Analisis Kecepatan Sorting Dengan Notasi Big O

Rama Aulia – NIM : 13506023

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : ramaaulia@yahoo.co.id

Abstrak

Sorting atau pengurutan adalah salah satu proses yang sangat dibutuhkan di dalam pemrograman. Sorting atau pengurutan ini adalah proses mengatur sekumpulan objek menurut urutan atau susunan tertentu. Adanya kebutuhan akan pengurutan melahirkan beberapa macam pengurutan. Metode-metode pengurutan antara lain, yaitu Bubble Sort, Selection Sort(Maximum dan Minimum sort), Insertion Sort, Heap Sort, Shell Sort, Quick Sort, Merge Sort, Radix Sort, Tree Sort. Masing-masing dari metode pengurutan ini mempunyai kelebihan dan kelemahan. Pada suatu masalah pengurutan dapat dipakai berbagai macam metode. Namun, efisien dari suatu algoritma sorting tetap harus dipertimbangkan. Pada makalah ini akan dibahas kecepatan sorting tersebut dengan notasi Big O.

Notasi Big O adalah notasi matematika yang digunakan untuk menggambarkan suatu fungsi asimptotik. Notasi Big O sering digunakan untuk menjelaskan berapa besar ukuran dari suatu data mempengaruhi penggunaan sebuah algoritma dari sumber komputasi. Notasi Big O juga biasa disebut sebagai notasi Landau (Landau notation), Bachman-Landau notation, dan notasi asimptotik (Asimptotik notation).

Notasi Big O mempunyai aplikasi pada dua buah bidang. Pada bidang matematika, notasi tersebut biasanya digunakan untuk menjelaskan tahap sisa dari deret tak terhingga, khususnya pada deret asimptotik. Pada bidang ilmu komputer, notasi ini sangat berguna dalam analisis dari kompleksitas algoritma. Notasi ini pertama kali diperkenalkan pada tahun 1894 oleh Paul Bachman, yaitu pada volume kedua dari bukunya *Analytische Zahlentheorie* (analisis teori bilangan). Simbol O digunakan untuk menggambarkan dan menjelaskan batas atas dari asimptotik suatu jarak dari sebuah fungsi untuk suatu fungsi yang lebih sederhana. Selain itu juga terdapat simbol Ω, ω, dan Θ yang berguna untuk menggambarkan batas atas, batas bawah, dan rata-rata.

1. Efisiensi Algoritma

Efisiensi di dalam algoritma sangat dipertimbangkan karena suatu masalah dapat diselesaikan dengan berbagai macam cara yang dalam hal ini disebut sebagai algoritma(langkah penyelesaian masalah)

Algoritma yang bagus adalah algoritma yang efisien dimana algoritma tersebut dikatakan bagus karena dinilai dari aspek kebutuhan waktu dan ruang membutuhkan jumlah yang sedikit.

Di atas telah kita ketahui bahwa terdapat begitu banyak pilihan jika kita ingin menyelesaikan masalah pengurutan. Aspek yang harus dipertimbangkan di dalam pemilihan algoritma sorting berhubungan dengan kemangkusan atau keefisienan algoritma. Hal inilah yang mendasari hubungan antara kecepatan algoritma sorting dengan keefisienan algoritma yang dalam hal ini digunakan fungsi dari notasi Big O.

Kemangkusan algoritma dapat diukur dari orde yang terdapat di dalam persamaan kompleksitas waktu. Kompleksitas waktu diukur dari jumlah tahapan komputasi di yang dibutuhkan dalam menjalankan algoritma dimana kompleksitas waktu tersebut merupakan fungsi dari jumlah masukan n.

Hal–hal yang mempengaruhi kompleksitas waktu :

1. Jumlah masukan data untuk suatu algoritma(n).
2. Waktu yang dibutuhkan untuk menjalankan algoritma tersebut.

3. Ruang memori yang dibutuhkan untuk menjalankan algoritma yang berkaitan dengan strutur data dari program.

Kompleksitas mempengaruhi performa atau kinerja dari suatu algoritma. Kompleksitas dibagi menjadi 3 jenis, yaitu worst case, best case, dan average case. Masing-masing jenis kompleksitas ini menunjukkan kecepatan atau waktu yang dibutuhkan algoritma untuk mengeksekusi sejumlah kode.

2. Notasi Big O

Notasi Big O sangat berguna di dalam menganalisa efisiensi dari suatu algoritma.

T(n) = O(f(n)), artinya *T(n)* berorde paling besar *f(n)* bila terdapat konstanta *C* dan n_o sehingga

$T(n) \leq C(f(n))$

Untuk $n \geq n_o$

Terdapat 2 jenis penggunaan notasi Big O, yaitu :

1. Infinite asymptotics
2. Infinitesimal asymptotics

Perbedaan kedua jenis penggunaan notasi ini hanya pada aplikasi. Sebagai contoh, pada infinite asymptotics dengan persamaan

$$T(n) = 2n^2 - 2n + 2$$

Untuk n yang besar, pertumbuhan T(n) akan sebanding dengan n^2 dan dengan mengabaikan suku yang tidak mendominasi kita, maka kita tuliskan

$$T(n) = O(n^2)$$

Pada infinitesimal asymptotic**s,** Big O digunakan untuk menjelaskan kesalahan dalam aproksimasi untuk sebuah fungsi matematika, sebagai contoh

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2} + \ O(x^3), \qquad x \to 0$$

Kesalahannya memiliki selisih

$$e^x - \ (1 + \frac{x}{1} + \frac{x^2}{2})$$

3. Analisis Algoritma Sorting

Tujuan dari analisis algoritma adalah untuk mengetahui efisiensi dari algoritma. Dalam hal ini dilakukan pembandingan antara dua atau lebih algoritma pengurutan.

Pembandingan yang dilakukan adalah running time atau waktu komputasi antara algoritma yang satu dengan algoritma yang lain dan juga jumlah proses atau langkah atau pembandingan data. Hasil dari analisis yang didapat adalah jumlah running time atau waktu komputasi dan jumlah proses dari masing-masing algoritma.

Tahap analisis adalah melakukan pengecekan program untuk memastikan bahwa program telah benar secara logika maupun sintak (tahap tracing atau debugging). Tahap selanjutnya yaitu menjalankan program untuk mengetahui running time atau waktu komputasi dalam hal ini termasuk jumlah langkah. Data uji yang digunakan adalah data yang tidak terurut atau data random, terurut membesar/, dan terurut mengecil.

Sebagai contoh, pada bahasan ini akan dianalisa algoritma selection sort dan quick sort.

Selection Sort

PSEUDOCODE

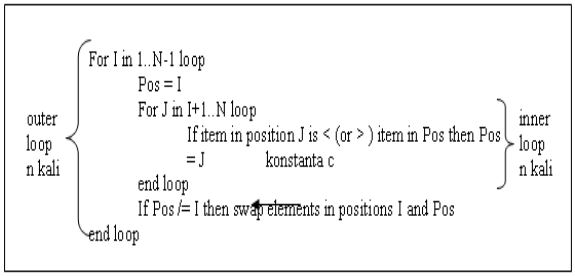
```
For I in 1..N-1 loop
        Pos = I
        For J in I+1..N loop
                If item in position J is <
                (or > ) item in Pos then
                Pos = J

        end loop
        If Pos /= I then swap elements in
        positions I and Pos

end loop
```

Pengurutan pada algoritma selection sort, dilakukan dengan cara mencari suatu nilai ekstrim (minimal atau maksimal) untuk ditukarkan dengan elemen terujung yang ada pada suatu proses loop. Elemen terujung ini tidak akan dimasukkan pada proses loop berikutnya karena nilai terujung ini sudah menjadi nilai maksimum atau minimum. Proses ini dilakukan juga terhadap sisa dari elemen array.

Penjelasan program :
Sebagai contoh array tersebut bernama A. Pada setiap saat terdapat i buah data terurut pada A[0], A[1], A[2], …, A[i-1], dan data tak terurut pada A[i], A[i+1], …, A[n-1]. Algoritma melakukan pencarian terhadap A[j] terkecil dari data set yang belum terurut tersebut, misalnya didapat A[m] setelah itu melakukan swaping (penukaran) A[i] dengan A[m] sehingga sudah terurut i+1 data dalam A. Hasil pengurutan bergantung terhadap data nilai ekstrim yang diinginkan.



Dengan demikian, total running timenya adalah c x n x n = $cn^2 = O(n^2)$

Quick Sort

PSEUDOCODE

```
Function QuickSort (Array(First..Last))
    if Array contains only one element then return Array;
    //Partitioning step
    Choose a pivot and put swap it with Array[First];
    LeftPtr = First + 1;
    RightPtr = Last;
    while (RightPtr > = LeftPtr)
    while (Array[LeftPtr] < Pivot)
    increment LeftPtr;
    while (Array[RightPtr] > Pivot)
    decrement RightPtr;
    swap Array[LeftPtr] and Array[RightPtr];
    swap Array[First] and Array[RightPtr];
    //End partitioning step
    Array[First..RightPtr-1] = QuickSort(Array[First ..
    RightPtr-1]);
    Array[RightPtr+1..Last] =
    QuickSort(Array[RightPtr+1..Last]);
```

Quick sort adalah algoritma pengurutan tercepat diantara metode pengurutan yang lain. Quick sort menggunakan metode *divide-and-conquer recursive algorithm*.

4 tahap dasar dalam mengurutkan array misal A di dalam algoritma quick sort :

1. Jika jumlah elemen A sama dengan 0 atau 1, lalu stop.

2. Ambil secara random elemen c di dalam array A. Hal ini disebut pivot.

3. Bagi menjadi 2 bagian array A, elemen selain c di dalam array A menjadi 2 g kelompok. A1 = {x? A – {c}? x ≤ c} dan A2 = {x? A – {c}? x ≥ c}

4. Lakukan quick sort pada A1 dan A2, sehinggan didapat persamaan

   $T(n) = 2T(\frac{n}{2}) + \theta(n)$

   Notasi Big O dapat dihitung dengan metode master :

   $T(n) = 2T(\frac{n}{2}) + \theta(n)$? $a = 2$, $b = 2$ dan $f(n) = \theta(n) = n$
   $n \log ba = n \log 2.2 = n$
   sehingga sesuai dengan kasus 2 teorema master karena
   $n \log ba = n = f(n)$

   jadi total running timenya :
   $T(n) = \theta(n \log n)$

Penggunaan Big O terhadap kasus rata-rata(average case) dari quick sort :

Diberikan *C(n)* yang merupakan jumlah perbandingan yang dibutuhkan untuk mengurutkan sebuah array dengan n elemen. Karena array dengan 0 atau 1 elemen tidak diurutkan maka *C(0) = C(1) = 0*. Asumsi diberikan sebuah array dengan n elemen. Setiap elemen bisa dipilih sebagai batas. Peluang setiap elemen untuk menjadi batas adalah sama. *C(i-1)* dan *C(n-i)* menyatakan jumlah perbandingan yang dibutuhkan untuk mengurutkan 2 buah sub-array ada

$C(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(C(i-1) + C(n-i))$,
untuk n ≥ 2

perbandingan, dimana *n-1* adalah jumlah perbandingan di dalam partisi dari array dengan jumlah elemen n. Dengan beberapa

penyederhanan, maka persamaan di atas menjadi seperti berikut

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

$C(n) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} C(i)$

$nC(n) = n(n-1) = 2 \sum_{i=0}^{n-1} C(i)$

Untuk menyelesaikan persamaan, maka operator penjumlahan dibuang terlebih dahulu. Persamaan terakhir dikurangi dengan persamaan berikut

$(n+1)C(n+1) = (n+1)n + 2 \sum_{i=0}^{n} C(i)$

yang menghasilkan

$(n+1)C(n+1) - nC(n) = (n+1)n - n(n-1) + 2$

$(\sum_{i=0}^{n} C(i) - \sum_{i=0}^{n-1} C(i))$

$= 2C(n) + 2n$

dimana

$$\frac{C(n+1)}{n+2} = \frac{C(n)}{n+1} + \frac{2n}{(n+1)(n+2)}$$

$$= \frac{C(n)}{n+1} + \frac{4}{n+2} - \frac{2}{n+1}$$

Persamaan ini bisa dikembangkan, yang memberikan

$$\frac{C(2)}{3} = \frac{C(1)}{2} + \frac{4}{3} - \frac{2}{2} = \frac{4}{3} - \frac{2}{2}$$

$$\frac{C(3)}{4} = \frac{C(2)}{3} + \frac{4}{4} - \frac{2}{3}$$

$$\frac{C(4)}{5} = \frac{C(3)}{4} + \frac{4}{5} - \frac{2}{4}$$

$$\vdots$$

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{4}{n+1} - \frac{2}{n}$$

$$\frac{C(n+1)}{n+2} = \frac{C(n)}{n+1} + \frac{4}{n+2} - \frac{2}{n+1}$$

dimana

$$\frac{C(n+1)}{n+2} = (\frac{4}{3} - \frac{2}{3}) + (\frac{4}{4} - \frac{2}{3}) + (\frac{4}{5} - \frac{2}{4}) + \dots + (\frac{4}{n+1} - \frac{2}{n}) + (\frac{4}{n+2} - \frac{2}{n+1})$$

$$= -\frac{2}{2} + \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \dots + \frac{2}{n} + \frac{2}{n+1} + \frac{4}{n+2}$$

$$= -4 + 2H_{n+2} + \frac{2}{n+2}$$

H_{n+2} adalah sebuah bilangan harmonic. Dengan menggunakan aproksimasi untuk bilangan ini

$$C(n) = (n+1)(-4 + 2H_{n+1} + \frac{2}{n+1})$$

$$C(n) = (n+1)(-4 + 2O(\ln n) + \frac{2}{n+1})$$

$C(n) = O(n \log n)$

Jadi, terbukti dari rangkaian penyelesaian persamaan di atas kompleksitas waktu dengan kata lain notasi Big O untuk penyelesaian permasalahan pengurutan dengan quick sort adalah $T(n)$ atau $C(n) = O(n \log n)$.

Penjelasan tentang kecepatan beberapa algoritma pengurutan dengan notasi Big O :

1. Insertion sort
   Kompleksitas untuk kasus terbaiknya adalah $O(n)$. Pada kasus terbaik ini, elemen-elemen pada array sudah terurut. Elemen di kiri lebih kecil daripada elemen di kanan. Keterurutan pada elemen tersebut adalah menaik(ascending).

   Kompleksitas untuk kasus terburuk adalah $O(n^2)$. Pada kasus terburuk ini keterurutan awal elemen-elemen pada array adalah menurun(descending).

   Kompleksitas untuk kasus rata-rata adalah $O(n^2)$. Hal ini terjadi jika array terurut secara acak.

   Insertion sort lebih simple dan lebih baik digunakan untuk jumlah data yang sangat kecil. Untuk jumlah data yang besar, insertion sort tidak cocok untuk diterapkan karena kompleksitas waktu dari algoritma tersebut adalah $O(n^2)$.

2. Merge Sort
   Merge sort merupakan algoritma pengurutan yang bagus. Algoritma ini mengaplikasikan pembagian elemen array menjadi 2 buah sub-array. Lalu, menggabungkan data yang ada pada kedua buah sub-array tersebut.

   Kompleksitas waktu untuk semua kasus dari algoritma merge sort adalah *O(n log n)*. Kita bisa menerapkan pembagian general dan teorema Conquer untuk menghitung kompleksitas dari algoritma merge sort. Kita ketahui bahwa

   $$T(n) = aT(\frac{n}{b}) + O(n^k)$$

   n merupakan jumlah masalah utama
   a merupakan jumlah dari sub-masalah
   n/b merupakan jumlah dari sub-masalah
   dan $O(n^k)$ adalah kompleksitas dari pembagian dan operasi merging.

   $T(n) = O(n^{logba})$ jika $a>b^k$

   $T(n) = O(n^k log\ n)$ jika $a = b^k$

   $T(n) = O(n^k)$ jika $a<b^k$

   Pada merge sort, masalah utama dibagi kedalam 2 sub-masalah, jadi a = 2. Jumlah dari masalah utama dibagi dengan 2, jadi b = 2. Kompleksitas untuk pembagian adalah *O(1)* dan kompleksitas waktu untuk merging adalah *O(n)*. Total waktu untuk pembagian dan merging adalah $O(1) + O(n) = O(n) = O(n^1)$. Jadi, k = 1.

   Karena $a = b^k$ maka kompleksitas dari merge sort adalah $O(n^k log\ n) = O(n\ log\ n)$.

3. Quick sort
   Kompleksitas untuk kasus terbaik dan kasus rata-rata adalah *O(n log n)*.

   Kompleksitas untuk kasus terburuk adalah $O(n^2)$.

   Pemilihan pivot sangat penting karena bergantung pada pivot kita mendapatkan kompleksitas waktu kasus terbaik atau terburuk.

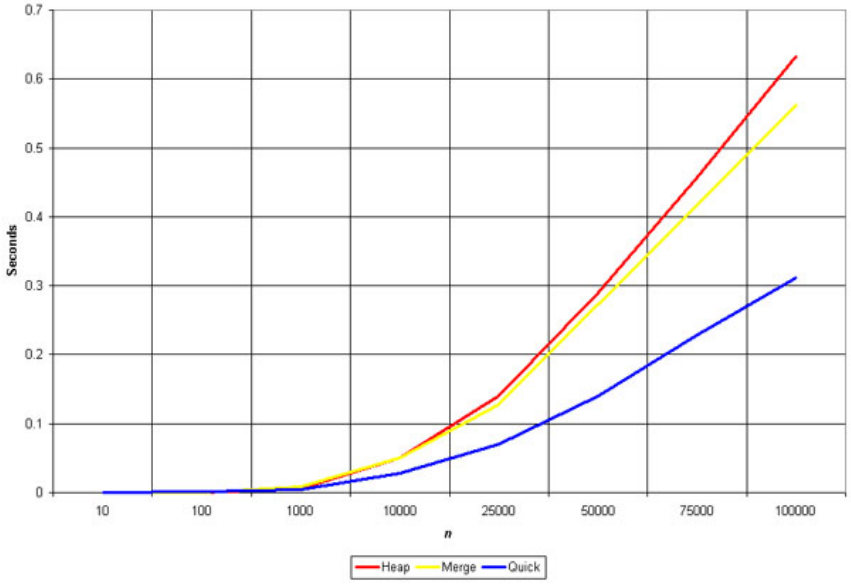
   Jika pivot yang dipilih merupakan elemen pertama atau terakhir dari array maka kompleksitas waktu yang didapat adalah $O(n^2)$.

Berikut tabel yang menunjukkan kompleksitas waktu dari algoritma sorting.

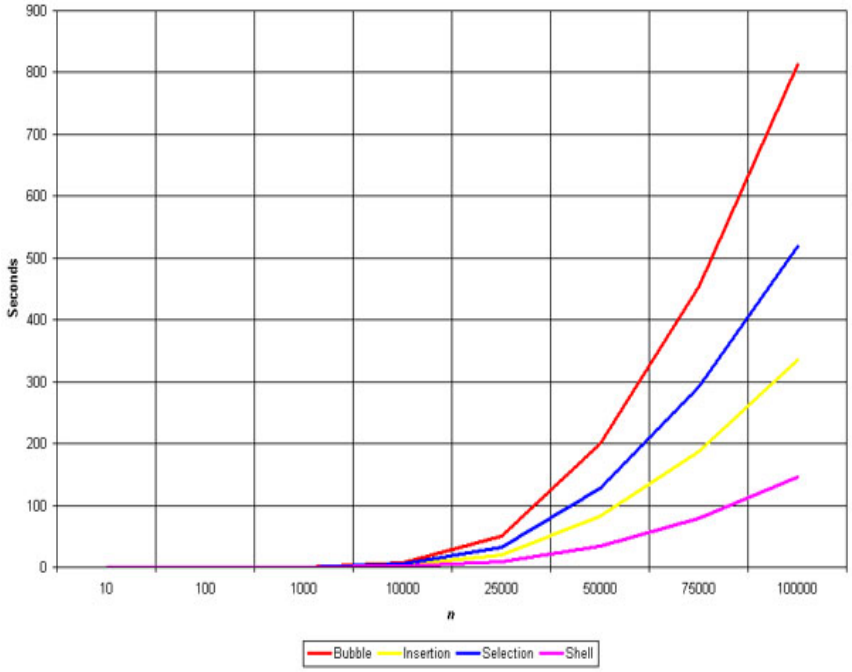
Nama	Average	Worst	Method
-Bubble sort	-	$O(n^2)$	Pertuka-ran
-Selection sort	$O(n^2)$	$O(n^2)$	Seleksi
-Insertion sort	$O(n+d)$	$O(n^2)$	Penyisi-pan
-Heap sort	$O(nlogn)$	$O(nlogn)$	Seleksi
-Shell sort	-	$O(nlog^2 n)$	Penyisi-pan
-Quick sort	$O(nlogn)$	$O(n^2)$	Pemisa-han
-Merge sort	$O(nlogn)$	$O(nlogn)$	Penggabu ngan
-LSD radix sort	$O(n.k/s)$	$O(n.k/s)$	
-MSD radix sort	$O(n.k/s)$	$O(n.(k/s).2^5)$	
-Binary Tree sort	$O(nlogn)$	$O(n^2)$	Penyisi-pan

Grafik yang menunjukkan kompleksitas waktu dari algoritma sorting :

O(n log n) sort
(kecepatan algoritma sorting yang cepat)



O(n^2) sort
(kecepatan algoritma sorting yang lambat)



Kesimpulan

Algoritma sorting merupakan algoritma yang penting didalam pemrograman. Terdapat banyak sekali jenis algoritma sorting. Namun dalam menjalankan program sangat perlu dipertimbangkan efisiensi dari program tersebut.

Salah satu cara untuk menganalisa kecepatan algoritma sorting saat running time adalah dengan menggunakan notasi Big O. Algoritma sorting mempunyai kompleksitas waktu terbaik, terburuk, dan rata-rata. Dengan notasi Big O, kita dapat mengoptimalkan penggunaan algoritma sorting. Sebagai contoh, untuk kasus dimana jumlah masukan untuk suatu pengurutan banyak, lebih baik digunakan algoritma sorting seperti quick sort, merge sort, atau heap sort karena kompleksitas waktu untuk kasuk terburuk adalah $O(n \log n)$. Hal ini tentu akan sangat berbeda jika kita menggunakan algoritma sorting insertion sort atau bubble sort dimana waktu yang dibutuhkan untuk melakukan pencarian akan sangat lama. Hal ini disebabkan kompleksitas waktu terburuk untuk algoritma sorting tersebut dengan jumlah masukan yang banyak adalah $O(n^2)$.

Daftar Pustaka

[1] Wikipedia, http://en.wikipedia.org/wiki/Sorting_algorithm/, Tanggal akses : 30 Desember 2007 pukul 18.30, 2006.

[2] Wikipedia, http://en.wikipedia.org/wiki/Big_O_notation/, Tanggal akses : 30 Desember 2007 pukul 18.30, 2006.

[3] R. Munir, *Bahan Kuliah IF2151 Matematika Diskrit*, Departemen Teknik Informatika, Institut Teknologi Bandung, 2004.

[4] A. Drozdek, *Data Structures and Algorithms in C++*, Brooks/Cole, 2001.

[5] Sorting Algorithm Analysis, http://ilmu-komputer.net/algorithms/sorting-algoritm-analysis/, Tanggal akses : 31 Desember 2007 pukul 09.00, 2006.

[6] Sort, http://linux.wku.edu/~lamonml/algor/sort/sort.html/, Tanggal akses : 31 Desember 2007 pukul 09.00, 2006.