

BAB I

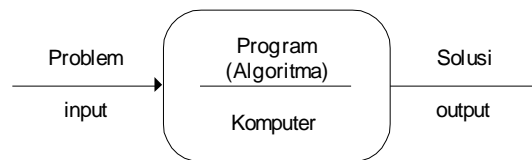
DESAIN DAN ANALISIS ALGORITMA

Pada bab ini kita akan membahas tentang apa dan mengapa algoritma. Desain algoritma dibahas secara sekilas dengan mengambil kasus sederhana, namun cukup menggambarkan bagaimana mendesain algoritma untuk suatu permasalahan. Pembahasan tentang analisis algoritma ditampilkan secara lengkap dengan mengambil permasalahan praktis seperti pencarian (*searching*), pengurutan (*sorting*), dan lainnya.

1. Algoritma

Algoritma (algoritma komputer) didefinisikan sebagai urutan langkah-langkah komputasi yang akan dieksekusi oleh komputer untuk mentransformasikan masukan (*input*) yang valid menjadi keluaran (*output*) yang diinginkan. Dengan demikian algoritma dapat kita pandang sebagai suatu alat untuk menyelesaikan permasalahan/problem komputasi.

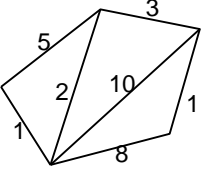
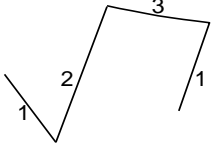
Problem kita abstraksikan sebagai sekelompok nilai (data) atau input, sedangkan penyelesaian (solusi) dari problem tersebut dapat kita pandang sebagai sekelompok nilai (informasi) atau output. Proses mentransformasikan input menjadi output adalah metoda untuk menyelesaikan problem tersebut. Dari penjelasan ini, problem-problem yang kita angkat/selesaikan adalah suatu problem yang sudah dipunyai/diketahui penyelesaiannya. Sebagai ilustrasi dapat dilihat Gambar 1.1



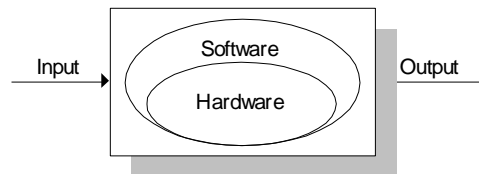
Gambar 1.1

Tabel 1.1 berikut ini berisi beberapa contoh problem yang diketahui penyelesaiannya. Pada kolom pertama adalah permasalahannya, kolom kedua merupakan abstraksi permasalahan dalam bentuk input dan kolom ke tiga adalah output atau abstraksi dari penyelesaian.

Tabel 1.1
Contoh Tranformasi Problem Ke Input & Output.

Problem	Input	Output
Mencari nilai maximum dari sekelompok bilangan $\{x_1, x_2, \dots, x_n\}$	Array $\{x_1, x_2, \dots, x_n\}$	x_{max} nilai terbesar dari $\{x_1, x_2, \dots, x_n\}$
Mengurut (sorting) naik sekelompok bilangan $\{x_1, x_2, \dots, x_n\}$	Array $\{x_1, x_2, \dots, x_n\}$ yang tidak terurut	Array $\{x_1, x_2, \dots, x_n\}$ yang sudah terurut naik
Minimum Spanning Tree		

Algoritma dalam kaitannya dengan penyelesaian permasalahan yang berbantuan komputer, memandang sistem komputer sebagai suatu kesatuan perangkat keras (*hardware*) dan perangkat lunak (*software*). Hardware melakukan langkah komputasi sederhana dengan kecepatan yang sangat tinggi, seperti operasi perbandingan, jumlah, dan kurang. Sedangkan software menentukan dan mengontrol langkah-langkah yang akan dieksekusi oleh hardware. Dalam hal ini software dapat terdiri dari sistem operasi dan bahasa pemrograman yang dipakai untuk implementasi algoritma. Hal ini dapat dilihat pada Gambar 1.2.



Gambar 1.2
Sistem Komputer Dipandang Dari Sisi Algoritma

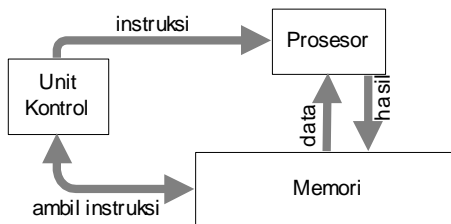
Sampai dengan saat ini perkembangan hardware dan software sudah sangat maju, mulai dari komputer dengan satu processor hingga banyak processor. Untuk itu kita lihat terlebih dahulu model komputasi pada seksi 2.

2. Model Komputasi

Setiap komputer baik sekuensial (satu prosesor) atau parallel (banyak prosesor) bekerja dengan cara mengerjakan instruksi pada suatu data. Sederetan insruksi-instruksi (algoritma) memerintah prosesor apa yang harus dikerjakan pada setiap langkah. Sederetan data (input algoritma) diolah oleh instruksi-instruksi ini. Berdasarkan pengerjaan instruksi ini, komputer dapat kita kita klasifikasikan kedalam empat kelas sebagai berikut.

1. Single Instruction stream Single Data stream (SISD).
2. Multiple Instruction stream Single Data stream (MISD).
3. Single Instruction stream Multiple Data stream (SIMD).
4. Multiple Instruction stream Multiple Data stream (MIMD).

Komputer SISD atau sekuensial komputer terdiri dari satu unit prosesor yang menerima satu deretan instruksi-instruksi yang mengolah satu deretan data. Setiap saat hanya ada satu eksekusi / pekerjaan, satu instruksi dari unit kontrol (*control unit*) dikerjakan pada satu datum yang diambil dari memory (*Random Acces Memory, RAM*). Sebagai contoh, prosesor melakukan operasi aritmatika pada datum dari memori untuk kemudian dikembalikan lagi ke memori. Sebagai gambaran dapat dilihat pada Gambar 1.3.



Gambar 1.3

Arus instruksi dan data pada komputer SISD

Algoritma yang bekerja pada kelas komputer SISD dikatakan algoritma sekuensial (*sequential algorithm*), dan pada buku ini hanya membahas tentang algoritma sekuensial. Untuk selanjutnya kita singkat saja algoritma.

Untuk kelas komputer MISD, SIMD, MIMD yang merupakan komputer parallel tidak dibahas lebih jauh. Ke tiga kelas komputer ini mempunyai prosesor banyak, hanya cara

bekerjanya saja yang berbeda. Algoritma yang bekerja pada komputer kelas ini disebut dengan algoritma paralel (*parallel algorithm*).

3. Desain Algoritma

Ada banyak cara dalam mendesain / merancang algoritma untuk suatu permasalahan / problem. Namun demikian belum ada suatu algoritma umum yang mentransformasikan permasalahan menjadi algoritma untuk permasalahan tersebut. Secara garis besar dalam merancang algoritma dilakukan langkah-langkah:

1. Transformasi permasalahan menjadi input.

Permasalahan kita identifikasi untuk menentukan objek input dan struktur datanya.

2. Transformasi solusi permasalahan menjadi output.

Bentuk solusi dari permasalahan harus sudah diketahui dan dari bentuk solusi ini dipakai untuk menentukan output beserta struktur datanya.

3. Langkah - langkah mengolah input menjadi output.

Dalam mengolah input menjadi output, kita memakai cara atau metoda atau teorema yang dipakai untuk menghasilkan solusi dari permasalahan. Hal ini sangat bergantung dari permasalahannya. Misal permasalahan penyelesaian sistem persamaan linier, kita dapat memakai metoda Eliminasi Gauss (dengan teorema - teorema yang mendasarinya) atau memakai metoda faktorisasi LU.

Langkah - langkah ini kita tulis dengan menggunakan kode semu (mirip bahasa pemrograman, namun belum dapat jalan). Dalam penulisan buku ini didekatkan pada bahasa pemrograman C.

Pembahasan lebih jauh tentang desain algoritma, akan dibahas melalui contoh - contoh kasus. Ada beberapa contoh kasus yang dibuatkan lebih dari satu algoritma, agar pembaca lebih mempunyai wawasan dalam desain algoritma.

Contoh 1:

Pandang sederetan n buah bilangan $\{a_1, a_2, \dots, a_n\}$. Ingin dicari bilangan x apakah ada pada sederetan bilangan tersebut. Kalau ada, x berada pada index ke berapa. Buatlah algoritma untuk permasalahan ini.

Sekarang kita ikuti saja langkah-langkah desain di atas.

Pertama, sederetan n bilangan kita nyatakan saja dalam sebuah array dengan n elemen (tentu saja dapat juga dinyatakan dengan sebuah *linked-list*). Sehingga inputnya adalah array A dengan elemen n buah bilangan dan bilangan x yang dicari. Perlu diingat bahwa array di dalam bahasa C berawal dari 0.

Kedua, solusi dari permasalahan di atas berupa sebuah bilangan bulat positif yang menyatakan index / letak bilangan x dalam array A . Sehingga outputnya adalah 0, atau 1, atau 2, ... , atau n yang merupakan urutan keberadaan x . Bagaimana kalau x tidak ada di dalam A ?. Kita buat saja, bila x tidak ditemukan di A , maka outputnya adalah -1.

Dari kedua langkah ini, kita tuliskan sebagai berikut.

Input : Array A dengan n elemen bilangan dan bilangan x yang dicari.

Output : Bilangan bulat non negatif j yang merupakan letak / index x di dalam array A .
Menghasilkan $j = -1$ bila x tidak ada di A .

Ketiga, salah satu cara untuk menghasilkan output yang dimaksud dari pemberian input di atas adalah masuk elemen index ke 0 dari array, $A[0]$, kemudian bandingkan dengan x . Bila sama maka hasilnya $j=0$ dan selesai, bila tidak sama maka masuk ke elemen berikutnya, $A[1]$, kemudian bandingkan dengan x . Bila sama maka hasilnya $j=1$ dan selesai, bila tidak sama diulang-ulang lagi hingga sampai dengan elemen ke n . Bila sampai dengan elemen ke n tidak ada yang sama, maka hasilnya adalah $j = -1$. Dengan demikian langkah ketiga ini dapat dituliskan sebagai berikut, anggap cara ini kita beri nama *SequentialSearch*.

Algoritma:

```
SequentialSearch( $A, n, x$ ) {  
1.  $j=0$ ;  
2. while ( $j < n$  &&  $x \neq A[j]$ )  
3.      $j++$ ;  
4. if ( $j < n$ ) return  $j$ ;  
   else return -1;  
}
```

Contoh 2:

Pandang sederetan n buah bilangan $\{a_1, a_2, \dots, a_n\}$. Ingin dicari bilangan *max* yang merupakan bilangan terbesar dari n buah sederetan bilangan tersebut. Buatlah algoritma untuk permasalahan ini.

Mirip contoh 1, penyelesaian dari permasalahan ini adalah:

Pertama, sederetan n bilangan kita nyatakan saja dalam sebuah array dengan n elemen. Sehingga inputnya adalah array A dengan elemen n .

Kedua, solusi dari permasalahan di atas berupa sebuah bilangan

$max = maks \{a_1, a_2, \dots, a_n\}$ atau bilangan terbesar yang ada pada array A .

Dari kedua langkah ini, kita tuliskan sebagai berikut.

Input : Array A dengan n elemen bilangan.

Output : Bilangan $max = maks \{a_1, a_2, \dots, a_n\}$

Ketiga, salah satu cara untuk menghasilkan output yang dimaksud dari pemberian input di atas adalah elemen pertama dianggap terlebih dahulu merupakan maksimal atau $max=A[0]$. Kemudian masuk ke elemen berikutnya, $A[1]$, bandingkan dengan max . Bila max lebih kecil dari $A[1]$ maka ganti max dengan $A[1]$, bila tidak maka nilai max tetap. Masuk lagi ke elemen berikutnya, lakukan cara yang sama hingga sampai dengan elemen ke n . Dengan demikian langkah ketiga ini dapat dituliskan sebagai berikut, anggap cara ini kita beri nama *CariMaximal*.

Algoritma:

CariMaximal(A, n, max) {

1. $max=A[0]$;

2. *for*($i=1$; $i < n$; $i++$)

3. *if* ($max < A[i]$) $max=A[i]$;

4. *return* max ;

}

Contoh 3:

Pandang sederetan n buah bilangan a_1, a_2, \dots, a_n . Ingin dilakukan pengurutan (urut naik) sehingga menjadi $a_{j_1}, a_{j_2}, \dots, a_{j_n}$ dengan $a_{j_1} \leq a_{j_2} \leq \dots \leq a_{j_n}$.

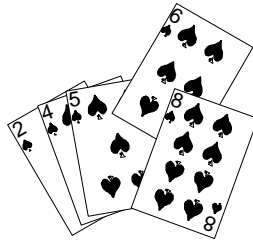
Sederetan n bilangan kita nyatakan dalam sebuah array dengan n elemen. Sehingga inputnya adalah array A dengan elemen n .

Solusi dari permasalahan di atas berupa sebuah array dengan elemen terurut naik $a_{j_1} \leq a_{j_2} \leq \dots \leq a_{j_n}$. Sehingga input dan outputnya sebagai berikut.

Input : Array A dengan n elemen bilangan.

Output : array A dengan n elemen terurut naik.

Ada banyak cara untuk melakukan pengurutan (*sorting*), di awal ini akan kita pakai metoda pengurutan penyisipan (*insertion sort*). Anggap kita sudah mempunyai i elemen yang sudah terurut, kita akan menyisipkan satu elemen lagi sehingga hasil $i+1$ elemen tersebut terurut, sebagaimana dalam Gambar 1.4. Elemen 5 akan disisipkan, pertama bandingkan dari kanan, yaitu dari elemen 6. Elemen 5 kalah besar, selanjutnya bandingkan dengan sebelah kirinya, yaitu dengan elemen 4. Elemen 5 lebih besar dari elemen 4, sehingga elemen 5 ditaruh diantara 4 dan



6.

Gambar 1.4 Gambaran Pengurutan Penyisipan

Algoritma pengurutan penyisipan dapat disusun sebagai berikut.

Algoritma:

```
InsertionSort(A,n) {
1. for (j=1; j<n; j++)
2.   key=A[j]
   /* menyisipkan A[j] kedalam urutan A[0] s/d A[j-1]
3.   i=j-1;
4.   while (i >= 0 && A[i] > key);
4.     A[i+1] = A[i];
5.     i--;
6.   A[i+1] = key;
}
```

Metoda Bagi-Serang

Akan dikenalkan pendekatan lain dalam membuat algoritma, yaitu **Metoda Bagi-Serang** (*divide-and-conquer*). Pendekatan ini akan kita pakai untuk algoritma contoh 2 dan contoh 3, untuk contoh 1 silahkan coba sendiri. Salah satu keuntungan metoda ini adalah cukup sederhana dan mudah dimengerti, selain itu kompleksitas algoritmanya dapat kita cari dengan mudah.

Dalam metoda *divide-and-conquer*, permasalahan yang besar dibagi menjadi beberapa sub permasalahan yang serupa dengan semula akan tetapi berukuran kecil. Sehingga sub-sub permasalahan ini dikerjakan seperti pengerjaan permasalahan semula (tinggal panggil dirinya sendiri, secara rekursif). Selanjutnya menggabung penyelesaian dari sub-sub permasalahan untuk mendapatkan penyelesaian permasalahan aslinya.

Ada tiga langkah utama dalam metoda *divide-and-conquer*, yaitu:

Divide (bagi) permasalahan kedalam sub-sub permasalahan (biasanya dua sub permasalahan, namun tidak harus).

Conquer (kerjakan) sub-sub permasalahan tersebut secara rekursif. Bila sub permasalahan dirasa sudah cukup kecil, dapat langsung diselesaikan.

Gabung penyelesaian-penyelesaian dari sub-sub permasalahan untuk mendapatkan penyelesaian permasalahan aslinya.

Misal permasalahan dinyatakan dalam bentuk input I . Untuk input yang berukuran kecil tidak kita selesaikan dengan cari *bagi-serang*, akan tetapi kita selesaikan secara langsung, anggap cara langsung ini kita beri nama *CaraLangsung*. Untuk membagi input I menjadi beberapa sub permasalahan kita namakan *Bagi* dan hasilnya menjadi sub permasalahan I_1, I_2, \dots, I_k . Penyelesaian dari sub-sub permasalahan ini dimisalkan S_1, S_2, \dots, S_k . Dan untuk mendapatkan penyelesaian akhir dengan cara menggabung (*Gabung*) penyelesaian S_1, S_2, \dots, S_k . Sehingga metoda *bagi-serang* dapat ditulis dalam sebagai berikut.

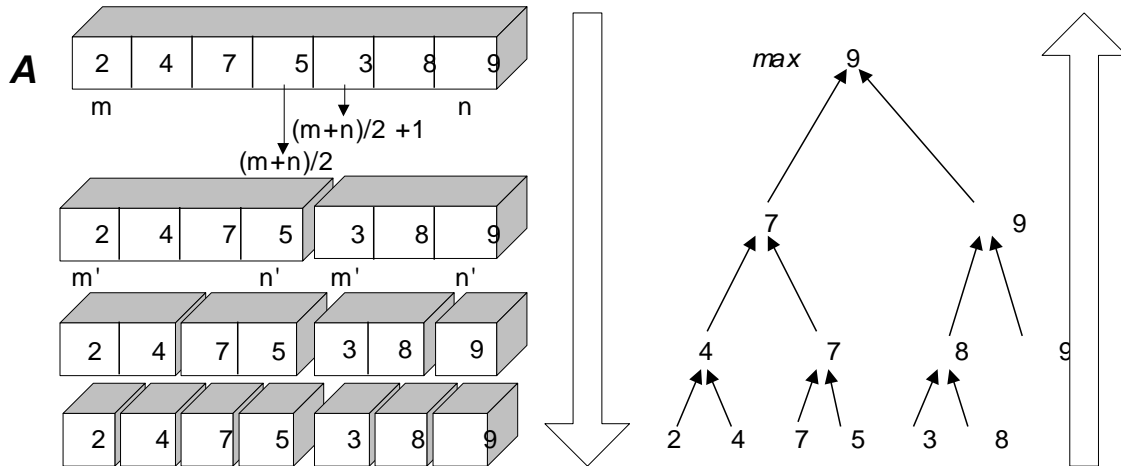
```
Bagi-Serang(Input) {  
1. if ( ukuran(Input) <= ukuranTerkecil)  
2.   return CaraLangsung(Input);  
3. else Bagi(Input,  $I_1, I_2, \dots, I_k$ );  
4.   for ( $i=1, i \leq k; i++$ )  
5.      $S_i = \text{Bagi-Serang}(I_i)$ ;  
6.   return Gabung(  $S_1, S_2, \dots, S_k$ );  
}
```

Contoh 4:

Permasalahan sama dengan contoh 2 di atas, namun akan kita buat dengan metoda *divide-and-conquer*.

Bagi array A menjadi dua sub array yang hampir sama besar. Array bagian pertama (kiri) dicari nilai maksimalnya, misal didapatkan hasil a . Dan array bagian kedua (kanan) dicari nilai

maksimalnya, misal didapatkan hasil b . Untuk mendapatkan nilai a dari bagian array kiri, caranya sama dengan pengerjaan array semula (seolah-olah seperti array semula). Begitu juga pengerjaan array bagian kanan. Untuk mendapatkan nilai masimal dari array, gabungkan hasil sub array sebelah kiri dan kanan. Bila $a > b$ maka maksimalnya adalah a , selain itu maksimalnya adalah b .



Gambar 1.5 Cara Kerja *CariMaksimal2*.

Sehingga secara lengkap algoritmanya sebagai berikut.

Input : Array A dengan n elemen bilangan.

Output : Bilangan $max = maks \{a_1, a_2, \dots, a_n\}$

Algoritma:

```

CariMaximal2(A,m,n,max) {
1. if ((n-m) == 0) /*Array hanya memuat 1 elemen */
2.   max=A[m];
3.   return max;
4. else
5.   CariMaximal2(A,m,[(m + n)/2],max);
6.   CariMaximal2(A, [(m + n)/2] + 1,n,max);
}

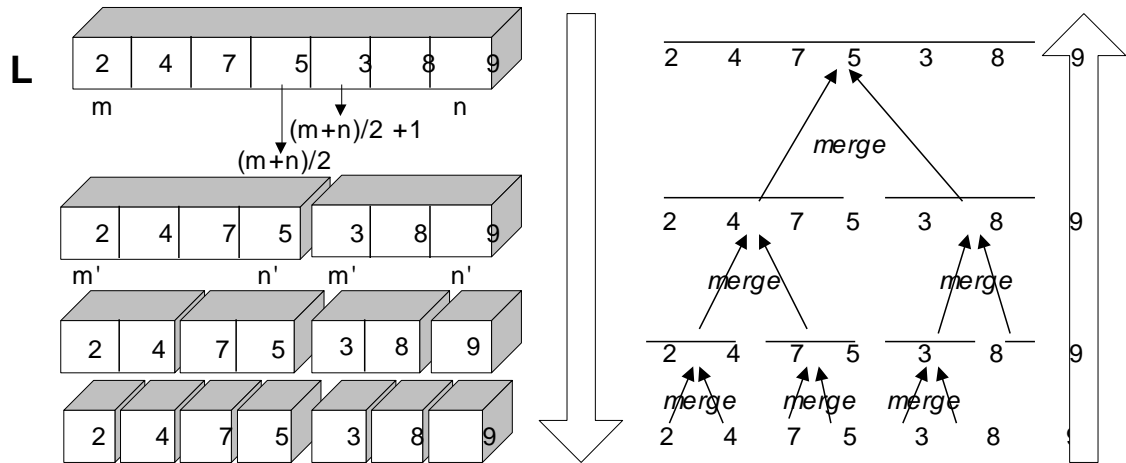
```

Contoh 5:

Permasalahan sama dengan contoh 3 di atas, namun akan kita buat dengan metoda *divide-and-conquer*.

Bagi array A menjadi dua sub array yang hampir sama besar. Array bagian pertama (kiri) diurut seperti cara pada array sebelumnya. Begitu juga array bagian kedua (kanan). Tinggal digabung antara dua array yang terurut menjadi satu array yang terurut. Cara ini yang disebut dengan

pengurutan penggabungan (*merge sort*). Proses penggabungan dua array kita beri nama saja proses '*merge*'. Isi dari '*merge*' dapat dipikirkan sendiri.



Gambar 1.6 Cara Kerja *merge-sort*.

Secara lengkap algoritma *merge-sort* adalah:

Input : Array A dengan n elemen bilangan.

Output : array A dengan elemen terurut naik.

Algoritma:

```

MergeSort(A,m,n) {
1. if (m < n) {
2.     k = (m+n)/2;
3.     MergeSort(A,m,k);
4.     MergeSort(A,k+1,n);
5.     Merge(A,m,n,k);
}
}

```

4. Analisis Algoritma

Analisis algoritma merupakan suatu metoda untuk mengetahui unjuk kerja suatu algoritma. Dapat berupa membandingkan biaya komputasi dari dua (atau lebih) algoritma untuk suatu permasalahan yang sama. Dari sudut pandang program komputer, analisis algoritma merupakan studi teoritis tentang unjuk kerja dari program komputer dan sumber daya yang digunakan program.. Analisis algoritma juga dapat kita pakai untuk memperkirakan apakah solusi yang kita berika dapat memenuhi kendala-kendala dari permasalahannya.

Analisis algoritma disini akan menekankan pada:

- Konsep laju pertumbuhan (*growth rate*).
Biaya komputasi algoritma akan naik seiring dengan naik ukuran input.
- Konsep batas atas dan batas bawah (*upper and lower bound*)
Bagaimana mengetahui batas ini, akan ditampilkan dengan menggunakan contoh-contoh untuk kasus sederhana.
- Konsep perbedaan antara biaya dari algoritma (program) dan biaya dari permasalahan.

Sebelum kita masuk pada apa yang harus kita lakukan dalam menganalisa algoritma, ada baiknya kita pikirkan hal-hal berikut ini.

- Bagaimana membandingkan (mencari efisiensi) dua algoritma untuk menyelesaikan suatu permasalahan?. Salah satu cara yang mudah adalah mengimplementasikan kedua algoritma tersebut dalam program komputer dan menjalankannya dengan diberi input yang sesuai. Selanjutnya diukur berapa banyak sumber daya (waktu, memori) yang dipakai untuk masing - masing program. Namun cara ini kurang begitu memuaskan, karena:
 - ✓ Seandainya kita hanya membutuhkan satu algoritma untuk menyelesaikan permasalahan, maka kita harus mengimplementasikan dan melakukan tes dua algoritma. Sudah barang tentu ini membuang banyak waktu dan tenaga.
 - ✓ Bisa jadi dalam implementasi, salah satu algoritma dapat diimplementasi dengan baik sedang implementasi algoritma satunya kurang begitu bagus atau kurang begitu mewakili algoritmanya (mungkin terlalu banyak menggunakan variabel pembantu atau operator sisipan).
 - ✓ Dalam melakukan tes kasus kedua algoritma tersebut, input yang dipakai bisa jadi baik untuk algoritma yang pertama namun tidak begitu baik untuk algoritma yang kedua. Hal ini akan mengakibatkan timbulnya kesimpulan yang tidak seharusnya.
- Dalam implementasi dan eksekusi suatu algoritma, membutuhkan tempat memori (*space*). Untuk permasalahan yang sama, dua algoritma dapat berbeda dalam jumlah ruang yang dipakai. Suatu algoritma dapat memakai memori lebih sedikit dari algoritma lainnya. Dan untuk besar input yang berbeda, berbeda pula banyaknya memori yang dipakai.

Dalam berbagai buku tentang analisis algoritma, suatu buku (penulis) mempunyai penekanan analisis yang berbeda dengan buku yang lainnya. Dalam buku ini akan melakukan analisis dengan memandang faktor - faktor sebagai berikut:

1. Kebenaran

2. Banyaknya kerja yang dilakukan
3. Banyaknya memori yang dipakai
4. Kesederhanaan
5. Optimalitas

Kelima faktor diatas akan dibahas secara lengkap untuk beberapa kasus seperti pencarian dan pengurutan. Hal ini untuk memberikan gambaran secara lengkap kepada pembaca bagaimana dan apa yang harus dilakukan dalam menganalisis algoritma. Namun untuk kasus - kasus yang lainnya hanya menekankan pada faktor kedua dan ketiga, yaitu masalah kompleksitas (*complexity*).

4.1. Kebenaran

Suatu algoritma harus menghasilkan output yang benar, ini harus. Sebelum menunjukkan bahwa suatu algoritma benar, maka harus dimengerti dahulu ‘apa yang dimaksud dengan benar?’. Suatu algoritma dikatakan **benar**, bila setiap diberikan input yang valid maka algoritma tersebut akan menghasilkan output yang diinginkan.

Input yang valid adalah suatu input yang sesuai dengan yang dispesifikasikan oleh algoritmanya. Misal suatu algoritma didesain dengan menggunakan input berupa array, kemudian diberikan input berupa senarai terkait (*linked list*). Input yang demikian ini tidak dikatakan valid. Dan kemudian apabila algoritma ini tidak menghasilkan output yang diinginkan belum bisa dikatakan bahwa algoritma ini salah (tidak benar). Contoh lain: suatu algoritma didesain dengan menggunakan input berupa dua buah bilangan bulat (*integer*), kemudian diberikan input berupa dua buah bilangan riil (*float*), maka input yang demikian ini dikatakan input yang tidak valid.

Untuk menunjukkan bahwa suatu algoritma benar, harus dibuktikan bahwa statemen - statemen yang ada dalam algoritma tersebut akan menghasilkan output yang diinginkan bila diberikan input yang valid. Ada kalanya, pembuktian ini membutuhkan banya lemma dan teorema. Untuk lebih jelasnya akan diberikan beberapa contoh.

Contoh 1:

Akan ditunjukkan bahwa algoritma *SequentialSearch* contoh didepan akan merupakan algoritma yang benar. Pandang kembali algoritmanya sebagai berikut.

```
SequentialSearch(A,n,x) {
1. j=0;
2. while (j<n && x != A[j])
3.     j++;
4. if (j<n) return j;
   else return -1;
```

}

Bukti kebenaran:

Jika kontrol pengerjaan algoritma mencapai baris 2 yang ke k kali, untuk $0 \leq k < n$, maka kondisi berikut ini akan dipenuhi:

- (i). $j = k$ dan $0 \leq i < k, A[i] \neq x$.
- (ii). Jika $k < n$ dan $A[k] = x$, algoritma akan berhenti dengan nilai $j = k$.
- (iii). Jika $k = n$, algoritma akan berhenti dengan nilai $j = -1$.

Ketiga kondisi di atas ini mewakili dari algoritma *SequentialSearch*, dan akan dibuktikan secara induksi bahwa statemen - statemen ini benar.

Basis induksi: ambil $k = 0$,

Kondisi (i) dipenuhi, yaitu $j=0$ dan x tidak sama dengan elemen A index sebelum 0 (karena index sebelum 0 ini tidak ada, secara otomatis dipenuhi).

Kondisi (ii), jika $0 < n$ dan $A[0]=x$, kontrol pada baris 2 gagal dan kontrol menuju baris 4 dengan nilai j tetap sama dengan 0.

Kondisi (iii), jika $k=n$ maka $j=n$. Sehingga tes pada baris 2 gagal dan kontrol menuju baris 4 dengan memberikan nilai $j = -1$.

Langkah asumsi : diasumsikan bahwa ketiga kondisi benar untuk $k < n$.

Akan dibuktikan benar untuk $k+1$.

Anggap kontrol sudah mencapai tes pada baris 2 yang ke $k+1$ kali, sehingga:

Kondisi (i), $j=k$ dan dari langkah asumsi mengatakan bahwa untuk $0 \leq i < k, A[i] \neq x$.

Ditambah lagi, oleh karena saat ini mencapai langkah 2 yang ke $k+1$ kali berarti $A[k] \neq x$ (kalau tidak demikian sudah berhenti terlebih dahulu). Sehingga dapat disimpulkan bahwa $0 \leq i < k + 1, A[i] \neq x$. Jadi dipenuhi kondisi (i).

Kondisi (ii), jika $k+1 < n$ dan $A[k+1]=x$, kontrol pada baris 2 gagal dan kontrol menuju baris 4 dengan nilai j tetap sama dengan $k+1$.

Kondisi (iii), jika $k+1=n$ maka $j=k+1=n$. Sehingga tes pada baris 2 gagal dan kontrol menuju baris 4 dengan memberikan nilai $j = -1$.

Dengan demikian algoritma di atas benar secara induksi.

Contoh 2:

Akan ditunjukkan bahwa algoritma *CariMaximal* contoh didepan akan merupakan algoritma yang benar. Pandang kembali algoritmanya sebagai berikut.

```
CariMaximal(A,n,max) {  
1. max=A[0];  
2. for(i=1; i<n ; i++)  
3.   if (max<A[i]) max=A[i];  
4. return max;  
}
```

Bukti kebenaran:

Algoritma di atas dapat kita ringkas dalam menjadi: setelah melakukan pengerjaan statemen *for*(*i*=1; *i*<*n* ; *i*++), bilangan yang berada pada variabel *max* adalah bilangan terbesar dari *A*[0], *A*[1], ..., *A*[*n*-1]. Ini akan kita buktikan kebenarannya dengan menggunakan bukti induksi.

Basis induksi: ambil *i* = 1 (dua elemen),

Jika *A*[0] > *A*[1] maka *max* = *A*[0], selainnya *max* = *A*[1]. Dengan demikian *max* adalah terbesar dari *A*[0] dan *A*[1].

Langkah asumsi : diasumsikan benar untuk *i*=*k*, yaitu *max* adalah bilangan terbesar dari *A*[0], *A*[1], ..., *A*[*k*].

Akan dibuktikan benar untuk *i*=*k*+1.

Pada saat mencapai baris 2 dengan *i*=*k*+1 dan kemudian mengerjakan baris 3 sebagai berikut.

```
if (max<A[k+1]) max=A[k+1];
```

Ingat bahwa (dari langkah asumsi) sampai disini variabel *max* berisi nilai terbesar dari *A*[0], *A*[01], ..., *A*[*k*]. Bila tes ekspresi pada baris 3 bernilai benar maka *max* = *A*[*k*+1], sehingga didapat *max* adalah nilai terbesar dari *A*[0], *A*[1], ..., *A*[*k*+1]. Dan apabila tes pada baris 3 tidak benar maka *max* masih tetap *Onilai* terbesar dari *A*[0], *A*[1], ..., *A*[*k*+1].

Jadi secara induksi algoritma ini terbukti benar.

4.2. Banyaknya Kerja Yang Dilakukan

Sebelum kita mengukur banyaknya kerja yang dilakukan oleh suatu algoritma, ada pertanyaan mendasar yaitu: kerja apa / bagaimana yang akan atau harus diukur dalam suatu algoritma?. Jawaban atas pertanyaan ini berbeda-beda untuk suatu algoritma dengan algoritma lainnya. Sebagai contoh pekerjaan / operasi yang dilakukan oleh suatu algoritma adalah:

pembandingan, penjumlahan / pengurangan, perkalian / pembagian, perpangkatan, mengunjungi alamat memori (pointer), dan yang lainnya.

Dalam satu algoritma mungkin ada lebih dari satu operasi, yang mana yang akan diukur / dihitung?. Lihat kembali algoritma *SequentialSearch*, disini ada oprasi penambahan nilai j , juga ada operasi pembandingan nilai x dengan elemen $A[j]$. Operasi pembandingan merupakan pekerjaan utama yang dilakukan oleh *SequentialSearch*, akan tetapi pekerjaan penambahan index j adalah pekerjaan sampingan (tambahan). Sudah barang tentu disini yang akan diukur adalah banyaknya operasi utama, yaitu pembandingan. Walaupun secara pekerjaan hardware, operasi penjumlahan lebih mahal dibandingkan dengan operasi pembandingan. Operasi penjumlahan dalam algoritma *SequentialSearch* ini akhirnya hanya dihitung sebagai konstanta pengali dari banyaknya operasi utama (pembandingan).

Kita ambil satu contoh lagi, yaitu algoritma untuk pengalian matrik. Ada dua operasi utama yaitu penjumlahan elemen dan perkalian elemen, namun ada juga operasi penambahan indek yang merupakan operasi sampingan. Diantara dua operasi utama ini mana yang harus diukur?. Idealnya keduanya diukur semua, yaitu: operasi penjumlahan ada berapa banyak dan operasi perkalian ada berapa banyak. Namun, mengingat operasi perkalian jauh lebih mahal dari pada operasi penjumlahan maka kita boleh mengukur banyaknya operasi perkalian saja dan mengabaikan pengukuran banyaknya operasi penjumlahan. Hal ini dapat dianalogikan dengan: apabila kita pergi untuk belanja mobil dan belanja parfum mobil, sudah barang tentu idealnya kita memperhitnugkan uang untuk belanja keduanya. Namun demikian kita juga boleh hanya memperhitungkan uang untuk belanja mobil saja, karena harga mobil jauh lebih mahal dari pada harga parfun mobil. Walaupun kita hanya akan belanja 30 mobil dan akan belanja 3000 perfum mobil.

Lihat kembali algoritma *CariMaksimal*, bila input array A hanya memuat 5 elemen maka algoritma tersebut akan mengerjakan pembandingan sebanyak 4 kali. Apabila input array A memuat 25 elemen maka algoritma tersebut akan mengerjakan pembandingan sebanyak 24 kali. Dan apabila input array A memuat n elemen maka algoritma tersebut akan mengerjakan pembandingan sebanyak $(n-1)$ kali. Hal ini menandakan bahwa banyaknya kerja yang dilakukan oleh suatu algoritma tergantung dari besarnya / ukuran input. Ukuran input dari algoritma *CariMaksimal* adalah banyaknya elemen pada array A atau n . Untuk lebih jelasnya lihat tabel 1.2 yang menampilkan conoth - contoh permasalahan dan operasi yang harus diukur dan ukuran inputnya.

No	Permasalahan	Operasi Yang Diukur	Ukuran Input
1	Cari x dalam array $A[1]$, $A[1]$, ... $A[n]$	Pembandingan	Banyaknya elemen, n .
2	Pengurutan elemen array $A[0]$, $A[1]$, ..., $A[n-1]$	Pembandingan	Banyaknya elemen, n .
3	Penjumlahan matrik $C_{m \times n} = A_{m \times n} + B_{m \times n}$	Penjumlahan	Banyaknya baris m dan banyaknya kolom n .
4	Perkalian matrik $C_{m \times n} = A_{m \times p} B_{p \times n}$	Perkalian, dan penjumlahan (bila perlu)	Banyaknya baris m , banyaknya kolom p dan n .
5	Penyelesaian sistem persamaan linier $A_{n \times n} b_{n \times 1} = b_{n \times 1}$	Perkalian, dan penjumlahan (bila perlu)	Banyaknya persamaan atau variabel, n .
6	Pengunjungan node pada pohon biner	Pointer	Banyaknya node pada pohon biner.

Sekarang kita akan mencoba menjalankan algoritma *SequentialSearch* dengan input berupa array $A[0]$, $A[1]$, ..., $A[n]$ dan bilangan yang dicari x . Akan terjadi kemungkinan - kemungkinan sebagai berikut.

- Apabila nilai x sama dengan nilai $A[0]$, maka algoritma akan melakukan kerja pembandingan (mencapai kontrol pada baris 2 dan mebandingkan x dengan $A[0]$) sebanyak 1 kali. Keadaan terbaik (*best case*)
- Apabila nilai x sama dengan nilai $A[1]$, maka algoritma akan melakukan kerja pembandingan sebanyak 2 kali.
- Apabila nilai x sama dengan nilai $A[2]$, maka algoritma akan melakukan kerja pembandingan sebanyak 3 kali. Dan seterusnya.
- Apabila nilai x tidak ada di array A , maka algoritma akan melakukan kerja pembandingan sebanyak n kali. Ini merupakan kasus terburuk (*worst case*).

Dari contoh percobaan di atas, kita dapat mengatakan bahwa algoritma *SequentialSearch* dalam keadaan / kasus terbaik (*best case*) melakukan kerja 1 kali. Dalam analisa algoritma informasi *best case* ini tidak berguna sama sekali, sehingga tidak perlu ditampilkan. Kita juga dapat mengatakan bahwa dalam kasus terburuk (*worst case*), algoritma *SequentialSeach*

melakukan kerja sebanyak $(n+1)$ kali. Atau bahkan kita dapat menghitung kerja secara rata - rata (*average case*).

Analisis Kerja Rata-Rata Dan Kerja Terburuk

Kita akan menghitung banyaknya kerja rata-rata (*average case* atau *average complexity*) dan banyaknya kerja terburuk (*worst case* atau *complexity*) dari suatu algoritma. Di atas telah ditunjukkan bahwa suatu algoritma yang diberi input berukuran n , dapat berakibat kerja terbaik dan terburuk. Hal ini mengatakan bahwa input yang berukuran n ada banyak macamnya yang akan mengakibatkan algoritmanya kerja berbeda-beda.

Himpunan macam-macam input (dari suatu algoritma) yang berukuran n dinamakan *domain input* dan disimbolkan dengan D_n . Kita misalkan I adalah elemen D_n , probabilitas bahwa input I akan dipakai adalah $P(I)$, dan banyaknya operasi yang dilakukan oleh algoritma bila diberi input I adalah $t(I)$. Dengan menggunakan simbol - simbol ini didefinisikan:

- Banyaknya kerja rata - rata (*average case*) untuk input yang berukuran n , disimbolkan dengan $A(n)$ dan didefinisikan:

$$A(n) = \sum_{I \in D_n} p(I)t(I)$$

Disini kita harus mendata semua elemen input I didalam D_n dan menghitung banyaknya kerja untuk masing-masing input I , yaitu $t(I)$ dan menentukan probabilitas untuk input I tersebut. Banyaknya kerja untuk input I , $t(I)$, dapat kita tentukan dengan mudah. Masukkan input I ke algoritma kemudian hitung berapa banyak kerjanya, itulah $t(I)$. Namun demikian untuk menentukan nilai $p(I)$ tidaklah mudah. Kita sulit untuk mengetahui berapa probabilitas bahwa input I akan dipakai oleh algoritmanya. Sebagai gambaran kita ambil contoh *SequentialSearch*, bila algoritma ini diimplementasi untuk permasalahan pencarian buku di perpustakaan, maka kemungkinan bahwa buku x yang dicari/ditanyakan ada di dalam array/database buku akan sangat besar sekali. Karena orang datang ke suatu perpustakaan mencari buku yang ditanyakan sudah memperkirakan akan ada. Namun demikian apabila algoritma tersebut diimplementasikan pada permasalahan / database kejahatan di kepolisian, kemudian ada seorang datang mencari kelakuan baik dan polisi akan melakukan pencarian apakah orang ini pernah berbuat jahat. Kemungkinan tidak ada akan besar sekali. Dengan ini dapat kita bayangkan akan sangat sulit untuk menentukan nilai $P(I)$. Oleh karena sulit, kita dapat memakai asumsi-asumsi.

- Banyaknya kerja terburuk (*worst case*) untuk input yang berukuran n , disimbolkan dengan $W(n)$ dan didefinisikan:

$$W(n) = \max_{I \in D_n} t(I)$$

Melihat rumusan di atas, kita harus mendata semua input didalam D_n dan menghitung banyaknya kerja untuk masing-masing input I , yaitu $t(I)$. Kemudian diambil yang terbesar. Ini sepertinya agak rumit, karena kita harus menentukan $t(I)$ untuk semua input I . Akan tetapi tidaklah demikian, cukup kita pilih sebuah input yang akan mengakibatkan algoritmanya kerja terburuk, kemudian hitung banyaknya kerja $t(I)$. Ini adalah nilai $W(n)$.

Contoh 1:

Tentukan nilai kerja rata-rata $A(n)$ dan kerja terburuk $W(n)$ dari algoritma *SequentialSearch* pada permasalahan pencarian lokasi elemen x di dalam array A ?

Operasi dasar yang akan kita hitung adalah perbandingan nilai x dengan elemen-elemen $A[i]$.

Kompleksitas rata-rata, $A(n)$:

Untuk menentukan D_n , macam-macam input dapat kita golongankan menurut dimana lokasi x pertama kali muncul di dalam array A . Sehingga D_n mempunyai elemen sebanyak $n+1$ buah input.

Untuk $0 \leq j < n$, I_j menyatakan suatu kasus dimana x ada pada lokasi/index ke j dengan input ini algoritma akan bekerja sebanyak $t(I_j) = j + 1$. Dan I_n menyatakan kasus dimana x tidak ada di dalam array A dengan input ini algoritma akan bekerja sebanyak $t(I_n) = n$.

Probabilitas untuk masing-masing input kita asumsikan *equally likely* / punya kesempatan sama. Misal q adalah probabilitas bahwa input x ada di dalam array A , dan probabilitas bahwa x tidak ada di dalam array A adalah $1-q$. Untuk $0 \leq j < n$, $p(I_j) = \frac{q}{n}$ dan $p(I_n) = 1 - q$.

$$A(n) = \sum_{I \in D_n} p(I)t(I)$$

$$A(n) = \sum_{j=0}^{n-1} \frac{q}{n} j + (1 - q)n$$

$$A(n) = \frac{q}{n} \sum_{j=0}^{n-1} j + (1-q)n$$

$$A(n) = \frac{q}{n} \frac{n(n+1)}{2} + (1-q)n$$

$$A(n) = \frac{q(n+1)}{2} + (1-q)n$$

Jika diketahui bahwa x selalu ada di dalam array ($q=1$), maka $A(n) = \frac{(n+1)}{2}$. Namun demikian, jika x ada di dalam array hanya 50%, maka $A(n) = \frac{(n+1)}{4} + \frac{n}{2} \approx \frac{3}{4}n$.

Kompleksitas rata-rata, $A(n)$:

Input yang mengakibatkan algoritma *SequentialSearch* kerja terburuk adalah I_n , yaitu x tidak ada di dalam array A . Sehingga $W(n) = n$ atau kalau kita hitung dengan menggunakan rumus.

$$W(n) = \max_{I \in D_n} t(I)$$

$$W(n) = \max\{t(I_j) | 0 \leq j \leq n\} = n$$

Contoh 2:

Permasalahan perkalian matriks $A_{n \times n} = (a_{ij})$ dan $B_{n \times n} = (b_{ij})$, elemen matrik adalah bilangan riil. Buat algoritma perkalian matrik $C=AB$ dan hitung kerja rata-rata dan terburuknya.

Input : Array dari matriks $A_{n \times n} = (a_{ij})$ dan $B_{n \times n} = (b_{ij})$, ukuran matriks n .

Output : Matriks $C_{n \times n} = (c_{ij})$, dengan $c_{ij} = \sum_{k=0}^{n-1} a_{ik}b_{kj}$,

Algoritma:

```
PerkalianMatriks(A,B,C,n) {
1. for(i=0; i<n; i++)
2.   for(j=0; j<n; j++){
3.     Cij=0;
4.     for(k=0; k<n; k++)
5.       Cij=Cij + Aik*Bkj;
   }
}
```

Operasi dasar yang ada adalah perkalian dan penjumlahan. Kedua operator ini boleh dihitung semuanya, atau cukup menghitung banyaknya operasi perkalian.

Domain dari input yang berukuran n , D_n , hanya memuat satu elemen (satu macam input). Yaitu pasangan input matriks A dan B yang berukuran n . Bagaimanapun bentuk pasangan matriks A

dan B , algoritma *Perkalianmatriks* akan melakukan proses secara sama, sehingga inputnya hanya ada satu macam. Karena hanya ada satu input, maka kerja rata-rata dan terburuknya sama.

Matriks C mempunyai elemen sebanyak n^2 , dan untuk memperoleh nilai satu elemen c_{ij} dibutuhkan perkalian sebanyak n kali dan operasi penjumlahan sebanyak $(n-1)$ kali. Sehingga banyaknya perkalian untuk menghasilkan matriks C ada n^3 , dan banyaknya operasi penjumlahan ada $n^2(n-1)$.

Banyaknya kerja untuk operasi perkalian adalah:

$$W(n) = A(n) = n^3$$

Banyaknya kerja untuk operasi penjumlahan adalah:

$$W(n) = A(n) = n^2(n - 1)$$

Dari penampilan contoh-contoh di atas, menekankan bagaimana menghitung *average case* dan *worst case*. Juga menunjukkan bahwa $A(n)$ itu bukannya setengah dari $W(n)$, bahkan mungkin bisa juga sama.

Untuk penghitungan $A(n)$ tidaklah mudah (untuk permasalahan secara umum), karena kita harus menghitung kerja algoritma untuk semua input dan menghitung probabilitasnya. Penghitungan nilai probabilitas disertai dengan asumsi-asumsi. Untuk kasus - kasus lain pada buku ini sering tidak menampilkan banyaknya kerja rata-rata.

4.3. Banyaknya Memori Yang Dipakai

Suatu program yang merupakan implementasi dari suatu algoritma memerlukan tempat penyimpanan di dalam memori, yaitu tempat untuk menyimpan instruksi, konstanta, dan variabel yang digunakan oleh program tersebut. Banyaknya sel memori yang dipakai oleh program sangat tergantung dari implementasinya. Implementasi yang satu bisa jadi lebih sedikit menggunakan tempat memori dari pada implementasi lainnya. Untuk menganalisis banyaknya memori yang dipakai oleh program, dapat juga kita analisis melalui algoritmanya.

Dalam menganalisis banyaknya sel memori yang dipakai, kita hanya menganalisis banyaknya sel memori yang dipakai oleh variabel/konstanta ekstra. Variabel/konstanta ekstra adalah variabel selain yang dipakai untuk mengelola input. Sel memori yang dipakai oleh variabel/konstanta ekstra dinamakan memori ekstra (*extra space*). Jika suatu algoritma mempunyai memori ekstra konstan (tidak tergantung dari ukuran input), maka algoritma tersebut

dikatakan kerja ditempat (*work in place*). Algoritma yang kerja ditempat ini adalah algoritma yang baik dari segi pemakaian memori.

Contoh 1:

Apakah algoritma *SequentialSearch* merupakan algoritma yang kerja ditempat?.

Lihat kembali algoritma *SequentialSearch*, variabel yang dipakai untuk menyimpan input adalah: array A , n yang menyatakan banyaknya elemen array, dan x adalah elemen yang dicari. Selain itu ada variabel ekstra j . Berapapun ukuran input (besarnya n) memori ekstranya akan konstan, yaitu yang dipakai untuk variabel j . Oleh karena itu algoritma *SequentialSearch* ini kerja ditempat, jadi algoritma ini baik dari segi pemakaian memori.

Contoh 2:

Pada permasalahan mencari nilai terbesar max dari semua elemen array A , di depan telah didesain dua algoritma yaitu: *CariMaksimal*, dan *CariMaksimal2*. Apakah kedua algoritma tersebut kerja ditempat?.

Pada algoritma *CariMaksimal*, memori ekstra-nya adalah sel memori untuk variabel i dan max . Memori ekstra ini tidak tergantung dari ukuran input, oleh karena itu *CariMaksimal* merupakan algoritma yang kerja ditempat.

Pada algoritma *CariMaksimal2*, memori ekstra-nya adalah sel memori untuk variabel indeks dari array m , n , dan variabel max . Ini kelihatannya konstan. Akan tetapi, karena algoritmanya adalah rekursif, setiap kali pemanggilan dirinya sendiri membentuk stack memori dan saat itu juga menciptakan parameter baru m , n , dan max (variabel lokal) pada stack tersebut. Proses pemanggilan dirinya sendiri ini tergantung dari besarnya input (banyaknya elemen A), semakin besar ukuran input semakin banyak pemanggilan dirinya sendiri. Sehingga algoritma *CariMaksimal2* merupakan algoritma yang tidak kerja ditempat.

Dari segi pemakaian memori, algoritma *CariMaksimal* lebih baik dari pada algoritma *CariMaksimal2*.

4.4. Kesederhanaan

Dalam suatu permasalahan, biasanya (tidak selalu) ada algoritma yang sederhana dan metoda penyelesaian yang dipakai terlihat secara langsung, namun waktu pengerjaannya kurang efisien. Ada pula algoritma yang sangat rumit namun efisien dalam hal kecepatan.

Algoritma yang sederhana biasanya mudah untuk diketahui kebenarannya, mudah untuk diprogram termasuk proses debug dan modifikasi program. Sehingga dalam implementasi tidak memakan waktu lama. Hal demikian ini menjadikan suatu pertimbangan dalam pemilihan algoritma. Apabila kasusnya jarang terjadi atau modul program jarang dipakai, maka kesederhanaan algoritma ini menjadi prioritas dalam pemilihan (walaupun kurang efisien). Namun demikian apabila modul programnya sangat sering dipakai, algoritma yang sederhana (tapi kurang efisien) menjadi sesuatu hal yang tidak menarik. Lebih baik pilih algoritma yang efisien kerjanya, walaupun membutuhkan waktu implementasi yang agak lama.

4.5. Optimalitas

Setiap permasalahan pasti mempunyai sejumlah langkah untuk menyelesaikannya, langkah penyelesaian ini dinamakan kompleksitas dari permasalahan. Dalam suatu permasalahan dimungkinkan adanya beberapa algoritma dengan tipe operasi dasar yang berbeda. Dua Algoritma yang mempunyai tipe operasi dasar sejenis (sama) dikatakan kedua algoritma tersebut berada dalam satu **kelas algoritma**. Sehingga dimungkinkan dalam satu permasalahan ada lebih dari satu kelas algoritma.

Untuk menganalisis kompleksitas permasalahan (bukan kompleksitas algoritma), kita pilih suatu kelas algoritma dan kita hitung kompleksitasnya. Dari sini kita bisa mengetahui berapa banyak operasi dasar yang diperlukan untuk menyelesaikan permasalahan ini. Ingat bahwa ini adalah kompleksitas permasalahan. Sehingga apapun algoritma yang telah dan akan dibuat orang, paling sedikit melakukan operasi dasar sebanyak kompleksitas permasalahan.

Suatu algoritma dikatakan **optimal** dalam kerja terburuk bila tidak ada algoritma lain dalam kelas yang sama yang mempunyai operasi dasar (dalam kerja terburuk) lebih sedikit. Jadi algoritma yang optimal bukan berarti ‘terbaik yang diketahui’, akan tetapi algoritma yang optimal adalah ‘terbaik dari yang mungkin’ termasuk yang belum diketemukan.

Mengukur Keoptimalan Algoritma.

Bagaimana mengukur bahwa suatu algoritma optimal ? Apakah kita harus mengukur satu per satu dari setiap algoritma yang mungkin?. Jelas tidak demikian, sebab bagaimana kita harus

menghitung kerja dari algoritma yang belum dipikirkan orang. Untuk itu, kita cukup membuktikan suatu teorema yang menyatakan bahwa untuk menyelesaikan suatu permasalahan diperlukan paling sedikit $F(n)$ operasi dasar. Sejumlah operasi dasar minimal yang dibutuhkan untuk menyelesaikan suatu permasalahan dinamakan **batas bawah** (*lower bound*) dari permasalahan tersebut. Setelah kita mengetahui batas bawah, selanjutnya adalah mengukur kerja dari algoritma yang akan diketahui keoptimalannya. Apabila kerja dari algoritma tersebut sama dengan batas bawahnya, maka algoritma tersebut dikatakan optimal.

Untuk lebih jelasnya, misal kita ingin menentukan apakah suatu algoritma A optimal?. Kita ikuti langkah berikut:

1. Pandang algoritma A yang kita anggap efisien. Analisis algoritma A dan hitung banyaknya operasi dasar yang dilakukannya bila diberikan input yang berukuran n , kita namakan $W(n)$.
2. Cari batas bawah untuk kelas algoritma A dengan cara: membuktikan suatu teorema yang menyatakan bahwa ‘setiap algoritma yang berada dalam satu kelas dengan algoritma A, maka paling sedikit mempunyai operasi dasar sebanyak $F(n)$ apabila diberi input berukuran n ’.
3. Jika $W(n) = F(n)$ maka algoritma A optimal. Jika tidak demikian maka ada algoritma yang lebih baik dari algoritma A atau akan dapat ditemukan batas bawah yang lebih besar.

Pada langkah 2, $F(n)$ merupakan batas bawah banyaknya operasi dasar yang diperlukan untuk menyelesaikan permasalahan. Sehingga algoritma yang sudah ditemukan ataupun yang belum ditemukan paling sedikit melakukan operasi dasar sebesar $F(n)$. Namun, jika kita menghitung banyaknya kerja yang dilakukan oleh algoritma tertentu, maka yang kita dapatkan sebagai batas atas (*upper bound*) untuk menyelesaikan permasalahan.

Apabila $W(n) > F(n)$, ada dua kemungkinan yaitu: dapat dibuat algoritma lain yang lebih baik atau dapat dicari batas bawah yang lebih besar. Oleh karena itu, langkah - langkah di atas juga dapat dipakai untuk menentukan apakah suatu algoritma masih dapat diperbaiki atau tidak.

Contoh 1:

Pandang kembali permasalahan mencari elemen maksimal (*max*) dari array A dengan algoritma yang akan kita analisis keoptimalannya adalah algoritma *CariMaksimal*. Kelas algoritmanya adalah: algoritma yang membandingkan elemen array A dan menyalinnya divariabel lain. Operasi

dasar yang diukur adalah membandingkan dua elemen array. Batas atas dari permasalahan ini misal sudah diketahui yaitu n (banyaknya elemen A).

Langkah 1, kerja terburuk dari algoritma *CariMaksimal* adalah $W(n) = n-1$. Dengan demikian ada perbaikan batas atas yaitu menjadi $n-1$.

Langkah 2, Untuk menentukan batas bawah kita anggap semua elemen array berbeda. Sehingga ada $n-1$ buah elemen yang bukan elemen maksimal. Suatu elemen dikatakan bukan elemen maksimal apabila elemen tersebut lebih kecil dari paling sedikit satu elemen dari array A . Dalam proses perbandingan, ada sebanyak $n-1$ elemen pecundang. Dan Setiap kali perbandingan hanya ada 1 elemen pecundang, sehingga paling sedikit harus terjadi perbandingan sebanyak $n-1$. Jadi didapatkan batas bawah $F(n) = n-1$. Algoritma apapun untuk menyelesaikan permasalahan ini pasti paling sedikit melakukan kerja perbandingan elemen sebanyak $n-1$ kali.

Langkah 3, dari langkah 1 dan 2 didapatkan $W(n) = F(n)$. Oleh karena itu, algoritma *CariMaksimal* adalah optimal.

Contoh 2:

Pandang kembali permasalahan mengalikan dua matrik bujur sangkar yang berukuran n , yaitu $C_{n \times n} = A_{n \times n} B_{n \times n}$ dengan algoritma yang akan kita analisis keoptimalannya adalah algoritma *PerkalianMatrik*. Kelas algoritmanya adalah: algoritma yang melakukan perkalian, penjumlahan, dan pengurangan elemen matrik. Operasi dasar yang diukur adalah perkalian dua elemen matrik.

Langkah 1, kerja terburuk dari algoritma *PerkalianMatrik* adalah $W(n) = n^3$. Dengan demikian batas atas dari perkalian matrik ini adalah n^3 .

Langkah 2, Untuk mengalikan matrik paling sedikit dibutuhkan n^2 perkalian, sehingga $F(n) = n^2$.

Langkah 3, $W(n) > F(n)$, kita tidak dapat menarik kesimpulan apakah algoritma *PerkalianMatriks* optimal atau tidak. Ada dua kemungkinan yaitu: batas bawah dapat ditingkatkan atau dapat dicari algoritma yang lebih baik. Sementara ada peneliti yang mencoba membuktikan bahwa batas bawah perkalian matrik lebih dari n^2 . Saat ini telah ditemukan algoritma perkalian matrik yang melakukan perkalian sebanyak $n^{2.376}$. Oleh karena itu algoritma *PerkalianMatriks* tidak optimal. Namun apakah perkalian matrik dengan kerja $n^{2.376}$ optimal?, kita belum mengetahuinya.

Keoptimalan yang dibahas di atas hanya menyangkut keoptimalan dalam kerja terburuk. Bagaimana dengan keoptimalan dalam kerja rata-rata ?. Langkah - langkah pencarian dan pembuktiannya sama, hanya saja dihitung dalam kerja rata-rata.

Sebenarnya optimal tidak saja dalam hal banyaknya kerja (operasi dasar yang dilakukan), namun dapat juga kita lakukan analisis untuk keoptimalan dalam penggunaan memori. Langkah-langkahnya juga mirip. Sehingga dimungkinkan adanya algoritma optimal dalam kerja terburuk juga optimal dalam penggunaan memori atau dapat juga hanya optimal salah satu.

5. Klasifikasi Fungsi Berdasarkan Laju Pertumbuhannya

Didepan kita telah melakukan penghitungan banyaknya operasi dasar yang dilakukan oleh suatu algoritma. Penghitungan yang dilakukan tersebut masih belum teliti, karena ada operator sampingan tidak dihitung. Sehingga hasil analisis kita masih kasar. Jumlah keseluruhan operasi yang dilakukan oleh suatu algoritma sebanding dengan banyaknya operasi dasar yang dihitung. Sebagai contoh, algoritma *SequentialSearch* melakukan kerja sebanyak $W(n) = n$, jumlah keseluruhan operasi yang dikerjakan oleh *SequentialSearch* adalah cn dengan c adalah konstanta positif.

Pada saat kita membandingkan dua algoritma, katakan algoritma A dan algoritma B. Misal Algoritma A mempunyai operasi dasar $W_1(n) = 2n$ atau jumlah operasi totalnya adalah $2c_1n$ dan algoritma B mempunyai operasi dasar $W_2(n) = 6n$ atau jumlah operasi totalnya adalah $6c_2n$. Pertanyaan yang muncul adalah: bagaimana ketelitian hasil perbandingan kedua algoritma tersebut ?. Atau pertanyaan yang lebih spesifik: mana yang lebih cepat ?. Jelas kita tidak mengetahui jawabannya. Jika konstanta $c_1 > 3c_2$ (misal algoritma A banyak operasi tambahan) maka algoritma A lebih lambat dari algoritma B. Atau bisa sebaliknya. Dua fungsi yang hanya berbeda pada faktor pengali konstanta, kedua fungsi tersebut berada dalam satu kelas. Sehingga algoritmanya mempunyai kompleksitas yang sama.

Sekarang kita ambil pemisalan yang lain. Anggap algoritma A mempunyai $W_1(n) = n^3$ atau jumlah operasi totalnya adalah c_1n^3 dan algoritma B mempunyai operasi dasar $W_2(n) = 5n^2$ atau jumlah operasi totalnya adalah $5c_2n^2$. Mana yang lebih cepat ?. Untuk nilai yang kecil $n < \frac{5c_2}{c_1}$ maka algoritma A akan lebih cepat dari pada algoritma B. Namun apabila $n > \frac{5c_2}{c_1}$, maka algoritma B lebih cepat dari pada algoritma A. Kalau kita lihat pada laju pertumbuhan fungsinya, fungsi kubik mempunyai laju pertumbuhan lebih besar dari pada fungsi kuadratik. Sehingga untuk ukuran input yang sangat besar, algoritma B jauh lebih baik dari pada algoritma A.

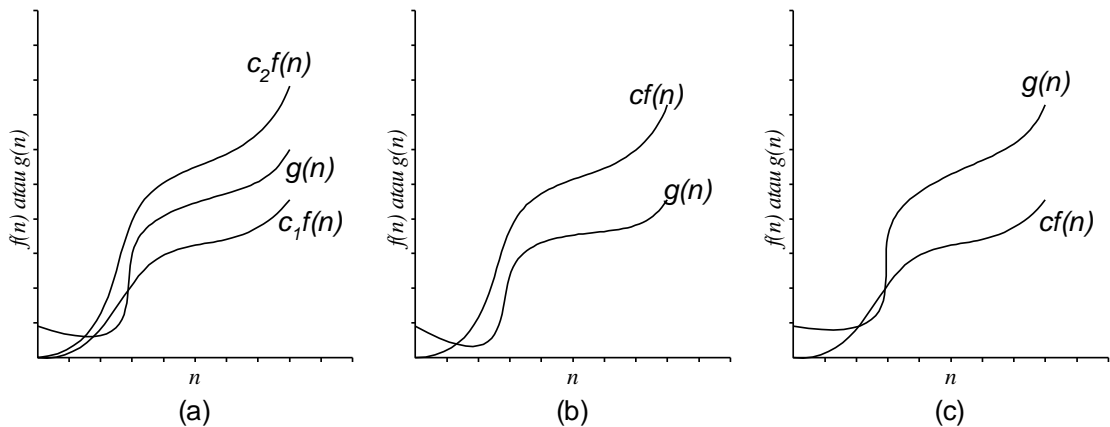
Dengan gambaran contoh di atas, kita ingin mempunyai suatu cara untuk mengklasifikasikan fungsi yang tidak tergantung pada konstanta dan mengabaikan nilai variabel bebas (ukuran input) yang kecil. Klasifikasi fungsi seperti ini berdasarkan **laju pertumbuhan asimptotik** atau **order** dari fungsinya.

Berikut ini akan ditampilkan beberapa definisi dan notasi yang umum dipakai. Misal N merupakan himpunan bilangan alam, R merupakan bilangan riil, R^+ merupakan bilangan riil positif, R^* merupakan bilangan riil non negatif, $f: N \rightarrow R^+$ dan $g: N \rightarrow R^+$ didefinisikan:

- $O(f(n)) = \{ g(n) / \text{untuk beberapa konstanta } c \in R^+ \text{ dan } n_0 \in N \text{ berlaku } g(n) \leq cf(n) \text{ untuk semua } n \geq n_0 \}$. Atau fungsi $g(n)$ didominasi secara asymtotik oleh fungsi $f(n)$.
- $\Omega(f(n)) = \{ g(n) / \text{untuk beberapa konstanta } c \in R^+ \text{ dan } n_0 \in N \text{ berlaku } g(n) \geq cf(n) \text{ untuk semua } n \geq n_0 \}$. Atau fungsi $g(n)$ mendominasi secara asymtotik fungsi $f(n)$.
- $\theta(f(n)) = \{ g(n) / \text{untuk beberapa konstanta } c_1, c_2 \in R^+ \text{ dan } n_0 \in N \text{ berlaku } c_1f(n) \leq g(n) \leq c_2f(n) \text{ untuk semua } n \geq n_0 \}$. Atau fungsi $g(n)$ mendominasi secara asymtotik fungsi $f(n)$ dan fungsi $g(n)$ didominasi secara asymtotik oleh fungsi $f(n)$.

Himpunan fungsi $O(f(n))$ sering disebut dengan ‘**big oh of $f(n)$** ’ atau ‘**oh $f(n)$** ’. Meskipun $O(f(n))$ merupakan himpunan, lebih paktis mengatakan ‘ $g(n)$ adalah oh $f(n)$ ’ dari pada mengatakan ‘ $g(n)$ adalah anggota dari oh $f(n)$ ’. Dan penulisan $g(n) \in O(f(n))$ lebih praktis ditulis dengan penulisan $g(n) = O(f(n))$. Himpunan fungsi $O(f(n))$ sering disebut dengan **Order $f(n)$** .

Kalau melihat definisi O dan Ω , ini merupakan dua definisi yang mirip karena pada



dasarnya hanya masalah ‘mendominasi’(kata kerja aktif) dan ‘didominasi’(kata kerja pasif). Oleh karena itu kita lebih sering menggunakan salah satu saja yaitu O .

Gambar 1.7

- (a) Fungsi $f(n)$ mendominasi $g(n)$ dan $f(n)$ didominasi oleh $g(n)$, (b) Fungsi $f(n)$ mendominasi $g(n)$, (c) Fungsi $f(n)$ didominasi $g(n)$.

Contoh 1:

Pandang fungsi $f(n) = \frac{1}{n} + 7$ dan $g(n) = n + 1$.

$f(n)$ didominasi oleh $g(n)$ atau $f(n) = O(g(n))$, karena: kalau diambil $c=1$ berlaku $f(n) \leq g(n)$ untuk semua nilai $n \geq n_0$ ($n_0 = 7$).

$$\frac{1}{n} + 7 \leq c(n + 1)$$

$$1 + 7n \leq c(n^2 + n)$$

$$1 \leq c(n^2 + n) - 7n$$

$$1 \leq n^2 - 6n \text{ untuk } c = 1 \text{ dipenuhi untuk semua nilai } n \geq 7.$$

Akan tetapi $f(n)$ tidak mendominasi $g(n)$ atau $f(n) \geq cg(n)$, karena: kalau dianggap $f(n)$ mendominasi $g(n)$ atau $g(n) \leq cf(n)$ untuk $n \geq n_0$, maka:

$$n + 1 \leq c\left(\frac{1}{n} + 7\right) \text{ untuk } n \geq n_0$$

$$n^2 + n \leq c(1 + 7n) \text{ untuk } n \geq n_0$$

Untuk $n = 7c$ didapat $49c^2 + 7c \leq c + 49c^2$. Hal ini tidak mungkin, oleh karena itu $f(n)$ tidak mendominasi $g(n)$.

Ada cara lain untuk menunjukkan bahwa $g(n) = O(f(n))$, yaitu:

$$g(n) = O(f(n)) \text{ jika}$$

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = K \text{ untuk } K \in R^+$$

Ini mempunyai arti bahwa: jika nilai limit diatas ada (bukan tak berhingga), maka laju pertumbuhan fungsi $g(n)$ tidak lebih cepat dari laju $f(n)$. Dan jika nilai limitnya tak berhingga maka laju pertumbuhan fungsi $g(n)$ lebih cepat dari laju $f(n)$.

Contoh 2:

Pandang fungsi $f(n) = n^3$ dan $g(n) = 100n^2 + 10n + 25$. Akan ditunjukkan $g(n) = O(f(n))$, akan tetapi $f(n) \neq O(g(n))$.

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{100n^2 + 10n + 25}{n^3}$$

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{200n + 10}{3n^2}$$

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{200}{6n} = 0$$

Kita lihat, karena nilai limitnya ada yaitu 0, maka $g(n) = O(f(n))$.

Sekarang kita lihat:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{n^3}{100n^2 + 10n + 25}$$

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{3n^2}{200n + 10}$$

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{6n}{200} = \infty$$

Karena nilai limitnya tidak ada atau ∞ , maka $f(n) \neq O(g(n))$.

Contoh 3:

Algoritma Bubble Sort mempunyai kerja terburuk $\frac{n(n-1)}{2}$. Tunjukkan bahwa $\frac{n(n-1)}{2} = O(n^2)$.

Kalau kita lihat definisi order, $g(n) = O(f(n))$ bila $f(n)$ mendominasi $g(n)$ dan $g(n)$ didominasi oleh $f(n)$. Sehingga kita bisa menunjukkannya dengan cara:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = K \text{ untuk } K \in R^+.$$

$K \in R^+$ berarti bahwa K tidak boleh bernilai 0.

Pada contoh ini, $g(n) = \frac{n(n-1)}{2}$ dan $f(n) = n^2$. Sekarang kita lihat nilai limit berikut ini.

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{n(n-1)/2}{n^2}$$

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{n - \frac{1}{2}}{2n}$$

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{1}{2} = \frac{1}{2}$$

Dengan demikian $\frac{n(n-1)}{2} = O(n^2)$.

Dengan demikian, untuk mengetahui apakah suatu fungsi berada dalam satu kelas dengan fungsi lainnya, kita dapat memakai order O. Atau O dapat kita pakai untuk menentukan bahwa suatu algoritma mempunyai kompleksitas sama dengan algoritma lainnya. Misal algoritma A mempunyai kompleksitas $W_1(n)$ dan algoritma B mempunyai kompleksitas $W_2(n)$, jika $W_1(n) = O(f(n))$ dan $W_2(n) = O(f(n))$ maka algoritma A dan B berada dalam satu kelas kompleksitas.

6. Lebih Jauh Tentang Order.

Kita akan melihat perbedaan kecepatan algoritma yang berbeda order (kelas) dalam menyelesaikan permasalahan. Pada Tabel 1.2 menampilkan waktu eksekusi dari beberapa algoritma yang berbeda dalam banyaknya kerja apabila diberikan input sebesar n . Dan menampilkan banyaknya ukuran input yang mampu diselesaikan dalam waktu 1 detik dan 1 menit. Pembaca dapat membedakan sendiri laju pertumbuhan beberapa fungsi. Perlu diingat bahwa faktor konstanta yang besar yang berada pada kelas algoritma $O(n)$ dan $O(n \log n)$ tidak mengakibatkan algoritma tersebut lebih lambat dibandingkan kelas algoritma $O(n^2)$, $O(n^3)$, dan $O(2^n)$ yang mempunyai faktor konstanta kecil.

Tabel 1.2
Laju Pertumbuhan Beberapa Fungsi

		Banyaknya Kerja Algoritma Untuk Input Berukuran n				
		$33n$	$46n \log n$	$13n^2$	$2.4n^3$	2^n
Waktu Eksekusi, Bila Ukuran $n=$	10	.00033 dtk	.0015 dtk	.0013 dtk	.0034 dtk	.001 dtk
	100	.003 dtk	.03 dtk	.13 dtk	3.4 dtk	4×10^4 abad
	1,000	.033 dtk	.45 dtk	13 dtk	.94 jam	
	10,000	.33 dtk	6.1 dtk	22 mnt	39 hari	-
	100,000	3.3 dtk	1.3 mnt	1.5 hari	108 tahun	-
Maks Ukuran Input, Dalam	1 detik	30,000	2,000	280	67	20

Waktu=						
	1 menit	1,800,000	82,000	2,200	260	26

Sekarang kita lihat dari sisi lain, dengan melihat perkembangan prosesor yang sangat pesat, kita akan meningkatkan kecepatan proses penyelesaian permasalahan melalui peningkatan kecepatan prosesor. Apakah peningkatan kecepatan prosesor ini banyak membantu ?. Untuk menjawab ini akan kita lihat melalui contoh. Misal ada dua orang Pak Untung dan Pak Harto. Keduanya ingin melakukan pengurutan data sebanyak 1.000.000 data. Pak Untung memakai komputer PC lama yang mempunyai kecepatan proses 10 MIPS (Mega Instruction Per Second), dan dalam memprogram pengurutan menggunakan algoritma *merge sort* dengan kerja $50n \log n$. Pak Harto memakai komputer main frame yang mempunyai kecepatan proses 200 MIPS (20 kali lebih cepat dari komputer yang dipakai Pak Untung), dan dalam memprogram pengurutan menggunakan algoritma *insertion sort* dengan kerja $2n^2$. Akan kita lihat waktu yang dibutuhkan dari keduanya.

Waktu untuk hasil kerja Pak Harto:

$$\frac{2(10^6)^2 \text{ instruksi}}{2(10^8) \text{ instruksi/detik}} = 10.000 \text{ detik} = 2,78 \text{ jam}$$

Waktu untuk hasil kerja Pak Untung:

$$\frac{50(10^6) \log 10^6 \text{ instruksi}}{10^6 \text{ instruksi/detik}} = 996,6 \text{ detik} = 16,6 \text{ menit}$$

Dari contoh di atas menunjukkan bahwa peningkatan kecepatan algoritma (pemilihan algoritma) jauh lebih baik dari pada meningkatkan kecepatan prosesor. Hal ini menunjukkan bahwa perangkat lunak (algoritma) merupakan suatu teknologi.

BAB II

PENCARIAN ARRAY TERURUT

Pandang permasalahan pencarian elemen x pada suatu array A dengan n elemen terurut naik atau $A[1] < A[2] < \dots < A[n]$. Kita ingin mendapatkan index dari x pada array A , bila $x = A[i]$ maka hasilnya adalah i . Namun apabila x tidak ada di dalam array A hasilnya adalah -1.

Untuk menyelesaikan permasalahan ini, jelas algoritma *SequentialSearch* yang ada di depan dapat kita pakai. Kalau kita memakai algoritma *SequentialSearch* berarti tidak ada bedanya antara array yang sudah terurut maupun belum terurut. Mestinya dengan keadaan array yang terurut ini, kita dapat membuat algoritma yang memanfaatkan keterurutan elemen array ini. Barang kali algoritmanya lebih baik dari pada *SequentialSearch*.

Dalam perbandingan dengan x , kita langsung menuju elemen tengah (katakan pada index ke k) dan kita bandingkan $A[k]$ dengan x . Jika x sama dengan $A[k]$ maka keluarkan hasil k . Namun apabila x tidak sama dengan $A[k]$, kemungkinannya adalah x lebih kecil dari $A[k]$ atau lebih besar dari $A[k]$. Jika x lebih kecil dari $A[k]$, maka pasti x tidak sama dengan $A[j]$ untuk $j > k$. Untuk itu kita cari x pada $A[k+1]$ sampai dengan $A[n-1]$. Kemungkinan lainnya, yaitu apabila x lebih besar dari $A[k]$ maka x tidak sama dengan $A[i]$ untuk $i < k$. Untuk itu kita cari x pada $A[0]$ sampai dengan $A[k-1]$. Langkah ini kita ulang terus hingga kita dapatkan hasil. Langkah - langkah ini dikenal dengan sebutan algoritma *binary Search* dan dapat kita tuliskan sebagai berikut.

Input : Array A dengan n elemen bilangan terurut naik dan bilangan x yang dicari.

Output : Bilangan bulat non negatif j yang merupakan letak / index x di dalam array A .

Menghasilkan $j = -1$ bila x tidak ada di A .

Algoritma:

BinarySearch(A, n, x) {

1. $k=0; m=n;$

2. **while** ($k \leq m$) {

3. $j = \left\lfloor \frac{k+m}{2} \right\rfloor$

4. **if** ($x = A[j]$) **return** j ;

5. **else if** ($x < A[j]$) $m=j-1$;

6. **else** $k=j+1$;

```
}  
j=-1;  
}
```

Analisis Banyak Kerja Terburuk