Analisis Perbandingan Kompleksitas Algoritma Pengurutan Nilai (*Sorting*)

Panny Agustia Rahayuningsih
Manajemen Informatika, AMIK BSI Pontianak
panny.par@bsi.ac.id

Abstract *- The role of algorithms in software or programming is so important, so it is necessary to understand the basic concept of the algorithm. So a lot of logic programming that has been created, to the general case and also special. sequencing data can be used in sorting algorithms value (sorting) namely, selection sort (sorting by selecting), insertion sort (sorting by insertion), quick sort (fast sorting), (sorting the pile), shell sort (sorting shells, and bubble sort (sorting bubble). This value sorting algorithms are algorithms for sorting processing using integer data type. Each of these types of algorithms have different levels of effectiveness. the effectiveness of an algorithm can be measured by how much time and space (space / memory) required to run the algorithm. Effective algorithms are algorithms that can minimize the need for time and space. The less space required to run an algorithm, the more effective the algorithm. And the less time needed to run an algorithm, the more effective the algorithm. Yet needs time and space of an algorithm depends on the amount of data processed and algorithms used. the complexity of the space will not be discussed at this writing. this paper will only discuss and analyze the complexity of time for each types of algorithms. The algorithm is written in this paper are algorithms that are implemented in Visual Basic 6.0 programming language.*
Keywords: Algorithm Complexity, Sorting, Quick Sort, Shell Sort, Insertion Sort, Selection Sort, Bubble Sort

Abstrak - Peran algoritma dalam perangkat lunak atau pemrograman sangat penting, sehingga perlu untuk memahami konsep dasar dari algoritma. Jadi banyak logika pemrograman yang telah dibuat, untuk kasus umum dan juga khusus. Data sequencing dapat digunakan dalam memilah nilai algoritma (pengurutan) yaitu, selection sort (pengurutan dengan memilih), insertion sort (pengurutan oleh penyisipan), semacam cepat (menyortir cepat), (menyortir tumpukan), shell sort (pengurutan kerang, dan bubble sort (pengurutan bubble). nilai ini algoritma pengurutan algoritma untuk menyortir pengolahan menggunakan tipe data integer. Masing-masing jenis algoritma memiliki berbagai tingkat efektivitas. efektivitas algoritma dapat diukur dengan berapa banyak waktu dan ruang (space / algoritma memori) yang diperlukan untuk menjalankan algoritma. efektif algoritma yang dapat meminimalkan kebutuhan ruang dan waktu. semakin sedikit ruang yang dibutuhkan untuk menjalankan sebuah algoritma, yang lebih efektif algoritma. dan sedikit waktu yang dibutuhkan untuk menjalankan sebuah algoritma, yang lebih efektif algoritma. Namun membutuhkan waktu dan ruang suatu algoritma bergantung pada jumlah data yang diolah dan algoritma yang digunakan. kompleksitas ruang tidak akan dibahas pada tulisan ini. tulisan ini hanya akan membahas dan menganalisis kompleksitas waktu untuk setiap jenis algoritma. Algoritma ini ditulis dalam makalah ini adalah algoritma yang diimplementasikan dalam Basic bahasa pemrograman Visual 6.0.
Kata Kunci: Kompleksitas Algoritma, Sorting, Quick Sort, Shell Sort, Insertion Sort, Selection Sort, Bubble Sort

A. PENDAHULUAN

Dalam beberapa konteks pemograman, algortima merupakan spesefikasi urutan langkah untuk melakukan pekerjaan tertentu. untuk menyelesaikan suatu masalah tidak cukup dengan menemukan algoritma yang hasil penyelesaian masalahnya terbukti benar. Artinya algoritma akan memberikan keluaran yang dikehendaki dari sejumlah masukan yang diberikan. Tidak peduli sebagus apapun algoritma, kalau memberikan keluaran yang salah, pastilah algoritma tersebut bukan algoritma yang baik. Maka dari itu, sebuah algortima yang baik akan menggunakan algoritma yang efektif, efisien , tepat sasaran dan terstruktur. Untuk memilih algoritma yang kualifikasinya seperti itu dapat diukur dari waktu eksekusi algoritma dan kebutuhan ruang memori. Algoritma yang efisien adalah algoritma yang meminimukan kebutuhan waktu dan ruang. Digunakan untuk menjelaskan model pengukuran waktu dan ruang ini adalah kompleksitas algoritma. Namun, kebutuhan waktu dan ruang dari suatu algoritma bergantung pada jumlah data yang diproseskan dan algoritma yang digunakan. Kompleksitas Waktu, T(n), adalah jumlah operasi yang dilakukan untuk melaksanakan algoritma sebagai fungsi dari ukuran masukan n. Maka, dalam mengukur kompleksitas waktu dihitunglah banyaknya operasi yang dilakukan oleh algoritma. Kompleksitas waktu untuk algoritma-algoritma yang dibahas akan dinyatakan dengan notasi O besar (Big-O notation). Definisi dari notasi O besar adalah, jika sebuah algoritma mempunyai waktu

asimptotik O(f(n)), maka jika n dibuat semakin besar, waktu yang dibutuhkan tidak akan pernah melebihi suatu konstanta C dikali dengan f(n). Jadi f(n) adalah adalah batas atas (upper bound) dari T(n) untuk n yang besar. O(n) dihitung berdasarkan banyaknya jumlah operasi perbandingan yang dilakukan dalam algoritma tersebut.

Salah satu algoritma dasar yang sering dipakai untuk menyelesaikan masalah adalah algoritma pengurutan atau *sorting algorithm.* Pengurutan data atau sorting merupakan salah satu jenis operasi penting dalam pengolahan data. Pengurutan data sangat penting digunakan, sehinga sampai saat ini telah banyak metode-metode pengurutan data dan mungkin akan tetap bermunculan metode – metode yang baru. ada banyak metode pengurutan data antara lain : *bubble sort, bi-directional bubble sort, selection sort, shaker sort, insertion sort, inplace merge sort, double storage merge sort, comb sort 11, shell sort, heap sort*, *exchange sort*, *merge sort, quick sort, quick sort with bubblesort, enhange quick sort, fast quick sort, radix sort, swap sort*, dan lain sebagainya. Untuk membatasi luasnya pembahasan, maka dalam karya ilmiah ini hanya akan membahas 5 metode, yaitu *: quick sort, shell sort, insertion sort, selection sort dan bubble sort.* Pembahasan karya ilmiah ini akan difokuskan untuk menganalisis seberapa efisien suatu algoritma dengan melakukan pengujian waktu eksekusi dari ke 5 algoritma tersebut. Dengan sekali inputan data berupa bilangan bulat (*integer*) dalam *list* atau *array* secara acak (*random).*

Dari ke 5 algoritma tersebut pasti mempunyai kecepatan waktu yang berbeda-beda, ada yang lebih cepat dan lebih lama dalam melakukan pengurutan. Oleh karena itu, untuk mengetahui kecepatan waktu dari setiap algoritma diperlukan suatu bahasa pemrograman sebagai pendukung untuk melakukan perhitungan kecepatan algoritma. Bahasa pemrograman yang digunakan adalah *visual basic 6.0*. alasan menggunakan *visual basic 6.0* karena bahasa pemograman tingkat tinggi yang sederhana dan mudah dipelajari. mampu menciptakan suatu program dengan produktifitas, kualitas, pengembangan perangkat lunak, kecepatan compiler, serta pola desain yang menarik.

B. TINJAUAN PUSTAKA

1. Algoritma

Menurut Saputra, dkk (2010:1) menjelaskan bahwa algoritma adalah deretan instruksi yang jelas untuk memecakan masalah, yaitu untuk memperoleh keluaran yang diinginkan dari suatu masukan.

Ada 3 definisi tentang algoritma yang dijelaskan oleh Suarga (2012:1), diantaranya :

a) Teknik penyusunan langkah-langkah penyelesaian masalah dalam bentuk kalimat dengan jumlah kata terbatas tetapi tersusun secara logis dan sistematis.
b) Suatu prosedur yang jelas untuk menyelesaikan suatu persoalan dengan menggunakan langkah-langkah tertentu dan terbatas jumlahnya
c) Susunan langkah yang pasti, yang bila diikuti maka akan mentransformasi data input menjadi output yang beruapa informasi.

Algoritma merupakan suatu prosedur untuk menyelesaikan suatu masalah yang tersusun secara logis dan sistematis serta akan memperoleh data masukan menjadi keluaran yang diinginkan berupa informasi.

2. Pengurutan Data (*Sorting*)

Menurut Yahya (2014:135) Sorting adalah proses pengurutan data yang sebelumnya disusun secara acak atau tidak teratur menjadi urut dan teratur menurut suatu aturan tertentu. Biasanya pengurutan terbagi menjadi dua yaitu *Ascending* (pengurutan dari karakter/angka kecil ke karakter/angka besar dan Descending (pengurutan dari karakter/angka besar ke karakter/angka kecil).

Menurut Saputra, dkk (2010:1) juga menjelaskan bahwa algoritma sorting didefinisikan sebagai algoritma pengurutan sejumlah data berdasarkan nilai kunci tertentu. Pengurutan dapat dilakukan dari nilai terkecil ke nilai terbesar (*ascending*) atau sebaliknya (*descending*).

Pengurutan data (*sorting)* adalah suatu proses pengurutan data yang tersusun secara acak pada suatu pola tertentu, sehingga tersusun secara teratur menurut aturan tertentu. pengurutan ini dapat dilakukan dengan cara *Ascending* dan *descending* serta digunakan juga untuk mengurutkan data yang bertipe angka atau karakter.

3. Kompleksitas Waktu

Menurut Traju (2010:1) Kompleksitas waktu, T(n), adalah jumlah operasi yang dilakukan untuk melaksanakan algoritma sebagai fungsi dari ukuran masukan n. Maka, dalam mengukur kompleksitas waktu dihitunglah banyaknya operasi yang dilakukan oleh algoritma. Pada algoritma pengurutan, terutama pada pengurutan dengan perbandingan, operasi dasar adalah operasi-operasi perbandingan elemen-elemen suatu larik dan operasi pertukaran elemen. Kedua hal itu dihitung secara terpisah, karena jumlah

keduanya tidaklah sama. Biasanya kompleksitas algoritma dinyatakan secara asimptotik dengan notasi big-O. Jika kompleksitas waktu untuk menjalankan suatu algoritma dinyatakan dengan T(n), dan memenuhi

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

untuk n ≥ n0, maka kompleksitas dapat dinyatakan dengan

$$T(n) = O(f(n))$$

Menurut Tjaru (2010:2) menyatakan bahwa terdapat 2 jenis penggunaan notasi Big O, yaitu:

a) *Infinite asymptotics*
b) *Infinitesimal asymptotics*

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

$$T(n) = 2n2 - 2n + 2$$

Untuk n yang besar, pertumbuhan T(n) akan sebanding dengan n2 dan dengan mengabaikan suku yang tidak mendominasi kita, maka kita tuliskan

$$T(n) = O(n2)$$

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

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

Kesalahannya memiliki selisih

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

4. *Quick Sort*

Menurut Saputra, dkk (2010:1) menjelaskan bahwa *Quick Sort* adalah sebuah algoritma sorting dari model *Divide and Conquer* yaitu dengan cara mereduksi tahap demi tahap sehingga menjadi 2 bagian yang lebih kecil.

Quick Sort merupakan algoritma yang sangat cepat dibandingkan dengan algirtma *sorting* lainnya, karena algoritma *quick sort* ini melakukan *sorting* dengan membagi masalah menjadi sub masalah dan sub masalah dibagi lagi menjadi sub-sub masalah sehingga *sorting* tersebut menjadi lebih cepat walaupun memakan ruang memori yang besar.

5. *Shell Sort*

Menurut Atrinawati (2007:6) menjelaskan *Shell Sort* adalah algoritma dengan kompleksitas algoritma $O(n^2)$ dan yang paling efisien dibandingkan algortima-algoritma lain dengan kompleksitas algoritma yang sama. Algortima *shell sort* lima kali lebih cepat dibandingkan algoritma pengurutan gelembung dan dua kali lebih cepat dibandingkan algoritma pengurutan dengan penyisipan.

Algortima *shell sort* merupakan metode pengurutan data yang lebih efisien dan memiliki kompleksitas algoritma yang sama dengan algoritma yang lain. Cara kerja *shell sort* yaitu dengan cara membandingkan suatu data lain yang memiliki jarak tertentu sehingga membentuk sebuah sub-list.

6. *Insertion Sort*

Insertion Sort adalah sebuah algortima sederhana yang cukup efisien untuk mengurutkan sebuah list yang hampir terurut. Algoritma ini juga bisa digunakan sebagai bagian algoritma yang lebih canggih (Traju, 2010:3). Cara kerja algoritma ini adalah dengan mengambil elemen list satu-per-satu dan memasukkannya di posisi yang benar seperti namanya. Pada array, list yang baru dan elemen sisanya dapat berbagi tempat di array, meskipun cukup rumit. Untuk menghemat memori, implementasinya menggunakan pengurutan di tempat yang membandingkan elemen saat itu dengan elemen sebelumnya yang sudah diurut, lalu menukarnya terus sampai posisinya tepat. Hal ini terus dilakukan sampai tidak ada elemen tersisa di input (Traju, 2010:3).

Metode pengurutan *insertion sort* merupakan pengurutan data yang membandingkan dengan dua elemen data pertama, kemudian membandingkan elemen-elemen data yang sudah diurutkan, kemudian perbandingan tersebut akan terus diulang hingga tidak ada elemen data yang tersisa.

7. *Selection Sort*

Menurut Yahya (2014:136) menjelaskan bahwa *selection sort* adalah suatu metode pengurutan yang membandingkan elemen yang sekarang dengan elemen berikutnya sampai ke elemen yang terakhir. Jika ditemukan elemen lain yang lebih kecil dari elemen sekarang maka dicatat posisinya dan langsung ditukar.

Metode *selection sort* adalah melakukan pemilihan dari suatu nilai yang terkecil dan kemudian menukarnya dengan elemen paling awal, lalu membandingkan dengan elemen yang sekarang dengan elemen berikutnya sampai dengan elemen terakhir, perbandingan dilakukan terus sampai tidak ada lagi pertukaran data.

8. *Bubble Sort*

Menurut Yahya (2014:136) menjelaskan *bubble sort* adalah suatu metode pengurutan yang membandingkan elemen yang sekarang dengan elemen berikunya, jika elemen

sekarang > elemen berikutnya maka posisinya ditukar, kalau tidak, tidak perlu ditukar, misalnya untuk n = 7 maka akan dilakukan (n – 1) = 6 tahap (mulai dari 0 sampai dengan n - 2).

Algoritma *bubble sort* ini melakukan perbandingan antara setiap elemen, kemudian melakukan penukaran jika terdapat elemen yang tidak sesuai urutannya atau salah. Perbandingan akan terus dilakukan sehingga tidak ada lagi pertukaran data.

9. Bahasa Pemrograman *Visual Basic* 6.0

Visual Basic 6.0 memiliki bahasa pemograman BASIC (*Beginners All-purpose Symbolic Instruction Code*) yang merupakan bahasa pemograman tingkat tinggi yang sederhana dan mudah dipelajari, sehingga memungkinkan pengguna bisa membuat aplikasi, baik aplikasi kecil dan sederhana untuk keperluan pribadi, hingga aplikasi pengolahan database client-server perkantoran atau pertokoan. Selain itu, visual basic 6.0 juga mampu menciptakan suatu program dengan beberapa keunggulan, antara lain : pada produktifitas, kualitas, pengembangan perangkat lunak, kecepatan compiler, serta pola desain yang menarik (Budi, 2010:iii).

Visual basic 6.0 merupakan perangkat lunak yang dapat digunakan untuk membuat sebuah aplikasi yang mempunyai kualitas yang bagus.baik itu aplikasi yang kecil, sederhana hingga ke aplikasi pengolahan database. Selain itu dengan menggunakan *visual basic* 6.0 ini dapat juga membuat program dengan aplikasi GUI (*Graphical User Interface*) serta dapat menggunakan grafik atau gambar.

C. METODE PENELITIAN

Metode yang digunakan pada penelitian ini yaitu:

1. Studi litelatur, yaitu suatu metode untuk mendapatkan informasi dan melakukan pengumpulan data dengan membaca dan mempelajari berbagai litelatur-litelatur antara lain bersumber dari buku, jurnal,modul, refrensi internet, dan lain-lain yang mana sumber-sumber tersebut berhubungan dengan masalah yang diangkat sehingga dapat membantu dalam meyelesaikan permasalahan yang ada.
2. Metode pengembangan perangkat lunak menggunakan metode SDLC (*Software Development Life Cycle*) dengan model *Waterfall* yaitu: analisa, rancangan, pengkodean, dan implementasi.

D. HASIL DAN PEMBAHASAN

1. *Quick Sort*
   a) Konsep *Quick Sort*

   Sistem algoritma Quick Sort sendiri adalah membagi kumpulan suatu data menjadi beberapa sub bagian/partisi. Pembagian partisi ini berdasarkan letak dari suatu pivot yang dapat dipilih secara acak. Akan tetapi justru penentuan pivot inilah yang sangat mempengaruhi dalam proses kecepatan sorting. Pemilihan pivot bisa dengan berbagai cara. Bisa dari elemen pertama, elemen tengah, elemen terakhir atau secara acak. Cara yang diangap paling bagus dan lazim adalah pemilihan pivot pada elemen tengah dari suatu tabel. Karena dengan memilih elemen tengah, tabel tersebut akan dibagi menjadi 2 partisi yang sama besar. Penentuan elemen tengah dapat dirumuskan sebagai berikut:

   Pivot = ^{a}[(indeks awal + indeks akhir) div 2]

   Langkah-langkah mempartisi tabel dalam *Quick Sort* adalah sebagai berikut:

   1) pilih x □ {a1, a2, ..., an} sebagai elemen pivot
   2) pindai (scan) tabel dari kiri sampai ditemukan elemen ap ≥ x
   3) pindai tabel dari kanan sampai ditemukan elemen aq ≤ x
   4) pertukarkan ap dan aq
   5) ulangi langkah 2 dari posisi p+1, dan langkah 3 dari posisi q-1, sampai kedua pemindaian bertemu di tengah tabel

   b) Algoritma dan Pseudocode *Quick Sort*

   Algoritma Quick Sort adalah sebagai berikut:

```
Procedure QuickSort (input / output a : array
[1 . . n] of integer, input i, j : integer)
{ mengurutkan tabel a[i . . j] dengan
algoritma quick sort .
Masukkan: Tabel a[i . . j] yang sudah
terdefinisi elemen – elemennya.
Keluaran: Tabel a[i . . j] yang terurut menaik.
}
```

```
Deklarasi :
        K : integer;
Algoritma :
   If ( i < j ) then
        Partisi ( a, i, j, k ) { ukuran (a) > 1 }
        Quicksort ( a, i, k )
        Quicksort ( a, k + 1, j )
   Endif
```

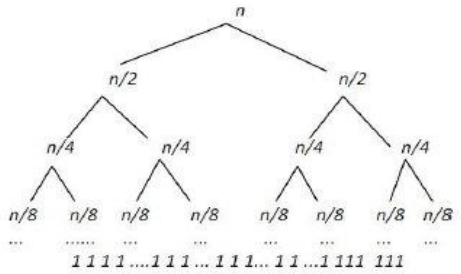
```
Procedure Partisi ( input / output:    a    :
array [ 1 . . n ] of integer, input i, j : integer,
output q : integer)

{ membagi tabel a[ i . . j ] menjadi uptabel
a[ i . . j ] dan a[ q + 1 . . j]
keluaran uptabel a[ i . . q ] dan uptabel a[ q +
1 . . j ]
```

```
Sedemikian sehingga elemen tabel a[ i . . q ]
Lebih kecil dari elemen tabel a[ q + 1 . . j ]
}
Deklarasi :
        Pivot, temp : integer
Algoritma :
        Pivot <- A[( i + I ) div 2 ]
        { Pivot = elemen tengah }

p <- i
q <- j
```

```
repeat
        while a[p] < pivot do
            p <- P + 1
          endwhile
          { Ap >= pibot }
         while a[q] > pivot do
            q <- q – 1
          endwhile
         { Aq >= pivot }

If (p ≤ q) then
        { pertukaran a[p] dengan a[q] }
   temp <- a[p]
   a[p]  <- a[q]
   a[q]  <- temp
{ tentukan awal pemidaian berikutnya }
        p <- p + 1
        q <- q - 1
  endif
until p > q
```

c) Kompleksitas *Quick Sort*

Efisiensi algoritma Quick Sort sangat dipengaruhi oleh pemilihan elemen pivot. Pemilihan pivot akan menentukan jumlah dan besar partisi pada setiap tahap rekursif. Kasus terbaik (*best case*) terjadi bila pivot berada pada elemen tengah dan n adalah 2k dimana k=konstanta, sehingga kedua tabel akan selalu berukuran sama setiap pemartisian.



Kompleksitanya :

$$T(n) = \begin{cases} 1 & , n = 1 \text{ (initial cond)} \\ 2T(n/2) + cn & , n > 1 \end{cases}$$

$T(n) = 2T(n/2) + cn$

$= 2(2T(n/4) + cn/2) + cn = 4(T(n/4) + 2cn$

$= 4(2(T(n/8) + cn/4) + 2cn = 8T(n/8) + 3cn$

$= ..$

$= 2k(T(n/2k) + kcn$

Pemartisisan tabel pada akhirnya akan menghasilkan tabel berukuran 1. Sedangkan n=1 adalah kondisi inisial yang menghasilkan T(n)=1, sehingga :

$$n/2k = 1 \; k = 2\log n$$

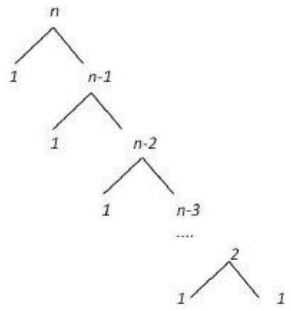
maka:

$T(n) = nT(1) + cn\ 2\log n$

$= n.1 + cn\ 2\log n$

$T(n) = O(n\ 2\log n)$

Kasus terburuk (*worst case*) terjadi jika terdapat kondisi dimana komposisi sub masalah adalah (n-1). Hal ini terjadi apabila pemilhan pivot adalah elemen pertama atau elemen terakhir sehingga partisi pertama sebesar 1 dan partisi kedua sebesar n-1.



Kompleksitasnya:

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

$T(n) = cn + T(n-1)$
$= cn + \{ c.(n-1) + T(n-2) \}$
$= cn + c(n-1) + \{c(n-2) + T(n-3)\}$
$=...$

$= c(n+(n-1)+(n-2)..+2) +1$
$= c\{(n-1)(n+2)/2\} + 1$
$= cn2/2 + cn/2 + (1-c)$

$T(n) = O(n2)$

Kasus rata-rata (*average case*) terjadi jika pivot dipilih secara acak dari unsur tabel, dan peluang setiap unsur dipilih menjadi pivot adalah sama. Rata-rata jumlah tingkatan perbandingan diatas semua permutasi urutan masukan dapat diperkirakan dengan teliti dengan pemecahan hubungan perulangan.

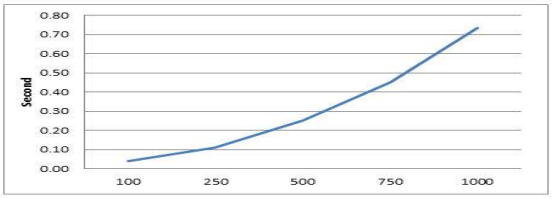
Tavg (n) = O(n n)= O(n log n)

d) Analisa *Quick Sort*
Algoritma *quick sort* mengurutkan dengan sangat cepat, namu algoritma ini sangat kompleks dan diproses secara rekursif. Algoritma ini sangat lebih cepat untuk beberaapa kasus khusus, namun untuk kasus umum, sampai saat ini tidak ada yang lebih cepat dibandingkan algoritma *quick sort.* Walaupun begitu algoritma *quick sort* ini tidak selalu menjadi pilihan yang terbaik. Karena algoritma ini dilakukan secara rekursif yang berarti jika dilakukan untuk tabel yang berukuran sangat besar, walaupun cepat, dapat menghabiskan memori yang besar pula. Selain itu, algoritma ini adalah algoritma yang terlalu kompleks untuk mengurutkan tabel yang berukuran kecil (hanya puluhan elemen misalnya). Dengan interval data antara 100 sampai dengan 1.000 elemen. Waktu eksekusi diukur dengan satuan *Second* (s).

Tabel 1. Waktu eksekusi algoritma *Quick Sort*

Jumlah elemen *array*	Waktu (s)
100	0.04
250	0.11
500	0.25
750	0.45
1000	0.73

Kompleksitas algoritma *quick sort* dapat di gambarkan seperti grafik dibawah ini:



Gambar 1. Grafik kompleksitas algoritma *quick sort*

2. *Shell Sort*
   a) Konsep *Shell Sort*
   Langkah – langkah pengurutan algoritma *shell sort* sebagai berikut:
      1) Untuk jarak (N/2)+1:
      Data pertama (i=0) dibandingkan dengan data jarak (N/2)+1. Apabila data pertama lebih besar dari data ke (N/2)+1 tersebut maka kedua data tersebut ditukar. Kemudian data kedua (i=1) dibandingkan dengan jarak yang sama yaitu (N/2)+1) = elemen ke-(i+N/2)+1. Demikian seterusnya sampai seluruh data dibandingkan sehingga semua data ke-i selalu lebih kecil dari pada data ke-(i+N/2)+1.
      2) Ulangi langkah-langkah diatas untuk jarak = (N/4)+1 kemudian lakukan perbandingan dan pengurutan sehingga semua data ke-i lebih kecil daripada data ke-(i+N/4)+1.
      3) Ulangi langkah-langkah diatas untuk jarak =(N/8)+1 kemudian lakukan perbandingan dan pengurutan sehingga semua data ke-i lebih kecil daripada data ke-(i+N/8)+1
      4) Demikian seterusnya samapi jarak yang digunakan adalah I atau data sudah terurut.

b) Algoritma dan Pseudocode *Shell Sort*
Algoritma *shell sort* adalah sebagai berikut:

```
void ShellSort (int *T, int Nmax)
/*I.S. T tabel dgn elemen bertipe*/
 /* integer, T tidak kosong*/
/*F.S. T terurut menaik*/
/*Proses : melakukan pengurutan*/
/* dengan metode shell sort*/
{
    /*kamus lokal*/
   int i, j, increment, temp;
   /*algoritma*/
  increment = 3;
  while (increment > 0)
  {
    for (i=0; i < Nmax; i++)
    {
       j = i;
       temp = *T[i];
      while ((j >= increment) &&
(*T[j-increment] > temp))
       {
         *T[j] = *T[j - increment];
          j = j - increment;
       } *T[j] = temp;
    }
if (increment/2 != 0)
     increment = increment/2;
     else if (increment == 1)
        increment = 0;
     else
        increment = 1;
     }
}
```

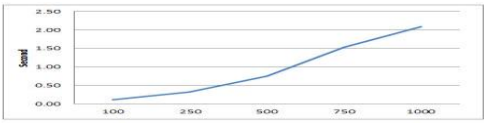
c) Kompleksitas *Shell Sort*
Algoritma *shell sort* adalah algoritma dengan kompleksitas algoritma $O(n^2)$ dan yang paling efisien dibandingkan algoritma – algoritma yang lain dengan kompleksitas algoritma yang sama. Algoritma *shell sort* lima kali lebih cepat dibandingkan algoritma pengurutan *bubble sort* dan dua kali lebih cepat dibandingkan algoritma pengurutan dengan *insertion sort.* Dan algoritma *shell sort* juga merupakan algoritma yang paling kompleks dan sulit dipahami.

d) Analisa *Shell Sort*
Algoritma *shell sort* melalukan *pass* atau traversal berkali-kali, dan setiap kali *pass* mengurutkan sejumlah nilai yang sama dengan ukuran set menggunakan *insertion sort*. Ukuran dari set yang harus diurutkan semakin membesar setiap kali melakukan pass pada tabel, sampai set tersebut mencakup seluruh elemen tabel. Ketika ukuran dari set semakin membesar, sejumlah nilai yang harus diurutkan semakin mengecil. Ukuran dari set yang digunakan untuk setiap kali iterasi (iteration) mempunyai efek besar terhadap efisiensi pengurutan. Algoritma shell sort adalah algoritma yang relatif sederhana. Hal ini menjadikan algoritma shell sort adalah pilihan yang baik dan efisien untuk mengurutkan nilai dalam suatu tabel berukuran sedang. Dengan interval data antara 100 sampai dengan 1.000 elemen. Waktu eksekusi diukur dengan satuan *Second* (s).

Tabel 2. Waktu eksekusi algoritma *Shell Sort*

Jumlah elemen *array*	Waktu (s)
100	0.11
250	0.32
500	0.75
750	1.53
1000	2.09

Kompleksitas algoritma *Shell sort* dapat di gambarkan seperti grafik dibawah ini:



Gambar 2. Grafik kompleksitas algoritma *shell sort*

3. *Insertion Sort*
   a) Konsep *Insertion Sort*
   Cara kerja algoritma ini yaitu pengurutan dengan penyisipan bekerja dengan cara menyisipkan masing-masing nilai di tempat yang sesuai (di antara elemen yang lebih kecil atau sama dengan nilai tersebut. Untuk menghemat memori, implementasinya menggunakan pengurutan di tempat yang membandingkan elemen saat itu dengan elemen sebelumnya yang sudah diurut, lalu menukarnya terus sampai posisinya tepat. Hal ini terus dilakukan sampai tidak ada elemen tersisa di input. Contoh dari algoritma ini dapat kita ambil dalam kehidupan sehari-hari, misalnya mengurutkan kartu remi. Langkah-langkah pengurutannya adalah:
   1) Elemen awal di masukkan sembarang, lalu elemen berikutnya dimasukkan dibagian paling akhir.
   2) Elemen tersebut dibandingkan dengan elemen ke (x-1). Bila belum terurut posisi elemen sebelumnya digeser sekali ke kanan terus sampai elemen yang sedang diproses

menemukan posisi yang tepat atau sampai elemen pertama.

3) Setiap pergeseran akan mengganti nilai elemen berikutnya, namun hal ini tidak menjadi persoalan sebab elemen berikutnya sudah diproses lebih dahulu.

b) Algoritma dan Pseudocode *Insertion Sort*

Algoritma *insertion sort* adalah sebagai berikut:

```
Procedure InsertionSort
(Input/Output T: TabInt, Input N: integer)
{mengurut tabel integer [1 .. N] dengan
Insertion Sort secara ascending}
Kamus:
i: integer
Pass: integer
Temp: integer
Algoritma:
Pass traversal [2..N]
Temp ←TPass
 i ← pass - 1
while (j ≥ 1) and (Ti > Temp) do Ti+1
←Ti i ← i-1
 depend on (T, i, Temp)
 Temp ≥ Ti : Ti+1 ← Temp
Temp < Ti : Ti+1 ← Ti
Ti ← Temp
{T[1..Pass-1] terurut}
```

c) Kompleksitas *Insertion Sort*

Kondisi terbaik (best case) tercapai jika data telah terurut. Hanya satu perbandingan dilakukan untuk setiap posisi i, sehingga terdapat n – 1 perbandingan, atau O (n). Kondisi terburuk (worst case) tercapai jika data telah urut namun dengan urutan yang terbalik. Pada kasus ini, untuk setiap i, elemen data[i] lebih kecil dari elemen data[0], …, data[i-1], masing-masing dari elemen dipindahkan satu posisi. Untuk setiap iterasi i pada kalang for terluar, selalu ada perbandingan i, sehingga jumlah total perbandingan untuk seluruh iterasi pada kalang ini adalah:

$$T(n) = 1 + 2 + \ldots + n - 1 = \sum_{f=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

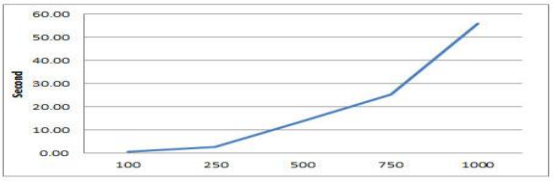
d) Analisa *Insertion Sort*

Untuk kasus terbaik algoritma ini berjalan 1 kali, yaitu jika elemen dalam tabel telah terurut. Kalang (loop) while tidak pernah dijalankan. Untuk kasus terburuk algoritma ini berjalan Nmax kali. Sehingga, seperti pengurutan gelembung, pengurutan dengan penyisipan mempunyai kompleksitas algoritma O(n2). Walaupun mempunyai kompleksitas algoritma yang sama, namun jika dijalankan dengan data input yang sama, algoritma pengurutan dengan penyisipan dua kali lebih cepat dan efisien dibandingkan dengan pengurutan gelembung. Namun, algoritma ini tetap kurang efisien untuk tabel berukuran besar (menyimpan banyak nilai). Dengan interval data antara 100 sampai dengan 1.000 elemen. Waktu eksekusi diukur dengan satuan *Second* (s).

Tabel 3. Waktu eksekusi algoritma *Insertion Sort*

Jumlah elemen *array*	Waktu (s)
100	0.56
250	2.67
500	13.73
750	25.34
1000	56.03

Kompleksitas algoritma *Insertion sort* dapat di gambarkan seperti grafik dibawah ini:



Gambar 3. Grafik kompleksitas algoritma *insertion sort*

4. *Selection Sort*

a) Konsep *Selection Sort*

Algoritma ini bekerja sebagai berikut:

1) Mencari nilai minimum (jika ascending) atau maksimum (jika descending) dalam sebuah list
2) Menukarkan nilai ini dengan elemen pertama list
3) Mengulangi langkah di atas untuk sisa list dengan dimulai pada posisi kedua.

Secara efisien kita membagi list menjadi dua bagian yaitu bagian yang sudah diurutkan, yang didapat dengan membangun dari kiri ke kanan dan dilakukan pada saat awal, dan bagian list yang elemennya akan diurutkan.

b) Algoritma dan Pseudocode *Selection Sort*

Algoritma *selection sort* adalah sebagai berikut:

```
Procedure SelectionSort (Input/Output
T: TabInt, Input N:integer)
{mengurut tabel integer [1 .. N] dengan
Selection Sort secara ascending}
Kamus:
 i: integer
 Pass: integer
 min: integer
 Temp: integer
Algoritma:
Pass traversal [1..N-1]
Min ← Pass
  i traversal [Pass+1..N]
if (Ti < Tmin) then
min ← i
    Temp ← TPass
    TPass ← Tmin
    Tmin ← Temp
    {T[1..Pass] terurut}
```

c) Kompleksitas *Selection Sort*
Algoritma di dalam *Selection Sort* terdiri dari kalang bersarang. Dimana kalang tingkat pertama (disebut pass) berlangsung N-1 kali. Di dalam kalang kedua, dicari elemen dengan nilai terkecil. Jika didapat, indeks yang didapat ditimpakan ke variabel min. Lalu dilakukan proses penukaran. Begitu seterusnya untuk setiap Pass. Pass sendiri makin berkurang hingga nilainya menjadi semakin kecil. Berdasarkan operasi perbandingan elemennya:

$$T(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \sum_{f=1}^{n-1} n - i$$

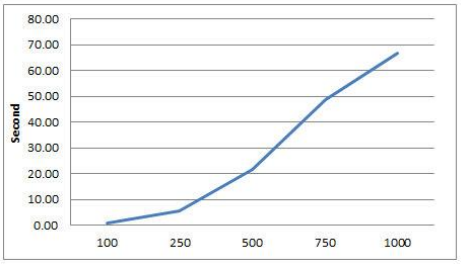
$$= \frac{n(n-1)}{2} = O(n^2)$$

d) Analisa *Selection Sort*
Sama seperti algoritma pengurutan gelembung, algoritma ini mempunyai dua buah kalang, satu kalang di dalam kalang yang lainnya. Banyaknya perbandingan yang harus dilakukan untuk siklus pertama adalah n, perbandingan yang harus dilakukan untuk siklus yang kedua n-1, dan seterusnya. Sehingga jumlah keseluruhan perbandingan adalah n(n+1)/2-1 perbandingan. Menggunakan algoritma pengurutan seleksi, hindari pengurutan nilai dengan data pada tabel lebih besar dari 1000 buah, dan hindari mengurutkan tabel lebih dari beberapa ratus kali. Berikut Dengan interval data antara 100 sampai dengan 1.000 elemen. Waktu eksekusi diukur dengan satuan *Second* (s).

Tabel 4. Waktu eksekusi algoritma *Selection Sort*

Jumlah elemen *array*	Waktu (s)
100	0.87
250	5.59
500	21.54
750	48.62
1000	66.86

Kompleksitas algoritma *Selection sort* dapat di gambarkan seperti grafik dibawah ini:



Gambar 4. Grafik kompleksitas algoritma *selection sort*

5. *Bubble Sort*
a) Konsep *Bubble Sort*
Dalam melakukan pengurutan data algoritma Bubble Sort bekerja sebagai berikut :
1) Bandingkan A [1] dengan A [2] dan susun sehingga A [1] < A [2]
2) Bandingkan A [2] dengan A [3] dan susun sehingga A [2] < A [3]
3) Bandingkan A [n-1] dengan A [n] dan susun sehingga A [n-1] < A [n] setelah (n-1) kali perbandingan, A [n] akan merupakan elemen terbesar peratama terurut. Langkah ke-2
4) Ulangi step 2 sampai kita telah mebandingkan dan kemungkinan menyusun A [n- 2], A [n-1]. Setelah (n-2) perbandingan, (n-1) akan merupakan elemen terbesar kedua.
5) Dan seterusnya. Langkah ke(n-1)
6) Bandingkan A [1] dengan A [2] dan susun sehingga A [1] < A[2].
7) Sesudah (n-1) langkah, array akan tersususn dalam urutan naik.

b) Algoritma dan Pseudocode *Bubble Sort*
Algoritma *Bubble Sort* adalah sebagai berikut :

```
procedure bubbleSort( A : list of
sortable items ) defined as:
do
    swapped := false
    for each i in 0 to length(A) - 2
inclusive do:
```

```
        if A[i] > A[i+1] then
        swap( A[i], A[i+1] )
        swapped := true
       end if
      end for
     while swapped
    end procedure
```

c) Kompleksitas *Bubble Sort*
Kompleksitas Algoritma Bubble Sort dapat dilihat dari beberapa jenis kasus, yaitu worst-case, average-case, dan best-case. Kondisi *best-case* data yang akan disorting telah terurut sebelumnya, sehingga proses perbandingan hanya dilakukan sebanyak (n-1) kali, dengan satu kali pass. Proses perbandingan pada bubble sort ini hanya dilakukan sebanyak (n-1) kali. Persamaan Big-O yang diperoleh dari proses ini adalah O(n). Dengan kata lain, pada kondisi Best-Case algoritma Bubble Sort termasuk pada algoritma lanjar. Kondisi *worst-case,* data terkecil berada pada ujung *array.* setiap kali melakukan satu pass, data terkecil akan bergeser ke arah awal sebanyak satu step. Dengan kata lain, untuk menggeser data terkecil dari urutan keempat menuju urutan pertama, dibutuhkan pass sebanyak tiga kali, ditambah satu kali pass untuk memverifikasi. Sehingga jumlah proses pada kondisi best case dapat dirumuskan sebagai berikut.

$$Jumlah\ proses = n^2 + n$$

Pada kondisi average-case, jumlah pass ditentukan dari elemen mana yang mengalami penggeseran ke kiri paling banyak. Dengan kata lain, jumlah proses perbandingan dapat dihitung sebagai berikut.

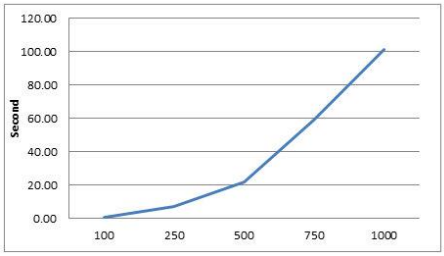
$$Jumlah\ proses = x^2+x$$

d) Analisa *Bubble Sort*
Pengurutan gelembung adalah algoritma pengurutan yang paling tua dan sederhana untuk diimplementasikan. Algoritma ini juga cukup mudah untuk dimengerti. Algoritma ini bekerja dengan cara membandingkan nilai tiap elemen dalam tabel dengan elemen setelahnya, dan menukar nilainya jika sesuai dengan kondisi yang diperlukan. Proses ini akan terus berulang hingga seluruh elemen dalam tabel telah diproses dan elemen dalam tabel telah terurut. Algoritma pengurutan gelembung ini adalah algoritma yang paling lamban dan tidak mangkus dibandingkan dengan algoritma pengurutan yang lain dalam penggunaan secara umum. Dalam kasus terbaik (yaitu list sudah terurut), kompleksitas algoritma pengurutan gelembung adalah O(n). Namun, untuk kasus umum, kompleksitas algoritma pengurutan gelembung adalah O(n2). Dengan interval data antara 100 sampai dengan 1.000 elemen. Waktu eksekusi diukur dengan satuan *Second* (s).

Tabel 5. Waktu eksekusi algoritma *Bubble Sort*

Jumlah elemen *array*	Waktu (s)
100	0.89
250	6.84
500	21.67
750	59.35
1000	101.50

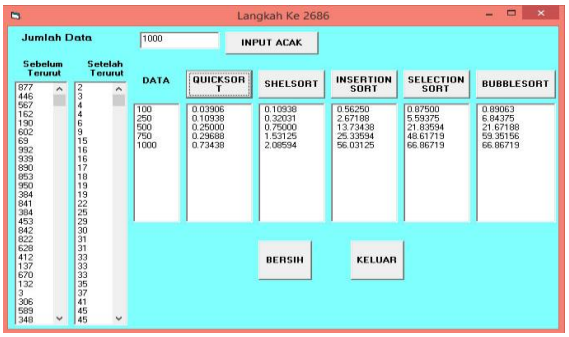
Kompleksitas algoritma *Bubble sort* dapat di gambarkan seperti grafik dibawah ini:



Gambar 5. Grafik kompleksitas algoritma *bubble sort*

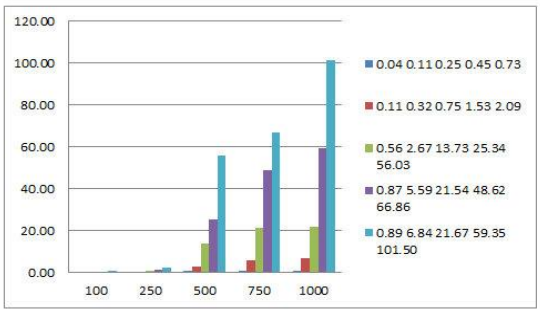
e) Perbandingan Kompleksitas Waktu
Masing-masing metode mempunyai kelebihan dan kekurangan, dari panjang pendeknya kode, kompleksitas kode, waktu pemrosesan, memori yang digunakan, kompatibilitas, dan lain sebagainya. Namun kompleksitas waktu yang dibutuhkan tetap menjadi masalah utama yang harus menjadi pertimbangan untuk menentukan algoitma mana yang lebih baik dan tepat untuk digunakan. waktu tiap algoritma, maka digunakan perangkat lunak penghitung kecepatan algoritma dalam bahasa pemograman *visual basic*. Hal ini digunakan untuk mendukung analisis kecepatan sorting data. Dengan mengatur perangkat lunak agar banyaknya jumlah n sesuai yang diinput secara acak, sehingga kemunculan data setiap kali program dijalankan tidak sama dan tidak dapat diprediksi dan

interval data antara 100 -– 1000. Pengujian dilakukan disebuah laptop Asus X200MA dengan spesifikasi : Windows 8.1 Pro 64-bit, processor Intel®, Celeron® CPU N2840 2.16Ghz, dan 2GB RAM.



Gambar 6. Hasil Program

Dari gambar hasil program diatas bisa kita lihat perbedaan waktu yang cukup jelas di antara algoritma-algoritma tersebut. Pengurutan data dengan metode quicksort lebih cepat jika dibandingkan dengan algritma yang lain. Hal ini dibuktikan dengan keadaan grafik waktu perbandingan setiap algoritma.



Gambar 7. Grafik perbandingan kompleksitas algoritma

E. KESIMPULAN DAN SARAN

1. Kesimpulan

Teknik – teknik pengurutan data memang cukup beragam, namun tentunya dalam proses ada metode yang tercepat. masing-masing metode mempunyai kelebihan dan kelemahan. Apalagi dalam pembuatan program komputer tidak lepas dari algoritma, karena program yang dibuat sangat kompleks. Program dapat dibuat tanpa menggunakan algoritma, akan tetapi program tersebut memiliki akses yang lambat atau memakai banyak memori. Berdasarkan logika proses pengurutan data dengan menggunakan algoritma *Quick Sort, Shell Sort, Insertion Sort, Selection Sort danBubble Sort,* maka dapat disimpulkan sebagai berikut :

a) Algoritma *Quick Sort* lebih cepat dalam melakukan pengurutan data jika dibandingkan dengan *Shell Sort, Insertion Sort, Selection Sort* dan *Bubble Sort.*
b) Dalam Pengurutan data, algoritma *shell sort* yang paling efisien dibandingkan algoritma – algoritma yang lain dengan kompleksitas algoritma yang sama.
c) Algoritma pengurutan dengan *insertion sort* dua kali lebih cepat dan efisien dibandingkan dengan pengurutan bubble sort. Namun, algoritma ini tetap kurang efisien untuk tabel berukuran besar (menyimpan banyak nilai).
d) *Selection Sort* lebih cepat mengurutkan data dibandingkan algoritma *bubble Sort*. Hal ini ditunjukkan kecilnya nilai yang didapat oleh algoritma tersebut.
e) Algoritma *bubble Sort* mempunyai algoritma yang sederhana sehingga lebih mudah untuk dipahami. Untuk kasus-kasus sederhana dengan jumlah data sedikit, *selection Sort* dapat diunggulkan bila dibandingkan dengan *bubble Sort*.

2. Saran

Adapun saran yang penulis berikan untuk pengembangan penelitian ini sebagai berikut :

a) Dalam penelitian selanjutnya dapat menggunakan tipe data string sehingga penerapannya akan menjadi lebih beragam.
b) Untuk mendapatkan waktu yang lebih optimal, penulis menyarankan agar algoritma dari metode diatas dapat lebih di optimasikan.
c) Dalam pengurutan data pada penelitian ini penulis menggunakan bahasa pemograman *visual basic 6*.0. disarankan dapat menggunakan bahasa pemograman lain selain *visual basic* 6.0.

DAFTAR PUSTAKA

[1] Atrinawati, Lovinta Happy. 2007. *Analisis Kompleksitas Algoritma Untuk Berbagai Macam Metode Pencarian Nilai (Searching) Dan Pengurutan Nilai (Sorting) Pada Tabel* (http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2006-2007/Makalah/Makalah0607-77.pdf, diakses 20 Januari 2016)

[2] Budi. 2010. *Programming With Microsoft Visual Basic 6.0*. Yogyakarta: Skripta Media Creative.

[3] Saputra, dkk. 2010. *Analisis Algoritma Rekursif Quick Sort* (http://www.mediafire.com/download/py3q5jevwp59vj/DAA+2010+IF32_01+Analisis+Algoritma+Rekursif+Quick+Sort.pdf, diakses 20 Januari 2016)
[4] Suarga. 2012. *Algoritma Pemrograman*. Yogyakarta: Andi.
[5] Tjaru, Setia Negara B. 2010. *Kompleksitas Algoritma Pengurutan Selection Sort dan Insertion Sort*. Makalah IF2091 Strategi Algoritmik Tahun 2009 (http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2009-2010/Makalah0910/MakalahStrukdis0910-074.pdf, diakses 20 Januari 2016)
[6] Yahya, Sofyansyah Yusari. 2014. *Analisa Perbandingan Algoritma Bubble Sort dan Selection Sort Dengan Metode Perbandingan Eksponensial*. Jurnal Pelita Informatika Budi Darma, Vol : VI, No : 3, April 2014 (http://pelita-informatika.com/berkas/jurnal/28.%20Sofyansayah.pdf, diakses 20 Januari 2016)
[7] Wahyu Eko Susanto, *Pendekatan Keamanan Serta Kecepatan Akses Data Pada Cloud Dengan Algoritma Huffman Dan Aes*, Vol 2, No 2 (2014): Jurnal Bianglala Informatika 2014
[8] Saifudin, *Penerapan Algoritma C4.5 Dalam Prediksi Penyewa Sepeda*, Vol 2, No 2 (2014): Jurnal Evolusi 2014
[9] Pudji Widodo, *Rule-Based Classifier Untuk Mendeteksi Penyakit Liver*, Vol 2, No 1 (2014): Jurnal Bianglala Informatika 2014
[10] Sardiarinto, *Aplikasi Sistem Pendukung Keputusan Kelayakan Peminjaman Kredit Nasabah Koperasi Berbasis Android*, Vol 1, No 1 (2013): Jurnal Bianglala Informatika 2013