Kompleksitas Algoritma Quick Sort Guna Menemukan Efisiensi Waktu Dan Memori

Wahju Tjahjo Saputro*
Teknologi Informasi, Universitas Muhammadiyah Purworejo, Purworejo 54111, Indonesia

Abstrak

Banyak metode algoritma dalam menyelesaikan suatu permasalahan. Banyak penelitian telah melakukan berbagai performa setiap algoritma, hingga saat ini telah berkembang dengan pesat performa algoritma dalam bidang ilmu komputer. Dari sekian algoritma, salah satu metode algoritma yang sering digunakan yaitu Algoritma Qucik Sort.

Algoritma Quick Sort dalam melakukan proses pengurutan data menggunakan teknik divide and conquer. Dalam teknik tersebut dapat ditentukan pivot dari kiri, kanan atau tengah. Sehingga memiliki 3 performa yaitu: *best case* $O(n^2 \log n)$, *worst case* $O(n^2)$ dan *average case* $T_{avg}=O(n^2 \log n)$. Pada kasus tertentu *worst case* ternyata justru mampu bekerja dengan baik secara *in place* dan dalam virtual memori.

Kata kunci: *Quicksort, divide and conquer, performance, memory*

Abstract

Many algorithm methods slve a problem many research have done various performance of each algorithm, until now has growth rapidly the performance of algorithms in the field of computer science. One of the methods from all of algorithms is used such as the quick sort algorithm.

Qucik Sort Algorithm in the process of sorting data used divide and conquer techniques. Divide and conquer techniques can be determined right left or middle pivot. So they have 3 performance such as base case $O(n^2 \log n)$, worst case $O(n^2)$ and average case $Tavg = O(n^2 \log n)$. In certain, case the worst case was able to work well in place and in virtual memory

Keywords: *Quicksort, Divide and conquer, Performance, Memory*

1. PENDAHULUAN

Banyak penelitian yang membahas tentang berbagai macam algoritma, mengenai algortima mana yang lebih baik dalam menyelesaikan masalah tertentu. Untuk menjawab masalah diatas tentu ada beberapa hal yang harus diukur supaya bisa menilai apakah algoritma tersebut lebih baik atau tidak. Menurut (Syaukani, 2011) algoritma pada dasarnya merupakan alur pikiran dalam menyelesaikan pekerjaan yang dituangkan dalam bentuk tertulis. Maka dapat dikatakan algoritma adalah langkah yang diambil dalam menyelesaikan pekerjaan. Sedangkan (Purbasari, 2007) menyatakan prosedur komputasi yang mentransformasikan sejumlah input menjadi sejumlah output.

Bila dicoba mengeksekusi program dengan algoritma X pada komputer A dan algoritma Y pada komputer B. Hal ini tidak dapat dikatakan bahwa algoritma X lebih baik dibandingkan algoritma Y hanya karena algoritma X jauh lebih cepat dieksekusi pada komputer A. Komputer yang digunakan tidak semua memiliki spesifikasi yang sama. Sehingga waktu komputasi pun juga berbeda. Kompiler bahasa pemrograman pun juga berbeda dalam menghasilkan kode mesin. Sehingga pada kasus di atas, tidak dapat dikatakan bila ada sebuah program menggunakan algoritma X jauh lebih cepat dieksekusi karena spesifikasi komputer A lebih baik dibandingkan spesifikasi komputer B.

Oleh karena itu diperlukan model dalam pengukuran waktu atau ruang yang bebas dari pertimbangan arsitektur komputer, spesifikasi komputer dan kompiler bahasa pemrograman. Besaran yang dipakai untuk menerangkan model

pengukuran waktu atau ruang adalah kompleksitas algoritma.

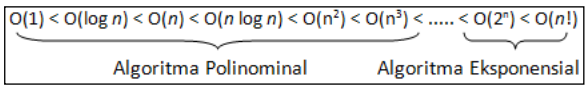
Ada dua macam kompleksitas algoritma yaitu: kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu disimbolkan dengan *T(n)* dan kompleksitas ruang *S(n)*. Kompleksitas waktu, *T(n)*, diukur dari jumlah tahapan komputasi yang dibutuhkan untuk menjalankkan algoritma sebagai fungsi dari ukuran masukan n. Sedangkan kompleksitas ruang, *S(n)*, diukur dari memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari ukuran masukan n (Purbasari, 2007). Dengan menggunakan besaran kompleksitas waktu atau ruang algoritma, dapat menentukan laju peningkatan waktu, dalam hal ini ruang, yang diperlukan algoritma dengan meningkatnya ukuran masukan n^1.

Dalam praktek, hanya dihitung jumlah operasi khas (tipikal) yang mendasari suatu algoritma, sehingga sebuah algoritma terdapat berbagai jenis operasi, diantaranya yaitu:

1. Operasi baca tulis,
2. Operasi aritmetika (+ , - , * , /)
3. Operasi pengisian nilai (*assignment*)
4. Operasi pengaksesan elemen larik
5. Operasi pemanggilan fungsi atau prosedur

Dalam struktur data terkadang tidak diperlukankan kompleksitas waktu yang detail dari sebuah algoritma, namun yang diperlukan adalah besaran kompleksitas waktu yang menghampiri kompleksitas waktu yang sebenarnya. Hal demikian disebut kompleksitas waktu asimptotik yang dinotasikan dengan "*O*" (Big O) (Cormane & dkk, 2001) (Syaukani M. , 2011).

Kompleksitas waktu asimptotik, diperoleh dengan mengambil term yang memberikan kompleksitas waktu terbesar. Misalkan $T(n) = 3n^3 + 2n^2 + n + 1$. Maka kompleksitas waktu asimptotiknya adalah *O(n3)*. Karena n^3 yang memberikan kompleksitas waktu terbesar. Maka tidak perlu menambahkan pengali dari term pada n^3. Spektrum kompleksitas waktu algoritma ditunjukkan pada Gambar 1, dimana sisi kiri merupakan kelompok algoritma polinominal dan sisi kanan merupakan kelompok algoritma eksponensial (Purbasari, 2007).



Gambar 1. Spektrum kompleksitas waktu algoritma (Cormane & dkk, 2001)

Sedangkan pada Tabel 1 menunjukkan pengelompokan algoritma berdasarkan notasi Big-O.

Tabel 1. Pengelompokan algoritma berdasarkan notasi Big-O (Cormane & dkk, 2001)

No	Kelompok	Nama
1	O(1)	Konstan
2	O(log *n*)	Logaritmik
3	O(*n*)	Lanjar
4	O(*n* log *n*)	n log n
5	O(n^2)	Kuadratik
6	O(n^3)	Kubik
7	O(2^n)	Eksponensial
8	O(*n*!)	Faktorial

2. PEMBAHASAN

2.1. Algoritma Quick Sort

Quick Sort disebut juga sebagai *Partition Exchange Sort*. Disebut *Quick Sort* karena terbukti mempunyai kemampuan "*average behaviour*" yang terbaik diantara metode mengurutan algoritma yang lain (Syaukani M. , 2011). Disebut *Partition Exchange Sort* karena proses pengurutan menggunakan partisi dan pengurutan dilakukan pada setiap partisi (Rinaldi, 2011). Selain itu menurut (Purbasari, 2007) algoritma Quick Sort merupakan algoritma pengurutan terbaik dengan metode Divide-Conquer. Tahapan dalam melakukan partisi pada Algoritma *Quick Sort* ada lima yaitu (Purbasari, 2007):

1. Pilih x ∈ {A1, A2, ... , An} sebagai elemen *pivot*.
2. Lakukan *scaning* tabel dari kiri ke kanan sampai ditemukan Ap ≥ x.
3. Lakukan *scaning* pada tabel dari kanan ke kiri sampai ditemukan Aq ≤ x.
4. Lakukan pertukaran Ap ↔ Aq.
5. Ulangi langkah ke-2 dari posisi p+1 dan langkah ke-3 dari posisi q-1, sampai kedua *scaning* bertemu di tengah tabel.

Dalam Algoritma *Qucik Sort* pemilihan *pivot* sangat menentukan, apakah *Quick Sort* memberikan kinerja terbaik atau terburuk.

Berikut ini cara memilih *pivot* dalam Algoritma *Quick Sort*.

1. Pivot dapat diambil dari elemen pertama, elemen terakhir, atau elemen tengah tabel. Cara ini hanya optimal bila elemen tabel tersusun secara acak, tetapi tidak optimal bila elemen tabel semula sudah terurut.
2. Pivot dipilih secara acak dari salah satu elemen tabel. Cara ini baik, tetapi mahal, sebab memerlukan biaya untuk pembangkitan prosedur acak. Selain itu, ita tidak mengurangi kompleksitas waktu algoritma.
3. Pivot ditentukan dari elemen median tabel. Cara ini paling baik, karena hasil partisi menghasilkan dua bagian tabel yang berukuran sama, dimana setiap bagian tabel berukuran n/2. Cara ini memberikan kompleksitas waktu yang minimum. Sebab mencari median dari elemen tabel yang belum terurut adalah persoalan tersendiri.

Gambar 2 menunjukkan *pseudo-code* Algoritma *Quick Sort*, dan gambar 3 menunjukkan *psuedo-code Quick Sort* dalam melakukan partisi.

```
QUICKSORT (A, p, r)
1. if p < r then
2.     q <- PARTITION(A, p, r)
3.     QICKSORT(A, P, q-1)
4.     QUCKSORT(A, q+1, r)
```

Gambar 2. Algoritma Quick Sort (Purbasari, 2007)

Pada gambar 2 dan gambar 3, guna mengurutkan semua data pada array A dengan memangil fungsi QUICKSORT(). Sedangkan kunci utama dari algoritma ini adalah prosedur PARTISI yang digunakan untuk mengurutkan ulang elemen pada array sesuai urutan.

```
PARTITION(A, p, r)
x <- A(r)
i <- p-1
for j <- p to r-1 do
    if A[j] <= x then
        i <- i+1
        exchange A[i] <-> A[j]
        exchange A[i+1] <-> A[r]
return i+1
```

Gambar 3. Pseudo-code melakukan partisi (Purbasari, 2007)

Dibawah ini akan ditunjukkan kinerja Algoritma *Quick Sort* dalam melakukan partisi dan mengurutkan data dengan menggunakan data pada Gambar 4. Misal data yang diurutkan susunan awal tampak pada Gambar 4.

A1	A2	A3	A4	A5	A6	A7
11	**4**	**55**	**15**	**3**	**5**	**31**

Gambar 4. Struktur data awal

Seperti langkah (1) menentukan nilai *pivot* sebelum mempartisi terhadap tabel data yang ada pada gambar 4, maka dari x ∈ {a1, a2, a3, a4, a5, a6, a7} diperoleh A4 sebagai *pivot* tengah. Sehingga hasilnya seperti Gambar 5.

A1	A2	A3	A4	A5	A6	A7
11 p	**4**	**55**	**15** pivot	**3**	**5**	**31** q

Gambar 5. Struktur data awal

Seperti langkah (2) dan (3) melakukan *scaning* tabel data dari kiri dengan Ap ≥ x dan *scaning* tabel data dari kanan dengan Aq ≤ x. Dimana Ap ≥ x menyatakan lebih besar dari *pivot* dan Aq ≤ x lebih kecil dari *pivot*. Hasilnya ditunjukkan pada Gambar 6.

A1	A2	A3	A4	A5	A6	A7
11 p	**4**	**55**	**15** pivot	**3**	**5**	**31** q

Gambar 6. Hasil setelah $A_p \geq x$ dan $A_q \leq x$

Seperti langkah (4), pertukaran dilakukan antara data yang ditunjuk p dan data yang ditunjuk oleh q, sehingga p berada pada A3 dan q berada pada A6, ditunjukkan Gambar 7.

A1	A2	A3	A4	A5	A6	A7
11	**4**	**55** p	**15** pivot	**3**	**5** q	**31**

Gambar 7. Hasil setelah $A_3 \geq x$ dan $A_6 \leq x$

Dari hasil pertukaran Ap ↔ Aq maka diperoleh seperti Gambar 8, dimana A3 = 5 dan A6 = 55. Artinya angka 5 telah berada disebelah kiri pivot dan angka 55 telah berada disebelah kanan pivot.

A1	A2	A3	A4	A5	A6	A7
11	**4**	**5** p	**15** pivot	**3**	**55** q	**31**

Gambar 8. Hasil setelah $A_3 \leftrightarrow A_6$

Berikutnya mengikuti langkah (5) dimana p+1 dan q-1 artinya p berada di A4 dan q berada di A5, selanjutnya masing-masing p dan q melakukan *scaning*. Proses pertukaran juga dilakukan terhadap Ap ↔ Aq untuk A4 ↔ A5, maka hasil pertukaran tampak pada Gambar 9.

A1	A2	A3	A4	A5	A6	A7
11	**4**	**5** p	**15** pivot	**3**	**55** q	**31**
11	**4**	**5**	**15** p	**3** q	**55**	**31**
11	**4**	**5**	**3** p	**15** q	**55**	**31**

Gambar 9. Hasil setelah Ap ↔ Aq kedua

Proses berhenti ketika q ≥ p dan selanjutnya menghasilkan dua partisi seperti Gambar 10. Dimana partisi pertama < 15 dan partisi kedua ≥ 15. Langkah berikutnya setiap partisi dapat ditentukan *pivot* baru. Bila dilanjutkan maka proses rekursif masing-masing terjadi pada partisi pertama dan kedua.

A1	A2	A3	A4
11	**4**	**5**	**3**

A5	A6	A7
15	**55**	**31**

Gambar 10. Hasil setelah dilakukan partisi

Langkah yang sama juga dilakukan pada partisi kedua, sehingga hasilnya tampak pada Gambar 11. Kemudian hasil akhir tabel data yang telah urut tampak pada Gambar 12.

A5	A6	A7
15 p	**55** pivot	**31** q
15	**55** p	**33** q
15	**33** p	**55** q

Gambar 11. Hasil setelah dilakukan partisi

A1	A2	A3	A4	A5	A6	A7
3	**4**	**5**	**11**	**15**	**33**	**55**

Gambar 12. Hasil akhir data telah urit naik

2.2. Implementasi Algoritma Quick Sort

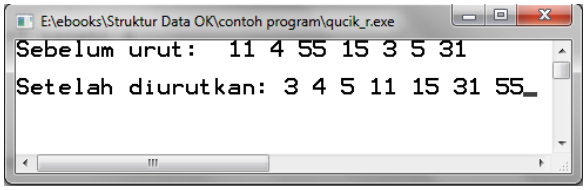
Implementasi Algoritma *Quick Sort* dilakukan menggunaan Borland C++ 5.02 yang ditunjukkan pada Gambar 13.

Dari Gambar 13 dijelaskan sebagai berikut: baris 1 – 5 digunakan untuk mendeklarasikan *file header* yang dipanggil, mendeklarasikan jumlah array yang akan diurutkan. Baris 6 digunakan untuk mendeklarasikan variabel l dan r bertipe integer. Baris 7 – 17 merupakan program utama dari *Quick Sort*. Berisi dua bagian perulangan. Perulangan baris 9 – 13 digunakan untuk menampilkan data awal, sedangkan baris 14 – 16 digunakan untuk menampilkan data yang telah diurutkan.

Baris 18 – 32 merupakan fungsi buatan sendiri yang digunakan untuk menentukan pivot dan melakukan pertukaran data. Baris 20 digunakan untuk mendeklarasikan i dan j sebagai penunjuk. Baris 21 menentukan pivot dari kiri. Baris 23 – 24 digunakan untuk menggerakan penunjuk ke arah kanan dan baris 25 – 26 digunakan untuk menggerakkan penunjuk ke kiri. Baris 27 – 32 digunakan untuk melakukan pertukaran data. Hasil implementasi ditunjukkan Gambar 14.

```
01 #include <stdio.h>
02 #include <conio.h>
03 #include <iostream.h>
04 #define n 7
05 int a[n]={11,4,55,15,3,5,31};
06 void sort(int l, int r);
07 void main() {
08  int i;
09  printf("Sebelum urut: ");
10  for(i=0; i<=n-1; i++)
11    printf(" %d", a[i]);
12    printf("\n");
13    sort(0,n-1);
14    printf("\nSetelah diurutkan:");
15    for(i=0; i<=n-1; i++)
16      printf(" %d",a[i]);
17  getch(); }
18  void sort(int kiri,int kanan) {
19    int i,j,pivot,w;
20    i=kiri; j=kanan;
21    pivot=a[i];
22    while(i <= j) {
23      while(a[i] < pivot)
24        i++;
25        while(pivot < a[j])
26          j--;
27          if(i <= j) {
28            w=a[i];
29            a[i]=a[j]; a[j]=w;
30            i++; j++; } }
31    if(kiri < j) sort(kiri,j);
32    if(i < kanan) sort(i,kanan);  }
```

Gambar 13. Implementasi program untuk Algoritma Quick Sort



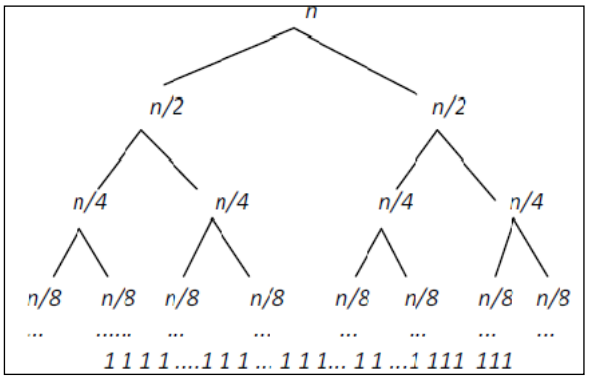
Gambar 14. Hasil implementasi program Algoritma Quick Sort

2.3. Kompleksitas Algoritma Quick Sort

Terdapat tiga kemungkinan kasus dari performa algoritma *Quick Sort* yaitu terbaik (*best case*), terburuk (*worst case*) dan rata-rata (*average case*) yang dijelaskan berikut ini (Purbasari, 2007) (Cormane & dkk, 2001) (Syaukani M. , 2011) (Lafore, 1999).

2.3.1. Kasus Terbaik (*Best Case*)

Kasus terbaik terjadi bila *pivot* adalah elemen median sedemikian sehingga kedua bagian tabel berukuran relatif sama setiap kali dilakukan partisi. Menentukan median tabel adalah persoalan tersendiri, sebab harus ditentukan median dari tabel yang belum terurut. Pohon pada gambar 15 menggambarkan bagian tabel kiri dan bagian tabel kanan setiap kali partisi sampai menghasilkan tabel terurut.



Gambar 15. Pohon untuk mempartisi dengan n/2 (Cormane & dkk, 2001)

Kompleksitas waktu pengurutan dihitung dari jumlah perbandingan elemen-elemen tabel berikut Tmin(n) = waktu partisi + waktu pemanggilan *rekurens* yaitu *Quick Sort* untuk dua bagian tabel hasil partisi. Kompleksitas prosedur partisi adalah t(n) = cn = O(*n*), sehingga kompleksitas algoritma *Quick Sort* dalam bentuk relasi rekurens, menjadi seperti Gambar 16 sedangkan penyelesaiannya ditunjukkan pada Gambar 17.

$$T(n) = \begin{cases} a & ,n=1 \\ 2T\left(\frac{n}{2}\right) + cn & ,n>1 \end{cases}$$

Gambar 16. Quick Sort dalam bentuk rekurens (Rinaldi, 2011)

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2(2T(n/4) + cn/2) + cn = 4(T(n/4) + 2cn \\ &= 4(2(T(n/8) + cn/4) + 2cn = 8T(n/8) + 3cn \\ &= .. \\ &= 2^k(T(n/2^k) + kcn \end{aligned}$$

Gambar 17. Penyelesaian dari relasi rekurens (Rinaldi, 2011)

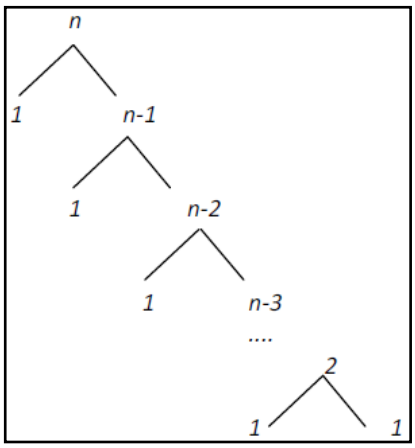
Selanjutnya persamaan terakhir pada gambar 17 dapat diselesaikan, karena basis rekursif adalah ketika ukuran tabel=1, dimana $n/2^k = 1 \rightarrow k = {}^2\log n$, sehingga diselesaikan seperti Gambar 18.

$$\begin{aligned} T(n) &= nT(1) + cn\ {}^2\log n \\ &= na + c\,n\ {}^2\log n \\ &= O(n\ {}^2log\ n) \end{aligned}$$

Gambar 18. Penyelesaian dari persamaan $2^k(T(n/2^k)+kcn$ (Rinaldi, 2011)

2.3.2. Kasus Terburuk (*Worst Case*)

Kasus ini terjadi bila pada setiap partisi *pivot* selalu menunjukkan elemen maksimum atau elemen minimum tabel. Hal ini menyebabkan pembagian menghasilkan sisi tabel kiri atau kanan berukuran satu elemen dan sisi tabel kanan atau kiri mempunyai ukuran n – 1 elemen. Keadaan kedua sisi tabel ini digambarkan dengan pohon, seperti Gambar 19.



Gambar 19. Pohon yang menunjukkan n-1 (Cormane & dkk, 2001)

Dari Gambar 19, kompleksitas waktu mengurutan mempunyai persamaan dan dapat diselesaikan seperti Gambar 20.

$$T(n) = \begin{cases} a & ,n=1 \\ T(n-1) + cn & ,n>1 \end{cases}$$

$$\begin{aligned} T(n) &= cn + T(n-1) \\ &= bn + \{ b.(n-1) + T(n-2) \} \\ &= bn + b(n-1) + \{b(n-2) + T(n-3)\} \\ &= ... \\ &= b(n+(n-1)+(n-2)..+2) + a \\ &= b\{(n-1)(n+2)/2\} + a \\ &= bn^2/2 + bn/2 + ((a-b) \\ &= O(n^2) \end{aligned}$$

Gambar 20. Kompleksitas waktu pengurutan pada worst case (Cormane & dkk, 2001)

2.3.3. Rata-rata (*avarege case*)

Kasus rata-rata terjadi bila *pivot* dipilih secara acak dari elemen tabel dan peluang setiap

elemen yang dipilih menjadi *pivot* adalah sama. Kompleksitas waktunya yaitu $T_{avg}(n) = O(n\ ^2\log n)$ (Cormane & dkk, 2001).

3. KESIMPULAN

Algoritma mampu membantu dalam memahami *scalability*, performa yang dihasilkan algoritma dapar membedakan yang *feasible* dan *impossible*. Performa adalah *currency* dari komputasi selain itu performa program dapat digeneralisasi pada sumber daya komputasi yang lain, kecepatan dalam *running time* merupakan hal penting dalam algoritma.

Performa algoritma dapat diketahui sebagai algoritma yang handal dalam melakukan pengurutan data dari besarnya asimptotik yang diperlukan bila diberikan n buah masukan. Dimana kompleksitas algoritma ini memiliki tiga kasus yaitu $O(n^2 \log n)$ sebagai *best case*, $O(n^2)$ sebagai *worst case* dan $T_{avg}=O(n^2 \log n)$ untuk kasus *average case*. Kompleksitas tersebut dipengaruhi karena pemilihan *pivot*, sehingga pemilihan *pivot* perlu dipertimbangkan. Namun pada beberapa kasus nyata *pivot* diambil dari elemen tengah tabel yang akan diurutkan.

Meskipun Quick Sort memiliki *worst case running time* $O(n^2)$, sering digunakan sebagai pilihan dalam mengurutkan data karena sangat efisien pada kasus *average* yang memiliki $T_{avg}=O(n\ ^2\log n)$. *Quick Sort* juga efisien dalam mengurutkan data secara *in place* dan mampu bekerja dengan baik pada lingkungan virtual memori.

DAFTAR PUSTAKA

Cormane, T., & dkk. (2001). *Introduction to Algorithm.* Massachusetts, England: The MIT Cambridge Press

Lafore, R. (1999). *Teach Your Self Data Structure and Algorithmin 24 Hours.* Indianapolis, USA: Sams Publishing

Purbasari, I. (2007). *Desain dan Analisis Algoritma.* Yogyakarta, DIY, Indonesia: Graha Ilmu

Rinaldi, M. (2011). *Diktat Kuliah Analisis Algoritma*. Bandung: STT-Telkom

Syaukani, M. (2011). *Algoritma dengan C C++ dan JAVA.* Bogor: Mitra Wacana Media

Syaukani, M. (2011). *Struktur Data dengan C dan C++*. Bogor, Jawa Barat, Indonesia: Mitra Wacana Indonesia