Analisis Kompleksitas Waktu Untuk Beberapa Algoritma Pengurutan

Dibi Khairurrazi Budiarsyah, 13509013[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13509013@stei.itb.ac.id

***Abstract*—Tidak semua data yang kita miliki pada sebuah array telah terurut dengan baik. Terkadang masih ada data yang letaknya acak. Untuk beberapa pengolahan data, data terurut merupakan suatu kebutuhan yang harus dipenuhi. Oleh karena itu dibutuhkanlah sebuah algoritma pengurutan yang dapat mengurutkan semua data pada sebuah array dengan cepat dan tepat. Beberapa algoritma pengurutan yang tersedia antara lain *quick sort, merge sort, heap sort, bubble sort,* dll. Pada makalah ini dilakukan analisis terhadap enam algoritma pengurutan yang sering digunakan yakni *quick sort, merge sort, heap sort, bubble sort, insertion sort,* dan *selection sort.* Keenam algoritma ini di analisis kebutuhan waktu (T(n)) dan kompleksitas waktu asimptotiknya (big-O). Simpulannya, algoritma yang cocok digunakan di berbagai kondisi adalah *merge sort* dan *heap sort* karena kompleksitas waktu asimptotiknya sebesar O(n log n).**

***Index Terms*—Array, data, pengurutan, kompleksitas waktu.**

I. PENDAHULUAN

Pengaksesan data pada sebuah *array* dilakukan secara sekuensial. Dengan data yang terurut, pengambilan data akan menjadi lebih efisien dan cepat. Untuk itulah dibutuhkan suatu algoritma yang dapat mengurutkan elemen-elemen pada *array*.

Algoritma pengurutan inipun banyak jenisnya. Ada yang menggunakan perbandingan dan ada yang tidak menggunakan perbandingan. Contoh algoritma pengurutan dengan perbandingan adalah *quick sort*, *bubble sort*, *merge sort*, *heap sort*, *insertion sort*, *selection sort*, *cocktail sort*, *comb sort*, *gnome sort*, dll.

Selain algoritma pengurutan dengan perbandingan, terdapat algoritma pengurutan tanpa perbandingan. Beberapa contohnya antara lain *bucket sort*, *radix sort*, *counting sort*, *spread sort*, *pigeon hole sort*, dll. Algoritma pengurutan yang dibahas pada makalah ini hanyalah algoritma pengurutan dengan perbandingan.

Dengan adanya banyak algoritma pengurutan yang beredar tentunya memberikan kita banyak pilihan untuk menentukan algoritma mana yang ingin digunakan. Algoritma manakah yang memiliki waktu pengerjaan tercepat untuk mengurutkan array dengan n elemen?. Penghitungan waktu tercepat tidak dapat dilakukan dengan menghitung runtime dari program yang kita buat. Karena tiap komputer memiliki kemampuannya masing-masing dan tentunya tiap komputer akan menghasilkan runtime yang berbeda.

Oleh karena itu, penghitungan kecepatan algoritma ditentukan dari besar kompleksitas waktu dari algoritma tersebut (biasanya dinotasikan dengan T(n)). Selain itu dianalisis pula besar dari kompleksitas waktu asimptotik dari suatu algoritma yang dinotasikan dengan big-O. Semakin besar nilai big-O, maka semakin tidak efektiflah algoritma tersebut digunakan. Pada makalah ini, akan dilakukan analisis besarnya kompleksitas waktu algoritma (T(n)) dan kompleksitas waktu asimptotik (O) terhadap beberapa algoritma pengurutan yang populer.

II. ALGORITMA PENGURUTAN

Terdapat beberapa algoritma pengurutan yang tersedia. Algoritma tersebut biasanya dikelompokkan berdasarkan kompleksitasnya, penggunaan memori, sifat rekursif, dan *adaptibility*. Pada makalah ini, algoritma yang dibahas hanyalah algoritma pengurutan dengan perbandingan.

Algoritma pengurutan yang tergolong kepada kelompok ini merupakan algoritma yang menggunakan perbandingan dua buah elemen atau lebih untuk melakukan pengurutan. Beberapa algoritma pengurutan yang termasuk kedalam kelompok ini antara lain :

1. *Quick sort*

   Quick sort merupakan algoritma yang ditemukan oleh C. A. R. Hoare pada tahun 1960 dan kemudian dikenalkan secara luas pada tahun 1962. Algoritma ini menggunakan prinsip *divide and conquer*. Algoritma ini sering dijadikan pilihan karena mudah dan hemat untuk diimplementasikan [2].

   Algoritma ini bekerja dengan membagi *array* menjadi dua bagian. Pertama, sebuah elemen dipilih untuk dijadikan *pivot*. Kemudian elemen-elemen

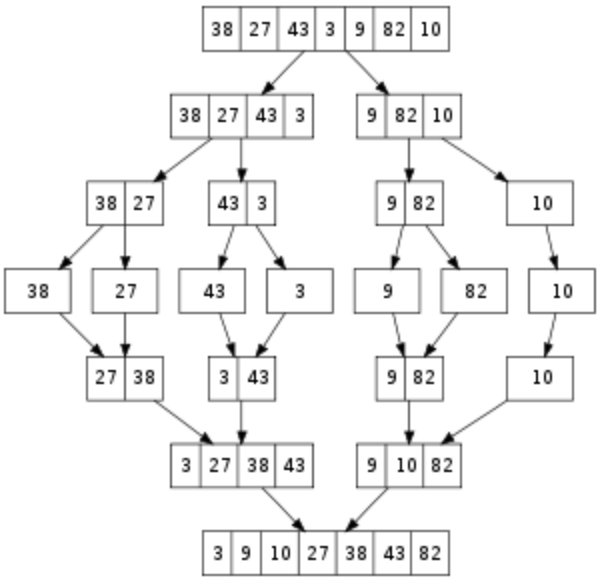
yang nilainya lebih kecil daripada nilai *pivot* tersebut akan diletakkan di sebelah kiri *pivot*, sedangkan yang lebih besar diletakkan di sebelah kanan *pivot*.

2. Merge Sort

Merge sort merupakan algoritma yang dicetuskan oleh John von Neuman pada tahun 1945 [4]. *Merge sort* juga menggunakan prinsip *divide and conquer* [3].

Merge sort membagi array menjadi dua secara terus menerus hingga hanya tersisa satu elemen pada *sub-array* yang terbentuk. Kemudian elemen-elemen tersebut di urutkan lalu di gabung secara terus menerus hingga terbentuk *array* dengan ukuran yang sama dengan *array* asal.

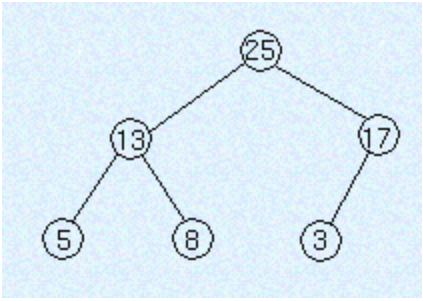
Proses pengurutan dan penggabungan elemen pada *merge sort* dapat dilihat pada gambar dibawah ini.



Gambar 1. Proses penggabungan dan pengurutan dari *merge sort*
Sumber :
http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Merge_sort.html

3. Heap Sort

Heap sort adalah algoritma pengurutan yang dicetuskan oleh J. W. J. William pada tahun 1964 [8]. Algoritma ini bekerja dalam tiga tahap yakni *heapify*, *build heap* dan pembentukan *array* terurut. Heap sendiri adalah representasi array dalam bentuk pohon biner lengkap [9]. Contoh *heap* dari sebuah *array* dapat dilihat pada gambar dibawah ini.



Gambar 2. *Heap* dari *array* A = [25, 13, 17, 5, 8, 3]
Sumber :
http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/heapSort.htm

Heap dibentuk dari *array* yang dimiliki. Untuk memenuhi karakteristik dari *heap*, setiap ada elemen baru yang dimasukkan pada heap, maka akan dilakukan pengecekan elemen. Apabila elemen baru yang dimasukkan ke dalam *heap* nilainya lebih besar daripada simpul akar dari heap tersebut, maka nilai pada simpul akar akan ditukar dengan nilai elemen yang baru dimasukkan pada *heap*. Dengan demikian, nilai simpul akar pada *heap* akan selalu lebih besar dibanding simpul anak-anaknya.

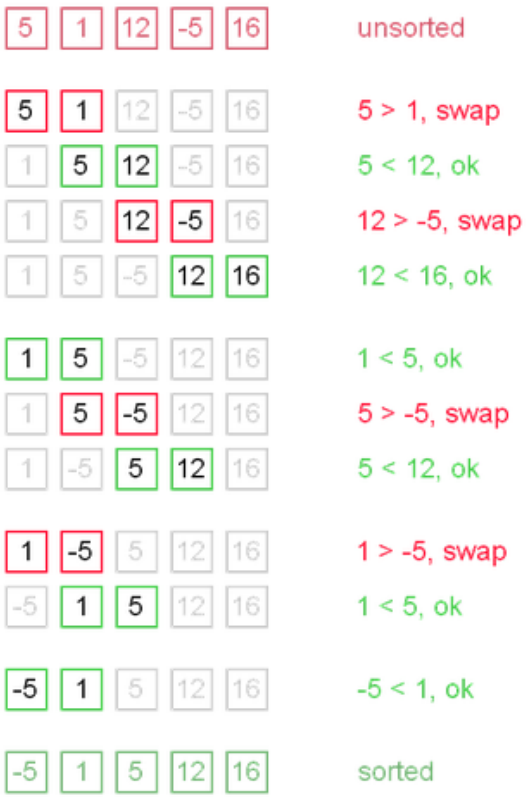
Setelah itu, pembentukan *array* terurut dilakukan dengan cara memindahkan elemen terbesar pada *heap* ke suatu *array*. Setiap terjadi proses pemindahan ini, *heap* akan dibentuk ulang. Pemindahan ini dilakukan terus menerus hingga terbentuklah *array* yang telah terurut. Pengurutan *array* tergantung dari jenis *heap* yang dipilih, apakah *min-heap* atau *max-heap*.

4. Bubble Sort

Bubble sort merupakan salah satu algoritma pengurutan yang simpel dan populer. Algoritma ini biasa digunakan untuk pendahuluan ketika belajar tentang algoritma pengurutan. *Bubble sort* merupakan algoritma yang stabil dan adaptif [5].

Cara kerja algoritma ini adalah dengan membandingkan dua buah elemen pada *array* dimulai dari elemen pertama dan kedua. Apabila elemen tersebut kedudukannya terbalik, maka kedua elemen tersebut akan ditukar posisinya. Hal ini dilakukan berulang-ulang sampai semua elemen pada *array* telah terurut.

Proses yang terjadi pada algoritma *bubble sort* dapat dilihat pada gambar 3.



Gambar 3. Proses pengurutan menggunakan algoritma *bubble sort*
Sumber :
http://www.algolist.net/Algorithms/Sorting/Bubble_sort

5. Insertion Sort

Insertion sort merupakan salah satu jenis algoritma pengurutan yang simpel. Pembuatan array yang terurut dilakukan dengan cara mengambil elemen pada iterasi ke-n lalu meletakkannya pada tempat yang sesuai. Proses ini dilakukan satu demi satu sehingga tidak efisien untuk data yang besar [6].

Keuntungan menggunakan *insertion sort* antara lain mudah untuk diimplementasikan, efisien untuk data yang kecil, adaptif, dan stabil. Gambaran penggunaan dari algoritma ini dapat dilihat pada gambar 4.

5	2	7	3	10	1
2	**5**	7	3	10	1
2	**5**	**7**	3	10	1
2	**3**	**5**	**7**	10	1
2	**3**	**5**	**7**	**10**	1
1	**2**	**3**	**5**	**7**	**10**

Gambar 4. Proses pengurutan menggunakan algoritma *insertion sort*

Dari gambar 4 diatas. dapat dilihat elemen pada *array* masih berada dalam kondisi acak. Pada iterasi kedua, elemen kedua pada *array* dipilih lalu dibandingkan dengan *array* yang sudah di cetak tebal. Iterasi kemudian dilanjutkan dengan elemen ketiga hingga elemen terakhir. Tiap elemen pada iterasi yang bersangkutan akan dibandingkan dengan *array* yang dicetak tebal untuk kemudian ditentukan akan diletakkan pada posisi keberapa dari *array* tersebut.

6. Selection Sort

Selection sort merupakan salah satu algoritma pengurutan yang bekerja dengan cara mencari elemen dengan nilai terkecil pada sebuah *array* dan memindahkannya kedepan. Proses ini dilakukan berulang-ulang hingga terbentuk *array* yang terurut [7].

Array dibagi menjadi dua bagian yakni bagian yang terurut dan yang belum terurut. Iterasi dilakukan pada bagian *array* yang belum terurut dan memindahkan elemen terkecil pada *array* tersebut ke *array* yang telah terurut. Proses pengurutan *array* dengan menggunakan *selection sort* dapat dilihat pada gambar dibawah ini.

5	2	7	3	16	10	1
1	5	2	7	3	16	10
1	2	5	7	3	16	10
1	2	3	5	7	16	10
1	2	3	5	7	16	10
1	2	3	5	7	16	10
1	2	3	5	7	10	16

Gambar 4. Proses pengurutan menggunakan algoritma *selection sort*

Pada gambar 4, dapat dilihat *array* yang telah terurut menggunakan warna dasar hitam sedangkan *array* yang belum terurut menggunakan warna dasar putih. Iterasi dilakukan sebanyak n-1 kali dikarenakan elemen terakhir pada *array* yang belum terurut sudah pasti menjadi elemen terakhir pada *array* yang sudah terurut.

III. ANALISIS KOMPLEKSITAS WAKTU ALGORITMA PENGURUTAN

Kompleksitas algoritma terdiri dari dua jenis yakni kompleksitas waktu dan kompleksitas ruang. Menghitung kebutuhan waktu suatu algoritma adalah hal yang penting dikarenakan setiap komputer dengan arsitektur berbeda memiliki bahasa mesin yang berbeda pula. Hal ini menyebabkan waktu untuk setiap operasi yang sama

antara satu komputer dengan komputer yang lain dapat berbeda. Selain itu, compiler bahasa pemrograman yang digunakan menghasilkan bahasa mesin yang berbeda dapat menyebabkan waktu operasi antar compiler yang satu dengan yang lain tidak sama. Kompleksitas waktu T(n) diukur dari jumlah tahapan komputasi yang dibutuhkan untuk menjalankan suatu algoritma sebagai fungsi dari masukan n [13].

Kompleksitas waktu dibedakan menjadi tiga macam, yakni kompleksitas waktu terbaik (*best case*), kompleksitas waktu terburuk (*worst case*), dan kompleksitas waktu rata-rata (*average case*). Kompleksitas waktu asimptotik dinotasikan dengan big-O. Semakin besar nilai kompleksitas waktu asimptotiknya, maka semakin tidak efektif algoritma tersebut digunakan.

A. Quick Sort

Pada *quick sort*, sebuah elemen diambil untuk dijadikan *pivot* dan memindahkan elemen dengan nilai lebih kecil ke sebelah kiri *pivot*. Elemen dengan nilai yang lebih besar akan diletakkan pada sebelah kanan *pivot*.

Waktu yang dibutuhkan untuk menyusun kembali array adalah sebesar Cn dengan C adalah sebuah konstanta. *Array* dibagi dua berdasarkan letak *pivot* yakni berukuran k dan n-k. Kedua sub-array tersebut juga butuh untuk di urutkan. Dengan demikian didapat sebuah persamaan

$$T(n) = T(n-k) + T(k) + Cn \quad (1)$$

T(n) adalah waktu yang dibutuhkan algoritma ini untuk mengurutkan n buah elemen.

1. *Worst case analysis*

   Worst case terjadi apabila *pivot* yang dipilih ternyata merupakan elemen terakhir dari *array*. Dengan demikian didapat dua array berukuran 1 dan n-1. Oleh karena itu dari persamaan (1) didapat

   $$T(n) = T(n-1) + T(1) + Cn \quad (2)$$

   Dari persamaan (2) didapat rekurens

   $$T(n) = T(n-1) + T(1) + Cn = [T(n-2) + T(1) + C(n-1)] + T(1) + Cn \quad (3)$$

   Setelah dilakukan iterasi pada persamaan (2) hingga iterasi ke-i, maka didapat

   $$T(n) = T(n-i) + iT(1) + C\sum_{j=0}^{i-1}(n-j) \quad (4)$$

   Rekurensi ini hanya akan terjadi sampai i = n – 1. Oleh karena itu dengan mensubstitusi i = n – 1 ke persamaan (4) maka akan didapat

   $$T(n) = T(1) + (n-1)T(1) + C\sum_{j=0}^{n-2}(n-j) = nT(1) + C\left(n(n-2) - \frac{(n-2)(n-1)}{2}\right) \quad (5)$$

   Dari persamaan (5), dapat ditarik kesimpulan bahwa algoritma quick sort dalam kondisi worst case memiliki kompleksitas waktu asimptotik sebesar $O(n^2)$ [1].

2. *Best case analysis*

   Best case pada algoritma quick sort didapat apabila nilai pivot yang diambil tepat membagi array menjadi dua sub-array dengan jumlah elemen yang sama. Dengan kata lain didapat k = n / 2. dan n – k = n / 2. Dari persamaan (1) didapat rekurensi

   $$T(n) = 2T\left(\frac{n}{2}\right) + Cn = 2\left(2T\left(\frac{n}{4}\right) + \left(\frac{Cn}{2}\right)\right) + Cn \quad (6)$$

   Dengan menjabarkan rekurensi pada persamaan (6) hingga iterasi ke-k maka didapat

   $$T(n) = 2^kT\left(\frac{n}{2^k}\right) + kCn \quad (7)$$

   Iterasi ini akan berhenti sampai nilai k = log n. Dengan mensubstitusi k = log n ke persamaan (7) maka didapat

   $$T(n) = nT(1) + Cn\log n \quad (8)$$

   Dari persamaan (8) didapat nilai kompleksitas waktu asimptotik O(n log n) [1].

B. Merge Sort

Kompleksitas waktu asimptotik dari *merge sort* dapat dicari menggunakan beberapa cara. Diantaranya adalah dengan menggunakan pohon rekursif dan melihat sifat rekursifnya. Jumlah perbandingan yang dilakukan menggunakan *merge sort* adalah jumlah perbandingan di kiri, jumlah perbandingan di kanan, dan jumlah penggabungan.

$$T(n) = \frac{n}{2} + \frac{n}{2} + n \quad (9)$$

Dengan rekurens yang terjadi pada *merge sort* adalah

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad (10)$$

Nilai n > 1 dan T(1) = 0. Dari penjabaran persamaan (6),

telah diketahui persamaan rekurens (10) memiliki nilai kompleksitas waktu asimptotik O(n log n). Hal ini dapat dibuktikan dengan menelusuri rekurensi yang terjadi ataupun menggunakan pohon rekursif.

Dengan menelusuri persamaan rekursif yang terjadi, akan didapat

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

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

$$T\left(\frac{n}{n}\right) = \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} + 1$$

$$T\left(\frac{n}{n}\right) = \frac{T\left(\frac{n}{4}\right)}{\frac{n}{4}} + 1 + 1$$

....

$$T\left(\frac{n}{n}\right) = \frac{T\left(\frac{n}{n}\right)}{\frac{n}{n}} + 1 + 1 + \cdots + 1 = \log n \quad (11)$$

Dengan demikan didapat T(n) = n log n dan O(n log n). Kompleksitas algoritma ini berlaku untuk semua kasus, baik *best case*, *worst case*, maupun *average case* [11].

C. Heap Sort

Algoritma *heap sort* membutuhkan n langkah untuk membentuk *heap*. Selain itu, heap sort juga membutuhkan (n-1) log n langkah untuk mengeluarkan elemen dari *heap* dan menambahkannya pada *array* baru. Oleh karena itu dibutuhkan total langkah sebesar

$$T(n) = n + (n-1)\log n \approx n \log n \quad (12)$$

Dari sumber lain, dimisalkan T(n) adalah waktu yang dibutuhkan untuk menjalankan algoritma *heap sort* pada *array* dengan n elemen. Maka persamaan T(n) adalah

$$T(n) = T_{buildheap}(n) + \sum_{k=1}^{n-1} T_{heapify}(k) + \theta(n-1) \quad (13)$$

Proses *heapify* juga digunakan didalam *buildheap*. Oleh karena itu *heapify* akan dijabarkan terlebih dahulu

$$T_{heapify}(n) = \theta(1) + T_{heapify}(size\ of\ subtree) \quad (14)$$

Jika suatu heap A memiliki ukuran n, maka ukuran dari subtree heap tersebut kurang dari atau sama dengan 2n/3 [12]. Dengan demikian, nila nilai tersebut disubstitusikan pada persamaan (14), maka didapat

$$T_{heapify}(n) = \theta(1) + T_{heapify}\left(\frac{2n}{3}\right)$$

$$T_{heapify}(n) = \theta(\log n) \quad (15)$$

Kompleksitas waktu *buildheap* sebesar O(n log n). Jika persamaan (15) disubstitusikan pada persamaan (13) maka didapat

$$T(n) = T_{buildheap}(n) + \sum_{k=1}^{n-1} T_{heapify}(k) + \theta(n-1)$$

$$T(n) = \theta(n \log n) + \sum_{k=1}^{n-1} \log k + \theta(n-1)$$

$$T(n) = \theta(n \log n) \quad (16)$$

Didapat nilai kompleksitas waktu asimptotik Θ(n log n) = O(n log n). Kompleksitas waktu ini berlaku untuk semua kondisi baik *best case*, *worst case*, maupun *average case* [11].

D. Bubble Sort

Bubble sort menggunakan dua buah *loop* yakni *inner loop* dan *outer loop*. *Outer loop* akan melakukan iterasi sebanyak n-1 kali, sedangkan *inner loop* akan melakukan penelusuran dan pertukaran nilai yang urutannya tidak benar.

1. Best case analysis
   Kondisi ini tercapai apabila *array* sudah dalam keadaan terurut. Iterasi yang bekerja hanyalah *outer loop* sebanyak n-1 kali. Oleh karena itu nilai kompleksitas algoritma yang didapat adalah O(n).

2. Worst case analysis
   Kondisi ini tercapai apabila *array* dalam keadaan terurut namun terbalik. Oleh karena itu dibutuhkan pengulangan sebanyak n-1 dan pertukaran dalam *inner loop*. Nilai T(n) yang didapat adalah

$$T(n) = (n-1) + (n-2) + (n-3) + \cdots + 2 + 1$$

$$T(n) = \frac{n(n-1)}{2} \quad (17)$$

   Dari persamaan (17) didapat nilai kompleksitas algoritmanya sebesar $O(n^2)$.

E. Insertion Sort

Insertion sort menggunakan sebuah kunci untuk menentukan batasan antara elemen yang telah diurutkan dan yang belum terurut. Contoh penggunaan insertion sort dapat dilihat pada algoritma dibawah ini [9].

```
INSERTION-SORT(A)
    for j ← 2 to n
        do key ← A[ j ]
         Insert A[ j ] into the sorted sequence A[1 . . j -1]
           i ← j - 1
           while i > 0 and A[i] > key
                do A[i + 1] ← A[i]
                   i ← i – 1
           A[i + 1] ← key
```

Misalkan b adalah baris dari algoritma diatas. b1 akan dilakukan sebanyak n kali, b2, b4 dan b8 akan dilakukan sebanyak n-1 kali. b3 dapat diabaikan. b5 dilakukan sebanyak $\sum_{j=2}^{n} t_j$ kali. b6, dan b7 akan dilakukan sebanyak $\sum_{j=2}^{n} (t_j - 1)$ [10]. Nilai T(n) adalah

$$T(n) = b_1 n + b_2(n-1) + b_4(n-1) + b_5 \sum_{j=2}^{n} t_j + b_6 \sum_{j=2}^{n} (t_j - 1) + b_7 \sum_{j=2}^{n} (t_j - 1) + b_8(n-1) \quad (18)$$

1. *Best case analysis*

   Kondisi ini akan tercapai apabila array yang dimiliki telah berada dalam kondisi terurut. Dengan demikian nilai $t_j = 1$. Dengan substitusi ke persamaan (11) maka didapat

   $$T(n) = b_1 n + b_2(n-1) + b_4(n-1) + b_5(n-1) + b_8(n-1) \quad (19)$$

   Dari persamaan (19) dapat disederhanakan menjadi T(n) = an + b. Dengan demikian, kompleksitas waktu asimptotiknya adalah O(n).

2. *Worst case analysis*

   Kondisi ini tercapai apabila array telah terurut namun dalam posisi terbalik. Elemen yang dijadikan kunci harus dibandingkan dengan j-1 elemen ($t_j = j$).

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

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

   $$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2} \quad (20)$$

   Dengan menggunakan persamaan (20), maka persamaan (18) dapat dimodifikasi menjadi

   $$T(n) = b_1 n + b_2(n-1) + b_4(n-1) + b_5\left(\frac{n(n+1)}{2} - 1\right) + b_6\frac{n(n-1)}{2} + b_7\frac{n(n-1)}{2} + b_8(n-1)$$

   $$T(n) = an^2 + bn + c \quad (21)$$

   Dari persamaan (20) dapat disimpulkan bahwa nilai waktu asimptotiknya adalah $O(n^2)$. Nilai ini juga berlaku untuk *average case* [10].

F. Selection Sort

Untuk menentukan kompleksitas waktu dari algoritma *selection sort*, dilakukanlah penghitungan data yang diuji pada *array* dengan n elemen. Terdapat *outer loop* yang dieksekusi sebanyak n-1 kali. Setiap loop, satu elemen diambil dan diletakkan pada posisi yang sesuai.

Setelah iterasi dilakukan sebanyak k kali. *Array* yang sudah terurut memiliki k-1 elemen, sedangkan *array* yang belum terurut memiliki n-k+1 elemen. Pada *inner loop*, sisa n-k elemen akan dibandingkan dengan elemen pada index pertama *array* yang telah terurut. Terdapat dua elemen yang dibandingkan pada *inner loop*. Jadi, total terdapat 2(n-k) elemen yang dibandingkan pada iterasi *outer loop* ke-k [14].

Simpulannya, pada iterasi *outer loop* ke-k, terdapat 2(n-k) elemen yang diperiksa. Nilai k sendiri berada diantara 1 sampai n-1. Dengan demikian, bisa diambil simpulan nilai T(n) adalah sebagai berikut

$$T(n) = 2(n-1) + 2(n-2) + 2(n-3) + \cdots + 2(n-(n-2)) + 2(n-(n-1)) \quad (22)$$

Dari persamaan (22), didapatkan

$$T(n) = 2((n-1) + (n-2) + \cdots + 2 + 1)$$

$$T(n) = 2\left(\frac{n(n-1)}{2}\right)$$

$$T(n) = n^2 - n \quad (23)$$

Dari nilai T(n) yang didapat dari persamaan (23) bisa disimpulkan bahwa nilai kompleksitas waktu asimptotik dari algoritma *selection sort* adalah $O(n^2)$. Kompleksitas waktu ini berlaku untuk segala kondisi baik *best case*, *average case*, maupun *worst case*.

IV. SIMPULAN

Dari analisis yang dilakukan untuk menemukan besarnya kompleksitas waktu algoritma dan kompleksitas waktu asimptotik tiap algoritma yang telah dilakukan sebelumnya, dapat dibentuk sebuah tabel untuk menampilkan nilai kompleksitas waktu asimptotik untuk setiap algoritma.

Tabel 1. Nilai kompleksitas algoritma

Algoritma	***Best Case***	***Average Case***	***Worst Case***
Quick sort	O(n log n)	O(n log n)	$O(n^2)$
Merge sort	O(n log n)	O(n log n)	O(n log n)
Heap sort	O(n log n)	O(n log n)	O(n log n)
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$

Quick sort cocok digunakan untuk mengolah data yang tidak terlalu besar. Untuk mengolah data yang besar, disarankan untuk menggunakan *merge sort* atau *heap sort* dikarenakan nilai kompleksitas waktu asimptotiknya sebesar O(n log n). Bubble sort, insertion sort, dan selection sort tidak cocok digunakan baik untuk data yang besar maupun data kecil karena kompleksitas waktunya masuk kedalam kelompok waktu kuadratik ($O(n^2)$).

REFERENSI

[1] http://intranet.daiict.ac.in%2F~ajit_r%2FIT623%2Fquicksort_analysis.pdf&ei=BD2pUufXJciWrgeB_YDYCw&usg=AFQjCNGxxANuVYHipQkGx7sDcmBHsESUaw&bvm=bv.57967247,d.bmk Waktu akses : 12 Desember 2013, jam 11.44.

[2] http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/quickSort.htm. Waktu akses : 12 Desember 2013, jam 12.14.

[3] http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/mergeSort.htm. Waktu akses : 12 Desember 2013, jam 12.33.

[4] Knuth, Donald (1998). "Section 5.2.4: Sorting by Merging". Sorting and Searching. The Art of Computer Programming 3 (2nd ed.). Addison-Wesley. pp. 158–168. ISBN 0-201-89685-0.

[5] http://www.algolist.net/Algorithms/Sorting/Bubble_sort. Waktu akses : 12 Desember 2013, jam 12.57.

[6] http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Insertion_sort.html. Waktu akses : 13 Desember 2013, jam 11.08.

[7] http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/selectionSort.htm. Waktu akses : 13 Desember 2013, jam 12.47.

[8] Williams, J. W. J. (1964), "Algorithm 232 - Heapsort", Communications of the ACM 7 (6): 347–348

[9] http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/heapSort.htm. Waktu akses : 13 Desember, jam 13.13.

[10] http://www.cse.unr.edu/~bebis/CS477/Lect/InsertionSortBubbleSortSelectionSort.ppt. Waktu akses : 14 Desember 2013, Jam 00.32.

[11] http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/04MergeQuick.pdf. Waktu akses : 14 Desember 2013, Jam 09.50.

[12] http://cs.txstate.edu/~ch04/webtest/teaching/courses/5329/lectures/heap-comp.pdf. Waktu akses : 14 Desember 2013, Jam 10.10.

[13] http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2013-2014/Kompleksitas%20Algoritma.ppt. Waktu akses : 14 Desember 2013, Jam 12.05.

[14] http://www.csd.uwo.ca/courses/CS1037a/notes/topic13_AnalysisOfAlgs.pdf. Waktu akses : 14 Desember 2013, Jam 13.34.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 14 Desember 2013

Dibi Khairurrazi Budiarsyah
13509013