Analisis Algoritma Bubble Sort

Ryan Rheinadi – NIM : 13508005
Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Jalan Ganesha 10, Bandung
e-mail: if18005@students.if.itb.ac.id

ABSTRAK

Makalah ini membahas efektifitas dari algoritma bubble sort yang merupakan salah satu bentuk algoritma pengurutan. Efektifitas yang akan ditinjau di makalah ini yaitu mengenai kompleksitas algoritma serta tingkat kesulitan koding dari algoritma bubble sort.

Kata kunci: Efektifitas, algoritma pengurutan, kompleksitas algoritma,

1. PENDAHULUAN

1.1 Kompleksitas Algoritma

Untuk menyelesaikan suatu masalah, akan terdapat berbagai algoritma yang dapat digunakan, sesuai dengan salah satu pepatah popular, "Ada banyak jalan menuju Roma." Akan tetapi, algoritma manakah yang harus dipilih agar masalah itu dapat diselesaikan dengan efektif? Tentu harus ada parameter yang bisa dibandingkan.

Dalam aplikasinya, setiap algoritma memiliki dua buah ciri khas yang dapat digunakan sebagai parameter pembanding, yaitu jumlah proses yang dilakukan dan jumlah memori yang digunakan untuk melakukan proses. Jumlah proses ini dikenal sebagai kompleksitas waktu yang disimbolkan dengan T(n), sedangkan jumlah memori ini dikenal sebagai kompleksitas ruang yang disimbolkan dengan S(n).

Kompleksitas waktu diukur berdasarkan jumlah proses khas suatu algoritma, bukan berdasarkan *run-time* secara nyata ketika aplikasi dilakukan. Hal ini disebabkan oleh arsitektur komputer dan kompiler yang berbeda-beda, sehingga suatu algoritma yang sama akan menghasilkan waktu eksekusi yang berbeda, pada komputer dan kompiler yang berbeda.

Dalam setiap algoritma, terdapat berbagai jenis operasi, di antaranya :

- Operasi baca tulis.
- Operasi aritmatika (+ - / *)
- Operasi pengisian nilai (*assignment*)
- Operasi pengaksesan elemen larik
- Operasi pemanggilan fungsi ataupun prosedur

Dalam perhitungan kompleksitas algoritma, kita hanya menghitung operasi khas (tipikal) yang mendasari algoritma tersebut.

1.2 Kompleksitas Waktu Asimptotik

Kenyataannya, jarang sekali kita membutuhkan kompleksitas waktu yang detil dari suatu algoritma. Biasanya yang kita butuhkan hanyalah hampiran dari kompleksitas waktu yang sebenarnya. Kompleksitas waktu ini dinamakan kompleksitas waktu asimptotik yang dinotasikan dengan "O" (baca : "O-besar"). Kompleksitas waktu asimptotik ini diperoleh dengan mengambil *term* terbesar dari suatu persamaan kompleksitas waktu. Sebagai contoh, dapat dilihat pada persamaan di bawah ini.

$$T(n)=4n^3+5n^2+7n+3 \quad (1)$$

$$O(n^3) \quad (2)$$

Dari persamaan (1) di atas diperoleh persamaan (2). Dapat dilihat bahwa nilai O adalah *term* terbesar dari T(n), tanpa faktor pengalinya. Berikut ini adalah daftar dari beberapa kelompok algoritma berdasarkan nilai O nya.

Kelompok Algoritma	Nama
O(1)	konstan
O(log n)	logaritmik
O(n)	lanjar
O(n log n)	n log n
$O(n^2)$	kuadratik
$O(n^3)$	kubik
$O(2^n)$	eksponensial
O(n!)	faktorial

Tabel 1. Pengelompokan Algoritma Berdasarkan Notasi O-Besar

Kompleksitas algoritma tersebut memiliki suatu spektrum, yang menunjukkan tingkat kompleksitas suatu algoritma, dengan urutan sebagai berikut.

$O(1)<O(\log n)<O(n)<O(n \log n)<O(n^2)<\ldots<O(2^n)<O(n!)$

2. ALGORITMA BUBBLE SORT

2.1 Ide Dasar Algoritma Bubble Sort

2.1.1 Langkah pengurutan dalam Bubble Sort

Algoritma bubble sort adalah salah satu algoritma pengurutan yang paling simple, baik dalam hal pengertian maupun penerapannya. Ide dari algoritma ini adalah mengulang proses pembandingan antara tiap-tiap elemen *array* dan menukarnya apabila urutannya salah. Pembandingan elemen-elemen ini akan terus diulang hingga tidak perlu dilakukan penukaran lagi. Algoritma ini termasuk dalam golongan algoritma *comparison sort,* karena menggunakan perbandingan dalam operasi antar elemennya. Berikut ini adalah gambaran dari algoritma bubble sort.

Misalkan kita mempunyai sebuah *array* dengan elemen-elemen "4 2 5 3 9". Proses yang akan terjadi apabila digunakan algoritma bubblesort adalah sebagai berikut.

Pass pertama
(**4 2** 5 3 9) menjadi (**2 4** 5 3 9)
(2 **4 5** 3 9) menjadi (2 **4 5** 3 9)
(2 4 **5 3** 9) menjadi (2 4 **3 5** 9)
(2 4 3 **5 9**) menjadi (2 4 3 **5 9**)

Pass kedua
(**2 4** 3 5 9) menjadi (**2 4** 3 5 9)
(2 **4 3** 5 9) menjadi (2 **3 4** 5 9)
(2 3 **4 5** 9) menjadi (2 3 **4 5** 9)
(2 3 4 **5 9**) menjadi (2 3 4 **5 9**)

Pass ketiga
(**2 3** 4 5 9) menjadi (**2 3** 4 5 9)
(2 **3 4 5** 9) menjadi (2 **3 4** 5 9)
(2 3 **4 5** 9) menjadi (2 3 **4 5** 9)
(2 3 4 **5 9**) menjadi (2 3 4 **5 9**)

Dapat dilihat pada proses di atas, sebenarnya pada pass kedua, langkah kedua, *array* telah terurut. Namun algoritma tetap dilanjutkan hingga pass kedua berakhir. Pass ketiga dilakukan karena definisi terurut dalam algoritma bubblesort adalah tidak ada satupun penukaran pada suatu pass, sehingga pass ketiga dibutuhkan untuk memverifikasi keurutan *array* tersebut.

2.1.1 Kura-kura dan Kelinci pada Bubble Sort

Dalam algoritma Bubble Sort ini, terdapat beberapa ciri khas yang cukup menonjol, Ciri khas dari algoritma Bubble Sort ini adalah cepatnya elemen-elemen besar menempati posisi yang tepat dan lambatnya elemen-elemen yang lebih kecil dalam menempati posisi yang tepat. Hal ini dapat ditunjukkan pada contoh data "9 2 4 1" yang akan diurutkan berikut ini menggunakan algoritma Bubble Sort.

Pass Pertama
(**9 2** 4 1) menjadi (**2 9** 4 1)
(2 **9 4** 1) menjadi (2 **4 9** 1)
(2 4 **9 1**) menjadi (2 4 **1 9**)

Pass Kedua
(**2 4** 1 9) menjadi (**2 4** 1 9)
(2 **4 1** 9) menjadi (2 **1 4** 9)
(2 1 **4 9**) menjadi (2 1 **4 9**)

Pass Ketiga
(**2 1** 4 9) menjadi (**1 2** 4 9)
(1 **2 4** 9) menjadi (1 **2 4** 9)
(1 2 **4 9**) menjadi (1 2 **4 9**)

Pass Keempat
(**1 2** 4 9) menjadi (**1 2** 4 9)
(1 **2 4** 9) menjadi (1 **2 4** 9)
(1 2 **4 9**) menjadi (1 2 **4 9**)

Dari proses pengurutan di atas, dapat dilihat bahwa elemen terbesar, "9", langsung menempati posisi akhir pada *pass* pertama. Akan tetapi elemen terkecil, "1", baru menempati posisi pertama pada *pass* keempat, yaitu *pass* yang terakhir.

Oleh karena itu, muncullah istilah "kura-kura" dan "kelinci" dalam algoritma Bubble Sort. Pada contoh di atas, "1" berperan sebagai "kura-kura", sedangkan "9" berperan sebagai "kelinci".

Fenomena "kura-kura dan kelinci" ini sering kali mengakibatkan proses pengurutan menjadi lama, terutama elemen "kura-kura". Hal ini disebabkan oleh "kura-kura" membutuhkan satu kali *pass* hanya untuk bergeser posisi ke sebelah kiri.

2.2 Implementasi dalam Pseudo-Code

Setiap algoritma akan memiliki implementasi yang berbeda, tergantung dari bahasa program yang dipakai. Oleh karena itu berikut ini adalah pseudo-code dari algoritma bubblesort, untuk memudahkan implementasi bubblesort pada bahasa apapun.

```
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
```

2.3 Kompleksitas Algoritma Bubble Sort

Kompleksitas Algoritma Bubble Sort dapat dilihat dari beberapa jenis kasus, yaitu *worst-case, average-case,* dan *best-case*.

2.3.1 Kondisi *Best-Case*

Dalam kasus ini, data yang akan disorting telah terurut sebelumnya, sehingga proses perbandingan hanya dilakukan sebanyak (n-1) kali, dengan satu kali pass. Proses perbandingan dilakukan hanya untuk memverifikasi keurutan data. Contoh *Best-Case* dapat dilihat pada pengurutan data "1 2 3 4" di bawah ini.

Pass Pertama
(**1 2** 3 4) menjadi (**1 2** 3 4)
(1 **2 3** 4) menjadi (1 **2 3** 4)
(1 2 **3 4**) menjadi (1 2 **3 4**)

Dari proses di atas, dapat dilihat bahwa tidak terjadi penukaran posisi satu kalipun, sehingga tidak dilakukan pass selanjutnya. Perbandingan elemen dilakukan sebanyak tiga kali.

Proses perbandingan pada kondisi 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.

2.3.2 Kondisi *Worst-Case*

Dalam kasus ini, data terkecil berada pada ujung *array*. Contoh *Worst-Case* dapat dilihat pada pengurutan data "4 3 2 1" di bawah ini.

Pass Pertama
(**4 3** 2 1) menjadi (**3 4** 2 1)
(3 **4 2** 1) menjadi (3 **2 4** 1)
(3 2 **4 1**) menjadi (3 2 **1 4**)

Pass Kedua
(**3 2** 1 4) menjadi (**2 3** 1 4)
(2 **3 1** 4) menjadi (2 **1 3** 4)
(2 1 **3 4**) menjadi (2 1 **3 4**)

Pass Ketiga
(**2 1** 3 4) menjadi (**1 2** 3 4)
(1 **2 3** 4) menjadi (1 **2 3** 4)
(1 2 **3 4**) menjadi (1 2 **3 4**)

Pass Keempat
(**1 2** 3 4) menjadi (**1 2** 3 4)
(1 **2 3** 4) menjadi (1 **2 3** 4)
(1 2 **3 4**) menjadi (1 2 **3 4**)

Dari langkah pengurutan di atas, terlihat bahwa 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.

$$Jumlah\ proses = n^2+n \quad (3)$$

Dalam persamaan (3) di atas, *n* adalah jumlah elemen yang akan diurutkan. Sehingga notasi *Big-O* yang didapat adalah $O(n^2)$. Dengan kata lain, pada kondisi *worst-case*, algoritma Bubble Sort termasuk dalam kategori algoritma kuadratik.

2.3.3 Kondisi *Average-Case*

Pada kondisi average-case, jumlah pass ditentukan dari elemen mana yang mengalami penggeseran ke kiri paling banyak. Hal ini dapat ditunjukkan oleh proses pengurutan suatu *array,* misalkan saja (1 8 6 2). Dari (1 8 6 2), dapat dilihat bahwa yang akan mengalami proses penggeseran paling banyak adalah elemen 2, yaitu sebanyak dua kali.

Pass Pertama
(**1 8** 6 2) menjadi (**1 8** 6 2)
(1 **8 6** 2) menjadi (1 **6 8** 2)
(1 6 **8 2**) menjadi (1 6 **2 8**)

Pass Kedua
(**1 6** 2 8) menjadi (**1 6** 2 8)
(1 **6 2** 8) menjadi (1 **2 6** 8)
(1 2 **6 8**) menjadi (1 2 **6 8**)

Pass Ketiga
(**1 2** 6 8) menjadi (**1 2** 6 8)
(1 **2 6** 8) menjadi (1 **2 6** 8)
(1 2 **6 8**) menjadi (1 2 **6 8**)

Dari proses pengurutan di atas, dapat dilihat bahwa untuk mengurutkan diperlukan dua buah *passing*, ditambah satu buah *passing* untuk memverifikasi. Dengan kata lain, jumlah proses perbandingan dapat dihitung sebagai berikut.

$$Jumlah\ proses = x^2+x \quad (4)$$

Dalam persamaan (4) di atas, x adalah jumlah penggeseran terbanyak. Dalam hal ini, x tidak pernah lebih besar dari n, sehingga x dapat dirumuskan sebagai

Dari persamaan (4) dan (5) di atas, dapat disimpulkan bahwa notasi big-O nya adalah $O(n^2)$. Dengan kata lain, pada kondisi *average case* algoritma Bubble Sort termasuk dalam algoritma kuadratik.

2.4 Kelebihan dan Kekurangan Algoritma Bubble Sort

Setiap algoritma memiliki kelebihan dan kekurangannya masing-masing, demikian pula dengan algoritma Bubble Sort. Kelebihan dan kekurangan dari algoritma Bubble Sort dapat dilihat dari karakteristik algoritma Bubble Sort itu sendiri. Berikut ini adalah

beberapa kelebihan dan kekurangan dari algoritma Bubble Sort.

2.4.1 Kelebihan Bubble Sort

Beberapa kelebihan dari algoritma Bubble Sort adalah sebagai berikut :

- Algoritma yang simpel.
- Mudah untuk diubah menjadi kode.
- Definisi terurut terdapat dengan jelas dalam algoritma.
- Cocok untuk pengurutan data dengan elemen kecil telah terurut.

Algoritma yang simpel. Hal ini dilihat dari proses pengurutan yang hanya menggunakan rekurens dan perbandingan, tanpa penggunaan proses lain. Algoritma pengurutan lain cenderung menggunakan proses lain, misalnya proses partisi pada algoritma *Quick Sort*.[4]

Mudah untuk diubah menjadi kode. Hal ini diakibatkan oleh simpelnya algoritma Bubble Sort, sehingga kecil kemungkinan terjadi kesalahan sintax dalam pembuatan kode.

Definisi terurut terdapat dengan jelas dalam algoritma. Definisi terurut ini adalah tidak adanya satu kalipun *swap* pada satu kali *pass*. Berbeda dengan algoritma lain yang seringkali tidak memiliki definisi terurut yang jelas tertera pada algoritmanya, misalnya *Quick Sort* yang hanya melakukan partisi hingga hanya ada dua buah nilai yang bisa dibandingkan.

Cocok untuk pengurutan data dengan elemen kecil telah terurut. Algoritma Bubble Sort memiliki kondisi *best case* dengan kompleksitas algoritma O(n).

2.4.2 Kekurangan Bubble Sort

Beberapa kekurangan dari algoritma Bubble Sort adalah sebagai berikut :

- Tidak efektif dalam pengurutan data berskala besar.
- Langkah pengurutan yang terlalu panjang.

Kekurangan terbesar dari Bubble Sort adalah kompleksitas algoritma yang terlalu besar, baik dalam *average case* maupun *worst case*, yaitu $O(n^2)$, sehingga seringkali disebut sebagai algoritma primitif, *brute-force*, maupun algoritma naïf.[1] Untuk 1000 buah data misalnya, maka akan terjadi proses tidak lebih dari satu juta proses perbandingan.

Kompleksitas yang besar ini juga seringkali membuat algoritma Bubble Sort sebagai "*the general bad algorithm*".[2]

Bahkan, diantara algoritma pengurutan lain yang memiliki kompleksitas algoritma $O(n^2)$, *insertion sort* cenderung lebih efisien.

2.4.2 Modifikasi Bubble Sort

Akibat dari ketidakefektifan algoritma Bubble Sort, muncul berbagai cara agar algoritma Bubble Sort lebih efisien. Dari berbagai cara ini muncul variasi-variasi baru dari Bubble Sort, beberapa diantaranya adalah:

- Modifikasi algoritma bubble sort
- Cocktail sort
- Comp sort

Modifikasi algoritma bubble sort dilakukan sehingga *pseudocode* menjadi seperti berikut.

```
procedure bubbleSort( A : list of
sortable items ) defined as:
  n := length( A )
  do
    swapped := false
    for each i in 0 to n - 1  inclusive
do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
        swapped := true
      end if
    end for
    n := n - 1
  while swapped
end procedure
```

Proses algoritma yang terjadi akan menjadi

$$T(n)=n(n-1)/2 \qquad (5)$$

Pada proses di atas, kompleksitas algoritma tetap $O(n^2)$. Akan tetapi, pada *worst case*, waktu pemrosesan akan berjalan dua kali lebih cepat daripada algoritma bubble sort biasa.

Pada *cocktail sort*, *pseudocode*-nya adalah sebagai berikut.

```
procedure cocktailSort( A : list of
sortable items ) defined as:
  do
    swapped := false
    for each i in 0 to length( A ) - 2
do:
      if A[ i ] > A[ i + 1 ] then //
test whether the two elements are in
the wrong order
        swap( A[ i ], A[ i + 1 ] ) //
let the two elements change places
        swapped := true
      end if
    end for
    if swapped = false then
      // we can exit the outer loop
here if no swaps occurred.
      break do-while loop
    end if
    swapped := false
    for each i in length( A ) - 2 to 0
do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
```

```
        swapped := true
      end if
    end for
  while swapped // if no elements have
been swapped, then the list is sorted
end procedure
```

Pada algoritma *cocktail-sort*, ide dasarnya adalah dalam satu *pass,* elemen terkecil dan terbesar akan berada pada tempat yang tepat. Setelah itu, algoritma diulang tanpa melibatkan elemen awal dan akhir. Hal ini menghilangkan "kura-kura" dan mengakibatkan jumlah operasi berkurang. Akan tetapi, pada *worst case,* algoritma ini masih memiliki kompleksitas algoritma $O(n^2)$.

Sedangkan pada *comb sort, pseudocode*-nya adalah sebagai berikut.

```
function combsort(array input)
    gap := input.size //initialize gap
size

    loop until gap <= 1 and swaps = 0
        //update the gap value for a
next comb. Below is an example
        gap := int(gap / 1.25)

        i := 0
        swaps := 0 //see bubblesort for
an explanation

        //a single "comb" over the
input list
        loop until i + gap >=
input.size //see shellsort for similar
idea
            if input[i] > input[i+gap]
                swap(input[i],
input[i+gap])
                swaps := 1 // Flag a
swap has occurred, so the
                           // list is
not guaranteed sorted
            end if
            i := i + 1
        end loop

    end loop
end function
```

Ide dari combsort adalah perbandingan antara elemen tidak dengan elemen sebelahnya, namun dimulai dengan gap sebesar panjang list data yang akan diurut, dibagi dengan suatu faktor. Faktor itu disebut *shrink factor*.

Sebagai contoh, suatu list dengan jumlah elemen 7, maka dengan *shrink factor* sebesar 1.3, masing –masing gap adalah 5,3,2,1. Dengan kata lain, pada awalnya elemen ke-1 dibandingkan dengan elemen ke-6, kemudian dilihat apakah ditukar atau tidak. Setelah itu ulangi dengan melakukan *sorting* dengan gap 3, kemudian 2, 1, dan seterusnya. Hasil kompleksitas algoritma *worst case* dari *comb sort* adalah O(n log n).

3. KESIMPULAN

Algoritma BubbleSort adalah algoritma yang simpel dan mudah dipelajari, selain itu memiliki definisi terurut yang jelas dalam algoritmanya. Algoritma ini juga memiliki ciri khas, yaitu "kura-kura dan kelinci".

Akan tetapi, algoritma BubbleSort memiliki kelemahan, yaitu kompleksitas algoritma $O(n^2)$ pada *average case* dan *worst case*, sehingga menjadikan algoritma ini tidak efektif dalam pengurutan.

Oleh karena itu, banyak diciptakan variasi BubbleSort, mulai dari modifikasi algoritma hingga penambahan langkah baru dalam bentuk *comb sort* dan *cocktail sort*.

REFERENSI

[1] http://www.cs.duke.edu/~ola/papers/bubble.pdf tanggal akses 14 dan 20 Desember 2009.

[2] http://www.jargon.net/jargonfile/b/bogo-sort.html tanggal akses 14 dan 20 Desember 2009.

[3] Knuth, Donald. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 106–110 of section 5.2.2: Sorting by Exchanging.

[4] http://www.informatika.org/~rinaldi/Matdis/2008-2009/Makalah2008/Makalah0809-019.pdf tanggal akses 14 dan 20 Desember 2009.