EFISIENSI ALGORITMA DAN NOTASI O-BESAR

Subandijo

Computer Science Department, School of Computer Science Binus University
Jl. K.H. Syahdan No. 9, Palmerah, Jakarta Barat 11480
subandijo1030@gmail.com

ABSTRACT

Efficiency or the running time of an algorithm is usually calculated with time complexity or space complexity as a function of various inputs. It is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Brute-force algorithm is the easiest way to calculate the performance of the algorithm. However, it is not recommended since it does not sufficiently explain the efficiency of the algorithm. Asymptotic estimaties are used because different implementations of the same algorithm may differ in efficiency. The big-O notation is used to generate the estimation.

Keywords: *algorithm complexity, algorithm efficiency, Brute-force algorithm, asymptotic estimation, big-O notation*

ABSTRAK

Efisiensi atau waktu eksekusi suatu algoritma biasanya diukur menggunakan kompleksitas waktu dan kompleksitas memori sebagai fungsi dari banyak masukan. Adalah hal yang umum untuk mengestimasi kompleksitas algoritma menggunakan pendekatan asimptotik dalam arti mengestimasi fungsi kompleksitas untuk data yang besar. Algoritma brute-force merupakan cara paling mudah untuk menghitung kinerja algoritma tetapi tidak dianjurkan karena tidak cukup menerangkan efisiensi algoritma. Estimasi asimptotik digunakan karena implementasi berbeda untuk algoritma yang sama dapat menghasilkan efisiensi yang berbeda. Notasi O-besar digunakan dalam penelitian ini untuk menyajikan estimasi.

Kata kunci: *kompleksitas algoritma, efsiensi algoritma, algoritma Brute-force, estimasi asymptotic, notasi O-Besar*

PENDAHULUAN

Seringkali pertanyaan yang muncul pada pemrograman komputer bukan bagaimana menyelesaikan suatu masalah, tetapi bagaimana menyelesaikan masalah dengan efisien. Sebagai contoh yang kerap dikutip adalah masalah pengurutan data. Banyak metode pengurutan data yang dikenal. Masalahnya bukan bagaimana menemukan metode untuk mengurutkan data tetapi bagaimana menemukan metode yang paling efisien dilihat dari sisi waktu komputasi (*time complexity*) dan besar memori yang digunakan (*space complexity*).

Dua algoritma pengurutan data, *merge sort* dan *quick sort* adalah dua metode yang sebanding efisiensinya yaitu O (*n* log *n*) tetapi banyak orang yang lebih suka menggunakan *quick sort* yang sangat populer karena tidak ada memori tambahan yang dibutuhkan meskipun *quick sort* bisa sangat tidak efisien dengan kinerja turun drastis menjadi O (n^2) jika data yang akan diurutkan telah urut (Sedgewick, 1998).

Biasanya *space complexity* dianggap tetap sehingga artikel ini hanya membahas pemahaman tentang *time complexity* dengan cara bagaimana mengestimasi waktu efisiensi relatif suatu algoritma dan mengapa hal ini perlu dilakukan. Estimasi disajikan menggunakan konsep notasi O-Besar untuk mendefinisikan efisiensi suatu algoritma.

Landasan Teori

Kompleksitas Algoritma

Secara sederhana algoritma didefinisikan sebagai prosedur selangkah demi selangkah untuk mencari solusi suatu masalah dalam waktu yang terbatas. Umumnya algoritma mentransformasi obyek masukan menjadi obyek keluaran dengan waktu eksekusi yang merupakan fungsi dari obyek masukkan. Dalam arti makin besar obyek masukkan makin lama waktu yang dibutuhkan untuk menghasilkan obyek keluaran.

Kompleksitas algoritma diukur berdasarkan kinerjanya dengan menghitung waktu eksekusi suatu algoritma. Menurut Goldreich (2008) waktu eksekusi algoritma dapat diklasifikasikan menjadi tiga kelompok besar, yaitu *best-case* (kasus terbaik), *average-case* (kasus rerata) dan *worst-case* (kasus terjelek). Pada pemrograman yang dimaksud dengan kasus terbaik, kasus terjelek dan kasus rerata suatu algoritma adalah besar kecilnya atau banyak sedikitnya sumber-sumber yang digunakan oleh suatu algoritma. Makin sedikit makin baik; makin banyak makin jelek. Biasanya sumber-sumber yang paling dipertimbangkan tak hanya waktu eksekusi tetapi bisa juga besar memori, catu-daya dan sumber-sumber lain.

Istilah kinerja terbaik digunakan untuk menerangkan keadaan optimal suatu algoritma. Sebagai contoh, kinerja terbaik untuk pencarian linear sederhana terjadi saat elemen yang dicari berada di posisi pertama dalam daftar. Kenyataannya dalam komputasi *real-time*, kinerja paling baik jarang digunakan. Manfaat paling utama adalah jika kasus terbaik untuk tugas individual telah diketahui, dapat digunakan untuk memperbaiki akurasi analisis kasus terjelek secara keseluruhan. Selain itu, untuk banyak algoritma waktu terbaik lebih mudah diketahui dengan pasti sehingga dapat dijadikan sebagai patokan atau *bench mark* bagi algoritma lain. Dengan demikian, pengembangan algoritma jarang didasarkan pada kinerja terbaiknya. Kebanyakan akademisi dan praktisi komersial lebih tertarik untuk mengembangkan kinerja rerata dan kinerja terjelek suatu algoritma.

Analisis kinerja waktu rerata dan terjelek mempunyai sejumlah kemiripan tetapi pada prakteknya mereka membutuhkan alat dan pendekatan berbeda. Menentukan apa yang disebut

masukan rerata bukanlah pekerjaan yang mudah dan kerapkali ia memiliki sifat-sifat yang membuatnya sukar dikarakterisir secara matematika. Kasus ini berlaku untuk algoritma yang dirancang beroperasi pada teks *string* karakter.

Karena waktu eksekusi rerata relatif lebih sukar dihitung secara pasti, waktu eksekusi terjelek adalah yang menjadi perhatian utama. Hal ini karena sangat penting untuk mengetahui berapa banyak waktu yang dibutuhkan dalam keadaan terjelek untuk menjamin bahwa suatu algoritma akan selalu berakhir secara tepat waktu. Alasan lain mengapa memilih pokok bahasan dikonsentrasikan pada keadaan terjelek suatu algoritma adalah lebih mudah dianalisis dan merupakan hal yang sangat krusial pada sejumlah aplikasi seperti *gam*e, *robotic* dan *finance*.

Analisis keadaan terjelek mempunyai masalah yang sama dengan pendahulunya: sukar sekali menemukan, bahkan dapat dikatakan tidak mungkin untuk menemukan skenario eksak kasus terjelek. Pilihan paling gampang adalah skenario tersebut paling sedikit sama jeleknya dengan kasus terjelek. Sebagai contoh, saat melakukan analisis algoritma, sangat mungkin untuk menemukan jalur terpanjang yang mungkin terjadi dengan memaksimalkan banyak perulangan meskipun tidak mungkin menemukan masukan yang tepat yang bisa me-*generate* jalur ini karena memang masukan seperti itu tidak pernah ada. Hal ini memunculkan ide *safe analysis*, karena kasus terjelek tidak pernah *underestimate* bagi mereka yang pesimistik karena mungkin tidak ada masukkan yang membutuhkan jalur ini.

Alternatifnya adalah skenario yang dipandang dekat, tetapi tidak perlu lebih jelek dengan kasus terjelek, sebenarnya perlu dipertimbangkan. Hal ini bisa mengarah ke hasil optimistik dalam arti analisis sebenarnya *underestimate* kasus terjelek yang sebenarnya. Dalam sejumlah keadaan mungkin perlu menggunakan analisis pesimistis untuk menjamin keamanan. Tetapi kerap kali analisis pesismistik terlalu pesimistik sehingga analisis yang menemukan lebih dekat ke nilai riil tetapi mungkin optimistik lebih banyak ke pendekatan praktis, tidak teoritis lagi.

Saat melalukan analisis algoritma yang kerap menemukan waktu pendek untuk penyelesaian – tetapi secara periodik membutuhkan lebih banyak waktu – analisis *amortized* dapat digunakan untuk menemukan waktu eksekusi terjelek melalui sejumlah operasi lanjutan. Analisis kasus *amortized* terjelek bisa lebih dekat ke kasus rerata tetapi tetap menjamin batas atas waktu eksekusi.

Algoritma Brute-Force

Cara yang paling mudah untuk menghitung kinerja algoritma adalah menggunakan algoritma *brute-force*. Metode ini banyak disukai oleh pemula karena sederhana tetapi tidak dianjurkan untuk diimplementasikan karena sangat tidak efisien sehingga dipandang tidak cukup untuk menerangkan efisiensi algoritma. Metode *brute-force* – dikenal juga dengan nama metode *exhaustive* atau metode *generate and test* – adalah metode yang sangat umum untuk mencari solusi suatu masalah yang berbentuk enumerasi secara sistematis semua kandidat yang mungkin dan uji apakah setiap kandidat memenuhi kondisi yang diinginkan. Contoh klasik adalah mencari pembagi bilangan natural *n* adalah mencek satu persatu apakah *n* habis dibagi oleh bilangan 1 sampai n. Contoh lain yang popular adalah *eight queen puzzle*. Pendekatan *brute-force* akan mengeksaminasi semua susunan yang mungkin dari 8 buah *queen* dalam 64 kotak papan catur dan di setiap susunan, yaitu apakah sembarang *queen* meng-*attack queen* yang lain atau tidak.

Metode ini sederhana untuk diimplementasikan dan akan selalu menemukan solusi jika ada. Akan tetapi metode ini sangat *costly* karena secara proposional dengan banyak kandidat solusi, di mana dalam banyak hal praktis cenderung tumbuh sangat cepat jika nilai *n* meningkat. Karena itu metode ini hanya digunakan jika masalah terbatas ukurannya atau ada metode yang dapat digunakan untuk mengurangi kandidat solusi. Selain itu, metode ini juga digunakan jika implementasi yang

sederhana dipandang lebih penting dari pada kecepatan eksekusi. Kasus ini umumnya dijumpai pada komputasi numerik di mana eror di algoritma mempunyai pengaruh yang signifikan atau kasus penggunaan komputer untuk membuktikan teori matematika seperti *4 color theorem.* Metode *brute-force* juga bermanfaat sebagai metode basis untuk *benchmarking* algoritma lain. Metode *brute force* juga tidak perlu dipertentangkan dengan metode *backtracking* di mana sebagian besar kandidat solusi dapat diabaikan tanpa melakukan enumerasi secara eksplisit. Metode *brute force* untuk mencari item dalam tabel dikenal dengan nama pencarian linear.

Untuk memahami lebih lanjut metode *brute-force*, empat prosedur perlu diimplementasikan yaitu *first*, *next*, *valid* dan *output* yang masing-masing membutuhkan parameter P sebagai obyek tertentu suatu masalah:

first(P): generate kandidat solusi pertama untuk P
next(P,c): generate kandidat solusi berikutnya untuk P sesudah kandidat c
valid(P,c): uji apakah kandidat c adalah solusi untuk P
output(P,c): gunakan solusi c dari P sebagai aplikasi

Prosedur *next()* juga harus mengembalikan nilai jika tidak ada kandidat lain untuk P sesudah kandidat c. Cara yang lazim digunakan adalah dengan mengembalikan nilai "NULL". Hal yang sama juga berlaku untuk *first()* jika tidak ada satupun kandidat untuk P. Secara singkat metode *brute-force* adalah sebagai berikut:

```
c <- first(P)
while c <> NULL do if valid(P,c) then output(P,c)  c <- next(P,c)
```

Sebagai contoh, untuk mencari pembagi dari integer *n*, instant *P* adalah bilangan *n*. Prosedur *first(n)* harus mengembalikan integer 1 jika $n >= 1$ dan NULL jika tidak. Pemanggilan *next(n,c)* mengembalikan $c+1$ jika $c < n$ dan NULL jika tidak. Prosedur *valid(n.c)* mengembalikan *True* jika dan hanya jika *c* adalah pembagi dari *n* . Prosedur *brute-force* akan memanggil *output()* untuk setiap kandidat yang merupakan solusi dari *P*. Algoritma mudah dimodifikasi sesudah menemukan solusi pertama, atau sejumlah solusi tertentu atau sesudah menghabiskan sejumlah waktu CPU.

Jika masih mungkin menggunakan metode *brute-force* untuk menyelesaikan suatu masalah, seperti mencoba semua kombinasi yang mungkin, mengapa masih perlu untuk mencari metode yang lebih baik? Jawaban yang paling sederhana adalah jika tersedia komputer yang cukup cepat, hal ini tidak perlu dilakukan. Faktanya adalah sebaliknya. Sampai saat ini tidak ada komputer yang cukup cepat untuk mencari semua solusi yang mungkin untuk mengurutkan 100 kata berbeda karena kasus ini memerlukan 100! (100 faktorial), urutan kata.

Secara teoritis proses ini membutuhkan 100 slot yang harus disediakan dan masing-masing slot harus diisi dengan satu kata. Untuk slot pertama tersedia 100 kata yang siap diisikan. Untuk slot kedua tersedia 99 kata sedangkan untuk slot ketiga tersedia 98 kata. Jadi untuk slot pertama ada 100 opsi dan untuk 99 slot sisanya, untuk setiap 100 opsi, 99! Opsi karena untuk setiap 99 opsi di slot kedua, ada 98! opsi untuk slot ketiga. Proses berlangsung terus sampai dicapai slot terakhir yang harus diisi dengan huruf terakhir. 10^{149}

Seratus faktorial bukanlah bilangan yang kecil. Ia adalah suatu bilangan dengan 158 digit. Katakanlah ada komputer yang mampu menghitung sampai 1,000,000,000 kemungkinan per detik, masih tersisa $1x10^{149}$ detik yang harus ditunggu, lebih lama daripada umur bumi yang diperkirakan. Dengan demikian jelaslah bahwa mempunyai algoritma yang efisien untuk mengurutkan 100 kata selama masih ada kehidupan sangat diharapkan.

METODE

Notasi O-Besar adalah cara yang digunakan untuk menguraikan laju pertumbuhan suatu fungsi yang tidak lain adalah *time complexity* suatu algoritma. Notasi O-Besar pertama kali dikenalkan oleh pakar teori bilangan Paul Bachman pada tahun 1894 dalam bukunya *Analytische Zahlentheorie* ("*analytic number theory*") volume dua untuk menyatakan *asymptotic upper bounds* suatu fungsi. Volume pertamanya diterbitkan tahun 1892 tetapi belum memuat notasi O. Notasi ini dipopulerkan *pada* teori bilangan oleh Edmund Landau yang kemudian kerap disebut simbol Landau. pada ilmu komputer notasi O-Besar diklaim sudah dikenal sejak tahun 1927 yang kemudian dipopulerkan kembali penggunaannya oleh Knuth (1976, 1998, 1999) yang juga mengenalkan kembali notasi Omega dan Theta. Notasi O-Besar, dibaca sebagai "*order of*", aslinya menggunakan simbol *omikron* besar. Saat ini digunakan huruf besar Latin O yang kelihatannya mirip dengan *omikron.*

Dalam matematika dan ilmu komputer, notasi O-Besar juga dikenal dengan notasi Omikron besar, notasi Landau, notasi Bachman-Landau dan notasi asimpotik. Bersama dengan notasi yang berkaitan seperti notasi Omega besar, notasi Theta besar dan notasi o kecil, notasi O-Besar digunakan untuk menguraikan perilaku batas suatu fungsi jika argumen fungsi bergerak menuju nilai tak terbatas. Notasi O-Besar mengklasifikasikan fungsi berdasarkan kecepatan pertumbuhan argumennya, yaitu fungsi berbeda dengan kecepatan pertumbuhan yang sama akan disajikan dengan notasi O yang sama. Uraian fungsi yang berkaitan dengan notasi O-Besar biasanya hanya menyajikan batas atas kecepatan pertumbuhan fungsi. (Higham, 1998).

HASIL DAN PEMBAHASAN

Notasi O-Besar

Notasi O-Besar adalah cara yang digunakan untuk menguraikan laju pertumbuhan suatu fungsi yang tidak lain adalah *time complexity* suatu algoritma. Secara sederhana notasi O-Besar didefinisikan sebagai berikut:

Jika *n* adalah ukuran masukan dan f(n) serta g(n) adalah fungsi positif dari *n* maka f(n) adalah O(g(n)) jika dan hanya jika terdapat konstanta positif *c* dan integer positif n_0 sedemikian rupa sehingga $f(n) \leq c\ g(n)$ untuk semua $n \geq n_0$.

Secara tidak formal, suatu algoritma disebut menunjukkan laju pertumbuhannya merupakan order suatu fungsi matematika jika untuk ukuran masukan *n*, fungsi f(n) dikalikan konstanta positif merupakan batas atas atau limit dari waktu eksekusi algoritma tersebut. Dengan kata lain untuk ukuran masukan *n* yang lebih besar daripada n_0 dan konstanta *c*, waktu eksekusi algoritma tidak akan melampaui c * f(n). Sebagai contoh, karena waktu eksekusi *insertion sort* tumbuh secara kuadratik dengan besarnya masukan *n* ketika ukuran masukan naik, *insertion sort* dikatakan mempunyai order O (n^2).

Kerap kali order disingkat dengan huruf O besar. Notasi ini, dikenal dengan nama notasi O-Besar yang secara khusus digunakan untuk menguraikan efisiensi algoritma. Notasi O-Besar tidak secara spesifik memasukkan suku konstanta. Jika hasil estimasi efisiensi algoritma mencakup jumlah beberapa suku, notasi ini hanya akan mengambil suku dengan order tertinggi. Sebagai contoh algoritma dengan efisiensi $T(n) = n^2+n+10$ maka order algoritma tersebut adalah $O(n^2)$. Ilustrasi berikut menujukkan bagaimana dominasi n^2 terhadap nilai T(n) secara keseluruhan saat nilai *n* menuju tak berhingga.

n	n^2	n	10	T(n)
0	0	0	10	10
10	100	10	10	120
100	10000	100	10	10110
1000	1000000	1000	10	1001010
10000	100000000	10000	10	100010010

Meskipun notasi ini dikembangkan sebagai bagian dari matematika murni, notasi ini kerap dipakai pada analisis algoritma untuk menguraikan algoritma pemakaian sumber-sumber komputasi: kasus terburuk atau kasus rerata waktu eksekusi atau pemakaian memori suatu algoritma kerap disajikan sebagai fungsi dari besaran masukan menggunakan notasi O-Besar. Hal ini memungkinkan perancang algoritma memprediksi perilaku algoritmanya dan menentukan algoritma mana yang akan digunakan tak tergantung pada arsitektur komputer dan *clock rate*. Karena notasi O-Besar mengabaikan nilai konstanta dan kelipatannya dan juga mengabaikan efisiensi untuk argumen-argumen dalam ukuran yang lebih kecil ordernya, maka notasi O besar tidak selalu mencerminkan algortima yang paling cepat pada data tertentu, tetapi pendekatan ini tetap sangat efektif untuk membandingkan berbagai algoritma saat ukuran data masukan menuju tak terhingga.

Konsep ini, O(n), dikenal sebagai salah satu metode *asymptotic analysis* untuk menyatakan batas atas asimptotik. Sebagai contoh, $T(n) = 13\ n^3 + 42\ n^2 + 2\ n \log n + 4\ n$ di mana T merupakan fungsi dari obyek masukan *n*, ditulis sebagai T(n). Jika *n* tumbuh menjadi lebih besar maka nilai n^3 akan jauh lebih besar daripada n^2, n log n dan n sehingga n^3 mendominasi T(n). Waktu eksekusi T(n) secara garis besar mempunyai order n^3 dan notasinya ditulis sebagai $O(n^3)$. Dua metode lain yang dikenal adalah Ω (n) untuk menyatakan batas bawah asimtotik dan Θ(n) untuk menyatakan *tight bound asymptotic*.

Secara formal T(n) didefinisikan sebagai berikut: T(n) = O(f(n)) jika terdapat konstanta c dan n0 sedemikian rupa sehingga $T(n) < c * f(n)$ untuk $n > n_0$. Ini berarti bahwa untuk $n > n_0$ maka c * f(n) merupakan batas atas dari T(n). Sebagai contoh, jika T(n) = 1000n dan $f(n) = n^2$, $n_0 = 1000$ dan c = 1 maka $T(n) \leq 1 * f(n)$ untuk $n > 1000$ dan dikatakan bahwa T(n) = O(f(n)).

Notasi O-Besar adalah cara yang sangat menyenangkan untuk menyajikan skenario keadaan terjelek suatu algoritma, meskipun ia juga dapat digunakan untuk menyajikan kasus rerata. Sebagai contoh, skienario kasus terjelek *quick sort* adalah $O(n^2)$, tetapi rerata waktu eksekusinya adalah O(n log n).

Efisiensi Algoritma

Ada sejumlah terminologi yang terlebih dulu perlu dipahami untuk mengukur efisiensi algoritma. Biasanya efisiensi algoritma dinyatakan sebagai berapa lama ia dieksekusi berdasarkan banyak data masukan yang dinyatakan dalam *n* (Dave and Dave, 2007). Sebagai contoh untuk kasus mengurutkan kata menggunakan metode *brute-force* di atas, efisiensi algoritma adalah n! Bagaimana dengan kasus jika sebagian datanya telah terurut? Apakah efisiensinya bisa menjadi n!/2? Jawabnya adalah tidak karena pembagian dengan 2 sangat tidak signifikan. Jika nilai *n* tumbuh menjadi sangat besarm biasanya suku konstanta kerap diabaikan sehingga efisiensi tetap n!, dengan perkecualian suku konstan adalah nol.

Berdasarkan uraian di atas, kita dapat menguraikan efisiensi sembarang algoritma sebagai fungsi dari data masukan dengan membandingkan algoritma berdasarkan pada "order" nya. Di sini "order" merujuk pada metode matematika yang digunakan untuk membandingkan efisiensi. Sebagai contoh, n^2 adalah order *n* kuadrat dan n! adalah order faktorial *n*. Jelaslah bahwa algoritma order n^2

kurang efisien dibandingjan ndengan algoritma order *n*. Selain order polinomial dan juga order faktorial kita juga mengenal order log n dan order 2^n.

Komputasi Efisiensi

Ada dua pendekatan utama yang dapat digunakan untuk menghitung efisiensi yaitu studi eksprimental dan analisis teoritis (Arora and Barak, 2009). Studi eksprimental dilakukan dengan menulis program untuk mengimplementasikan algoritma. Eksekusi program dengan berbagai ukuran dan komposisi obyek masukan sebagai data empiris. Kemudian gunakan metode seperti *System.currentTimeMillis()* untuk mengukur secara tepat waktu eksekusi program dan gambar hasilnya. Pendekatan ini mempunyai sejumlah batasan, dan diantaranya yang perlu dicatat adalah: (1) perlu mengimplementasikan algoritma meskipun sangat sukar ditulis, (2) hasil tidak dapat dijadikan indikator waktu eksekusi untuk data yang bukan merupakan obyek masukan dan (3) untuk membandingkan dua algoritma diperlukan lingkungan *hardware* dan *software* yang sama.

Sebagai contoh, ambil program untuk melihat isi suatu *array* terurut dengan *size n*. Andaikan program diimplementasikan pada komputer X yang sangat cepat menggunakan algoritma pencarian linear dan pada komputer Y yang lebih lambat menggunakan algoritma pencarian biner. Simulasi waktu eksekusi kedua komputer yang dilakukan oleh Wikipedia adalah sebagai berikut:

n	Komputer X	Komputer Y
15	7 ns	100,000 ns
65	32 ns	150,000 ns
250	125 ns	200,000 ns
1000	500 ns	250,000 ns
…		
1,000,000	500,000 ns	500,000 ns
4,000,000	2,000,000 ns	550,000 ns
16,000,000	8,000,000 ns	600,000 ns
…		
$63{,}072 \times 10^{12}$	31,536 x 1012 ns atau 1 tahun	1,375,000 ns atau 1.375 ms

Berdasarkan data empiris di atas tampak bahwa komputer X lebih superior dari pada komputer Y hanya untuk $n < 1{,}000{,}000$. Akan tetapi jika ukuran data masukkan naik secara signifikan kesimpulan akan berubah secara drastis. Komputer X yang mengeksekusi algoritma pencarian linear menunjukkan laju pertumbuhan yang linear. Waktu eksekusi program berbanding langsung dengan ukuran data masukan. Di sisi lain, komputer Y yang mengeksekusi pencarian biner menunjukkan laju pertumbuhan yang logaritmik. Ukuran data masukan dikalikan dua tidak akan menyebabkan waktu eksekusi menjadi dua kali lebih lama seperti yang dialami oleh komputer X tetapi hanya akan menaikkan waktu eksekusi dengan tambahan suatu konstanta, dalam contoh ini 25,000 ns. Meskipun komputer X jauh lebih cepat daripada komputer Y, waktu eksekusi komputer Y akan lebih lambat komputer X karena laju pertumbuhan algoritmanya lebih lambat.

Karena batasan ini, studi eksprimental bukanlah merupakan pilihan terbaik. Alternatifnya adalah analisis teoritis. Pendekatan ini tidak menggunakan implementasi dengan cara menulis program tetapi menggunakan deskripsi algoritma di aras atas yang dikenal dengan nama *pseudocode* dengan karakteristik utama waktu eksekusi sebagai fungsi dari ukuran obyek masukan *n* sehingga memungkinkan memasukkan semua obyek data yang mungkin. Pendekatan ini memungkinkan mengestimasi kecepatan algoritma yang tidak tergantung pada lingkungan pengembangan program seperti *hardware* dan *software*. Hal ini disebabkan karena algoritma adalah *platform independent*,

dalam arti dapat diimplementasikan dalam sembarang bahasa pemrograman pada sembarang komputer yang menggunakan sembarang sistem operasi.

Pseudocode dipilih karena dapat mendeskripsikan algoritma dengan baik, lebih terstruktur daripada bahasa natural, kurang detil dibandingkan dengan program, menggunakan notasi yang tepat untuk menguraikan program dan menyembunyikan isu perancangan program. Secara rinci *pseudocode* terdiri dari lima komponen yaitu *control flow*, *method declaration*, *method call*, *return value* dan ekspresi seperti *assignment*, *equality testing*, *subscripts* dan notasi-notasi matematika lain yang dimungkinkan.

Operasi dasar yang dilakukan oleh algoritma yang dapat diidentifikasi pada *pseudocode* disebut operasi primitif. Contoh operasi primitif yang banyak dikenal adalah evaluasi ekspresi, penugasan nilai ke variabel, pengindeksan pada *array*, pemanggilan metode, nilai balik ke metode dan sebagainya. Sebagian besar dari mereka tidak tergantung pada bahasa pemrograman dan diasumsikan memerlukan waktu akses konstan pada memori. Dengan mengamati *pseudocode* banyak operasi primitif maksimum suatu algoritma sebagai fungsi dari besaran masukan dapat dihitung. Sebagai contoh:

Algorithm arrayMax(A,n)	# operasi primitif
curMax ← A[0]	1
for i ← 1 to n-1 do	n-1
if A[i] > curMax then	n-1
curMax ← A[i]	n-1
{ increment counter i }	n-1
return curMax	1

Algoritma di atas mengeksekusi 4n-2 operasi primitif. Jika didefinisikan *a* dan *b* sebagai waktu tercepat dan waktu terlambat berturut-turut yang dilakukan oleh operasi primitif dan T(n) sebagai *worst-case time* dari *arrayMax* maka didapat.

$a (4n-2) \leq T (n) \leq b (4n-2)$

Ini berarti bahwa waktu eksekusi T(n) dibatasi oleh dua fungsi linear n. Perubahan *hardware* dan *software* hanya akan mempengaruhi T(n) dengan faktor konstan tetapi tidak akan mempengaruhi laju pertumbuhan T(n) karena laju pertumbuhan linear waktu eksekusi T(n) merupakan karakteristik intrinsik dari *arrayMax*.

Pendekatan lain yang dapat digunakan untuk menentukan nilai order algoritma O-Besar adalah berasumsi bahwa order algoritma adalah O(1), algoritma tidak melakukan apapun dan langsung berakhir apapun bentuk data masukannya. Kemudian cari bagian kode yang diduga mempunyai order tertinggi. Mulai dari sini, penyelidikan efisiensi algoritma dimulai dari luar ke dalam. Hitung efisiensi perulangan luar atau bagian rekursif, kemudian temukan efisiensi kode bagian dalam. Efisiensi total adalah hasil kali efisiensi di setiap lapisan kode. Metode ini dapat digunakan untuk menghitung efisiensi *selection sort* berikut ini.

```
for(int x = 0; x < n; x++) {
    int min = x;
  for (int y = x; y < n; y++) {
    if(array[y] < array[min]) min = y;
  }
  int temp = array[x];
  array[x] = array[min];
  array[min] = temp;
}
```

Contoh Algoritma dengan Nilai O-Besar Berbeda

Berikut ini adalah sejumlah contoh nilai O-Besar yang dapat ditemui di beberapa literatur termasuk Greene and Knuth (1982).

O(1): konstan. Algoritma dengan O-Besar O(1) dieksekusi di kecepatan yang sama tidak tergantung pada data masukannya. Sebagai contoh, algoritma yang selalu menghasilkan nilai yang sama apapun nilai masukannya dapat dipandang sebagai algoritma dengan efisiensi O(1). Contoh lain adalah algoritma untuk menentukan suatu bilangan genap atau ganjil, *look table* atau *hash table* adalah konstan ukuran tabel tetap. O(log n): logaritmik. Algoritma yang didasarkan pada pohon biner kerap mempunyai efisiensi O(log n). Hal ini disebabkan karena BST (*binary search tree*) yang sangat seimbang mempunyai banyak lapisan log dan untuk mencari sembarang elemen di BST memerlukan penelusuran satu simpul di setiap lapisan. Algoritma untuk mencari suatu item di *array* terurut menggunakan pencarian biner atau *balanced search tree* serta semua operasi di binomial heap adalah O(log n).

O(n): linear. Pencarian linear untuk menemukan suatu elemen dalam suatu *array* tak urut dengan *n* elemen. Algoritma tipe ini hanya memerlukan satu putaran untuk seluruh *array*. Keadaan terbaik terjadi jika yang dicari terletak di posisi pertama sehingga O(1). Keadaan terjelek terjadi jika elemen yang dicari terletak di posisi terakhir atau tidak ada dalam *array*. Untuk itu algoritma harus menelusuri seluruh *array* untuk mencek setiap elemen dalam *array* yang berarti O(n). Keadaan rerata berdasarkan pada asumsi bahwa elemen yang dicari ada dalam *array* dan setiap elemen mempunyai peluang yang sama untuk ditemukan. Pencarian hanya perlu mengunjungi n/2. Contoh lain adalah *link-list*. Akses elemen pada *link-list* adalah O(n) karena *link-list* tidak mendukung akses acak. Menambahkan dua integer n-bit menggunakan *ripple carry* adalah O(n).

O(n log n): *loglinear, quasilinear* atau *linearithmik*. Algoritma pengurutan yang baik kerap mempunyai order O(n log n). Contoh algoritma dengan efisiensi ini adalah algoritma yang tergabung dalam kelompok *devide and conquer* (DAC) seperti *quick sort* (*best* dan *average case*) dan *merge sort*. Algoritma *merge sort* membagi *array* menjadi dua bagian, urutkan kedua *subarray* secara rekursif dengan memanggil dirinya sendiri dan kemudian gabung kembali hasilnya kedalam *array* tunggal. Karena setiap kali membagi *array* menjadi dua bagian maka perulangan luar mempunyai efisiensi log n, dan untuk setiap "level" *array* yang dibagi menjadi dua bagian maka harus menggabungkan kembali semua elemen ke dalam satu *array*, operasinya membutuhkan order *n*. Algoritma pengurutan paling popular *quick sort* mempunyai kinerja terbaik O(n log n) yang membuatnya menjadi algoritma pengurutan yang sangat cepat didasarkan pada asumsi bahwa semua nilai berbeda dan dalam keadaan acak. Tetapi data masukan yang paling jelek akan membuat kinerjanya menjadi $O(n^2)$. Contoh lain algoritma loglinear adalah *Fast Fourier Transform* (FFT) dan *heapsort.*

$O(n^2)$: kuadratik. Cukup efisien karena masih tetap dalam rentang waktu polinomial. Contoh biasanya ada dalam kelompok pengurutan data seperti s*election sort, insertion sort, quick sort (worst case), shell sort, bubble sort* (*worst case* atau implementasi naïf). Insertion sort yang diaplikasikan pada *array n* elemen berdasarkan pada asumsi semua nilai elemen berbeda dan dalam keadaan acak. Secara rerata, separuh dari elemen dalam *array* $A_1 \ldots A_j$ lebih kecil daripada elemen A_{j+1} dan separuh lainnya lebih besar. Dengan demikian algoritma membandingkan secara rerata elemen ke j+1 yang akan disisipkan separuh sub-*array* yang telah urut, sehingga $t_j = j/2$. Hasil akhirnya waktu eksekusi kasus rerata adalah fungsi kuadratik dari ukuran masukan; sama dengan waktu eksekusi kasus terjelek. Contoh lain algoritma kuadratik adalah mengalikan dua bilangan *n*-digit menggunakan algoritma sederhana.

$O(2^n)$: eksponensial. Efisiensi non-polinomial yang paling penting adalah *exponential time*. Banyak masalah penting yang hanya dapat diselesaikan oleh algoritma dengan efisiensi seperti ini.

Salah satu contohnya adalah faktorisasi bilangan sangat besar yang disajikan dalam notasi biner. Metode yang dikenal hanyalah *trial and error*. Contoh lain adalah mencari solusi eksak dari *traveling salesman problem* (TSP) menggunakan pemrograman dinamik, menentukan apakah dua pernyataan logikal adalah ekuivalen menggunakan algoritma *brute-force*. O(n!): faktorial. Mencari solusi TSP menggunakan algoritma *brute-force*, menentukan determinan menggunakan ekspansi minor. Pernyataan f(n) = O(n!) kadang-kadang disimplifikasi menjadi f(n) = $O(n^n)$ untuk memberikan formula kompleksitas yang lebih sederhana.

PENUTUP

Pakar komputer menggunakan notasi O-Besar sebagai alat untuk mendiskusikan berbagai tipe algoritma dan menganalisis kode yang ditulisnya. Kemampuan untuk melakukan analisis penggalan kode atau algoritma dan memahami efisiensinya sangat vital untuk memahami ilmu komputer sekaligus untuk meyakinkan bahwa program yang disusun dieksekusi lebih cepat dan atau lebih lambat daripada kode lain. Dengan demikian, efisiensi algoritma sangat krusial untuk dianalisis. Dalam tataran praktis notasi O-Besar sangat penting diketahui karena penggunaan secara tidak sengaja algoritma yang tidak efisien secara signifikan punya pengaruh pada kinerja sIstem. Pada aplikasi yang sangat sensitif terhadap waktu, algoritma yang membutuhkan waktu eksekusi yang lama bisa menghasilkan keluaran yang sia-sia karena sudah kadaluwarsa. Selain itu, algoritma yang tidak efisien juga membutuhkan catu daya dan ruang memori yang besar sehingga tidak lagi ekonomis jika dieksekusi.

DAFTAR PUSTAKA

Arora, S. and Barak, B. (2009). *Computational Complexity: A Modern Approach*. New York: Cambridge University Press.

Dave, P. H. and Dave, H. B. (2007). *Design and Analysis of Algorithms*: *Efficiency of Algorithms*, 371 – 404. Delhi**:** Pearson Education India.

Goldreich, Oded. (2008). *Computational Complexity: A Conceptual Perspective*. New York: Cambridge University Press.

Greene, D. A. & Knuth, D. E. (1982). *Mathematics for the Analysis of Algorithms* (2nd ed.). Boston: Birkhäuser.

Higham, N. J. (1998). *Handbook of Writing for the Mathematical Sciences*, (2nd ed.). Philadelphia: SIAM.

Knuth, D. E. (1976). Big Omicron and big Omega and big Theta*. ACM SIGACT News, 8* (2).

Knuth, D. E. (1998). Teach Calculus with Big O. *Notices of the American Mathematical Society*, *45* (6), 687. Diakses dari http://www.ams.org/notices/199806/commentary.pdf.

Knuth, D. E. (1999). *The Art of Computer Programming*, (3rd ed.). Boston: Addison-Wesley.

Sedgewick, R. (1998). Algorithms in C, Parts 1-4. *Fundamentals, Data Structures, Sorting, Searching,* (3rd ed.). Boston: Addison-Wesley.