BAB III
ANALISIS KOMPLEKSITAS ALGORITMA

3.1 Kompleksitas Algoritma

Suatu masalah dapat mempunyai banyak algoritma penyelesaian. Algoritma yang digunakan tidak saja harus benar, namun juga harus efisien. Efisiensi suatu algoritma dapat diukur dari waktu eksekusi algoritma dan kebutuhan ruang memori. Algoritma yang efisien adalah algoritma yang meminimumkan kebutuhan waktu dan ruang. Dengan menganalisis beberapa algoritma untuk suatu masalah, dapat diidentifikasi satu algoritma yang paling efisien. Besaran yang digunakan untuk menjelaskan model pengukuran waktu dan ruang ini adalah kompleksitas algoritma.

Kompleksitas dari suatu algoritma merupakan ukuran seberapa banyak komputasi yang dibutuhkan algoritma tersebut untuk menyelesaikan masalah. Secara informal, algoritma yang dapat menyelesaikan suatu permasalahan dalam waktu yang singkat memiliki kompleksitas yang rendah, sementara algoritma yang membutuhkan waktu lama untuk menyelesaikan masalahnya mempunyai kompleksitas yang tinggi. Kompleksitas algoritma terdiri dari dua macam yaitu kompleksitas waktu dan kompleksitas ruang.

Kompleksitas waktu, dinyatakan oleh $T(n)$, diukur dari jumlah tahapan komputasi yang dibutuhkan untuk menjalankan algoritma sebagai fungsi dari ukuran masukan n, di mana ukuran masukan (n) merupakan jumlah data yang diproses oleh sebuat algoritma. Sedangkan kompleksitas ruang, $S(n)$, diukur dari memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari masukan n. Dengan menggunakan kompleksitas waktu atau kompleksitas ruang, dapat ditentukan laju peningkatan waktu atau ruang yang diperlukan algoritma, seiring dengan meningkatnya ukuran masukan (n).

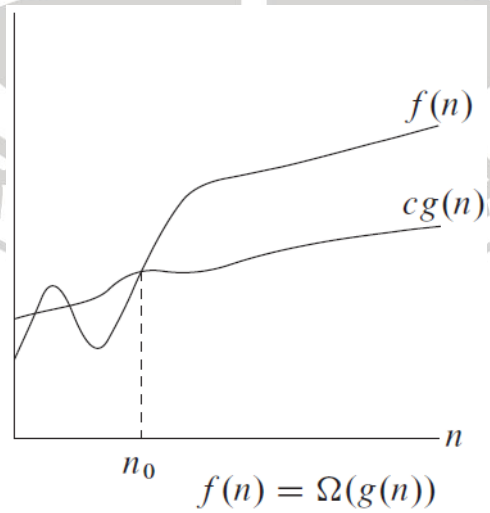
Kecenderungan saat ini, ruang (memori utama) yang disediakan semakin besar yang artinya kapasitas data yang diproses juga semakin besar. Namun,

waktu yang diperlukan untuk menjalankan suatu algoritma harus semakin cepat. Karena kompleksitas waktu menjadi hal yang sangat penting, maka analisis kompleksitas algoritma deteksi tepi akan dilakukan terhadap *running time* algoritma tersebut.

3.2 Notasi Asimptotik

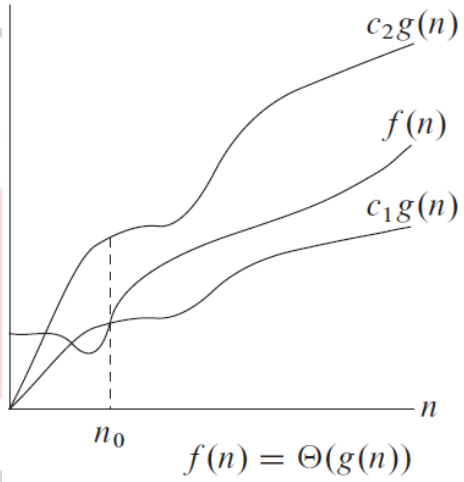
Untuk nilai n cukup besar, bahkan tidak terbatas, dilakukan analisis efisiensi asimptotik dari suatu algoritma untuk menentukan kompleksitas waktu yang sesuai atau disebut juga kompleksitas waktu asimptotik. Notasi yang digunakan untuk menentukan kompleksitas waktu asimptotik dengan melihat waktu tempuh (*running time*) algoritma adalah notasi asimptotik (*asimptotic notation*). Notasi asimptotik didefinisikan sebagai fungsi dengan domain himpunan bilangan asli $\mathbb{N} = \{0, 1, 2, \dots\}$ (Cormen *et al*., 2009: 43).

Kompleksitas waktu asimptotik terdiri dari tiga macam. Pertama, keadaan terbaik (*best case*) dinotasikan dengan $\Omega(g(n))$ (*Big-Omega*), keadaan rata-rata (*average case*) dilambangkan dengan notasi $\Theta(g(n))$ (*Big-Theta*) dan keadaan terburuk (*worst case*) dilambangkan dengan $O(g(n))$ (*Big-O*).



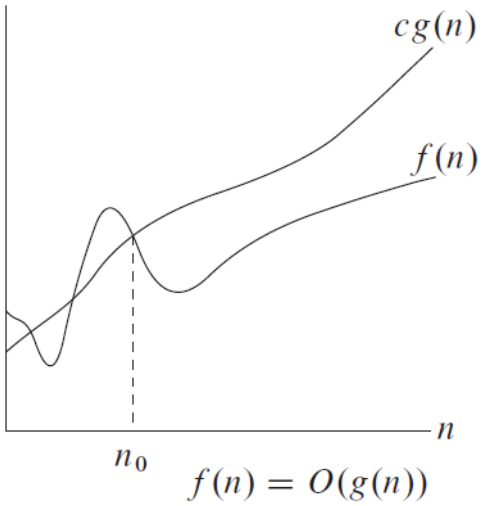
Gambar 3.1 Contoh Grafik dari Notasi Asimptotik Ω

Gambar 3.1 menunjukkan notasi Ω menjadi batas bawah dari suatu fungsi $f(n)$ agar berada dalam suatu faktor konstan. Dinyatakan $f(n) = O(g(n))$ jika terdapat konstanta positif n_0 dan c sedemikian sehingga pada n_0 dan di kanan n_0, nilai $f(n)$ selalu berada tepat pada $cg(n)$ atau di atas $cg(n)$.



Gambar 3.2 Contoh Grafik dari Notasi Asimptotik Θ

Pada gambar 3.2, n_0 merupakan nilai n minimum yang mungkin. Gambar 3.1 menunjukkan notasi Θ membatasi suatu fungsi $f(n)$ agar berada dalam faktor konstan. Dinyatakan $f(n) = \Theta(g(n))$ jika terdapat konstanta positif n_0, c_1, dan c_2 sedemikian sehingga pada n_0 dan di kanan n_0, nilai $f(n)$ selalu berada tepat pada $c_1g(n)$, tepat pada $c_2g(n)$, atau di antara $c_1g(n)$ dan $c_2g(n)$.



Gambar 3.3 Contoh Grafik dari Notasi Asimptotik O

Gambar 3.3 menunjukkan notasi O menjadi batas atas dari suatu fungsi $f(n)$ agar berada dalam suatu faktor konstan. Dinyatakan $f(n) = O(g(n))$ jika terdapat konstanta positif n_0 dan c sedemikian sehingga pada n_0 dan di kanan n_0, nilai $f(n)$ selalu berada tepat pada $cg(n)$ atau di bawah $cg(n)$. Kompleksitas waktu algoritma biasanya dihitung dengan menggunakan notasi $O(g(n))$, dibaca "*big-O* dari $g(n)$".

3.2.1 Notasi *O* (*Big-O)*

Notasi asimptotik O digunakan ketika hanya diketahui batas atas asimptotik. $O(g(n))$ didefinisikan:

$O(g(n)) = \{f(n)$: terdapat konstanta positif c dan n_0 sehingga $0 \leq f(n) \leq cg(n)$ untuk setiap $n \geq n_0\}$ (Cormen *et al*., 2009: 47).

Pada gambar 3.3 ditunjukkan bahwa untuk semua nilai n pada tepat dan di sebelah kanan n_0, nilai fungsi $f(n)$ berada tepat atau di bawah $cg(n)$. $f(n) = O(g(n))$ mengindikasikan bahwa $f(n)$ adalah anggota himpunan $O(g(n))$.

Notasi O menyatakan *running time* dari duatu algoritma untuk kemungkinan kasus terburuk. Notasi O memiliki dari beberapa bentuk. Notasi O dapat berupa salah satu bentuk maupun kombinasi dari bentuk-bentuk tersebut.

Bentuk $O(1)$ memiliki arti bahwa algoritma yang sedang dianalisis merupakan algoritma konstan. Hal ini mengindikasikan bahwa *running time* algoritma tersebut tetap, tidak bergantung pada n.

$O(n)$ berarti bahwa algoritma tersebut merupakan algoritma linier. Artinya, bila n menjadi $2n$ maka *running time* algoritma akan menjadi dua kali *running time* semula.

$O(n^2)$ berarti bahwa algoritma tersebut merupakan algoritma kuadratik. Algoritma kuadratik biasanya hanya digunakan untuk kasus dengan n yang berukuran kecil. Sebab, bila n dinaikkan menjadi dua kali semula, maka *running time* algoritma akan menjadi empat kali semula.

$O(n^3)$ berarti bahwa algoritma tersebut merupakan algoritma kubik. pada algoritma kubik, bila n dinaikkan menjadi dua kali semula, maka *running time* algoritma akan menjadi delapan kali semula.

Bentuk $O(2^n)$ berarti bahwa algoritma tersebut merupakan algoritma eksponensial. Pada kasus ini, bila n dinaikkan menjadi dua kali semula, maka *running time* algoritma akan menjadi kuadrat kali semula.

$O(\log n)$ berarti algoritma tersebut merupakan algoritma logaritmik. Pada kasus ini, laju pertumbuhan waktu lebih lambat dari pada pertumbuhan n. Algoritma yang termasuk algoritma logaritmik adalah algoritma yang memecahkan persoalan besar dengan mentransformasikannya menjadi beberapa persoalan yang lebih kecil dengan ukuran sama. Basis algoritma tidak terlalu penting, sebab bila misalkan n dinaikkan menjadi dua kali semula, $\log n$ meningkat sebesar jumlah tetapan.

Bentuk $O(n \log n)$, terdapat pada algoritma yang membagi persoalan menjadi beberapa persoalan yang lebih kecil, menyelesaikan setiap persoalan secara independen, kemudian menggabungkan solusi masing-masing persoalan.

Sedangkan $O(n!)$ berarti bahwa algoritma tersebut merupakan algoritma faktorial. Algoritma jenis ini akan memproses setiap masukan dan menghubungkannya dengan $n-1$ masukan lainnya. Bila n menjadi dua kali semula, maka *running time* algoritma akan menjadi faktorial dari $2n$.

3.3 Kompleksitas Waktu Algoritma

Untuk menentukan kompleksitas waktu suatu algoritma, diperlukan ukuran masukan n serta *running time* algoritma tersebut. Pada umumnya, *running time* algoritma meningkat seiring dengan bertambahnya ukuran n. Sehingga, *running time* suatu algoritma dapat dinyatakan sebagai fungsi dari n.

Ukuran masukan n untuk suatu algoritma bergantung pada masalah yang diselesaikan oleh algoritma tersebut. Pada banyak kasus, seperti pengurutan, ukuran yang paling alami adalah jumlah item dalam masukan. Dalam kasus lain, seperti mengalikan dua bilangan bulat, ukuran input terbaik adalah jumlah bit yang diperlukan untuk mewakili masukan dalam notasi biner biasa.

Running time algoritma pada masukan n tertentu merupakan jumlah operasi atau langkah yang dieksekusi. Selanjutnya, jumlah waktu yang konstan diperlukan untuk mengeksekusi setiap baris *pseudocode* (kode semu). Satu baris dapat memiliki jumlah waktu yang berbeda dari baris lain. Namun asumsikan bahwa setiap pelaksanaan baris ke-i membutuhkan waktu sebesar c_i, di mana c_i adalah konstanta.

Dalam menentukan *running time* suatu baris pada *pseudocode* (kode semu), kalikan konstanta c_i dengan jumlah waktu yang diperlukan untuk mengeksekusi baris tersebut. Untuk kasus di mana terdapat perintah *loop while* atau *for* dengan panjang n, maka perintah tersebut dieksekusi dengan waktu $n+1$. Sedangkan untuk baris berisi komentar, dinyatakan sebagai baris yang tidak dieksekusi, sehingga jumlah waktu untuk baris tersebut adalah nol.

Selanjutnya, *running time* dari algoritma adalah jumlah dari *running time* setiap perintah yang dieksekusi. Sebuah perintah yang membutuhkan c_i langkah n

waktu untuk dieksekusi akan memiliki pengaruh sebesar $c_i n$ pada *running time* total $(T(n))$.

Setelah diperoleh bentuk fungsi $T(n)$, dapat ditentukan bentuk dari algoritma tersebut dengan menggunkan notasi asimptotik *O*. Dengan ditentukannya bentuk algoritma, maka dapat diramalkan berapa besar peningkatan *running time* jika ukuran masukan *n* ditingkatkan.

Contohnya, untuk suatu prosedur algoritma pengurutan *A* berikut, dimulai dengan menghitung nilai waktu yang digunakan oleh suatu perintah dan jumlah pengulangan perintah tersebut dieksekusi. Untuk setiap $j = 2, 3, \ldots, n$, di mana n adalah panjang dari *A* (*A.length*). t_j merupakan notasi dari jumlah banyaknya *loop while* yang dieksekusi untuk nilai j pada baris 5.

	PENGURUTAN(*A*)	nilai	waktu
1.	**for** $j = 2$ **to** *A.length*	c_1	n
2.	*key* = *A*[*j*]	c_2	$n-1$
3.	// Insert *A*[*j*] into the sorted sequence *A*[1 .. *j* – 1].	0	$n-1$
4.	*i* = *j* – 1	c_4	$n-1$
5.	***while*** *i* > 0 and *A*[*i*] > *key*	c_5	$\sum_{j=2}^{n} t_j$
6.	*A*[*i* + 1] = *A*[*i*]	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7.	*i* = *i* – *1*	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8.	*A*[*i*+1] = *key*	c_8	$n-1$

Untuk menghitung $T(n)$, r*unning time* dari algoritma pengurutan dengan nilai masukan n, jumlahkan hasil kali nilai dengan waktu. Diperoleh,

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Kasus terbaik untuk algoritma ini adalah jika *array* sudah berurutan. Untuk setiap $j = 2, 3, \dots, n$, diperoleh $A[i] \leq key$ pada baris ke 5 ketika i menjadi nilai awal dari $j - 1$. Maka $t_j = 1$ untuk $j = 2, 3, \dots, n$, dan *running time* untuk kasus pengurutan terbaik adalah

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Running time ini dapat dinyatakan sebagai $an + b$ untuk a dan b konstanta yang bergantung pada nilai c_i, artinya *running time* ini merupakan fungsi linier dari n, atau dinyatakan sebagai.

Jika *array* berada pada kondisi susuanan yang terbalik, maka algoritma tersebut melakukan pengurutan untuk kasus terburuk. Setiap elemen $A[j]$ harus dibandingkan dengan semua elemen lain yang sudah tersusun dalam $a[1..j-1]$, dan $t_j = j$ untuk setiap $j = 2, 3, \dots, n$. Perhatikan bahwa

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

dan

$$\sum_{j=2}^{n} (j-1) = \frac{n(n+1)}{2}$$

Pada kasus terburuk diperoleh *running time* untuk algoritma pengurutan adalah

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right)$$
$$+ c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$
$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4$$
$$+ c_5 + c_8)$$

Running time untuk kasus terburuk ini dapat dinyatakan sebagai $an^2 + bn + c$ untuk a, b, dan c konstanta yang bergantung pada nilai c_i. Sehingga, *running time* tersebut merupakan fungsi kuadratik dari n.

Kompleksitas waktu yang dinyatakan dalam notasi asimptotik O menyatakan kemungkinan waktu terburuk yang dapat dicapai. Maka, kompleksitas waktu untuk algoritma pengurutan ini berbentuk $O(n^2)$ atau kuadratik.