KOMPLEKSITAS ALGORITMA PENGURUTAN (*SORTING ALGORITHM*)

Andi Kurniawan Dwi Putranto / 13508028

Program Studi Teknik Informatika,
Sekolah Teknik Elektro dan Informatika Institut Teknologi Bandung
Jln. Ganesha No. 10 Bandung 40135
e-mail: if18028@students.if.itb.ac.id

ABSTRAK

Pada Makalah ini akan dibahas kompleksitas beberapa algoritma pengurutan, antara lain : *Count Sort, Insertion Sort, dan Bubble Sort.* Penulis akan mengawali makalah ini dengan memberikan penjelasan singkat mengenai istilah-istilah yang akan dipakai selama penulisan makalah. Kemudian dilanjutkan dengan pembahasan algoritma pengurutan. Pembahasan yang akan diulas di antaranya adalah penjelasan mengenai algoritma pengurutan yang dipakai, simulasi pengurutan, dan komplekstiasnya.

Kata kunci: algoritma pengurutan, *Count Sort, Insertion Sort, Bubble Sort*, simulasi pengurutan, dan kompleksitas algoritma.

1. PENDAHULUAN

Pengurutan merupakan salah satu algoritma yang sangat penting di dalam dunia Teknologi Informasi terutama pada bagian pengolahan data. Pengolahan data, misalnya pembacaan, akaan lebih mudah dilakukan apabila data yang diolah sudah terurut. Oleh karena itu dibutuhkan algoritma pengurutan untuk mengurutkan data.

Performasi pengurutan data sangat menentukan performasi sistem, karena itu pemilihan metoda pengurutan yang cocok akan berperan dalam suatu aplikasi. Biasanya selain ditentukan performansi rata-rata, juga ditentukan performansi terjelek dan performansi terbagus. Untuk beberapa aplikasi, performansi yang "stabil" perlu juga dipertimbangkan.

Dari latar belakang di atas, Penulis berusaha melakukan analisa kompleksitas beberapa algoritma pengurutan (Count Sort, Insertion Sort, dan Bubble Sort).

2. METODE

Pada bab ini akan dibahas penjelasan singkat mengenai istilah-istilah yang dipakai di dalam pembuatan makalah ini.

2.1 Algoritma

Algoritma adalah deskripsi dapat terdiri dari suatu pola tingkah laku, dinyatakan dalam primitif, yaitu aksi-aksi yang didefinisikan sebelumnya dan diberi nama, dan diasumsikan sebelumnya bahwa aksi-aksi tersebut dapat dikerjakan sehingga dapat menyebabkan kejadian yang dapat diamati (dikutip dari "Draft Diktat Kuliah Dasar Pemrograman (Bagian Pemrograman Prosedural)").

Algoritma adalah urutan langkah-langkah logis penyelesaian masalah yang disusun secara sistematis (dikutip dari "Diktat Kuliah IF2091 Struktur Diskrit")

Suatu algoritma dapat terdiri dari beberapa sub-algoritma, jika setiap sub-aksi juga dapat diuraikan dalam urut-urutan yang dapat dimengerti dengan baik dan terbatas.

Urutan-uratan langkah harus dapat dimengerti dengan baik, oleh pembuat algoritma maupun oleh yang akan mengerjakan. Tidak boleh ada sedikit pun salah pengertian di antara keduanya supaya dapat dihasilkan efek yang diinginkan.

2.2 Algoritma Pengurutan (Sorting)

Sorting atau pengurutan data adalah proses yang sering harus dilakukan dalam pengolahan data. Pengurutan dapat dilakukan dari nilai terkecil ke nilai terbesar (*ascending*) atau sebaliknya (*descending*).

Algoritma pengurutan dibedakan menjadi 2 macam, yaitu :

- Pengurutan internal, yaitu pengurutan terhadap sekumpulan data yang disimpan dalam media internal komputer yang dapat diakses setiap elemennya secara langsung. Maka dapat dikatakan sebagai pengurutan tabel.
- Pengurutan eksternal, yaitu pengurutan data yang disimpan dalam memori sekunder, biasanya data

bervolume besar sehingga tidak mampu untuk dimuat semuanya di dalam memori.

Dalam makalah ini hanya akan dibahas jenis pengurutan internal. Algoritma pengurutan yang akan dibahas, antara lain adalah *Count Sort, Insertion Sort, dan Bubble Sort.*

Hal-hal yang menyebabkan suatu algoritma sering digunakan adalah kestabilan, kesesuaian dengan kebutuhan, kesesuaian dengan struktur data yang dipakai, kenaturalan, dan kemangkusan (efficiency). Ukuran untuk menyatakan kemangkusan algoritma tersebut dapat dinyatakan dengan kompleksitas algoritma.

2.3 Kompleksitas Algoritma

Sebuh algoritma tidak saja harus benar, tetapi juga harus mangkus (*efisien*). Algoritma yang benar sekalipun mungkin tidak berguna untuk jenis dan ukuran masukan tertentu karena waktu yang diperlukan untuk menjalankan algoritma tersebut atau ruang memori yang diperlukan untuk struktur datanya terlalu besar.

Kompleksitas algoritma adalah besaran yang dipakai untuk menerangkan model abstrak pengukuran waktu/ ruang. Dengan menggunakan besaran kompleksitas waktu/ruang algoritma, kita dapat menentukan laju peningkatan waktu/ ruang yang diperlukan algoritma dengan meningkatkan ukuran masukan n.

Terkadang kita tidak terlalu membutuhkan kompleksitas waktu yang detil dari sebuah algoritma. Yang kita butuhkan terkadang adalah besaran kompleksitas waktu yang menghampiri kompleksitas waktu yang sebenarnya. Kompleksitas waktu yang demikian disebut kompleksitas waktu asimptotik yang dinotasikan dengan "O" (baca : "O-Besar"). 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 $O(n^3)$. Karena n^3 yang memberikan kompleksitas waktu terbesar. Kita tidak perlu menambahkan pengali dari term dari n^3.

3. PEMBAHASAN

3.1 Count Sort

Count Sort adalah salah satu algoritma pengurutan yang menggunakan metode pencacahan. Pengurutan dengan pencacahan adalah pengurutan yang paling sederhana. Jika diketahui bahwa data yang akan diurut mempunyai daerah jelajah (range) tertentu, dan merupakan bilangan bulat, misalnya [Min..Max] maka cara paling sederhana untuk mengurut adalah :

1. Sediakan array TabCount [Min..Max] yang diinisialisasi dengan nol, dan pada akhir proses TabCount berisi banyaknya data pada tabel asal yang bernilai i.
2. Tabel dibentuk kembali dengan menuliskan kembali harga-harga yang ada.

3.1.1 Simulasi Pengurutan

Berikut adalah simulasi pengurutan menggunakan algoritma *Count Sort*. Sebagai contoh akan digunakan tabel TabInt

1	3	2	5	1	6	1	3	2	1

Pertama-tama dilakukan pembentukan tabel TabCount yang memiliki indeks maksimal = nilai maksimum yang ada pada TabInt. Kemudian TabCount diinisialisasi dengan nilai 0.

1	2	3	4	5	6
0	0	0	0	0	0

Kemudian TabCount diisi. Indeks ke-i pada TabCount diisi dengan banyaknya nilai i yang muncul pada TabInt.

Looping pertama :

TabInt

1	3	2	5	1	6	1	3	2	1

Mengisi TabCount

1	2	3	4	5	6
1	0	0	0	0	0

Looping kedua :

TabInt

1	**3**	2	5	1	6	1	3	2	1

Mengisi TabCount

1	2	3	4	5	6
1	0	**1**	0	0	0

Looping ketiga:

TabInt

1	3	**2**	5	1	6	1	3	2	1

Mengisi TabCount

1	2	3	4	5	6
1	**1**	1	0	0	0

Looping keempat:

TabInt

1	3	2	**5**	1	6	1	3	2	1

Mengisi TabCount

1	2	3	4	5	6
1	1	1	0	**1**	0

Looping kelima:

TabInt

1	3	2	5	**1**	6	1	3	2	1

Mengisi TabCount

1	2	3	4	5	6
2	1	1	0	1	0

Looping keenam:

TabInt

1	3	2	5	1	6	1	3	2	1

Mengisi TabCount

1	2	3	4	5	6
2	1	1	0	1	1

Looping ketujuh:

TabInt

1	3	2	5	1	6	1	3	2	1

Mengisi TabCount

1	2	3	4	5	6
3	1	1	0	1	1

Looping kedelapan:

TabInt

1	3	2	5	1	6	1	3	2	1

Mengisi TabCount

1	2	3	4	5	6
3	1	2	0	1	1

Looping kesembilan:

TabInt

1	3	2	5	1	6	1	3	2	1

Mengisi TabCount

1	2	3	4	5	6
3	2	2	0	1	1

Looping kesepuluh:

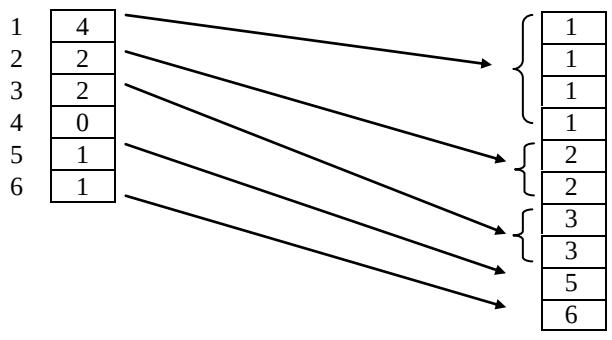
TabInt

1	3	2	5	1	6	1	3	2	1

Mengisi TabCount

1	2	3	4	5	6
4	2	2	0	1	1

Langkah selanjutnya adalah memdindahkan isi dari TabCount ke TabInt, sehingga elemen pada TabInt terurut menaik.



3.1.2 Algortima Count Sort

```
Procedure CountSort (input/ouput T : TabInt, input N : integer)
{ Mengurut tabel integer [1..N] dengan pencacahan}

Kamus Lokal
{ValMin dan ValMax adalah batas minimum dan maximum harga yang tersimpan dalam T, harus diketahui}
     TabCount : array [ValMin..ValMax] of integer [0..NMax]
     i : integer {indeks untuk traversal tabel}
     k :  integer {jumlah elemen T yang sudah diisi pada proses pembentukan kembali }

ALGORITMA
   {inisilaisasi TabCount}
   i traversal [ValMin..ValMax]
      TabCount_i ← 0
   {counting}
   i traversal [1..N]
      TabCount_Ti ← TabCount_Ti + 1
   {pengisian kembali : T1 ≤ T2 … ≤ TN}
   K ← 0
   i traversal [ValMin..ValMax]
      if (TabCount_i ≠ 0) then
         repeat TabCount_i times
            K ← K + 1
            T_K ← i
```

Catatan :
$TabCount_{Ti}$ dituliskan untuk menunjukkan bahwa indeks T adalah i, dan T_i merupakan indeks dari TabCount.

3.1.3 Kompleksitas Algortima Count Sort

Kompleksitas dari algoritma *Count Sort* dapat dilakukan dengan perhitungan sebagai berikut :
Kalang pertama membutuhkan waktu *O(ValMax-ValMin)*,
Kalang kedua membutuhkan waktu *O(n)*,
Kalang pertama membutuhkan waktu *O(ValMax-ValMin)*,
Jadi total waktu yang dibutuhkan adalah *O(ValMax-ValMin) + O(n) + O(ValMax-ValMin)*
Karena *O(ValMax-ValMin)* sering dianggap *O(n)* maka kompleksitas waktu yang dibutuhkan menjadi *O(n) + O(n) + O(n) = O(3n)* yang setara dengan *O(n)*.

Algoritma *Count Sort* adalah algoritma yang stabil. Di mana data-data dengan nilai yang sama akan diurutkan berdasar urutan kemunculan pada array asal. Properti ini sangat penting dalam pengurutan data majemuk. *Count Sort* melakukan pengurutan tanpa perbandingan data.

Kompleksitas waktu dari algoritma *Count Sort* adalah *O(n)*. Dari kompleksitas ini dapat kita lihat bahwa grafik kompleksitasnya berupa grafik linear, di mana perubahan n yang besar tidak menyebabkan T(n) naik signifikan. Walaupun kompleksitas *Count Sort* relatif lebih baik daripada algoritma pengurutan lainnya, namun jika kita

lihat dari sisi memori, maka algoritma ini adalah tergolong algoritma yang boros memori. Jka jumlah data yang banyak dan bilangan yang diurutkan adalah bilangan yang besar, maka kebutuhan akan memori sangatlah besar.

Metode pengurutan ini sangat cepat dan efekstif, akan tetapi memiliki kelemahan yaitu dibutuhkan memori tambahan sebagai array bantu dalam prosesnya. *Best case* bila nilai maksimal data tidak jauh lebih besar dari jumlah data yang diurutkan. *Worst case* bila nilai maksimal data juah lebih besar dari jumlah data yang diperlukan. *Worst case* yang terjadi hanya berkisar pada memori yang diperlukan, bukan pada waktu yang dihasilkan.

3.2 Insertion Sort

Idenya adalah mencari tempat yang tepat untuk suatu elemen data yang telah diketahui ke dalam subkumpulan data yang telah terurut, kemudian melakukan penyisipan (*insertion*) data di tempat yang tepat tersebut. Proses dilakukan sebanyak N tahapan (yang dalam sorting disebut sebagai "pass") :

1. T_1 dianggap sudah tepat tempatnya
2. T_2 harus dicarikan tempat yang tepat pada T[1..2], yaitu sisipkan T_2 pada T_i. Tp[1..2] terurut membesar
3. T_3 harus dicarikan tempat yang tepat pada T[1..3], yaitu sisipkan T_3 pada T_i. Tp[1..3] terurut membesar

…

N-1 T_{N-1} harus dicarikan tempat yang tepat pada T[1..N-1], yaitu sisipkan T_{N-1} pada T_i. T[1..N-1] terurut membesar.

N T[1..N] sudah terurut : $T_1 \leq T_2 \leq T_3 \leq \ldots \leq T_N$

Pada setiap Pass, tabel terdiri dari dua bagian : yang sudah terurut yaitu [1..Pass-1] dan sisanya [Pass..Nmax] yang belum terurut. Ambil elemen T_{Pass} , sisipkan ke dalam T[1..Pass-1] dengan tetap menjaga keterurutan. Untuk menyisipkan T_{Pass} ke T_i, harus terjadi "pergeseran" elemen tabel T[i..Pass]. pergeseran ini dapat dilakukan sekaligus dengan pencarian i. pencarian dapat dihentikan segera dengan memanfaatkan sigat keterurutan T[1..Pass]. Metoda pengurutan ini cukup efisisen terutama untuk N yang "kecil".

3.2.1 Simulasi Pengurutan

Berikut adalah simulasi pengurutan menggunakan algoritma *Insertion Sort*. Sebagai contoh akan digunakan larik

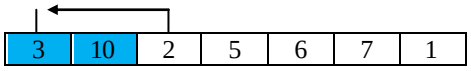
3	10	2	5	6	7	1

Pertama-tama dilakukan iterasi dimana di setiap iterasi insertion sort memindahkan nilai elemen, kemudian menyisipkannya berulang-ulang sampai ke tempat yang tepat. Begitu seterusnya dilakukan. Dari proses iterasi, seperti biasa, terbentuklah bagian yang telah di-*sorting* dan bagian yang belum.

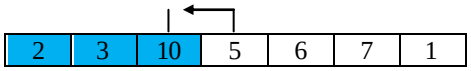
Sebelum *loop*

3	10	2	5	6	7	1

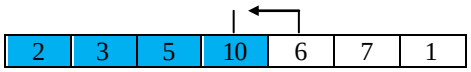
Pass = 2



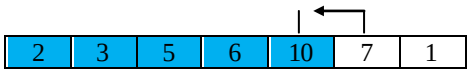
Pass = 3



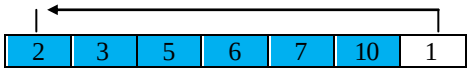
Pass = 4



Pass = 5



Pass = 6



Pass = 7

1	2	3	5	6	7	10

3.2.2 Algortima Insertion Sort

```
Procedure InsertionSort (input/output T : TabInt, input N : integer)
{mengurut tabel integer [1..N] dengan insertion sort}

KAMUS LOKAL
   i : integer   {indeks untuk traversal tabel}
   pass : integer{tahapan pengurutan}
   Temp : integer

ALGORITMA
   {T1 adalah terurut}
   Pass traversal [2..N]
     Temp ← TPass {simpan harga TPass supaya tidak tertimpa karena pergeseran}
     {sisipkan elemen ke Pass dalam T [1..Pass-1] sambil menggeser :}
     i ← Pass - 1
     {cari I, Temp < Ti and i > 1}
```

```
      while (Temp < Ti) and (i > 1) do
        Ti+1 ← Ti    {geser}
        i ← i - 1    {berikutnya}
      {Temp ≥ Ti (tempat yang tepat) or i = 1 (sisipkan sebagai element pertama)}
      depend on (T, i, Temp)
        Temp ≥ Ti  : Ti+1 ← Temp      {menemukan tempat yang tepat}
        Temp < Ti  :  Ti+1 ← Ti
                      Ti ← Temp       {sebagai elemen pertama}
      {T[1..Pass] terurut membesar : T1 ≤ T2 ≤ T3 ≤ ... ≤ TPass }
      {Seluruh tabel terurut, karena Pass = N : T1 ≤ T2 ≤ T3 ≤ ... ≤ TN }
```

3.2.2 Kompleksitas Algortima Insertion Sort

Pada algoritma ini terdapat 2 kalang bersarang. Di mana terjadi N-1 Pass (dengan N adalah banyak elemen struktur data), dengan masing-masing Pass terjadi i kali operasi perbandingan. i tersebut bernilai 1 untuk Pass pertama, bernilai 2 untuk Pass kedua, begitu seterusnya hingga Pass ke N-1.

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

3.3 Bubble Sort

Pengurutan gelembung adalah algoritma pengurutan yang paling tua dan sederhana untuk diimplementasikan. Algoritma ini juga cukup mudah untuk dimengerti.

Diberi nama ”Bubble ” karena proses pengurutan secara berangsur-angsur bergerak/ berpindah ke psosisinya yang tepat, seperti gelembung yang keluar dari sebuah gelas bersoda.

Bubble sort mengurutkan data dengan cara membandingkan elemen sekarang dengan elemen berikutnya. Jika elemen sekarang lebih besar dari elemen berikutnya maka kedua elemen tersebut ditukar, jika pengurutan ascending. Jika elemen sekarang lebih kecil dari elemen berikutnya, maka kedua elemen tersebut dituka, jika pengurutan descending.

Algoritma ini seolah-olah menggeser satu per satu elemen dari kanan ke kiri atau kiri ke kanan, tergantung jenis pengurutannya. Ketika satu proses selesai, maka bubble sort akan mengulangi proses, demikian seterusnya. Kapan berhentinya? *Bubble sort* berhenti jika seluruh array telah diperiksa dan tidak ada pertukaran lagi yang bisa dilakukan, serta tercapai pengurutan yang telah diinginkan.

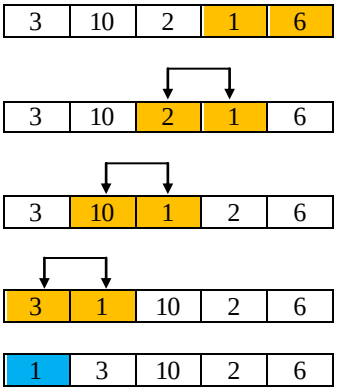
3.3.1 Simulasi Pengurutan

Berikut adalah simulasi pengurutan menggunakan algoritma *Bubble Sort*. Sebagai contoh akan digunakan larik

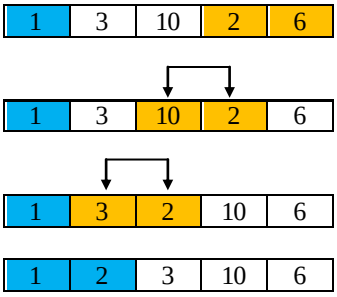
3	10	2	1	6

Proses pembandingan di dalam setiap *looping* digambarkan dengan warna jingga. Jika ternyata diperlukan pertukaran nilai. Hasil penukaran digambarkan di tabel simulasi berikutnya. Untuk membedakan dengan elemen tabel yang lain, bagian tabel yang telah diurutkan digambarkan dengan
warna biru muda.

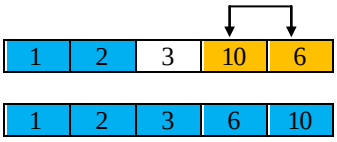
Looping pertama



Looping kedua



Looping ketiga



3.3.2 Algoritma Bubble Sort

```
Procedure BubbleSort (Input/Output T: TabInt, Input N: integer)
{mengurut tabel integer [1 .. N] dengan Bubble Sort}

KAMUS
  i: integer
  Pass: integer
  Temp: integer

ALGORITMA
  Pass traversal [1..N-1]
    i traversal [N..Pass+1]
```

```
    if (Ti < Ti-1) then
      Temp ← Ti
      Ti ← Ti-1
      Ti-1 ← Temp
{T[1..Pass] terurut}
```

3.3.3 Kompleksitas Algoritma Bubble Sort

Algortima di dalam *bubble sort* terdiri dari 2 kalang (loop) bertingkat. Kalang pertama berlangsung selama N-1 kali. Indeks untuk kalang pertama adalah Pass. Kemudian kalang tingkat kedua berlangsung dari N sampai dengan Pass + 1.

Dengan demikian, proses pembandingan yang terjadi sebanyak :

$$T(n) = (N - 1) + (N - 2) + \ldots + 2 + 1$$

$$= \sum_{i=1}^{N-1} N - 1 = \frac{N(N-1)}{2}$$

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

T(n) ini merupakan kompleksitas untuk kasus terbaik ataupun terburuk. Hanya saja pada kasus terbaik, yaitu pada kasus jika struktur data telah terurut sesuai perintah, proses Pass terjadi tanpa adanya *assignment*.

Hal ini jelas menunukkan bahwa algoritma akan berjalan lebih cepat. Hanya saja tidak terlalu signifikan.

4. KESIMPULAN

- Algoritma pengurutan (sorting) merupakan salah satu algoritma yang penting dalam pengolahan data
- Kemangkusan suatu algoritma dapat dihitung dengan menghitung kompleksitas waktunya (kompleksitas asimptotik)
- Kompleksitas waktu algoritma *Count Sort* = $O(n)$
- Metode *Count Sort* sangat cepat dan efekstif, akan tetapi memiliki kelemahan yaitu dibutuhkan memori tambahan sebagai array bantu dalam prosesnya
- Kompleksitas waktu algoritma *Insertion Sort* = $O(n^2)$
- Kompleksitas waktu algoritma *Bubble Sort* = $O(n^2)$
- Di antara ketiga algoritma pengurutan yang Penulis analisisis, algoritma yang paling mangkus adalah *Count Sort*

REFERENSI

[1] Munir, Rinaldi. 2008. Diktat Kuliah IF2091 Struktur Diskrit. Program Studi Teknik Informatika, Institut Teknologi Bandung. hlm. V-1 dan X-1 s.d. X-28

[2] Liem, Inggriani. 2007. Draft Diktat Kuliah Dasar Pemrograman (Bagian Pemrograman Prosedural). Program Studi Teknik Informatika, Institut Teknologi Bandung. hlm. 134-142

[3] Wikipedia, the free encyclopedia. 2009. Tanggal akses: 18 Desember 2009 pukul 21.00 WIB dan 19 Desember 2009 pukul 19.00 WIB
http://en.wikipedia.org/wiki/Bubble_sort
http://en.wikipedia.org/wiki/Insertion_sort

[4] Scribd. 2009. Tanggal akses: 19 Desember 2009 pukul 19.00 WIB
http://www.scribd.com/doc/9710352/Analisis-Algoritma-Pada-Masalah-Sorting

[5] fullsearching. 2009. Tanggal akses: 19 Desember 2009 pukul 19.00 WIB
http://www.fullsearching.co.cc/2009/08/metode-pengurutan-data-dan-contoh-array.html