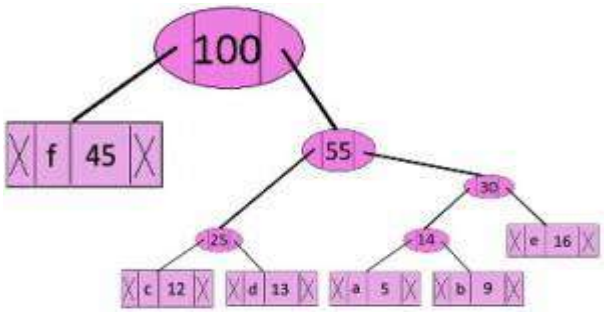


Gambar 5.5 Minimum heap 2 simpul

Sekarang min heap berisi 2 node.

```
character     Frequency
      f          45
Internal Node    55
```

Langkah 6: Ekstrak dua node frekuensi minimum. Tambahkan simpul internal baru dengan frekuensi 45 + 55 = 100



Gambar 5.6 Minimum heap 1 simpul

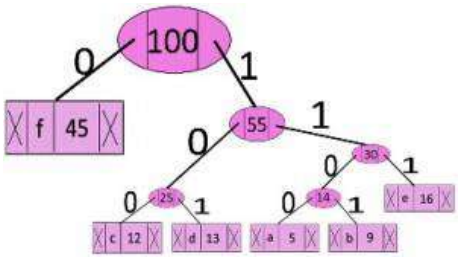
Sekarang min heap hanya mengandung satu node.

```
character       Frequency
Internal Node    100
```

Karena heap hanya berisi satu node, algoritma berhenti di sini.

Langkah-langkah untuk mencetak kode dari Huffman Tree:

Lintasi pohon yang terbentuk mulai dari akar. Pertahankan array bantu. Saat bergerak ke anak kiri, tulis 0 ke array. Saat pindah ke anak yang tepat, tulis 1 ke larik. Cetak array saat simpul daun ditemukan.



Gambar 5.7 Minimum heap 0 simpul

Kode-kode tersebut adalah sebagai berikut:

```
character   code-word
    f           0
    c           100
    d           101
    a           1100
    b           1101
    e           111
```

3. Listing Program

Listing berikut dengan bahasa pemrograman C++

Catatan ;

Listing boleh dirubah ke bahasa pemrograman lainnya

```
// C++ program untuk kode huffman
#include <iostream>
#include <cstdlib>
using namespace std;

// This constant can be avoided by explicitly
// calculating height of Huffman Tree
#define MAX_TREE_HT 100

// A Huffman tree node
struct MinHeapNode {

    // One of the input characters
    char data;

    // Frequency of the character
    unsigned freq;

    // Left and right child of this node
    struct MinHeapNode *left, *right;
};

// A Min Heap: Collection of
// min-heap (or Huffman tree) nodes
struct MinHeap {

    // Current size of min heap
    unsigned size;

    // capacity of min heap
    unsigned capacity;

    // Attay of minheap node pointers
    struct MinHeapNode** array;
};

// A utility function allocate a new
// min heap node with given character
// and frequency of the character
struct MinHeapNode* newNode(char data, unsigned freq)
{
    struct MinHeapNode* temp
        = (struct MinHeapNode*)malloc
(sizeof(struct MinHeapNode));

    temp->left = temp->right = NULL;
```

```
    temp->data = data;


    return temp;
}

// A utility function to create
// a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity)

{

    struct MinHeap* minHeap
        = (struct MinHeap*)malloc(sizeof(struct MinHeap));

    // current size is 0
    minHeap->size = 0;

    minHeap->capacity = capacity;

    minHeap->array
        = (struct MinHeapNode**)malloc(minHeap->
capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

// A utility function to
// swap two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a,
                     struct MinHeapNode** b)

{

    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// The standard minHeapify function.
void minHeapify(struct MinHeap* minHeap, int idx)

{

    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size && minHeap->array[left]->
freq < minHeap->array[smallest]->freq)
        smallest = left;
```

```
    if (right < minHeap->size && minHeap->array[right]->
freq < minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest],
                        &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

// A utility function to check
// if size of heap is 1 or not
int isSizeOne(struct MinHeap* minHeap)
{
    return (minHeap->size == 1);
}

// A standard function to extract
// minimum value node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0]
        = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);

    return temp;
}
// A utility function to insert
// a new node to Min Heap
void insertMinHeap(struct MinHeap* minHeap,
                struct MinHeapNode* minHeapNode)

{
    ++minHeap->size;
    int i = minHeap->size - 1;

    while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq)
{
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }

    minHeap->array[i] = minHeapNode;
}
// A standard function to build min heap
void buildMinHeap(struct MinHeap* minHeap)

{
```

```
    int n = minHeap->size - 1;
    int i;

    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

// A utility function to print an array of size n
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        cout<< arr[i];

    cout<<"\n";
}

// Utility function to check if this node is leaf
int isLeaf(struct MinHeapNode* root)

{

    return !(root->left) && !(root->right);
}

// Creates a min heap of capacity
// equal to size and inserts all character of
// data[] in min heap. Initially size of
// min heap is equal to capacity
struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int
size)

{

    struct MinHeap* minHeap = createMinHeap(size);

    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);

    minHeap->size = size;
    buildMinHeap(minHeap);

    return minHeap;
}

// The main function that builds Huffman tree
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int
size)

{
    struct MinHeapNode *left, *right, *top;
```

```
    // Step 1: Create a min heap of capacity
    // equal to size. Initially, there are
    // modes equal to size.
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap)) {

        // Step 2: Extract the two minimum
        // freq items from min heap
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        // Step 3: Create a new internal
        // node with frequency equal to the
        // sum of the two nodes frequencies.
        // Make the two extracted node as
        // left and right children of this new node.
        // Add this node to the min heap
        // '$' is a special value for internal nodes, not used
        top = newNode('$', left->freq + right->freq);

        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }

    // Step 4: The remaining node is the
    // root node and the tree is complete.
    return extractMin(minHeap);
}

// Prints huffman codes from the root of Huffman Tree.
// It uses arr[] to store codes
void printCodes(struct MinHeapNode* root, int arr[], int top)

{

    // Assign 0 to left edge and recur
    if (root->left) {

        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right) {

        arr[top] = 1;
```

```
        printCodes(root->right, arr, top + 1);
    }
    // If this is a leaf node, then
    // it contains one of the input
    // characters, print the character
    // and its code from arr[]
    if (isLeaf(root)) {

        cout<< root->data <<": ";
        printArr(arr, top);
    }
}

// The main function that builds a
// Huffman Tree and print codes by traversing
// the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)

{
    // Construct Huffman Tree
    struct MinHeapNode* root
        = buildHuffmanTree(data, freq, size);

    // Print Huffman codes using
    // the Huffman tree built above
    int arr[MAX_TREE_HT], top = 0;

    printCodes(root, arr, top);
}

// Driver program to test above functions
int main()
{

    char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int freq[] = { 5, 9, 12, 13, 16, 45 };

    int size = sizeof(arr) / sizeof(arr[0]);

    HuffmanCodes(arr, freq, size);

    return 0;
}
```

Hasil

```
f: 0

c: 100

d: 101

a: 1100
```

```
b: 1101
e: 111
```

Kompleksitas waktu: O (nlogn) di mana n adalah jumlah karakter unik. Jika ada n node, extractMin () disebut 2 * (n - 1) kali. extractMin () membutuhkan waktu O (logn) karena memanggil minHeapify (). Jadi, kompleksitas keseluruhan adalah O (nlogn).

5.11 ALGORITMA GREEDY UNTUK EGYPTIAN FRACTION (PECAHAN MESIR)

Setiap pecahan positif dapat direpresentasikan sebagai jumlah pecahan unit unik. Pecahan adalah pecahan satuan jika pembilangnya adalah 1 dan penyebutnya adalah bilangan bulat positif, misalnya 1/3 adalah pecahan satuan. Representasi seperti itu disebut Fraksi Mesir karena digunakan oleh orang Mesir kuno.

Berikut ini beberapa contoh:

```
Egyptian Fraction Representation of 2/3 is 1/2 + 1/6
Egyptian Fraction Representation of 6/14 is 1/3 + 1/11 + 1/231
Egyptian Fraction Representation of 12/13 is 1/2 + 1/3 + 1/12 +
1/156
```

Kita dapat menghasilkan Fraksi Mesir menggunakan Algoritma Greedy. Untuk jumlah tertentu dari bentuk ‘nr / dr’ di mana dr> nr, pertama-tama temukan fraksi unit terbesar, kemudian ulangi untuk bagian sisanya. Sebagai contoh, perhatikan 6/14, pertama-tama kita menemukan plafon 14/6, yaitu, 3. Jadi, fraksi satuan pertama menjadi 1/3, kemudian berulang untuk (6/14 - 1/3) yaitu, 4/42.

Listing Program

Listing berikut dengan bahasa pemrograman C++

Catatan ;

```
// C++ program untuk fraction in Egyptian menggunakan Greedy
// Algorithm
#include <iostream>
using namespace std;

void printEgyptian(int nr, int dr)
{
    // If either numerator or denominator is 0
    if (dr == 0 || nr == 1)
        return;

    // If numerator divides denominator, then simple division
```

```
    // makes the fraction in 1/n form
    if (dr%nr == 0)
    {
        cout << "1/" << dr/nr;
        return;
    }

    // If denominator divides numerator, then the given number
    // is not fraction
    if (nr%dr == 0)
    {
        cout << nr/dr;
        return;
    }

    // If numerator is more than denominator
    if (nr > dr)
    {
        cout << nr/dr << " + ";
        printEgyptian(nr%dr, dr);
        return;
    }

    // We reach here dr > nr and dr%nr is non-zero
    // Find ceiling of dr/nr and print it as first
    // fraction
    int n = dr/nr + 1;
    cout << "1/" << n << " + ";

    // Recur for remaining part
    printEgyptian(nr*n-dr, dr*n);
 }

// Driver Program
int main()
{
    int nr = 6, dr = 14;
    cout << "Egyptian Fraction Representation of "
         << nr << "/" << dr << " is\n ";
    printEgyptian(nr, dr);
    return 0;
      }
```

Output:

```
Egyptian Fraction Representation of 6/14 is
 1/3 + 1/11 + 1/231
```

Algoritma Greedy bekerja karena sebagian kecil selalu direduksi menjadi bentuk di mana penyebut lebih besar dari pembilang dan pembilang tidak membagi penyebut. Untuk bentuk yang dikurangi seperti itu, panggilan rekursif yang disorot dibuat untuk

pembilang yang dikurangi. Jadi panggilan rekursif terus mengurangi pembilang sampai mencapai 1

5.12 ALGORITMA GREEDY UNTUK KOIN-KOIN PECAHAN MINIMUM

Algoritma greedy merupakan algoritma yang digunakan untuk menemukan solusi optimal untuk masalah yang diberikan. Algoritma greedy bekerja dengan menemukan solusi optimal secara lokal (solusi optimal untuk sebagian masalah) pada setiap bagian sehingga menunjukkan solusi optimal Global dapat ditemukan.

Dalam masalah ini, kita akan menggunakan algoritma greedy untuk menemukan jumlah minimum koin /atau pengembalian coin yang dapat disesuaikan dengan jumlah yang diberikan. Untuk ini, kita akan mempertimbangkan semua koin atau catatan yang valid, yaitu denominasi {1, 2, 5, 10, 20, 50, 100, 200, 500, 2000}. Dan kita perlu mengembalikan jumlah koin/uang kertas ini yang kita butuhkan untuk membayar jumlahnya.

Kita dapat mengambil beberapa contoh untuk memahami konteks dengan lebih baik –

Contoh

```
Input: V = 70

Output: 2

Kita membutuhkan koin/uang kertas 50 dan 20 masing-masing 1.

Input: V = 121

Output: 3

Kita membutuhkan koin/uang kertas 100, 20 dan 1 masing-masing 1.
```

Input : V = 1231
Output : 7

Kita membutuhkan 2 coin/kertas **500,** 2 coin/kertas **100 ,** 1 coin/kertas **20 ,** 1 coin/kertas **10 dan 1** 2 coin/kertas

Sehingga jumlahnya adalah : 2+2+1+1+1 = 7

Untuk menyelesaikan masalah tersebut dengan menggunakan algoritma greedy, maka kita akan menemukan denominasi terbesar yang dapat digunakan. selanjutnya kita

akan mengurangi denominasi terbesar dari jumlah dan lagi melakukan proses yang sama sampai jumlahnya menjadi nol. Seperti yang ditunjukkan oleh algoritma berikut ini

Algoritma

1. Inisialisasi hasil sebagai kosong.
2. Temukan denominasi terbesar yang lebih kecil dari V.
3. Tambahkan denominasi yang ditemukan ke hasil. Kurangi nilai denominasi yang ditemukan dari V.
4. Jika V menjadi 0, maka hasil cetak.
5. Ulangi langkah 2 dan 3 untuk nilai V yang baru

Atau

1. Input: jumlah,
2. Inisialisasi koin = 0
3. Temukan denominasi terbesar yang dapat digunakan, yaitu lebih kecil dari jumlah.
4. Tambahkan denominasi dua koin dan kurangi dari Jumlah
5. Ulangi langkah 2 hingga jumlahnya menjadi 0.
6. Cetak setiap nilai dalam koin.
7.

Listing Program

Listing berikut dengan bahasa pemrograman C++

Catatan ;

```
#include <iostream>
using namespace std;

int deno[] = { 1, 2, 5, 10, 20};
int n = sizeof(deno) / sizeof(deno[0]);

void findMin(int V)
{
{
for (int i= 0; i < n-1; i++) {
for (int j= 0; j < n-i-1; j++){
if (deno[j] > deno[j+1])
swap(&deno[j], &deno[j+1]);
}
int ans[V];
for (int i = 0; i <n; i++) {
while (V >= deno[i]) {
```

```
V -= deno[i];
ans[i]=deno[i];
}
}

for (int i = 0; i < ans.size(); i++)
cout << ans[i] << " ";
}

// Main Program
int main()
{
int a;
cout<<"Enter you amount ";
cin>>a;
cout << "Following is minimal number of change for " << a<< " is ";
findMin(a);
return 0;
}
```

Output:

```
Enter you amount: 70
Following is minimal number of change for 70: 20 20 20 10
```

5.13 POHON MERENTANG MINIMUM (*MINIMUM SPANNING TREE*)

Secara umum Ada 3 (tiga) komponen graf, yaitu; (1) titik (vertice), atau noktah atau node atau simpul yang merepresentasikan objek pada suatugraf, (2) sisi (edge) yaitu garis yang menunjukkan titik yang terkait dengan titik tersebut, dan (3) loop atau sisi lain yang menghubungkannya pada diri sendiri

Sesuai dengan **Lemma**: Setiap graf yang terhubung tanpa sirkuit adalah pohon. Sangat mudah untuk menunjukkan bahwa graf G (V, E) dikatakan tree jika memenuhi sifat sifat berikut :

a. G(V,E) terhubung
b. G(V,E) tanpa sirkuit
c. $|E| = |V| - 1$

Sedangkan Pohon merentang (*spanning tree*) dari graf tak berarah yang terhubung G = (V, E), didefinisikan sebagai pohon T yang terdiri dari semua simpul graf G. Jika graf G terputus maka setiap komponen yang terhubung akan memiliki spanning tree Ti, di mana 'i' adalah jumlah komponen yang terhubung, koleksi yang

membentuk hutan merentang (*spanning forest*) dari grafik G dan tidak membentuk siklus. Suatu Graf boleh jadi memiliki banyak pohon merentang.

Jika kita mengaitkan bobot w_i dengan masing-masing tepi grafik edge (tepi) graf, maka pohon merentang mungkin memiliki bobot yang berbeda-beda. Di antara pohon tersebut ada yang terboboti minimum, sehingga sering disebut sebagai *Minimum Spanning Tree (*MST). MST memiliki berbagai macam aplikasi di berbagai bidang sains dan teknologi seperti desain jaringan yang digunakan dalam algoritma yang mendekati masalah Travelling Salesman person, masalah pemotongan minimum multi-terminal dan pencocokan sempurna dengan biaya terboboti minimum.

Aplikasi praktis lainnya adalah: Analisis Cluster Pengenalan tulisan tangan, Segmentasi gambar, Pemodelan jaringan listrik dengan bobot (panjang kabel) minimum, Pemodelan jaringan pipa PDAM, Pemodelan gardu sinyal (Tower) pada perusahaan telekomunikasi, Pemodelan pembangunan jalan raya, digunakan untuk memilih jalur dengan bobot pertimbangan., untuk memperbaiki biaya pembangunan jalan.

Selanjutnya definisi yang ekuivalen juga disampaikan oleh Bazlamacci (2010) yang menyatakan bahwa diberi graf tidak berarah G = (V, E), di mana V menunjukkan himpunan simpul dengan n = $\left|V\right|$ dan E himpunan tepi dengan m = $\left|E\right|$ dan bilangan real wi = w (e) untuk setiap tepi e ε E disebut bobot tepi e, MSTP secara resmi didefinisikan sebagai cara menemukan spanning tree T* pada G, sedemikian rupa sehingga $w(T^*) = min_T \sum_{e\epsilon T} w(e)$ adalah minimum yang diambil alih semua pohon merentang yang mungkin dari G.

Pencarian biaya minimum dari suatu graph sehingga membentuk *minimum spanning treen*ya memiliki syarat seperti berikut ini :

a. Graph harus terhubung
b. Ruasnya punya bobot
c. Graph tidak berarah
d. Tidak membentuk siklus/sirkel

Pendekatan tradisional untuk menyelesaiakan masalah MST adalah menggunakan algoritma greedy. Mayoritas algoritma klasik membangun MST *edge-wise*, menambahkan *edge* kecil yang sesuai dan tidak termasuk yang lebih besar. Sifat algoritma greedy tercermin dari fakta bahwa algoritma memilih *edge* terbaik untuk

dimasukkan ke dalam MST yang tidak menghasilkan siklus dalam sub-grafik yang dibangun sejauh ini, pada setiap tahap atau untuk penghapusan dari MST, menjaga grafik terhubung

Algoritma klasik yang terkenal seperti Boruvka (1926) dan Kruskal (1956) adalah algoritma penyatuan yang terpisah untuk komputasi MST. Prim (1957) mengimplementasikannya dengan binary heap, d-heap dan F-heap. Yao (1975), Cheriton (1976), dan Fredman (1987) memberikan peningkatan baru dari metode klasik ini. Tarjan (1983) menemukan proses konstruksi MST sebagai salah satu model pewarnaan tepi. Karger (1993) dan Karger (1995) mengusulkan algoritma waktu hampir linier pertama yang diketahui dengan bantuan algoritma acak dan rekursi. Survei yang baik dari semua metodologi ini dapat ditemukan dalam (Bazlamacci, dkk: 2001).

Buku ajar ini hanya membahas 3 Algoritma yang dipakai untuk menentukan minimum spanning tree :

a. Algoritma Prim
b. Algoritma Kruskal
c. Algoritma Boruvka

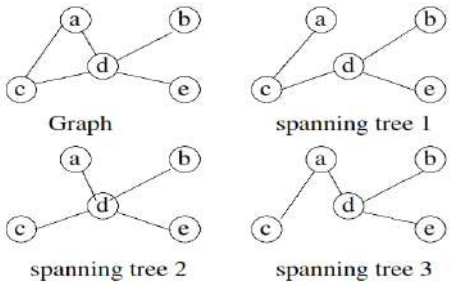
Beberapa istilah yang perlu diketahui dalam minimum spanning tree ini yaitu :

a. Graf terboboti: Graf terboboti adalah suatu graf di mana setiap sisi memiliki bobot (beberapa bilangan real).
b. Bobot suatu graf adalah : Jumlah bobot semua sisi..
c. Biaya spanning tree adalah jumlah dari bobot semua ujung di pohon. Mungkin ada banyak pohon yang merentang.
d. Minimum spanning tree adalah spanning tree di mana biayanya minimum di antara semua spanning tree. Mungkin juga ada banyak pohon merentang minimum.

Berikut beberapa contoh Graf dan spanning tree nya :

Contoh 9 :

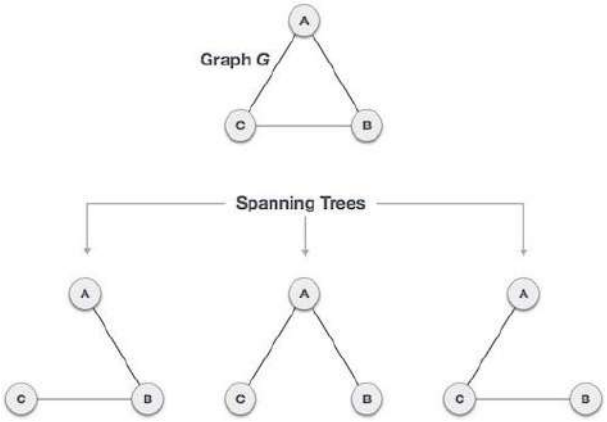
Graf dengan 4 simpul (node) yang belum terboboti, dimana dapat dilihat sesuai dengan definisinya maka ada 3 spanning tree yang dapat dibentuk dari graf pada gambar 5.8



Gambar 5.8. Graf dan spanning tree yang bisa dibentuk

Selanjutnya gambar 5.9 menunjukkan bahwa ada tiga spanning tree dari satu graf lengkap yaitu graf sederhana yang setiap titiknya terhubung ketitik yang lain. Grafik lengkap yang tidak diarahkan dapat memiliki jumlah pohon spanning maksimum n^{n-2}, di mana n adalah jumlah node. Dalam contoh 10 dengan n = 3, maka ada $3^{3-2} = 3$ spanning tree dimungkinkan

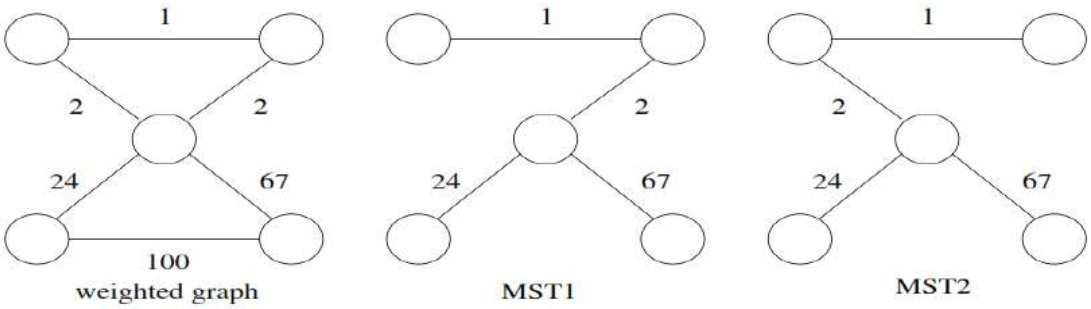
Contoh 10



Gambar 5.9. Graf dan spanning tree yang bisa dibentuk

Contoh 12

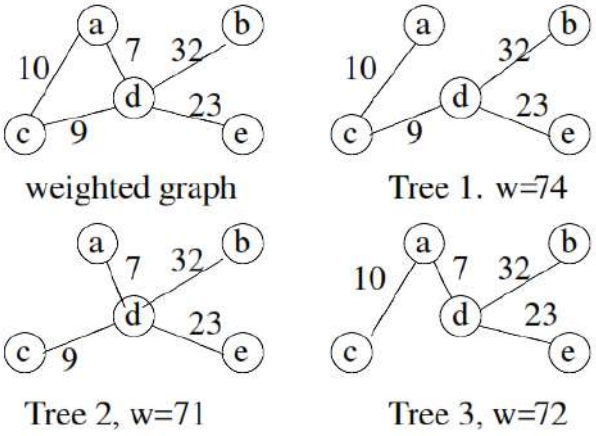
MST mungkin tidak unik jika bobot semua sisi berbeda secara berpasangan, namun akan unik, seperti Graf terboboti pada gambar 5.10, dapat dilihat bahwa ada dua sisi yang sama maka MST yang terbentuk boleh jadi memiliki bobot yang sama.



Gambar 5.10 Graf terboboti dan spanning tree dengan bobo MST-nya bernilai sama

Contoh 11

Graf dengan 5 simpul (node) yang sudah terboboti, dimana dapat dilihat sesuai dengan definisinya maka ada 3 spanning tree yang dapat dibentuk dari graf pada gambar 5.11



Gambar 5.11 Graf terboboti dan spanning tree yang bisa dibentuk

Gambar 5.11 dapat diambil kesimpulan terkait dengan MST nya, bahwa yang memiliki bobot minimum adalah spanning tree 2, dimana total bobotnya adala 71.

Kesimpulan yang dapat diambil dari uraian sebelumnya adalah bahwa satu graf dapat memiliki lebih dari satu spanning tree. Berikut adalah beberapa sifat dari spanning tree yang terhubung ke grafik G -

1. Grafik terhubung G dapat memiliki lebih dari satu Spanning tree.
2. Semua Spanning tree yang mungkin dari grafik G, memiliki jumlah sisi dan simpul yang sama.
3. Spanning tree tidak memiliki siklus (loop).

4. Menghapus satu sisi dari spanning tree akan membuat grafik terputus, misal spanning tree terhubung secara minimal.
5. Menambahkan satu tepi ke spanning tree akan membuat sirkuit atau loop, mis. Spanning tree maksimal asiklik.
6. Pohon rentang memiliki tepi n-1, di mana n adalah jumlah simpul (simpul).
7. Dari grafik yang lengkap, dengan menghapus batas e - n + 1 maksimum, kita dapat membuat spanning tree.
8. Graf lengkap dapat memiliki maksimum nn-2 jumlah spanning tree.

5.13.1 Algoritma Prim

Algoritma Prim merupakan algoritma greedy yang terkenal. Algoritma prims adalah suatu algoritma di dalam teori graph yang menemukan suatu *minimum spanning tree* untuk suatu graph yang terhubung dan terboboti serta tidak berarah. Metode ini menemukan suatu subset dari *edge* yang membentuk suatu pohon yang melibatkan tiap-tiap vertex, di mana total bobot dari semua edge di dalam tree diperkecil.

Jika graph tidak terhubung, maka akan hanya menemukan suatu *minimum spanning tree* untuk salah satu komponen yang terhubung.

Algoritma bekerja sebagai berikut:

Langkah-1:

a. Pilih titik atau node sebarang mana saja secara acak.
b. Node yang terhubung dicari yang melalui lintasan yang memiliki bobot paling minimal untuk dipilih.

Langkah-2:

a. Dapatkan semua lintasan yang menghubungkan node ke node baru.
b. Jika node yang dipilih lintasannya tersebut menghasilkan siklus, maka tolak lintasan tersebut dan cari lintasan berikutnya yang paling minimum bobotnya.

Langkah-3:

Mengulangi langkah-2 sampai semua simpul dikunjungi dan Minimum Spanning Tree (MST) diperoleh.

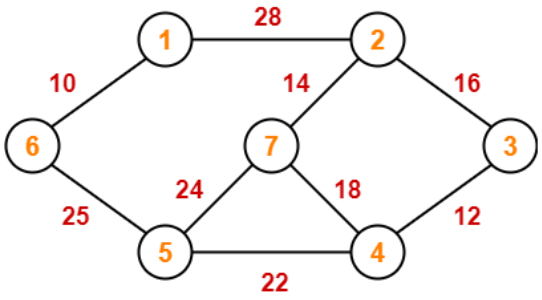
Strategi *greedy* yang digunakan: Pada setiap langkah, pilih sisi dari graf G yang mempunyai bobot minimum dengan syarat sisi tesrebut tetap terhubung dengan pohon merentang minimum T yang telah terbentuk.

```
procedure Prim(input G : graf, output T : pohon)
{ Membentuk pohon merentang minimum T dari graf terhubung G.
  Masukan: graf-berbobot terhubung G = (V, E), yang mana |V|= n
  Keluaran: pohon rentang minimum T = (V, E')
}
Deklarasi
  i, p, q, u, v : integer

Algoritma
  Cari sisi (p,q) dari E yang berbobot terkecil
  T ← {(p,q)}
  for i←1 to n-2 do
    Pilih sisi (u,v) dari E yang bobotnya terkecil namun bersisian
    dengan suatu simpul di dalam T
    T ← T ∪ {(u,v)}
  endfor
```

Kompleksitas waktu asimptotik: $O(n^2)$.

Contoh 13

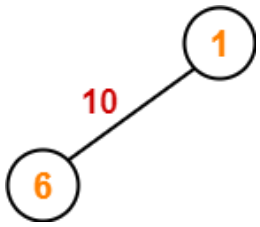
Perhatikan graf terhubung dan terboboti berikut :



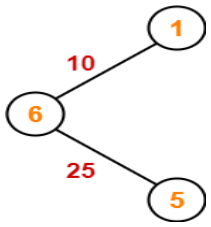
Carilah MST untuk kasus graf di atas.

Penyelesaian :

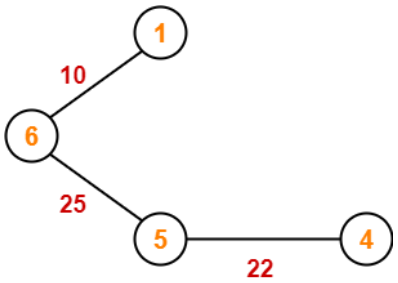
Langkah pertama adalah memilih sebarang node, atau memilih node yang menuju ke node lain dengan lintasan terpendek. Misalnya untuk kasus ini dapat dilihat bahwa node 1 terhubung ke node 6 dan node 2. Jika hendak dipilih node yang lintasannya terpendek ke node lain maka bisa dipilih node 1 ke node 6 yang panjang lintasannya adalah 10. Sehingga :



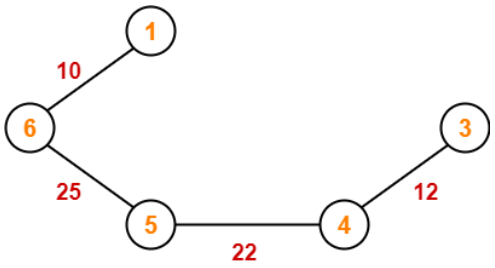
Langkah Berikutnya dapat dilihat bahwa dari node 1 dan node 6. Jika dilihat node 1 terhibung ke node 2 dengan panjang lintasan 28, dan node 6 terhubung ke node 5 dengan panjang lintasan 25, artinya bahwa 25<28 berarti yang diambil adalah node 6 ke node 5.



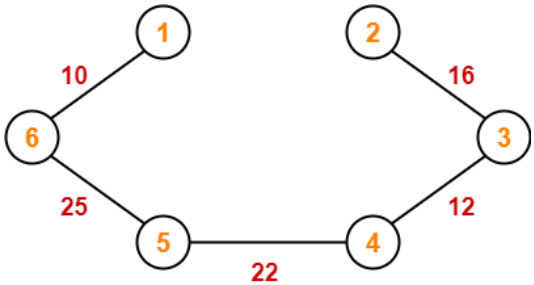
Selanjutnya antar 3 node tersebut dicek lagi dan diperoleh bahwa node 5 ke node 4 memiliki lintasan terpendek dibanding node 1 ke node 4, artinya yang diambil berikutnya adalah node 5 ke node 4.



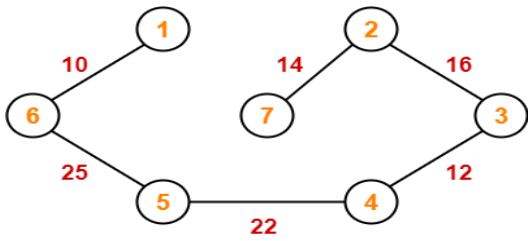
Selanjutnya node 4 ke node 3 dan node 7 dan node 1 ke node, maka didapatkan bahwa node 4 ke node 3 memiliki lintasan terpendek dibanding yang lain sehingga :



Berikutnya nodenya masih tersisa yaitu lintasan node 3 ke 2 dibanding node 1 ke node 2, terlihat bahwa node 3 ke node 2 memiliki lintasan terependek sehingga :



Selanjutnya dapat dilihat node 2 ke node 7 dan node 2 ke node 1, maka dapat dilihat bahwa node 2 ke node 7 memiliki lintasan terpendek dan diperoleh rutenya :



Karena semua simpul telah dimasukkan dalam MST, maka pencarian berhenti dengan lintasan minimumnya adalah dengan jalur nodenya : 1 -6 – 5 - 4 -3-2 - 7.

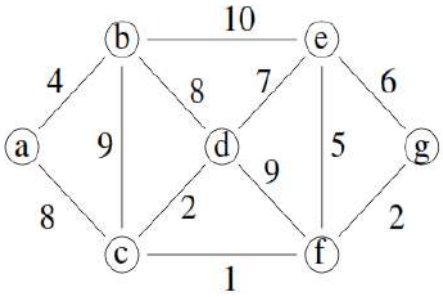
Biaya Spanning Tree Minimum = Jumlah semua bobot lintasan

= 10 + 25 + 22 + 12 + 16 + 14

= 99 unit

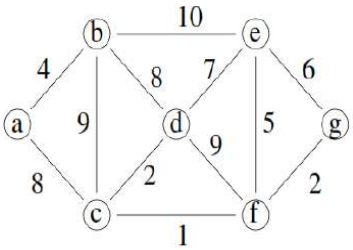
Contoh 14

Jika diketahui grfa berikut ini :

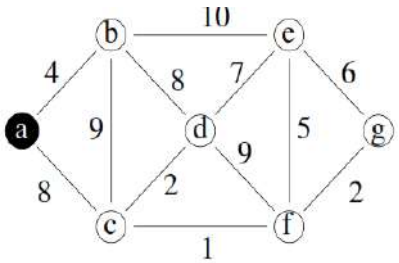


Tentukan MST nya.

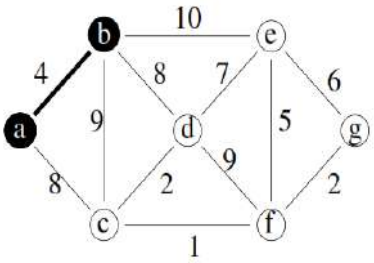
Penyelesaian.



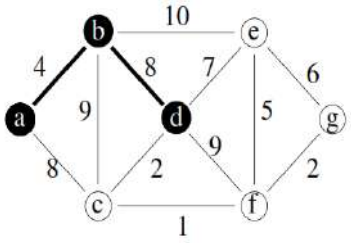
Graf yang ditampilkan terdapat 7 node, dimana akan dipilih secara acak node yang akan dijadikan node awal



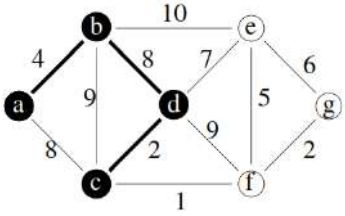
Node yang dipilih adalah **node a**, artinya bisa saja dipilih node c yang mempunyai lintasan terpendek ke f, namun untuk kasus ini kita akan coba pilih node sebarang untuk node awalnya dan dalam hal ini adalah node a



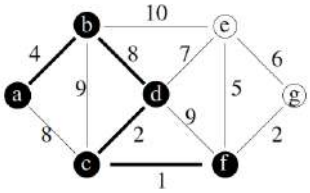
Jika dilihat gambar disamping ada dua lintasan dari node a yaitu node a ke node b, dan node a ke c, maka yang memiliki lintasan terpendek adalah node a ke node b. Jadi lintasannya menjadi {a,b}



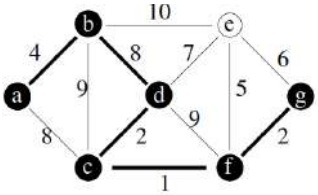
Node b ke c, node b ke d, maka yang lintasannya terpendek adalah node b ke d, sehingga lintasannya terpendeknya adalah node b ke d. Lintasannya menjadi {a,b}, {b, d}



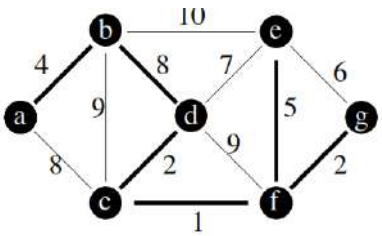
{d,c }=2 ; {d,f}= 9, jadi yang dipilih adalan node ke node f. Sehingga lintasan sementara menjadi {a,b}, {b, d}, {d, c}



{c,f}=1, {d,f} =9, maka lintasan terpendek adalah node c ke f. Sehingga lintasannya menjadi {a,b}, {b, d}, {d, c}, {c, f}



{f,g}=2, {f,d} =9, maka lintasan terpendek adalah node f ke g. Sehingga lintasannya menjadi {a,b}, {b, d}, {d, c}, {c, f}{f,g}



{f,e}=5, {f,d} =9, {g,e} =6, maka lintasan terpendek adalah node f ke e. Sehingga lintasannya menjadi :
MST =[{a,b}, {b, d}, {d, c}, {c, f}{f,g}, {f, e}]
Bobot MST lengkap =4+8+2+1+5+2=22

Contoh listing program untuk Algoritma Prim adalah :

```
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <bits/stdc++.h>
```

```
using namespace std;

// Number of vertices in the graph
#define V 5
// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout<<"Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices not yet included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
    {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
```

```
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}

// Driver code
int main()
{
    /* Contoh graph berikut:
        2      3
    (0)----(1)----(2)
    |     / \      |
   6|   8/   \5    |7
    |   /     \   /
    (3)---------(4)
          9         */
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    // Print the solution
    primMST(graph);

    return 0;
}
```

5.13.2 Algoritma Kruskal

Algoritma Kruskal dimulai dengan *forest* (hutan) yang terdiri dari n pohon. Setiap pohon hanya terdiri dari satu simpul dan tidak ada yang lain. Dalam setiap langkah algoritma, dua pohon berbeda dari hutan ini terhubung ke pohon yang lebih besar. Oleh karena itu, kami terus memiliki pohon yang lebih sedikit dan lebih besar di

forest kita sampai kita berakhir di pohon yang merupakan pohon genetik minimum (mgt). Dalam setiap langkah kita memilih sisi dengan biaya paling sedikit, yang berarti bahwa kita masih di bawah kebijakan serakah. Jika sisi yang dipilih menghubungkan simpul-simpul yang termasuk dalam pohon yang sama sisi ditolak, dan tidak diperiksa lagi karena dapat menghasilkan lingkaran yang akan menghancurkan pohon kita. dan ini kami masukkan menghubungkan dua pohon kecil ke yang lebih besar

Salah satu algoritma *greedy* yang dapat digunakan untuk menangani problem TSP ini adalah ***Kruskal's Algorithm***, dimana algoritma ini akan mencari sirkuit Hamilton minimum. Berikut adalah algoritma *Kruskal* .

Strategi *greedy* yang digunakan: Pada setiap langkah, pilih sisi dari graf G yang mempunyai bobot minimum tetapi sisi tersebut tidak memebentuk sirkuit di T.

```
procedure Kruskal(input G : graf, output T : pohon)
{ Membentuk pohon merentang minimum T dari graf terhubung G.
  Masukan: graf-berbobot terhubung G = (V, E), yang mana |V|= n
  Keluaran: pohon rentang minimum T = (V, E')
}
Deklarasi
  i, p, q, u, v : integer
Algoritma
  {Asumsi: sisi-sisi dari graf sudah diurut menaik berdasarkan
  bobotnya }
  T ← {}
  while jumlah sisi di dalam T < n-1 do
    Pilih sisi (u,v) dari E yang bobotnya terkecil
    if (u,v) tidak membentuk siklus di T then
       T ← T ∪ {(u,v)}
    endif
  endfor
```

Kompleksitas asimptotik $O(|E| \log |E|)$ dengan $|E|$ adalah jumlah sisi di dalam graf *G*.

Langkah-1:

Semua lintasan yang ada diurutkan mulai dari yang terkecil ke yang paling besar.

Langkah-2:

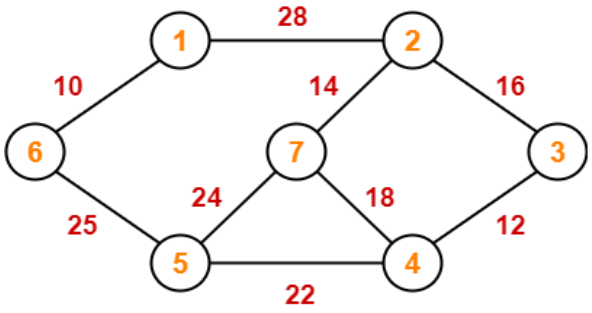
a. Ambil ujung dengan bobot terendah dan gunakan untuk menghubungkan simpul graf.
b. Jika menambahkan lintasan menciptakan siklus, maka tolak lintasan itu dan lanjutkan untuk bobot paling kecil berikutnya.

Langkah-3:

Terus tambahkan lintasan sampai semua simpul terhubung atau dikunjungi tanpa membentuk siklus dan diperoleh Pohon Spanning Minimum (MST)-nya

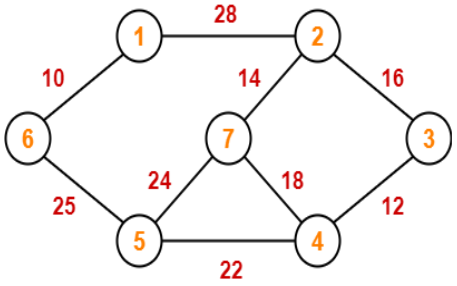
Contoh 15 :

Tentukan MST dengan menggunakan algoritma Kruskal dari graf berikut ini :

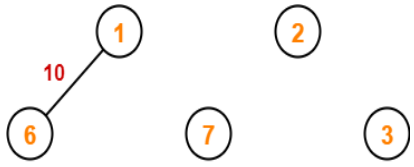


Penyelesaian

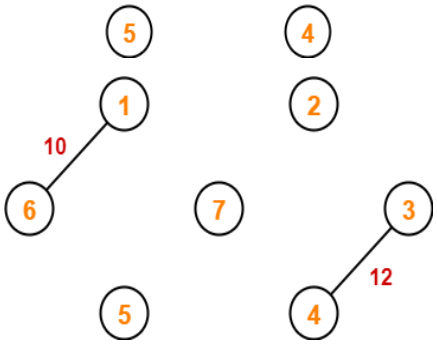
Berdasarkan data-data tersebut maka :



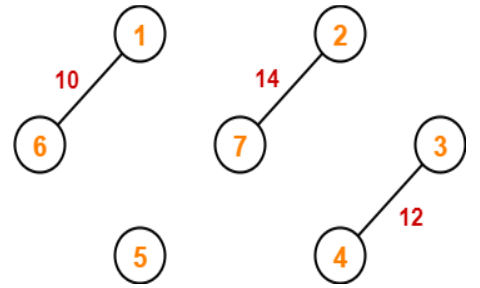
Berdasarkan Graf tersebut maka urutan lintasan dari yang terpendek ke yang terbesar :[{1,6} = 10 ; {3,4}=12 ; {2,7}=14 ; {2,3}=16 ; {4,7}=18 ; {4,5}=22 ; {5,7}=24 ; {5,6}=26 ; {1,2}=28]



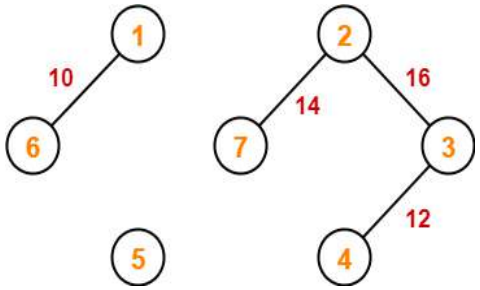
Pilih ujung 1-6: sebagai lintasan terpendek pertama juga karena tidak ada siklus yang terbentuk.



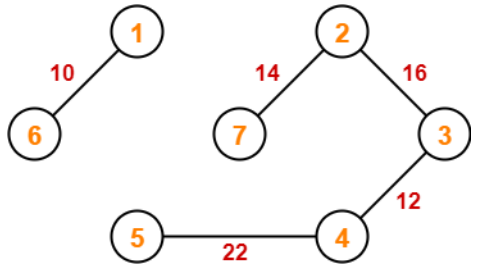
Pilih ujung 3-4: sebagai lintasan terpendek berikutnya juga karena tidak ada siklus yang terbentuk



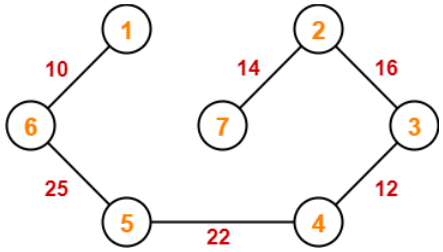
Pilih ujung 2-7: sebagai lintasan terpendek berikutnya juga karena tidak ada siklus yang terbentuk



Pilih ujung 2-3: sebagai lintasan terpendek berikutnya juga karena tidak ada siklus yang terbentuk



Pilih ujung 4-5: sebagai lintasan terpendek berikutnya juga karena tidak ada siklus yang terbentuk, {4,7} tidak dipilih karena akan membentuk siklus



Pilih ujung 5-6: sebagai lintasan terpendek berikutnya juga karena tidak ada siklus yang terbentuk, {4,7} tidak dipilih karena akan membentuk siklus

Karena semua simpul telah dimasukkan dalam MST, maka pencarian berhenti dengan lintasan minimumnya adalah dengan jalur nodenya : 1 -6 – 5 - 4 -3-2 - 7.

Biaya Spanning Tree Minimum = Jumlah semua bobot lintasan

= 10 + 25 + 22 + 12 + 16 + 14

= 99 unit

Baik algoritma Prim maupun algoritma Kruskal, keduanya selalu berhasil menemukan pohon merentang minimum. Lintasan Terpendek (*Shortest Path)*.

Listing program dengan menggunakan bahasa pemrograman C dapat dilihat seperti berikut ini :

```
// C program for Kruskal's algorithm to find Minimum Spanning Tree
```

```
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected
// and weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;

    graph->edge = new Edge[E];

    return graph;
}

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}
```

```
// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V];  // Tnis will store the resultant MST
    int e = 0;  // An index variable, used for result[]
    int i = 0;  // An index variable, used for sorted edges

    // Step 1:  Sort all the edges in non-decreasing
    // order of their weight. If we are not allowed to
    // change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V ssubsets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }
```

```
    // Number of edges to be taken is equal to V-1
    while (e < V - 1 && i < graph->E)
    {
        // Step 2: Pick the smallest edge. And increment
        // the index for next iteration
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge does't cause cycle,
        // include it in result and increment the index
        // of result for next edge
        if (x != y)
        {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
        // Else discard the next_edge
    }

    // print the contents of result[] to display the
    // built MST
    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest,
                                                  result[i].weight);
    return;
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
             10
        0--------1
        |  \     |
       6|   5\   |15
        |      \ |
        2--------3
            4       */
    int V = 4;  // Number of vertices in graph
    int E = 5;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;
```

```
    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;
}
```

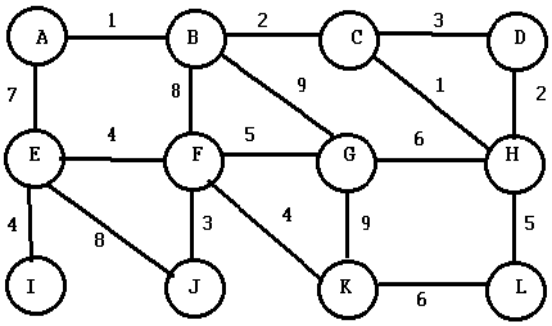
5.13.3 Algoritma Boruvka

Algoritma Boruvka adalah salah satu algoritma yang juga dapat digunakan untuk menemukan pohon rentang minimum - pohon rentang di mana jumlah bobot tepi diminimalkan. Ini adalah algoritma pertama yang dikembangkan (pada tahun 1926) untuk menemukan MST. Otakar Boruvka menggunakannya untuk menemukan rute yang paling efisien untuk jaringan listrik.

Algoritma Boruvka juga merupakan algoritma greedy yang mirip dengan algoritma Kruskal dan algoritma Prim. Algoritma ini pada dasarnya merupakan persilangan antara kedua algoritma dan juga dikenal sebagai algoritma Sollin, terutama dalam komputasi.

Contoh Algoritma Boruvka

Temukan MST untuk graf berikut menggunakan Algoritma Buruvka.



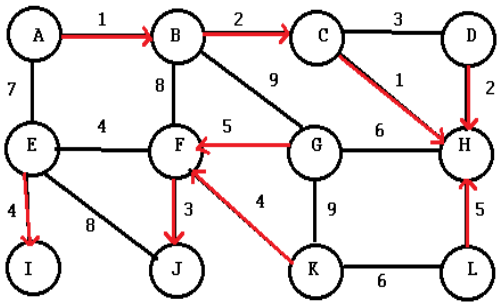
Langkah 1: Tulis daftar komponen. Untuk graf diatas, kita memiliki node/simpul : A, B, C, D, E, F, G, H, I, J, K, L. Langkah ini opsional tetapi membantu Anda melakukan pelacakan.

Langkah 2: Warnai lintasan/tepi yang memiliki bobot termurah untuk setiap node dalam daftar tersebut. Misalnya, simpul A memiliki tepi dengan bobot 1 dan 7, jadi yang dipilih/diwarnai adalah bobot 1. Lanjutkan secara berurutan (untuk daftar ini, buka B, lalu C…dst).

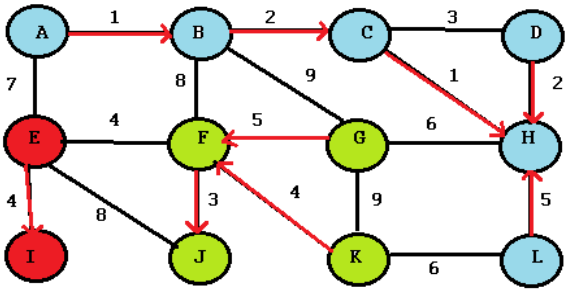
Pedoman:

Pada tahap ini, hanya sorot satu sisi termurah untuk setiap node. Untuk dasi (simpul E memiliki dua sisi dengan bobot 4), tetapkan bobot yang lebih rendah ke salah satu sisi. Ini dilakukan secara acak, tetapi harus tetap konsisten selama proses berlangsung. Untuk contoh ini, disini akan dibuat ujung ke kiri menjadi bobot terendah.

Sorot selalu sisi yang termurah, meskipun telah disorot sebelumnya. Jangan memilih ujung dengan bobot terendah berikutnya! Untuk tujuan praktis, ini berarti anda mungkin tidak perlu menyorot apa pun untuk beberapa node, jika bobot termurah sudah disorot.

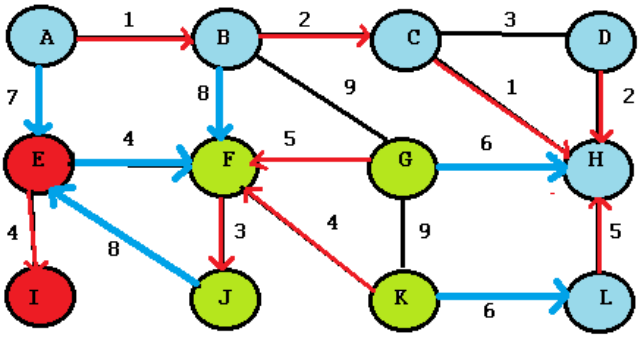


Langkah 3: Sorot kelompok pohon yang terpisah. Ini adalah set node yang terhubung, yang akan kita sebut komponen.



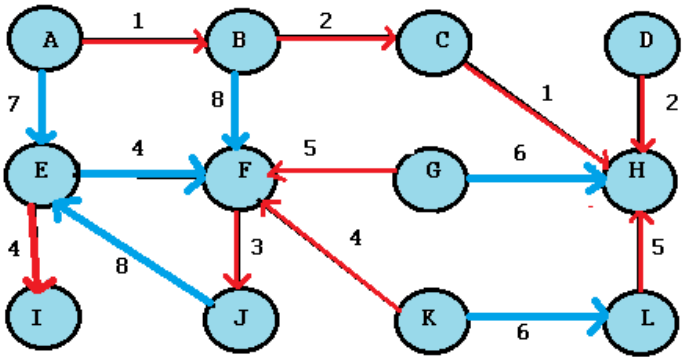
Langkah 4: Ulangi algoritma untuk setiap komponen (masing-masing set berwarna berbeda). Kali ini, untuk setiap node, pilih tepi termurah di luar komponen. Sebagai contoh, ABCDHL adalah satu komponen (titik biru). Untuk simpul A, tepi termurah di luar komponen adalah simpul 7 (karena simpul 1 terhubung di dalam komponen). Saya menyoroti dengan warna biru untuk kejelasan: Anda dapat menggunakan warna apa pun yang Anda suka.

Pedoman: Pastikan Anda memilih tepi termurah di luar komponen. Untuk grafik ini dan iterasi ini, tepinya berwarna hitam. Seperti dalam iterasi pertama, lewati menyoroti tepi jika sudah disorot kali ini.



Langkah 5: Jika semua node Anda terhubung dalam satu komponen, Anda selesai. Jika tidak, ulangi langkah 4.

Grafik berikut menunjukkan MST final. Saya telah menghapus setiap tepi (hitam) yang tidak digunakan.



Fakta Menarik tentang algoritma Boruvka:

1. Kompleksitas Waktu dari algoritma Boruvka adalah O (E log V) yang sama dengan algoritma Kruskal dan Prim.

2. Algoritma Boruvka digunakan sebagai langkah dalam algoritma acak yang lebih cepat yang bekerja dalam waktu linear O (E).
3. 1Algoritma Boruvka adalah algoritma spanning tree minimum tertua yang ditemukan oleh Boruuvka pada tahun 1926, jauh sebelum komputer ada. Algoritma ini diterbitkan sebagai metode membangun jaringan listrik yang efisien.

Atau :

1) Input adalah grafik yang terhubung, tertimbang, dan tidak diarahkan.
2) Inisialisasi semua simpul sebagai komponen individu (atau set).
3) Inisialisasi MST sebagai kosong.
4) Meskipun ada lebih dari satu komponen, lakukan hal berikut :

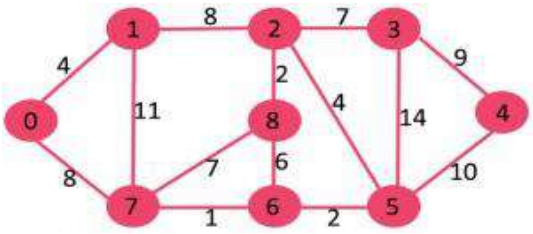
   untuk setiap komponen.

   a. Temukan ujung berat terdekat yang menghubungkan ini komponen ke komponen lain.
   b. Tambahkan tepi terdekat ini ke MST jika belum ditambahkan.

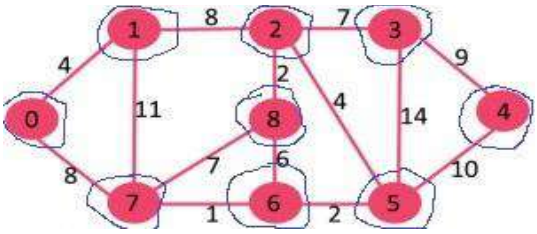
5) Kembalikan MST.

Pohon spanning berarti semua simpul harus terhubung. Jadi dua himpunan bagian yang terpisah (dibahas di atas) dari simpul harus terhubung untuk membuat Pohon Spanning. Dan mereka harus terhubung dengan batas bobot minimum untuk membuatnya menjadi Pohon Rentang Minimum.

Mari kita pahami algoritma dengan contoh di bawah ini.



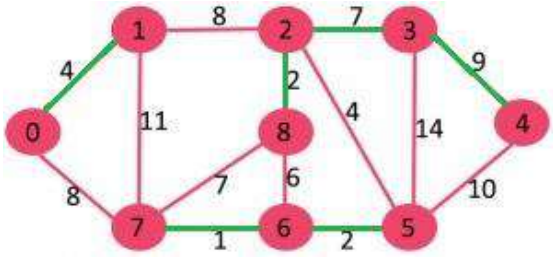
Awalnya MST kosong. Setiap simpul adalah komponen tunggal seperti yang disorot dalam warna biru pada diagram di bawah ini.



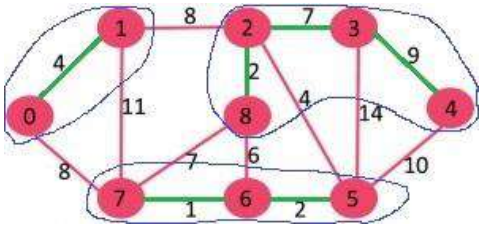
Untuk setiap komponen, temukan tepi termurah yang menghubungkannya ke beberapa komponen lain.

komponent	**lintasan termurah yang menghubungkannya ke beberapa komponen lain.**
{0}	0-1
{1}	0-1
{2}	2-8
{3}	2-3
{4}	3-4
{5}	5-6
{6}	6-7
{7}	6-7
{8}	2-8

Tepi termurah disorot dengan warna hijau. Sekarang MST menjadi {0-1, 2-8, 2-3, 3-4, 5-6, 6-7}



Setelah langkah di atas, komponennya adalah {{0,1}, {2,3,4,8}, {5,6,7}}. Komponen dikelilingi oleh warna biru.

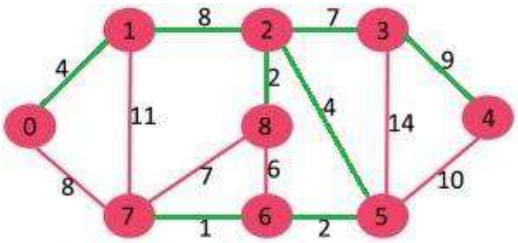


diulangi lagi langkahnya, mis., Untuk setiap komponen, temukan tepi termurah yang menghubungkannya ke beberapa komponen lain.

komponent	**lintasan termurah yang menghubungkannya ke**

	beberapa komponen lain
{0,1}	1-2 (or 0-7)
{2,3,4,8}	2-5
{5,6,7}	2-5

Tepi termurah disorot dengan warna hijau. Sekarang MST menjadi {0-1, 2-8, 2-3, 3-4, 5-6, 6-7, 1-2, 2-5}



Pada tahap ini, hanya ada satu komponen {0, 1, 2, 3, 4, 5, 6, 7, 8} yang memiliki semua sisi. Karena hanya ada satu komponen yang tersisa, kami berhenti dan mengembalikan MST.

Penerapan:

Di bawah ini adalah implementasi dari algoritma di atas. Grafik input direpresentasikan sebagai kumpulan tepi dan struktur data union-find digunakan untuk melacak komponen.

```
// algoritha Boruvka untuk Minimum Spanning Tree (MST)
// Tree of a given connected, undirected and
// weighted graph
#include <stdio.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected
// and weighted graph as a collection of edges.
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    Edge* edge;
};

// A structure to represent a subset for union-find
struct subset
```

```
{
    int parent;
    int rank;
};

// Function prototypes for union-find (These functions are defined
// after boruvkaMST() )
int find(struct subset subsets[], int i);
void Union(struct subset subsets[], int x, int y);

// The main function for MST using Boruvka's algorithm
void boruvkaMST(struct Graph* graph)
{
    // Get data of given graph
    int V = graph->V, E = graph->E;
    Edge *edge = graph->edge;

    // Allocate memory for creating V subsets.
    struct subset *subsets = new subset[V];

    // An array to store index of the cheapest edge of
    // subset.  The stored index for indexing array 'edge[]'
    int *cheapest = new int[V];

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
        cheapest[v] = -1;
    }

    // Initially there are V different trees.
    // Finally there will be one tree that will be MST
    int numTrees = V;
    int MSTweight = 0;

    // Keep combining components (or sets) until all
    // compnentes are not combined into single MST.
    while (numTrees > 1)
    {
         // Everytime initialize cheapest array
          for (int v = 0; v < V; ++v)
           {
               cheapest[v] = -1;
           }

        // Traverse through all edges and update
        // cheapest of every component
        for (int i=0; i<E; i++)
        {
            // Find components (or sets) of two corners
            // of current edge
            int set1 = find(subsets, edge[i].src);
            int set2 = find(subsets, edge[i].dest);
```

```
            // If two corners of current edge belong to
            // same set, ignore current edge
            if (set1 == set2)
                continue;

            // Else check if current edge is closer to previous
            // cheapest edges of set1 and set2
            else
            {
               if (cheapest[set1] == -1 ||
                   edge[cheapest[set1]].weight > edge[i].weight)
                 cheapest[set1] = i;

               if (cheapest[set2] == -1 ||
                   edge[cheapest[set2]].weight > edge[i].weight)
                 cheapest[set2] = i;
            }
        }

        // Consider the above picked cheapest edges and add them
        // to MST
        for (int i=0; i<V; i++)
        {
            // Check if cheapest for current set exists
            if (cheapest[i] != -1)
            {
                int set1 = find(subsets, edge[cheapest[i]].src);
                int set2 = find(subsets, edge[cheapest[i]].dest);

                if (set1 == set2)
                    continue;
                MSTweight += edge[cheapest[i]].weight;
                printf("Edge %d-%d included in MST\n",
                       edge[cheapest[i]].src, edge[cheapest[i]].dest);

                // Do a union of set1 and set2 and decrease number
                // of trees
                Union(subsets, set1, set2);
                numTrees--;
            }
        }
    }

    printf("Weight of MST is %d\n", MSTweight);
    return;
}
// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}
// A utility function to find set of an element i
```

```
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
      subsets[i].parent =
             find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
             10
        0--------1
        |  \     |
       6|   5\   |15
        |      \ |
        2--------3
            4       */
    int V = 4;  // Number of vertices in graph
    int E = 5;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
```

```
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    boruvkaMST(graph);

    return 0;
}
```

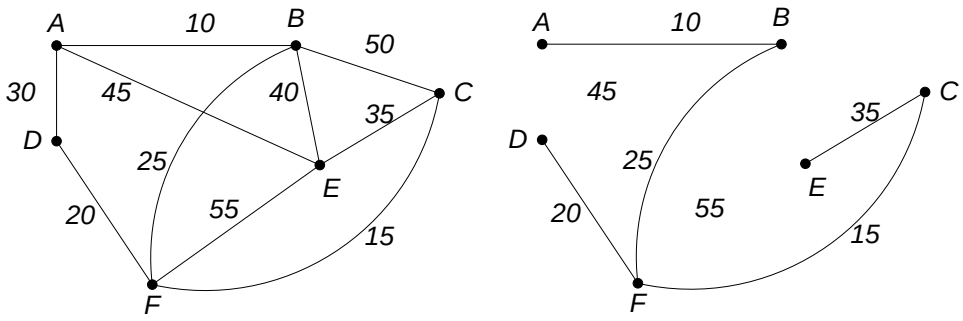
5.14 RANGKUMAN

1. Algoritma *Greedy* adalah suatu algoritma yang menyelesaikan masalah secara step by step sehingga ketika pada satu langkah telah diambil keputusan maka pada langkah selanjutnya keputusan itu tidak dapat diubah lagi. Jadi algoritma ini menggunakan pendekatan untuk mendapatkan solusi lokal yang optimum dengan harapan akan mengarah pada solusi global yang optimum. Dengan kata lain algoritma *greedy* tidak dapat menjamin solusi global yang optimum.
2. Penerapan algoritma *greedy* diantaranya dapat dilihat dalam kasus *Travelling Salesperson Problem (TSP), Minimum Spanning Tree (Pim's)* dan Minimasi Waktu dalam Sistem (*scheduling).* Pada ketiga contoh tersebut, kompleksitas tertinggi terdapat pada algoritma *prim's (Minimum Spanning Tree)* dan diikuti dengan algoritma *kruskal's (TSP)* dan yang terkecil adalah pada *scheduling.*
3. knapsack dengan Greedy Strategi merupakan salah satu cara untuk solusi ooptimum dari masalah knapsack tersebut
4. Solusi optimum dari knapsack Greedy Strategi lebih cepat dan efisien dari cara exhaustive search .
5. Faktor yang terpenting dari Knapsack adalah profit.

6. Pendekatan yang digunakan di dalam algoritma *greedy* adalah membuat pilihan yang "tampaknya" memberikan perolehan terbaik, yaitu dengan membuat pilihan **optimum lokal** (*local optimum*) pada setiap langkah dengan harapan bahwa sisanya mengarah ke solusi **optimum global** (*global optimm*).
7. Pohon merentang (*spanning tree*) dari graf tak berarah yang terhubung G = (V, E), didefinisikan sebagai pohon T yang terdiri dari semua simpul graf G. Jika graf G terputus maka setiap komponen yang terhubung akan memiliki spanning tree Ti, di mana 'i' adalah jumlah komponen yang terhubung, koleksi yang membentuk hutan merentang (*spanning forest*) dari grafik G dan tidak membentuk siklus. Suatu Graf boleh jadi memiliki banyak pohon merentang.
8. Pencarian biaya minimum dari suatu graph sehingga membentuk *minimum spanning treen*ya memiliki syarat seperti berikut ini :
    a. Graph harus terhubung
    b. Ruasnya punya bobot
    c. Graph tidak berarah
    d. Tidak membentuk siklus/sirkel
9. Algoritma Prim merupakan algoritma greedy yang terkenal. Algoritma prims adalah suatu algoritma di dalam teori graph yang menemukan suatu *minimum spanning tree* untuk suatu graph yang terhubung dan terboboti serta tidak berarah. Metode ini menemukan suatu subset dari *edge* yang membentuk suatu pohon yang melibatkan tiap-tiap vertex, di mana total bobot dari semua edge di dalam tree diperkecil
10. Salah satu algoritma *greedy* yang dapat digunakan untuk menangani problem TSP ini adalah ***Kruskal's Algorithm***, dimana algoritma ini akan mencari sirkuit Hamilton minimum.
11. Algoritma Boruvka juga merupakan algoritma greedy yang mirip dengan algoritma Kruskal dan algoritma Prim. Algoritma ini pada dasarnya merupakan persilangan antara kedua algoritma dan juga dikenal sebagai algoritma Sollin, terutama dalam komputasi.

5.15 TEST FORMATIF

1. Jelaskan apa yang anda ketahui tentang algoritma greedy
2. Jelaskan yang dimaksud dengan optimum lokal dan optimum global
3. Jelaskan Perbedaan antara Algoritma Prim dan Algoritma Kruskal
4. Jelaskan yang anda ketahui tentang graph, spanning tree , minimum spanning tree, vertex, edges, edge costs dan node.
5. Diketahui 3 program yang akan disimpan dalam media penyimpanan dengan panjang masing-masing 8, 15, dan 5. Bagaimana proses penyimpanan yang optimal dengan metode greedy.
6. Sebuah kapal akan diisi dengan muatan. Muatan tersebut disimpan di dalam peti kemas dan setiap peti kemas yang sama, tetapi berat peti kemas (yang sudah berisi muatan) berbeda belum tentu sama. Misalkan wiadalah berat peti kemas ke-i, 1 .i≤i≤n. Kapasitas kapal yang membawa muatan adalah C. Kita ingin kapal dengan jumlah yang mungkin terlihat seperti kemi. Rumuskan peluang analisis ini dengan algoritma greedy dengan menjelaskan properti algoritma greedy-nya (himpunan kandidat, himpunan solusi, fungsi seleksi, fungsi kelayakan, dan fungsi obyektif apa, lalu gunakan contoh-contoh aplikasi untuk perjanjian kerjasama sesuai dengan n = 8, w = (100,200,50, 90,150,50,20,80), dan C = 400
7. Gambar berikut merupakan graf yang terboboti dan hasil MST nya. Buktikan bahwa MST adalah seperti ditunjukkan pada bagian (b) dari gambar tersebut



(a) Graf *G* (b) Pohon merentang minimum

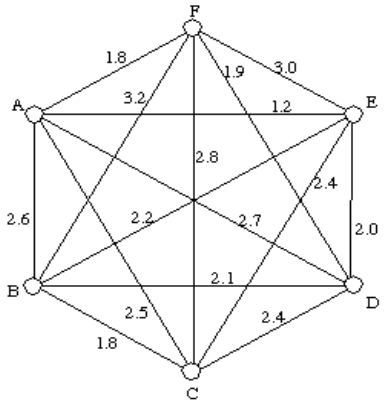
8. Sebuah studio musik membuka layanan sewa studio bagi sejumlah grup *band* anak muda yang ingin latihan di studio tersebut. Grup *band* yang ingin menyewa harus mendaftar dua hari sebelumnya untuk kemudian dijadwalkan. Andaikan studio musik itu hanya buka mulai dari jam 1 sampai jam 14. Setiap grup *band*

yang hendak menyewa harus menuliskan jam mulai dan jam selesai latihan (semua jam adalah bilangan bulat). Berhubung permintaan latihan cukup banyak sementara dalam satu waktu hanya satu grup *band* yang dapat dilayani, maka manajemen studio musik harus memilih dan menjadwalkan grup *band* yang akan menggunakan studionya itu sehingga sebanyak mungkin grup *band* yang dapat dilayani. Misalkan pada hari ini studio musik telah menerima permintaan sewa dari 10 grup *band* sebagai berikut:

Grup *band*	1	2	3	4	5	6	7	8	9	10
Jam Mulai	1	3	2	4	8	7	9	11	9	12
Jam Selesai	3	4	5	7	9	10	11	12	13	14

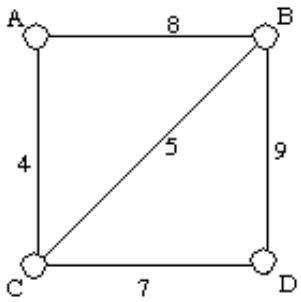
a. Jika persoalan di atas diselesaikan dengan algoritma *greedy*, jelaskan strategi *greedy* yang digunakan untuk memilih grup *band* yang dijadwalkan pada setiap langkah. Buat asumsi jika diperlukan.
b. Dengan strategi *greedy* di atas, selesaikan persoalan ini. Grup *band* mana saja yang dapat dijadwalkan

9. Diketahui graf terboboti berikut ini :



a. Gunakan algoritma Kruskal untuk menemukan pohon rentang minimum dan menunjukkan tepi pada grafik yang ditunjukkan di bawah ini: Tunjukkan pada edge yang dipilih urutan pilihannya.
b. Gunakan algoritma Prim untuk menemukan pohon rentang minimum dan menunjukkan tepi pada grafik yang ditunjukkan di bawah ini. Tunjukkan pada tepi yang dipilih urutan pilihan mereka

10. Diketahui graf berikut ini dengan n=4:



    a. Temukan subgrafnya dan gambarkan subgraf tersebut.
    b. Gambarkan semua spanning tree dari graf tersebut.

11. Tinjau persoalan 0/1 *Knapsack* dengan $n = 4$.

    $w_1 = 6$; $p_1 = 12$

    $w_2 = 5$; $p_1 = 15$

    $w_3 = 10$; $p_1 = 50$

    $w_4 = 5$; $p_1 = 10$

    Kapasitas *knapsack* $W = 16$

    Tentukan solusinya dengan algoritma knapsack by greedy dan exhaustive search

12. Sebuah tas hanya dapat menampung paling banyak 25 unit ruang. Berikut adalah daftar barang dan nilainya.

Item	**Size**	**Price**
Laptop	22	12
PlayStation	10	9
Textbook	9	9
Basketball	7	6

    Berdasarkan daftar barang yang ada, tentukan solunya dengan algoritma greedy dan exhaustive search barang-barang apa saja yang dapat dimasukkan ke dalam tas.

13. Buatlah Algoritma dan Program untuk kasus berikut ini :

    anda diberi wadah penuh air. Wadah dapat memiliki jumlah air yang terbatas. Anda juga memiliki botol N untuk diisi. Anda perlu menemukan jumlah botol maksimum yang dapat anda isi.

Input:

a. Baris pertama berisi satu bilangan bulat, T, jumlah kasus uji.
b. Baris pertama dari setiap test case berisi dua bilangan bulat, N dan X, jumlah botol dan kapasitas wadah.
c. Baris kedua dari setiap test case berisi bilangan bulat yang dipisahkan ruang N, kapasitas botol.

Keluaran:

Untuk setiap test case cetak jumlah botol maksimum yang dapat Anda isi.

Constraints:

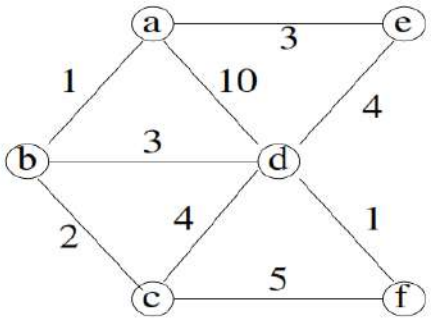
$1 \leq T \leq 100$

$1 \leq N \leq 10^4$

$1 \leq X \leq 10^9$

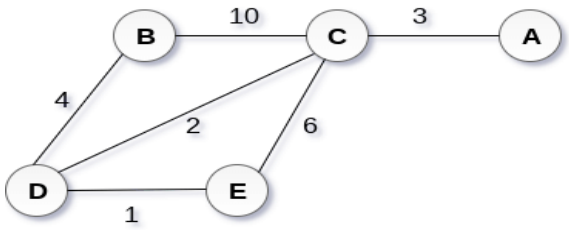
$1 \leq$ kapasitas botol $\leq 10^6$

14. Diketahui graf berikut ini :



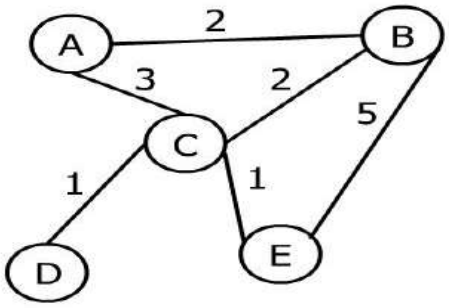
Tentukan MST nya dengan menggunakan Algoritma Prims dan Algoritma Kruskal

15. Diketahui graf berikut ini :



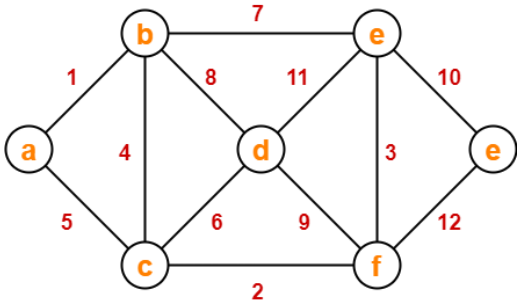
Tentukan MST nya dengan menggunakan Algoritma Prims dan Algoritma Kruskal

16. Diketahui graf berikut ini :



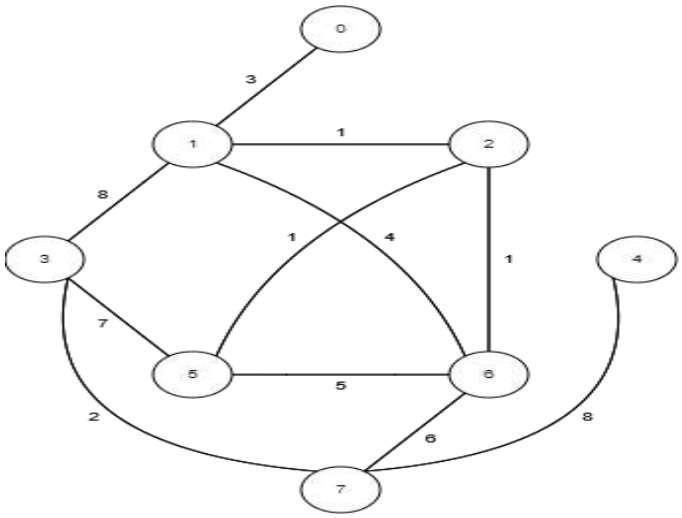
Tentukan MST nya dengan menggunakan Algoritma Prims dan Algoritma Kruskal

17. Diketahui graf berikut ini :



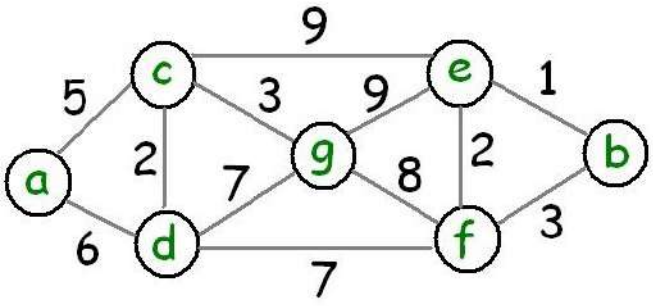
Tentukan MST nya dengan menggunakan Algoritma Prims dan Algoritma Kruskal

18. Diketahui graf berikut ini :



Tentukan MST nya dengan menggunakan Algoritma Prims dan Algoritma Kruskal dan boruvka

19. Diberikan graf berikut ini, kita akan menentukan lintasan terpendek dari simpul a ke semua simpul lainnya.

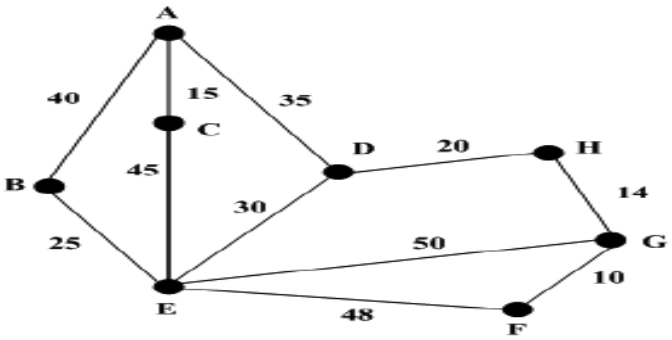


(a) Gunakanlah algoritma Kruskal untuk menentukan lintasan terpendek tersebut. Sebelum mengerjakan, tuliskanlah strategi greedy yang digunakan Kruskal.

(b) Gunakanlah algoritma Dijkstra untuk menentukan lintasan terpendek tersebut. Sebelum mengerjakan, tuliskanlah strategi greedy yang digunakan Dijkstra.

(c) Berikanlah kesimpulan dari hasil (a) dan (b).

20. Hitunglah jarak terpendek dari graph berikut ini dengan menggunakan Algoritma PRIM'S dan KRUSKAL



BAB VI
LINTASAN TERPENDEK (SHORTEST PATH)

a) Deskripsi Singkat

Permasalahan rute terpendek dari sebuah titik ke akhir titik lain adalah sebuah masalah klasik optimasi yang banyak digunakan untuk menguji sebuah algoritma yang diusulkan. Permasalahan rute terpendek dianggap cukup baik untuk mewakili masalah optimisasi, karena permasalahannya mudah dimengerti (hanya menjumlahkan seluruh edge yang dilalui) namun memiliki banyak pilihan solusi.

Persoalan mencari lintasan terpendek di dalam graf merupakan salah satu persoalan optimasi. Graf yang digunakan dalam pencarian lintasan terpendek adalah graf berbobot (weighted graph), yaitu graf yang setiap sisinya diberikan suatu nilai atau bobot. Bobot pada sisi graf dapat menyatakan jarak antar kota, waktu pengiriman pesan, ongkos pembangunan, dan sebagainya. Asumsi yang kita gunakan di sini adalah bahwa semua bobot bernilai positif. Kata terpendek berbeda-beda maknanya bergantung pada tipikal persoalan yang akan diselesaikan. Namun, secara umum terpendek berarti meminimisasi bobot pada suatu lintasan dalam graf.

b) Relevansi

Relevansinya adalah melalui metode ceramah, tanya jawab, duskusi dan praktikum. Berkaitan dengan materi pada Bab 6 ini, kepada mahasiswa akan dijelaskan dengan detail materinya untuk kemudian algoritmanya diimplementasikan ke program melalui praktikum menggunakan bahasa pemrograman C, C++, Java atau python atau bahasa pemrograman yang dikuasi mahasiswa. Kesempatan bertanya juga diberikan dan tugas mandiri sebagai pengayaan materi-materi pada bab 6 ini.

c) Capaian Pembelajaran

1. Mahasiswa mampu menyelesaikan masalah-masalah dengan menggunakan algoritma-algoritma lintas terpendek
2. Mahasiswa dapat menghitung jarak terpendek dengan menggunakan minimum algoritma lintas terpendek

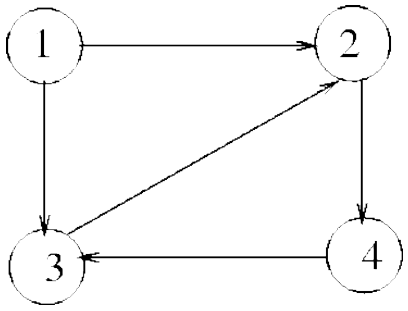
6.1 PENDAHULUAN

Sebuah graf dengan lintasan terboboti di mana ada keterkaitan antara bobot atau biaya dengan masing-masing lintasan. Jalur terpendek dari node S sebagai node awal ke node t adalah jalur langsung dari s ke t dengan sifat-sifat bahwa tidak ada jalur lain yang memiliki bobot lebih rendah

Graf terarah atau digraf terdiri :

1. satu set simpul V
2. satu set tepi atau busur terarah, E (busur adalah pasangan simpul berurutan)

Sebagai contoh dapat dilihat gambar 6.1



Gambar 6.1 Graf berarah dengan 4 node dan 5 tepi atau busur

Gambar 6.1 menunjukan bahwa V = {1, 2, 3, 4} dan E = {(1,2), (1,3), (2,4), (3,2), (4,3)}

Ada beberapa terkait dengan graf berarah :

a. Jika ada busur (v, w), dapat dikatakan bahwa w berbatasan dengan v.
b. Jalur /lintasan adalah urutan simpul v1, v2, ... vn sedemikian rupa sehingga pasangan simpul (v1, v2), (v2, v3), ..., (vn - 1, vn) adalah busur dalam graf.
c. Panjang lintasan adalah jumlah busur di jalan.
d. Sebuah simpul tunggal v dengan sendirinya menunjukkan jalur dengan panjang nol.
e. Jalur v1, v2,…,vn dikatakan sederhana jika simpulnya berbeda, kecuali kemungkinan v1 dan vn.
f. Jika v1 = vn dan n> 1, maka jalur seperti itu disebut sebagai siklus sederhana.

Masalah jalur terpendek adalah tentang menemukan jalur antara 2 simpul dalam graf sehingga diperoleh jumlah total bobot tepi/lintasannya adalah minimum.

Masalah ini dapat diselesaikan dengan mudah menggunakan (BFS) jika semua bobot tepi (1), tetapi di sini bobot dapat bernilai berapaun. Tiga algoritma berbeda dibahas di bawah ini tergantung pada use-case. Ketiga Algoritma tersebut adalah :

1. Algoritma Djikstra

2. Bellman Ford's Algorithm
3. Floyd–Warshall's Algorithm

6.2 ALGORITMA DJIKSTRA

Banyak algoritma-algoritma lain yang dapat digunakan untuk mencari rute atau lintasan terpendek seperti yang telah banyak dibahas dalam Subbab 5.10. namun tidak bisa dinafikan bahwa Algoritma Djikstra menjadi algoritma pencarian rute terpendek yang paling popular.

Algortima ini ditemukan oleh Edsger W. Dikstra dan di publikasi pada tahun 1959 pada sebuah jurnal Numerische Mathematik yang berjudul "*A Note on Two Problems in Connexion with Graphs*"[Djikstra]. Algoritma ini sering digambarkan sebagai algoritma greedy (tamak). Sebagai contoh, ada pada buku *Algorithmics* (Brassard and Bratley [1988, pp. 87-92]). Walaupun ditemukan pada sekitar tahun 1959, algoritma djikstra ini masih tetap popular dan banyak diimplementasikan dalam berbagai bidang.

Algoritma Dijkstra memiliki banyak varian tetapi yang paling umum adalah menemukan jalur terpendek dari titik sumber ke semua titik lainnya dalam graf yang terboboti yaitu graf yang setiap sisinya atau bususrnya atau tepinya diberi nilai atau bobot. Bobot pada sisi graf dapat menyatakan jarak antar kota, waktu pengiriman pesan, ongkos pembangunan, dan sebagainya. Asumsi yang digunakan di sini adalah bahwa semua bobot bernilai positif. Jadi tidak dapat dilalui oleh node negatif. Namun jika terjadi demikian, maka penyelesaian yang diberikan adalah infiniti (Tak Hingga) karena Node yang digunakan pada Algoritma Dijkstra adalah node graph berarah untuk penentuan rute listasan terpendek. Kata terpendek memiliki banyak arti dan berbeda-beda maknanya bergantung pada tipe permasalahannya yang akan diselesaikan, tetapi, secara umum untuk implementasi di graf terpendek berarti meminimisasi bobot pada suatu lintasan dalam graf.

Algoritma djikstra dalam teori graf dikategorikan kedalam *single-source shortest path problem* dimana dalam penyelesaiannya memiliki persyaratan seperti :

a. Graf berarah dan tidak terarah
b. Semua tepi harus memiliki bobot non-negatif
c. Graf harus terhubung

Algoritma djikstra juga menggunakan pendekatan greedy yang bermakna bahwa ditemukan solusi terbaik berikutnya dengan harapan hasil akhirnya merupakan solusi terbaik untuk seluruh masalah.

Algoritma djikstra bekerja dengan mempertahankan satu set S dari simpul khusus yang jarak terdekatnya dari sumber sudah diketahui. Pada setiap langkah, titik `` non-khusus " diserap ke dalam S. Penyerapan elemen V-S ke S dilakukan dengan strategi greedy. Permasalahan rute terpendek dari sebuah titik ke akhir titik lain adalah sebuah masalah klasik optimasi yang banyak digunakan untuk menguji sebuah algoritma yang diusulkan. Permasalahan rute terpendek dianggap cukup baik untuk mewakili masalah optimisasi, karena permasalahannya mudah dimengerti (hanya menjumlahkan seluruh edge yang dilalui) namun memiliki banyak pilihan solusi.

Menurut **Andrew Goldberg peneliti Microsoft Research Silicon Valley**, mengatakan ada banyak alasan mengapa peneliti terus mempelajari masalah pencarian jalan terpendek. "*Jalan terpendek adalah masalah optimasi yang relevan untuk berbagai macam aplikasi, seperti jaringan routing, game, desain sirkuit, dan pemetaan*".

Diskripsi matematis untuk grafik dapat diwakili *G = {V. E}*, yang berarti sebuah grafik (*G*) didefenisikan oleh satu set simpul (Vertex = *V*) dan koleksi Edge (*E*).

Algoritma Dijkstra bekerja dengan membuat jalur ke satu simpul optimal pada setiap langkah. Jadi pada langkah ke n, setidaknya ada n node yang sudah kita tahu jalur terpendek. Langkah-langkah algoritma Dijkstra dapat dilakukan dengan langkah-langkah berikut:

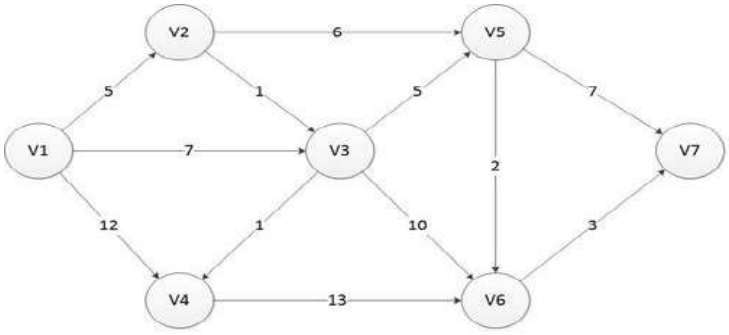
1. Tentukan titik mana yang akan menjadi node awal, lalu beri bobot jarak pada node pertama ke node terdekat satu per satu, Dijkstra akan melakukan pengembangan pencarian dari satu titik ke titik lain dan ke titik selanjutnya tahap demi tahap.
2. Beri nilai bobot (jarak) untuk setiap titik ke titik lainnya, lalu set nilai 0 pada node awal dan nilai tak hingga terhadap node lain (belum terisi) 2.
3. Set semua node yang belum dilalui dan set node awal sebagai "Node keberangkatan"
4. Dari node keberangkatan, pertimbangkan node tetangga yang belum dilalui dan hitung jaraknya dari titik keberangkatan. Jika jarak ini lebih kecil dari

jarak sebelumnya (yang telah terekam sebelumnya) hapus data lama, simpan ulang data jarak dengan jarak yang baru

5. Saat kita selesai mempertimbangkan setiap jarak terhadap node tetangga, tandai node yang telah dilalui sebagai "Node dilewati". Node yang dilewati tidak akan pernah di cek kembali, jarak yang disimpan adalah jarak terakhir dan yang paling minimal bobotnya.
6. Set "Node belum dilewati" dengan jarak terkecil (dari node keberangkatan) sebagai "Node Keberangkatan" selanjutnya dan ulangi langkah e.

Contoh 1

Sebagai contoh lainnya : hitunglah Jarak terdekat dari V1 ke V7 pada gambar 6.2 berikut ini.



Gambar 6.2 Jarak terdekat dari V1 ke V7

Hasil perhitungan jarak terdekat dengan menggunakan Algoritma Djikstra adalah seperti pada table 6.1

Tabel 6.1 Hasil perhitungan Gambar 6.2 dengan Algoritma Djikstra

Iterasi	Unvisited(Q)	Visited(S)	Current	Node : Min = (dist[node], prev[node]) iteration						
				V1	V2	V3	V4	V5	V6	V7
	Inisialisasi {V1,V2,V3,V4,V5,V6,V7}	{-}		(0,-)0	(∞,-)0	(∞,-)0	(∞,-)0	(∞,-)0	(∞,-)0	(∞,-)0
1	{V2,V3,V4,V5,V6,V7}	{V1}	V1		(5,V1)1	(7,V1)1	(12,V1)1	(∞,V1)1	(∞,V1)1	(∞,V1)1
2	{V3,V4,V5,V6,V7}	{V1,V2 }	V2			(6,V2)2	(12,V1)1	(11,V2)2	(∞,V2)2	(∞,V2)2
3	{V4,V5,V6,V7}	{V1,V2,V3 }	V3				(7,V3)3	(11,V3)3	(16,V3)3	(∞,V3)3
4	{V5,V6,V7}	{V1,V2,V3,V4}	V4					(11,V3)3	(16,V3)3	(∞,V3)3
5	{V6,V7}	{V1,V2,V3,V4,V5}	V5						(13,V5)5	(18,V5)5
6	{,V7}	{V1,V2,V3,V4,V5,V6}	V6							(16,V6)6

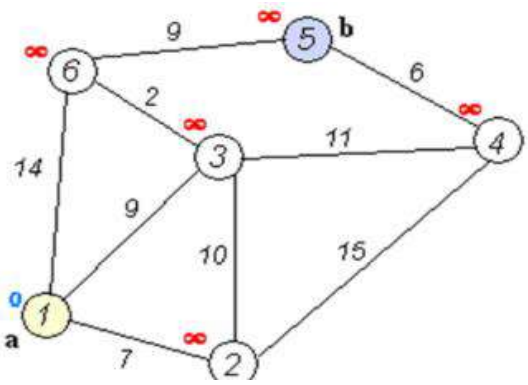
Algoritma Dijkstra diterapkan untuk mencari lintasan terpendek pada graf berarah. Namun, algoritma ini juga benar untuk graf tak berarah. Algoritma Dijkstra mencari lintasan terpendek dalam sejumlah langkah. Algoritma ini menggunakan prinsip greedy. Prinsip greedy pada algoritma dijkstra menyatakan bahwa pada setiap langkah kita memilih sisi yang berbobot minimum dan memasukannya dalam himpunan solusi.

Penerapan Algoritma Dijkstra pada Jaringan Komputer

Jaringan komputer dapat dimodelkan sebagai sebuah graf, dengan setiap simpul menyatakan sebuah komputer/router dan sisi di dalam graf menyatakan saluran komunikasi (sering disebut link). Setiap sisi mempunyai label nilai (yang disebut bobot). Bobot tersebut dapat menyatakan jarak geografis(dalam km), kecepatan transfer data, waktu pengiriman).

Contoh 2:

Node awal 1, Node tujuan 5. Setiap edge yang terhubung antar node dan telah diberi nilai



Gambar 6.3 Contoh keterhubungan antar titik dalam algoritma Dijkstra

Pertama-tama tentukan titik mana yang akan menjadi node awal, lalu beri bobot jarak pada node pertama ke node terdekat satu per satu, Dijkstra akan melakukan pengembangan pencarian dari satu titik ke titik lain dan ke titik selanjutnya tahap demi tahap. Inilah urutan logika dari algoritma Dijkstra:

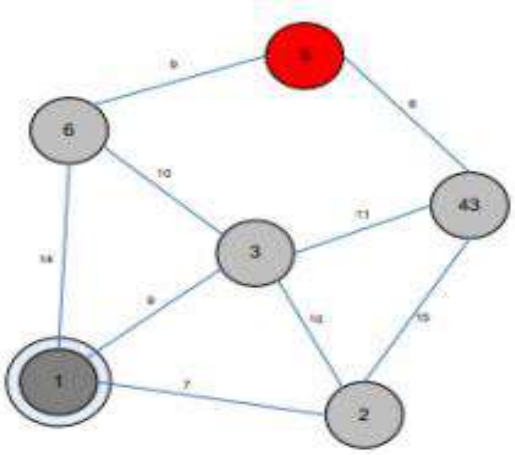
1. Beri nilai bobot (jarak) untuk setiap titik ke titik lainnya, lalu set nilai 0 pada node awal dan nilai tak hingga terhadap node lain (belum terisi)
2. Set semua node “Belum terjamah” dan set node awal sebagai “Node keberangkatan” Dari node keberangkatan, pertimbangkan node tetangga yang

belum terjamah dan hitung jaraknya dari titik keberangkatan. Sebagai contoh, jika titik keberangkatan A ke B memiliki bobot jarak 6 dan dari B ke node C berjarak 2, maka jarak ke C melewati B menjadi 6+2=8. Jika jarak ini lebih kecil dari jarak sebelumnya (yang telah terekam sebelumnya) hapus data lama, simpan ulang data jarak dengan jarak yang baru.

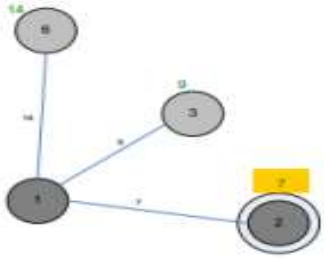
3. Saat kita selesai mempertimbangkan setiap jarak terhadap node tetangga, tandai node yang telah terjamah sebagai "Node terjamah". Node terjamah tidak akan pernah di cek kembali, jarak yang disimpan adalah jarak terakhir dan yang paling minimal bobotnya.
4. Set "Node belum terjamah" dengan jarak terkecil (dari node keberangkatan) sebagai "Node Keberangkatan" selanjutnya dan lanjutkan dengan kembali ke step 3

Dibawah ini penjelasan langkah per langkah pencarian jalur terpendek secara rinci dimulai dari node awal sampai node tujuan dengan nilai jarak terkecil.

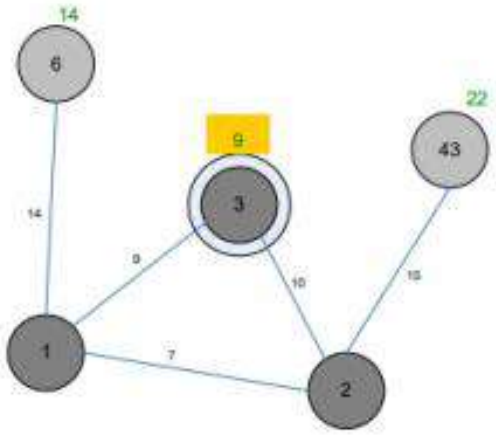
1. Node awal 1, Node tujuan 5. Setiap edge yang terhubung antar node telah diberi nilai



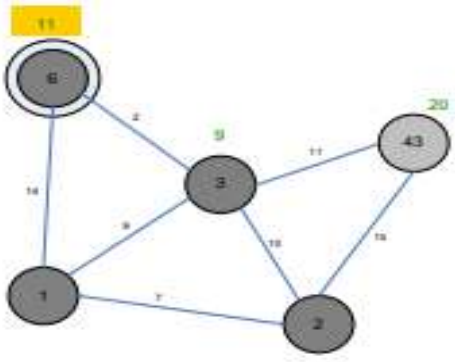
2. Dijkstra melakukan kalkulasi terhadap node tetangga yang terhubung langsung dengan node keberangkatan (node 1), dan hasil yang didapat adalah node 2 karena bobot nilai node 2 paling kecil dibandingkan nilai pada node lain, nilai = 7 (0+7).



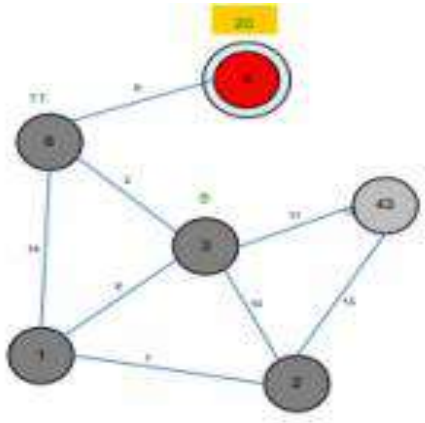
3. Node 2 diset menjadi node keberangkatan dan ditandai sebagi node yang telah terjamah. Dijkstra melakukan kalkulasi kembali terhadap node-node tetangga yang terhubung langsung dengan node yang telah terjamah. Dan kalkulasi dijkstra menunjukan bahwa node 3 yang menjadi node keberangkatan selanjutnya karena bobotnya yang paling kecil dari hasil kalkulasi terakhir, nilai 9 (0+9).



4. Perhitungan berlanjut dengan node 3 ditandai menjadi node yang telah terjamah. Dari semua node tetangga belum terjamah yang terhubung langsung dengan node terjamah, node selanjutnya yang ditandai menjadi node terjamah adalah node 6 karena nilai bobot yang terkecil, nilai 11 (9+2).



5. Node 6 menjadi node terjamah, dijkstra melakukan kalkulasi kembali, dan menemukan bahwa node 5 (node tujuan) telah tercapai lewat node 6. Jalur terpendeknya adalah 1-3-6-5, dan niilai bobot yang didapat adalah 20 (11+9). Bila node tujuan telah tercapai maka kalkulasi dijkstra dinyatakan selesai.



Contoh 3.

Jika diketahui ada 6 verteks dengan masing-masing panjang lintasannya seperti berikut ini :

Vertices	
Number	**Name**
1	a
2	b
3	c
4	d
5	e
6	f

Edges		
Start	**End**	**Cost**
a	b	7
a	c	9
a	f	14
b	c	10
b	d	15
c	d	11
c	f	2
d	e	6
e	f	9

Langkah-langkah Penyelesaiannya adalah :

\node awalnya adalah a, dan node tujuannya adalah f, maka, dapat dihitung jarak antar node-node tersebut :

```
a⟶ b,  cost=7,   NodeAkhir =a
a⟶ c,  cost=9,   NodeAkhir =a
a⟶ d,  cost=NA,  NodeAkhir =a
```

```
a⟶ e,  cost=NA,  NodeAkhir =a
a⟶ f,  cost=14,  NodeAkhir =a
cost yang paling kecil adalah a⟶b sehingga a⟶b ditambahka
n sebagai outputnya dan selanjutnya dimulai dari node b.
Koneksi b⟶d menghasilkan nilai jarak yang diupdate yaitu :
a⟶c,  cost=9,   NodeAkhir =a
a⟶d,  cost=22,  NodeAkhir =b
a⟶e,  cost=NA,  NodeAkhir =a
a⟶f,  cost=14,  NodeAkhir =a

cost yang paling kecil adalah    a⟶c sehingga a⟶c ditamba
hkan sebagai outputnya dan selanjutnya dimulai dari node c.
Lintasan dari node d dan f lebih murah/kecil melalui c sehingga
jarak terbarunya adalah
a⟶d,  cost=20,  NodeAkhir =c
a⟶e,  cost=NA,  NodeAkhir =a
a⟶f,  cost=11,  NodeAkhir =c
cost yang paling kecil adalah a⟶f sehingga c⟶f ditambahka
n sebagai outputnya dan selanjutnya dimulai dari node a.
Input update berikutnya adalah
a⟶d,  cost=20,  NodeAkhir =c
a⟶e,  cost=NA,  NodeAkhir =a
cost yang paling kecil adalah a⟶d sehingga   c⟶d ditambah
kan sebagai outputnya dan selanjutnya dimulai dari node a.
Ada koneksi dari d⟶e sehingga input updatenya menjadi:
a⟶e,  cost=26,  NodeAkhir =d
tinggal menambahkan lintasan d⟶e ke output
Sehingga lintasan terpendek akhirnya menjadi :
  [ d⟶e
    c⟶d
    c⟶f
    a⟶c
    a⟶b ]
```

6.3 RANGKUMAN

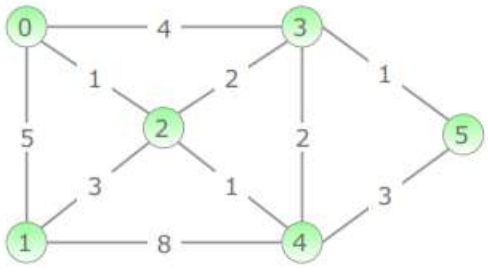
1. Metode atau algoritma yang umum digunakan dalam menyelesaikan masalah pencarian jalur atau lintasan terpendek adalah metode atau algoritma Dijkstra. Algoritma ini dinamakan seperti nama penemunya Edsger Dijkstra, seorang

ilmuwan dari Belanda. Algoritma ini dikembangkan di tahun 1956 dan dipublikasikan secara umum untuk pertama kalinya di tahun 1959 (Dijkstra, 1959).

2. Algoritma Dijkstra dapat digunakan untuk menyelesaikan permasalah rute terpendek dari sebuah titik asal menuju titik tujuan dalam sebuah grafik berbobot. Rute terpendek diperoleh dari dua buah titik yang total bobotnya paling mimum atau paling kecil jika ditotalkan dari semua titik dalam jaringan grafik.
3. Ide dari algoritma ini sangat sederhana. Algoritma Dijkstra memilih titik (sumber) dan menetapkan kemungkinan biaya optimum (yaitu tak terhingga) ke setiap titik lainnya. Langkah berikutnya akan mencoba untuk meminimalkan biaya untuk setiap titik. Biaya yang dimaksud disini adalah jarak atau waktu yang diambil untuk mencapai titik tujuan dari titik sumber.

6.4 TEST FORMATIF

1. Modifikasi program berikut ini malalui inputannya yang tidak dituliskan di dalam program tetapi melalui keyboard atau dengan membuat UI/UX yang user friendly.



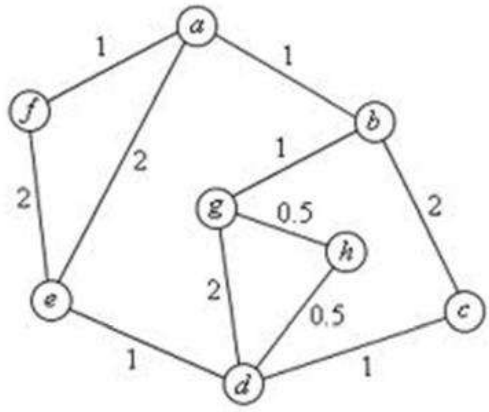

```
#include<iostream>
#include<vector>
#include<set>
using namespace std;
typedef pair<int,unsigned long long> PII;
typedef vector<PII> VPII;
typedef vector<VPII> VVPII;
void DijkstrasShortestPath (int source_node, int node_count, VVPII&
graph) {
```

```
    // Assume that the distance from source_node to other nodes is
infinite
    // in the beginnging, i.e initialize the distance vector to a max
value
    const long long INF = 999999999999;
    vector<unsigned long long> dist(node_count, INF);
    set<PII> set_length_node;

    // Distance from starting vertex to itself is 0
    dist[source_node] = 0;
    set_length_node.insert(PII(0,source_node));

    while (!set_length_node.empty()) {
        PII top = *set_length_node.begin();
        set_length_node.erase(set_length_node.begin());
        int source_node = top.second;
        for(auto& it : graph[source_node]) {
            int adj_node = it.first;
            int length_to_adjnode = it.second;
            // Edge relaxation
            if (dist[adj_node] > length_to_adjnode +
dist[source_node]) {
                // If the distance to the adjacent node is not INF,
means the pair <dist, node> is in the set
                // Remove the pair before updating it in the set.
                if (dist[adj_node] != INF) {
set_length_node.erase(set_length_node.find(PII(dist[adj_node],adj_node
)));
                }
                dist[adj_node] = length_to_adjnode +
dist[source_node];
                set_length_node.insert(PII(dist[adj_node], adj_node));
            }
        }
```

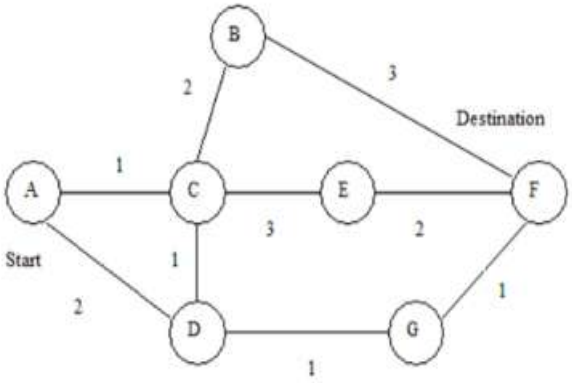
```
    }
    for (int i=0; i<node_count; i++)
        cout << "Source Node(" << source_node << ") -> Destination
Node(" << i << ") : " << dist[i] << endl;
}
int main(){
    vector<VPII> graph;
    // Node 0: <1,5> <2,1> <3,4>
    VPII a = {{1,5}, {2,1}, {3,4}};
    graph.push_back(a);
    // Node 1: <0,5> <2,3> <4,8>
    VPII b = {{0,5}, {2,3}, {4,8}};
    graph.push_back(b);
    // Node 2: <0,1> <1,3> <3,2> <4,1>
    VPII c = {{0,1}, {1,3}, {3,2}, {4,1}};
    graph.push_back(c);
    // Node 3: <0,4> <2,2> <4,2> <5,1>
    VPII d = {{0,4}, {2,2}, {4,2}, {5,1}};
    graph.push_back(d);
    // Node 4: <1,8> <2,1> <3,2> <5,3>
    VPII e = {{1,8}, {2,1}, {3,2}, {5,3}};
    graph.push_back(e);
    // Node 5: <3,1> <4,3>
    VPII f = {{3,1}, {4,3}};
    graph.push_back(f);
    int node_count  = 6;
    int source_node = 0;
    DijkstrasShortestPath(source_node, node_count, graph);
    cout << endl;
    source_node = 5;
    DijkstrasShortestPath(source_node, node_count, graph);
    return 0;
}
```

1. Selesaikan studi kasus berikut dengan menggunakan Algoritma Dijkstra !

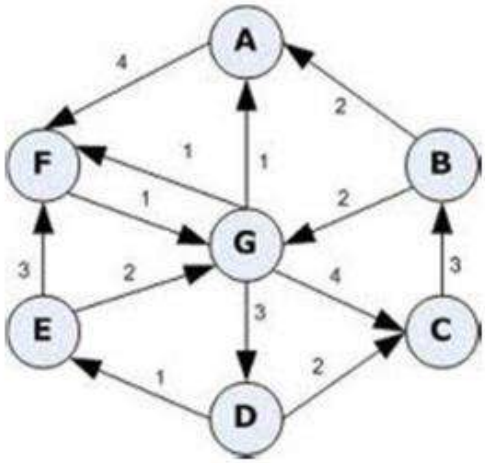
a. Studi kasus 1



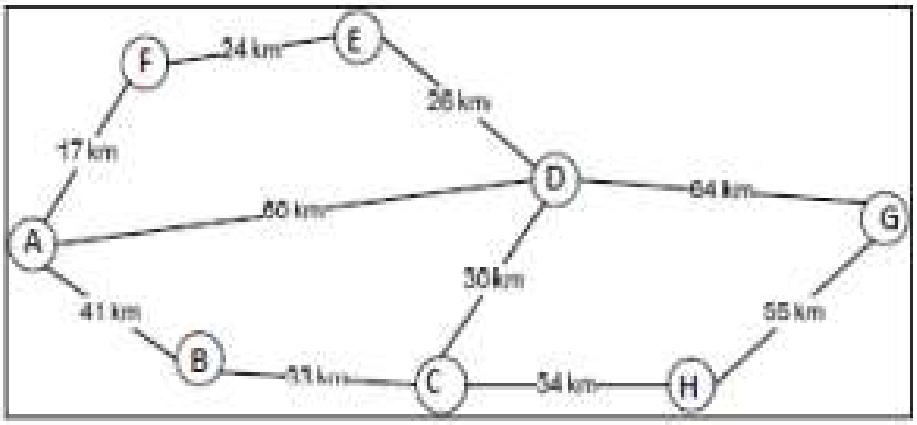
b. Studi kasus 2



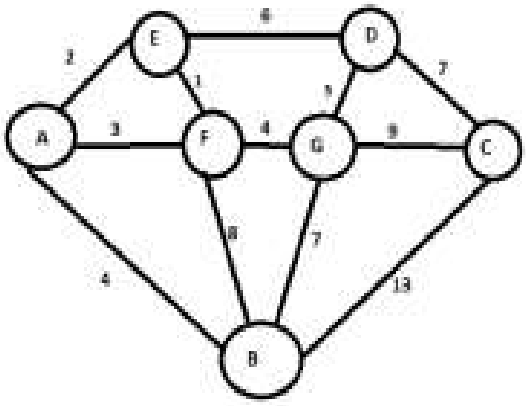
c. Studi kasus 3



d. Studi Kasus 4 :
Node awal F, Node tujuan H. Setiap edge yang terhubung antar node dan telah diberi nilai



e. Studi Kasus 5.
Node awal A, Node tujuan C. Setiap edge yang terhubung antar node dan telah diberi nilai



2. Selesaikan studi kasus pada soal no. 1 dengan menggunakan program anda masing-masing berdasarkan algoritma djikstra, selanjutnya lakukan analisis untuk kelima studi kasus tersebut.

DAFTAR PUSTAKA

Agustin, Dwi, Ririn. (2006). *Heuristic Search,* Materi kuliah AI. Bandung. Departemen Teknik Informatika Sekolah Tinggi Teknologi Telkom Bandung

Arhami. M. (2004). *Penggunaan Algoritma Genetika untuk Optimisasi Masalah Task Assignment dalam Sistem Terdistribusi*, Tesis, Magister Ilmu Komputer UGM.Jogjakarta

Bazlamacci C F, Hindi K S. (2001). Minimum-weight spanning tree algorithms a survey and empirical study. Computers and Operation Research.; 28: (pp 767-85)

Borůvka, Otakar. (1926): "O Jistém Problému Minimálním." Práce Moravské Přírodovědecké Společnosti III, no. 3 (pp. 37–58)

Jarník, Vojtěch. ((1930):). "O Jistém Problému Minimálním (Z Dopisu Panu O. Borůvkovi)." Práce Moravské Přírodovědecké Společnosti 6, No. 4 (pp. 57–63).

Cheriton D, Tarjan R E. (1976). Finding minimum spanning trees. SIAM Journal on Computing.; 5 (pp 724-42).

Cormen. Thomas .H,, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. (2009). Introduction to Algorithms, Third Edition, MIT Press

Dijkstra. E,.W. (1959).A note on two problems in connection with graphs. *Numerische Mathematik*, Volume 1, (pp. 269-271).

Fredman M L, Tarjan R E. (1987). heaps and their uses in improved network optimization algorithms. Journal of the Association of for Computing Machinery; 34/3:596-615

Gen, M ,and R. Cheng. (2000) *Genetic Algorithm and Engineering Optimization*, John Wiley and Sons, Inc, New York

Kusumadewi, S. (2002). *Artificial Intelligence (Teknik dan Aplikasinya)*, Graha Ilmu, Yogyakarta

Kruskal J B. (1956). On the shortest spanning subtree of a graph and the travelling salesman problem. Proceedings of the American Mathematical Society; 7:48-50

Karger D R. (1993.). Random sampling in matroids, with application to graph connectivity and minimum spanning trees. Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science. Los Alamitos, IEEE Computer Society Press, (pp. 84-93) California:.

Karger D R, Klein P N, Tarjan R E. (1995). A randomized linear-time algorithm to find minimum spanning trees. Journal of the Association for Computing Machinery.; 42/2:321-8.

Munir, Rinaldi. (2005). Strategi Algoritmik. Bandung. Institut Teknologi Bandung.

Prim R C. Shortest connection networks and some generalizations. Bell Systems Technical Journal. 1957; 4:53-7.

Sara Baase, Reading,Mass**. (**1993). *Computer Algorithms: introduction to design and analysis.* **2nd ed.**,:Addison-Wesley Company

Suryadi. (1992). Pengantar Analisis Algoritma, Gunadarma: Jakarta

Tarjan R E. (1983). Data structures and network algorithms. CBMS-NFS Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics,. p. 44, Philadelphia:

Thiang., Ronald K, Hany F. (2001). Implementasi Algoritma Genetika pada Mikrokontroler MCS51 Untuk Mencari Rute Terpendek, *Proceeding, Seminar of Intelligent Technology and Its Applications (SITIA 2001) Institut Teknologi Sepuluh Nopember, Surabaya, May 1, 2001*

Yao A. (1975). An O(|E| log log |V|) algorithm for finding minimum spanning trees. 4:21-3. Information Processing Letters.

GLOSSARIUM

Algoritma	Metode atau proses diikuti untuk memecahkan masalah

Analisis algoritma	Versi yang kurang formal dari istilah analisis algoritma asimptotik, umumnya digunakan sebagai sinonim untuk analisis asimptotik
algoritma aproksimasi	Dan algoritma untuk masalah optimisasi yang menemukan solusi yang baik, tetapi belum tentu termurah.
array	Tipe data yang digunakan untuk menyimpan elemen di lokasi memori berturut-turut dan merujuknya dengan indeks.
analisis algoritma asimptotik	Istilah yang lebih formal untuk analisis asimptotik.
analisis asimptotik	Metode untuk memperkirakan efisiensi suatu algoritma atau program komputer dengan mengidentifikasi tingkat pertumbuhannya. Analisis asimptotik juga memberikan cara untuk mendefinisikan kesulitan yang melekat pada suatu masalah
average case	Dalam analisis algoritma, rata-rata biaya untuk semua contoh masalah dari ukuran input yang diberikan n. Jika tidak semua instance masalah memiliki probabilitas yang sama untuk terjadi, maka case rata-rata harus dihitung menggunakan rata-rata tertimbang.
average seek time	Waktu yang diharapkan (rata-rata) untuk melakukan operasi pencarian pada disk drive, dengan asumsi bahwa pencarian berada di antara dua trek yang dipilih secara acak. Ini adalah salah satu dari dua metrik yang biasanya disediakan oleh vendor disk drive untuk kinerja disk drive, dengan yang lainnya adalah waktu pencarian track-to-track.
backtracking	Heuristik untuk pencarian ruang solusi dengan kekerasan. Ini pada dasarnya adalah pencarian mendalam-pertama dari ruang solusi. Ini dapat ditingkatkan dengan menggunakan algoritma cabang-dan-batas.
best case	Dalam analisis algoritma, instance masalah dari antara semua instance masalah untuk ukuran input yang diberikan n yang memiliki biaya paling sedikit. Perhatikan bahwa kasus terbaik bukan ketika n kecil, karena kami merujuk yang terbaik dari kelas input (yaitu, kami menginginkan yang terbaik dari input berukuran n).
big-Oh notation	Dalam analisis algoritma, notasi steno untuk menggambarkan batas atas untuk suatu algoritma atau masalah.
binary insert sort	Variasi pada jenis penyisipan di mana posisi nilai yang dimasukkan terletak oleh pencarian biner, dan kemudian dimasukkan ke tempatnya. Dalam penggunaan normal ini bukan peningkatan pada jenis

	penyisipan standar karena biaya memindahkan banyak item dalam array. Tetapi secara langsung bermanfaat jika biaya perbandingannya tinggi dibandingkan dengan memindahkan elemen, atau secara teori berguna jika kita hanya peduli untuk menghitung biaya perbandingan.
binary search	Algoritma rekursif standar untuk menemukan catatan dengan nilai kunci pencarian yang diberikan dalam daftar diurutkan. Ini berjalan dalam waktu O (logn). Pada setiap langkah, lihat bagian tengah sublist saat ini, dan buang setengah dari catatan yang kuncinya terlalu kecil atau terlalu besar.
binary search tree	Pohon biner yang menerapkan batasan berikut pada nilai-nilai simpulnya: Nilai kunci pencarian untuk setiap simpul A harus lebih besar dari nilai (kunci) untuk semua simpul di subtree kiri A, dan kurang dari nilai kunci untuk semua simpul di subtree kanan A. Beberapa konvensi harus diadopsi jika beberapa node dengan nilai kunci yang sama diizinkan, biasanya ini harus di subtree kanan.
binary tree	Seperangkat terbatas node yang kosong, atau memiliki simpul akar bersama dua pohon biner, yang disebut subtree kiri dan kanan, yang terpisah satu sama lain dan dari root.
Bubble sort	Jenis sederhana yang memerlukan waktu Theta (n2) dalam kasus terbaik, rata-rata, dan terburuk. Bahkan versi yang dioptimalkan biasanya akan berjalan lebih lambat daripada jenis penyisipan, sehingga tidak banyak merekomendasikannya.
complete graph	Grafik tempat setiap titik terhubung ke setiap titik lainnya.
CNF	Ekspresi Boolean ditulis sebagai serangkaian klausa AND yang DILAKUKAN bersama.
constant running time	Biaya fungsi yang waktu operasinya tidak terkait dengan ukuran inputnya. Dalam notasi Theta, ini secara tradisional ditulis sebagai Θ (1).
cost	Jumlah sumber daya yang dikonsumsi oleh solusi.
depth-first search	Algoritma traversal grafik. Kapanpun v dikunjungi selama traversal, DFS akan secara rekursif mengunjungi semua tetangga v yang tidak dikunjungi.
Dijkstra's algorithm	Algoritme untuk memecahkan masalah jalur terpendek satu sumber dalam grafik. Ini adalah algoritma serakah. Ini hampir identik dengan algoritma Prim untuk menemukan pohon rentang minimum, dengan satu-satunya perbedaan adalah perhitungan yang dilakukan untuk memperbarui jarak yang paling dikenal.

divide and conquer	Suatu teknik untuk merancang algoritma di mana solusi ditemukan dengan memecah masalah menjadi subproblem yang lebih kecil (serupa), menyelesaikan subproblem, kemudian menggabungkan solusi subproblem untuk membentuk solusi untuk masalah asli. Proses ini sering diimplementasikan menggunakan rekursi.
dynamic programming	Pendekatan untuk merancang algoritma yang bekerja dengan menyimpan tabel hasil untuk sub-masalah. Penyebab khas untuk biaya yang berlebihan dalam algoritma rekursif adalah cabang yang berbeda dari rekursi mungkin memecahkan subproblem yang sama. Pemrograman dinamis menggunakan tabel untuk menyimpan informasi tentang subproblem mana yang telah diselesaikan, dan menggunakan informasi yang tersimpan untuk segera memberikan jawaban untuk setiap upaya berulang untuk menyelesaikan subproblem itu.
edge	Koneksi yang menautkan dua node di pohon, daftar tertaut, atau grafik. sunting jarak. Diberikan string S dan T, jarak edit adalah ukuran untuk jumlah langkah pengeditan yang diperlukan untuk mengubah S menjadi T.
efficient	Suatu solusi dikatakan efisien jika memecahkan masalah dalam batasan sumber daya yang diperlukan. Suatu solusi kadang-kadang dikatakan efisien jika membutuhkan sumber daya lebih sedikit daripada alternatif yang diketahui, terlepas dari apakah ia memenuhi persyaratan tertentu.
Floyd's algorithm	Algoritme untuk memecahkan masalah jalur terpendek semua-pasangan. Ia menggunakan teknik algoritme pemrograman dinamis, dan berjalan dalam waktu Θ (n3). Seperti halnya algoritma pemrograman dinamis, masalah utama adalah untuk menghindari duplikasi pekerjaan dengan menggunakan pembukuan yang tepat pada kemajuan algoritma melalui ruang solusi. Ide dasarnya adalah pertama-tama menemukan semua biaya edge langsung, kemudian meningkatkan biaya-biaya tersebut dengan memungkinkan jalur melalui simpul 0, kemudian jalur termurah yang melibatkan jalur melalui simpul 0 dan 1, dan seterusnya.
forest	Koleksi satu atau lebih pohon.
graph	Grafik G = (V, E) terdiri dari satu set simpul V dan satu set tepi E, sedemikian sehingga setiap tepi dalam E adalah koneksi antara sepasang simpul dalam V.
greedy algorithm	Algoritma yang membuat pilihan optimal secara lokal di setiap langkah.

Huffman coding tree	Kode-kode yang diberikan kepada kumpulan huruf (atau simbol lainnya) melalui proses pengkodean Huffman. Pengodean Huffman menggunakan pohon pengkodean Huffman untuk menghasilkan kode. Kode dapat memiliki panjang variabel, sehingga huruf-huruf yang diharapkan muncul paling sering lebih pendek. Pengodean Huffman optimal setiap kali frekuensi sebenarnya diketahui, dan frekuensi surat tidak tergantung pada konteks surat itu dalam pesan.
Insertion Sort	Algoritma pengurutan dengan Θ (n2) biaya kasus rata-rata dan terburuk, dan Theta (n) biaya kasus terbaik. Biaya kasus terbaik ini berguna saat kami memiliki alasan untuk mengharapkan input hampir diurutkan..
knapsack problem	Meskipun ada banyak variasi dari masalah ini, berikut adalah versi yang khas: Diberikan knapsack dengan ukuran tetap, dan koleksi objek dari berbagai ukuran, apakah ada subset dari objek yang tepat masuk ke dalam ransel? Masalah ini dikenal sebagai NP-lengkap, tetapi dapat dipecahkan untuk contoh masalah dalam waktu praktis yang relatif cepat menggunakan pemrograman dinamis. Dengan demikian, dianggap memiliki biaya pseudo-polinomial. Versi masalah optimisasi adalah menemukan subset yang dapat sesuai dengan jumlah item terbesar, baik dalam hal ukuran totalnya, atau dalam hal jumlah nilai yang terkait dengan setiap item.
Kruskal's algorithm	Algoritma untuk menghitung MCST grafik. Selama pemrosesan, ia menggunakan proses UNION / FIND untuk menentukan dua simpul secara efisien dalam subgraph yang sama
merge insert sort	menggabungkan jenis sisipan Sinonim untuk jenis Ford dan Johnson.
Mergesort	Algoritma pengurutan yang membutuhkan Θ (nlogn) dalam kasus terbaik, rata-rata, dan terburuk. Secara konseptual sederhana: Membagi daftar menjadi dua, mengurutkan dua bagian, lalu menggabungkannya. Agak rumit untuk mengimplementasikan secara efisien pada array.
minimal-cost spanning tree	Disingkat MCST, atau terkadang sebagai MST. Berasal dari grafik berbobot, MCST adalah subset dari tepi grafik yang mempertahankan konektivitas grafik sementara memiliki total biaya terendah (seperti yang didefinisikan oleh jumlah bobot tepi dalam MCST). Hasilnya disebut sebagai pohon karena tidak akan pernah memiliki siklus (karena tepi dapat dihapus dari siklus dan masih menjaga konektivitas). Dua algoritma untuk mengatasi masalah ini adalah algoritma Prim

	dan algoritma Kruskal. berat jalur eksternal minimum
Node (simpul)	Objek yang membentuk struktur tertaut seperti daftar tertaut atau pohon biner. Biasanya, node dialokasikan menggunakan alokasi memori dinamis. Dalam terminologi grafik, node lebih sering disebut simpul.
Omega notation	Dalam analisis algoritma, notasi Ω digunakan untuk menggambarkan batas bawah. Secara kasar (tetapi tidak sepenuhnya) analog dengan notasi-Oh besar yang digunakan untuk mendefinisikan batas atas.
parent	Dalam sebuah pohon, simpul P yang terhubung langsung ke simpul A adalah induk dari A. A adalah anak dari P.
permutasi	Permutasi urutan S adalah elemen-elemen S yang diatur dalam suatu urutan.
pivot	Di Quicksort, nilai yang digunakan untuk membagi daftar menjadi sublists, satu dengan nilai lebih rendah dari pivot, yang lain dengan nilai lebih besar dari pivot.
program	Contoh, atau representasi konkret, dari suatu algoritma dalam beberapa bahasa pemrograman.
Quicksort	Jenis yang Θ (nlogn) dalam kasus terbaik dan rata-rata, meskipun Θ (n2) dalam kasus terburuk. Namun, penerapan yang masuk akal akan membuat kasus terburuk terjadi dalam keadaan yang sangat jarang. Karena loop dalam yang ketat, cenderung berjalan lebih baik daripada jenis lain yang dikenal dalam kasus umum. Jadi, ini adalah semacam populer untuk digunakan dalam pustaka kode. Ia bekerja dengan membagi dan menaklukkan, dengan memilih nilai pivot, memecah daftar menjadi bagian-bagian yang kurang dari atau lebih besar dari pivot, dan kemudian mengurutkan dua bagian.
root	Di pohon, simpul paling atas dari pohon. Semua simpul lain di pohon adalah keturunan dari akar.
search problem	iberikan nilai kunci tertentu K, masalah pencarian adalah untuk menemukan catatan (kj, Ij) dalam beberapa kumpulan catatan L sedemikian sehingga kj = K (jika ada). Pencarian adalah metode sistematis untuk menemukan catatan (atau catatan) dengan nilai kunci kj = K.
searching	Diberikan kunci pencarian K dan beberapa koleksi catatan L, pencarian adalah metode sistematis untuk menemukan catatan (atau catatan) di L dengan nilai kunci kj = K.
selection sort	Meskipun jenis ini membutuhkan waktu Θ (n2) dalam kasus terbaik, rata-rata, dan terburuk, ia hanya

	membutuhkan operasi pertukaran Θ (n). Dengan demikian, itu relatif baik dalam aplikasi di mana swap mahal. Ini dapat dilihat sebagai optimasi pada bubble sort, di mana swap ditunda sampai akhir setiap iterasi.
sequential search	Algoritma pencarian paling sederhana: Dalam sebuah array, cukup lihat elemen array dalam urutan yang muncul.
shortest path (jalur terpendek)	Diberikan grafik dengan jarak atau bobot di tepinya, jalur terpendek antara dua node adalah jalur dengan jarak atau berat total paling sedikit. Contoh dari masalah jalur terpendek adalah masalah jalur terpendek sumber tunggal dan masalah jalur terpendek semua-pasangan.
Strassen's algorithm	lgoritma rekursif untuk perkalian matriks. Ketika mengalikan dua matriks n × n, algoritma ini berjalan lebih cepat dari waktu Θ (n3) yang dibutuhkan oleh algoritma multiplikasi matriks standar. Secara khusus, algoritma Strassen membutuhkan waktu Theta (nlog27) waktu. Hal ini dicapai dengan refactoring operasi perkalian dan penambahan sub-matriks sehingga hanya membutuhkan 7 perkalian sub-matriks alih-alih 8, dengan biaya tambahan operasi penambahan sub-matriks. Dengan demikian, sementara biaya asimptotik lebih rendah, faktor konstan dalam persamaan laju pertumbuhan lebih tinggi. Ini membuat algoritma Strassen tidak efisien dalam praktik kecuali jika array yang dikalikan agak besar. Variasi pada algoritma Strassen ada yang mengurangi jumlah perkalian sub-matriks lebih jauh dengan biaya tambahan sub-matriks yang lebih banyak.
Theta notation	Dalam analisis algoritma, notasi Θ digunakan untuk menunjukkan bahwa batas atas dan batas bawah untuk suatu pencocokan algoritma atau masalah.
tree	Sebuah pohon T adalah himpunan terbatas dari satu atau lebih node sehingga ada satu node yang ditunjuk R, yang disebut root T. Jika set (T− {R}) tidak kosong, node ini dipartisi menjadi n> 0 disjoint set T0, T1, ..., Tn − 1, masing-masing adalah pohon, dan yang akarnya R1, R2, ..., Rn, masing-masing, adalah anak-anak dari R.
Weight (bobot)	Biaya atau jarak yang paling sering dikaitkan dengan keunggulan dalam grafik.
weighted graph (graf terboboti)	Grafik yang tepinya masing-masing memiliki bobot atau biaya terkait.
worst case	Dalam analisis algoritma, instance masalah dari antara semua instance masalah untuk ukuran input yang

diberikan n yang memiliki biaya terbesar. Perhatikan bahwa kasus terburuk bukanlah ketika n besar, karena kami merujuk pada wrost dari kelas input (yaitu, kami ingin yang terburuk dari input ukuran n).

INDEKS