BAB II

DASAR TEORI

2.1 Dasar – Dasar Algoritma

Kata algoritma diambil dari nama seorang penulis buku matematika dari Persia, Al khowarismi (825 M), yang diartikan sebagai suatu metode khusus untuk memecahkan masalah. Kemudian kata ini dimasukkan ke dalam bidang komputer dan diartikan sebagai langkah - langkah yang dapat digunakan oleh komputer untuk menyelesaikan suatu masalah. Algoritma merupakan suatu prosedur yang terdiri atas sejumlah terbatas langkah – langkah yang masing – masing memerlukan satu atau lebih operasi untuk memperoleh hasil.

Ada beberapa sifat algoritma sehingga dapat dioperasikan pada komputer, yaitu :

1. Setiap langkah atau operasi harus tertentu (*definite*), yang berarti harus jelas apa yang harus dikerjakan atau tidak mengandung lebih dari satu kemungkinan yang dikerjakan. Contoh perintah yang tidak dibenarkan misalnya " tambahkan 5 atau 3 pada x ", perintah tersebut tidak diijinkan karena tidak jelas bilangan yang akan ditambahkan pada x.
2. Setiap langkah atau operasi harus efektif yang berarti sedapat mungkin dengan sedikit prinsip dan ditulis dalam waktu terbatas, sehingga setiap langkah memang dibutuhkan untuk memperoleh hasil.

3. Algoritma memiliki *input*, artinya suatu algoritma berasal dari masalah kemudian diberi satu atau lebih masukan untuk diproses.
4. Algoritma memiliki *output*, artinya suatu algoritma menghasilkan satu atau lebih keluaran/hasil.
5. Algoritma bersifat *finite*, artinya algoritma berhenti setelah sejumlah terbatas langkah.

2.2 Kompleksitas Algoritma

Kompleksitas algoritma mengacu pada jumlah waktu dan ruang yang dibutuhkan untuk mengeksekusi algoritma. Beberapa *terminologi* yang digunakan untuk mendeskripsikan kompleksitas waktu algoritma tercantum dalam tabel 1 di bawah. Setiap *running time* dengan kompleksitas yang berbeda membutuhkan waktu yang berbeda untuk menyelesaikan suatu program.

Tabel 1. *Terminologi Kompleksitas Algoritma*

Kompleksitas	Terminologi
O(1)	Kompleksitas konstanta
O(log n)	Kompleksitas logaritma
O(n)	Kompleksitas linier
O(n log n)	Kompleksitas n log n
$O(n^b)$	Kompleksitas polinomial
$O(b^n)$	Kompleksitas eksponensial
O(n!)	Kompleksitas faktorial

2.2.1 Running time

Pada saat menyelesaikan masalah sering dihadapkan pada beberapa pilihan algoritma. Terdapat dua hal yang dijadikan dasar memilih algoritma yaitu :

1. Algoritma yang mudah dimengerti serta mudah dalam pengkodean dan pengecekan kesalahan, terutama untuk program yang digunakan lebih dari sekali atau program yang rumit.
2. Algoritma yang efisien dalam penggunaan komputer atau dapat dijalankan secepat mungkin. Ukuran untuk mengetahui efisiensi program adalah *running time* program.

Definisi 2.1

Running time adalah ukuran waktu sebagai fungsi dari besar data masukan untuk menjalankan suatu program sehingga menghasilkan *output*. *Running time* dinotasikan dengan notasi T(n), dengan n adalah besar data masukan (*input*). Running time program bergantung pada faktor – faktor seperti :

1. *Input* program.

   Running time program bergantung dari besar kecilnya masukan data (*input*). Untuk masukan data yang besar membutuhkan waktu lebih lama dibanding masukan data yang lebih kecil.

2. Kualitas kompiler (kecepatan mesin) dalam menjalankan program.

   Kecepatan kompiler mempengaruhi waktu yang dibutuhkan untuk menjalankan program. Biasanya untuk kompiler terbaru akan lebih cepat dalam menjalankan program dibanding kompiler lama.

3. Kompleksitas waktu.

   Untuk program yang mempunyai kompleksitas waktu yang berbeda akan menghasilkan running time yang berbeda pula, meskipun besar masukan dan kualitas kompilernya sama.

Untuk menghitung *running time* program digunakan fungsi pertumbuhan yang merupakan dasar dari *running time*.

2.2.2 Fungsi Pertumbuhan

Misalkan suatu program komputer dengan *input* n bilangan bulat. Pertimbangan penting yang berkaitan dengan program tersebut adalah bagaimana kecepatan fungsi pertumbuhan pada n. Notasi yang digunakan untuk menganalisa fungsi pertumbuhan disebut sebagai ' *big – Oh*'.

Definisi 2.2

T(n) adalah O(f(n)), jika terdapat konstanta c dan n_0 sedemikian sehingga $T(n) \leq c.\ (f(n))$, untuk setiap $n \geq n_0$. Program yang mempunyai *running time* O(f(n)) dikatakan mempunyai fungsi pertumbuhan f(n).

Contoh 2.1

Tunjukkan bahwa $T(n) = 3\ n^3 + 2n^2$ adalah $O(n^3)$.

Penyelesaian :

Untuk menerapkan definisi pada notasi *big-Oh*, ambil $n_0 = 0$ dan $c = 5$ sehingga dapat diperoleh pertidaksamaan : $3n^3 + 2n^2 \leq 3n^3 + 2n^3 = 5n^3$

Jadi T(n) adalah $O(n^3)$.

Suatu program biasanya dibuat dalam beberapa *subprogram* (prosedur), maka untuk menghitung *running time* program secara keseluruhan digunakan aturan untuk menjumlahkan dan mengalikan dalam notasi *big-Oh*.

Theorema 2.1

Misal $T_1(n)$ dan $T_2(n)$ adalah *running time* dari *subprogram* P_1 dan P_2 dengan $T_1(n)$ adalah $O(f(n))$ dan $T_2(n)$ adalah $O(g(n))$ maka $T_1(n) + T_2(n)$ adalah $O(\max(f(n), g(n)))$.

Bukti :

Untuk menunjukkan $T_1(n) + T_2(n)$ adalah $O(\max(f(n), g(n)))$, ambil konstanta c_1, c_2, n_1 dan n_2 sedemikian sehingga

$T_1(n) \leq c_1.f(n)$, untuk $n \geq n_1$

$T_2(n) \leq c_2.g(n)$, untuk $n \geq n_2$

Misalkan $n_0 = \max(n_1, n_2)$, maka berlaku

$T_1(n) \leq c_1.f(n)$, untuk $n \geq n_0$

$T_2(n) \leq c_2.g(n)$, untuk $n \geq n_0$

Sehingga berlaku :

$T_1(n) + T_2(n) \leq c_1.f(n) + c_2.g(n)$

Misalkan $c_3 = \max(c_1, c_2)$, maka

$T_1(n) + T_2(n) \leq c_3.f(n) + c_3.g(n)$

$\leq c_3(f(n) + g(n))$

$\leq c_3 (\max (f(n), g(n)) + \max (f(n), g(n)))$

$\leq 2c_3 (\max (f(n), g(n)))$

untuk $c = 2c_3$ dan $n \geq n_0$.

Jadi running time dari $T_1(n) + T_2(n)$ adalah $O (\max (f(n), g(n)))$.

Teorema 2.1 di atas disebut aturan penjumlahan (*rule of sum*).

Contoh 2.2

Aturan penjumlahan di atas dapat digunakan untuk menghitung *running time* dari barisan langkah program, dimana setiap langkah kemungkinan adalah bagian program dengan *loop* atau cabang. Misal terdapat tiga langkah yang mempunyai *running time* $O(n^2)$, $O(n^3)$ dan (n log n). Maka *running time* dari 2 langkah pertama tersebut adalah $O(\max (n^2, n^3))$ yaitu $O(n^3)$ dan *running time* dari ketiga langkah tersebut adalah $O(\max (n^3, n \log n))$ yaitu $O(n^3)$.

Secara umum, *running time* dari barisan langkah tersebut adalah running time dari langkah yang mempunyai *running time* terbesar.

Theorema 2.2

Misal $T_1(n)$ dan $T_2(n)$ adalah *running time* dari *subprogram* P_1 dan P_2 dengan $T_1(n)$ adalah $O(f(n))$ dan $T_2(n)$ adalah $O(g(n))$ maka $T_1(n).T_2(n)$ adalah $O(f(n).g(n))$.

Bukti :

Untuk menunjukkan $T_1(n).T_2(n)$ adalah $O(f(n).g(n))$, ambil konstanta c_1, c_2, n_1 dan n_2 sedemikian sehingga

$T_1(n) \leq c_1.f(n)$, untuk $n \geq n_1$

$T_2(n) \leq c_2.g(n)$, untuk $n \geq n_2$

Misalkan $n_0 = \max(n_1, n_2)$, maka berlaku

$T_1(n) \leq c_1.f(n)$, untuk $n \geq n_0$

$T_2(n) \leq c_2.g(n)$, untuk $n \geq n_0$

Sehingga berlaku :

$$T_1(n).T_2(n) \leq c_1.f(n).c_2.g(n)$$
$$\leq c_1.c_2\,(f(n).g(n))$$
$$\leq c\,(f(n).g(n))$$

untuk $c = c_1.c_2$ dan $n \geq n_0$.

Jadi running time dari $T_1(n).T_2(n)$ adalah $O(f(n).g(n))$.

Teorema 2.2 di atas disebut aturan perkalian (*rule of product*).

Secara umum, untuk menghitung *running time* dari *block statement* berdasarkan ketentuan sebagai berikut :

1. *Running time* setiap penugasan, *statement* membaca data dan mencetak mempunyai O(1).
2. *Running time* barisan *statement* ditentukan dengan aturan penjumlahan.
3. *Running time statement if* adalah waktu menjalankan *statement* ditambah waktu test kondisi. Waktu test kondisi adalah O(1). Waktu yang diperlukan *if – then – else* adalah waktu terbesar diantara dua kondisi bernilai benar atau kondisi bernilai salah. Waktu yang diperlukan untuk kondisi bernilai benar adalah waktu tes kondisi ditambah waktu yang diperlukan untuk menjalankan *statement* sesudah *then*, sedangkan waktu yang diperlukan untuk kondisi bernilai salah adalah waktu tes kondisi ditambah waktu untuk menjalankan *statement* sesudah *else*.
4. Waktu untuk menjalankan *loop* adalah waktu untuk menjalankan isi dikali waktu untuk tes kondisi.

2.3 Pengurutan

Pengurutan (*Sorting*) diartikan sebagai suatu proses penyusunan sekumpulan objek ke dalam suatu urutan tertentu. Tujuan pengurutan adalah untuk mendapatkan kemudahan dalam pencarian anggota dari suatu himpunan. Proses pengurutan ini merupakan aktivitas yang penting, khususnya dalam pengolahan data. Dimisalkan sekumpulan data : $a_1, a_2, a_3, \ldots, a_n$ maka pengurutan akan menukar posisi setiap data ke dalam suatu urutan $a_{k1}, a_{k2}, a_{k3}, \ldots, a_{kn}$

sedemikian rupa sehingga terdapat fungsi pengurutan $f(a_{k1}) < f(a_{k2}) < \ldots\ldots < f(a_{kn})$

Metode pengurutan dapat dimasukkan ke dalam dua kategori yaitu pengurutan larik (*array*) dan pengurutan berkas (*sequential*). Pengurutan larik adalah pengurutan dimana larik diurutkan ke dalam suatu memori utama berkecepatan tinggi yaitu *Random Access Memory*. Sedangkan pengurutan berkas adalah pengurutan dimana berkas diletakkan dalam suatu media penyimpan luar.

2.3.1 Bubble Sort

Algoritma *Bubble Sort* (gelembung) sering juga disebut dengan *Exchange Sort* (penukaran). *Bubble Sort* adalah pengurutan yang mendasarkan penukaran dua buah elemen untuk mencapai keadaan urut yang diinginkan. Untuk mengurutkan data bisa dilaksanakan dengan dua cara. Cara pertama adalah selalu meletakkan data dengan nilai paling besar pada posisi terakhir (posisi ke n). Kemudian data dengan nilai paling besar kedua diletakkan pada posisi n – 1, hal ini diulangi sampai semua data terurutkan semuanya. Cara kedua adalah kebalikan cara pertama. Dalam hal ini yang digunakan sebagai patokan adalah data dengan nilai terkecil. Dengan kata lain, pada iterasi pertama akan diletakkan data dengan nilai terkecil pada posisi ke-1. Kemudian data dengan nilai terkecil kedua diletakkan pada posisi ke-2 dan seterusnya, hal ini diulangi sampai semua data dalam keadaan urut.

Algoritma *Bubble Sort* dapat dijelaskan secara sederhana sebagai berikut :

Langkah 0 : Baca data yang akan diurutkan

Langkah 1 : Kerjakan langkah 2 untuk I = 1 sampai n – 1

Langkah 2 : Kerjakan langkah 3 untuk j = 1 sampai n – 1

Langkah 3 : Dilakukan tes kondisi apakah A[j] > A [j + 1]

Jika benar, tukarkan kedua nilai data tersebut.

Langkah 4 : Selesai.

Tabel 2 . *Iterasi Bubble Sort*

Iterasi	A[1]	A[2]	A[3]	A[4]	A[5]
Awal	24	23	56	45	12
1	23	24	56	45	12
	23	24	56	45	12
	23	24	45	56	12
	23	24	45	12	56
2	23	24	45	12	56
	23	24	45	12	56
	23	24	12	45	56
3	23	24	12	45	56
	23	12	24	45	56
4	12	23	24	45	56
Akhir	12	23	24	45	56

Keterangan : A[n] menunjukkan data ke-n

Tabel 2 menunjukkan pelacakan *Bubble Sort* untuk mengurutkan 5 data. Dari tabel di atas dapat dilihat bahwa untuk n data diperlukan iterasi n – 1 kali. Disamping itu bila diperhatikan maka nomor iterasi jika ditambah dengan

banyaknya langkah pada iterasi tersebut besarnya selalu tetap yaitu sama dengan n atau banyak data yang akan diurutkan. *Bubble Sort* sebenarnya merupakan algoritma yang tidak efisien karena bila akan diurutkan n buah data dan terdapat sebagian data yang sudah terurut sehingga iterasinya kurang dari n – 1, algoritma ini harus tetap melakukan iterasi tersebut sampai n – 1 kali.