Analisis Kompleksitas Waktu Beberapa Algoritma *Sorting*

Hayyu' Luthfi Hanifah 13512080[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13512080@std.stei.itb.ac.id

Abstrak*—Makalah ini berisi analisis kompleksitas waktu dari beberapa algoritma Sorting, *yaitu* Selection Sort, Insertion Sort, Quick Sort, *dan* Bubble Sort. *Notasi* Big-O *dari setiap algoritma tersebut akan dipaparkan agar dapat diketahui algoritma* sorting *mana yang paling mangkus di antara keempat algoritma tersebut.

***Kata kunci*— algoritma, kompleksitas, notasi Big-O, *sorting*.**

I. PENDAHULUAN

Kemangkusan suatu program sangat ditentukan oleh algoritmanya. Algoritma yang bagus bukan saja harus dapat menyelesaikan persoalan dengan benar (sesuai spesifikasi yang diminta) akan tetapi juga harus dapat meminimalisir pemakaian waktu dan memori dalam memproses suatu masukan dari pengguna. Algoritma seperti itulah yang disebut dengan algoritma yang mangkus (efisien).

Seorang *programmer* selalu memiliki beberapa alternatif algoritma yang bisa digunakan untuk menyelesaikan suatu permasalahan. Untuk menyelesaikan permasalahan *sorting* sendiri ada belasan algoritma. Akan tetapi, seperti yang telah disebutkan di atas, jika ingin menghasilkan program yang mangkus, *programmer* harus memilih algoritma yang mangkus dan sesuai dengan permasalahan yang ingin dipecahkannya.

II. DASAR TEORI

A. Kompleksitas Algoritma

Kompleksitas algoritma merupakan besaran yang dapat digunakan untuk menentukan tingkat kemangkusan suatu algoritma. Ada dua hal yang perlu dipertimbangkan untuk mengukur kompleksitas suatu algoritma, yaitu waktu dan ruang (memori) yang dibutuhkan oleh algoritma tersebut untuk memproses masukan dengan ukuran tertentu.

Kompleksitas waktu dari suatu algoritma tidak diukur dengan cara menjalankan program yang menggunakan algoritma tersebut lalu menghitung waktu yang diperlukan oleh program tersebut untuk menghasilkan solusi (dalam satuan detik, menit, jam, dll). Jika dilakukan dengan cara seperti itu, nilai kompleksitas waktu dari satu algoritma bisa jadi berbeda antara pengukuran di satu komputer dengan komputer yang lain. Hal tersebut bisa terjadi karena perbedaan arsitektur komputer dan *compiler* dapat memengaruhi waktu eksekusi suatu program.

Kompleksitas waktu dari suatu algoritma diukur dengan menghitung banyaknya operasi yang khas pada suatu algoritma. Misal, kompleksitas waktu dari algoritma *searching* diukur dengan menghitung banyaknya operasi pembandingan yang terjadi di dalam eksekusi algoritma tersebut untuk masukan dengan ukuran tertentu sedangkan kompleksitas waktu algoritma *sorting* diukur dengan menghitung banyaknya operasi pembandingan dan penukaran. Notasi untuk menyatakan kompleksitas waktu algoritma adalah T(n). Biasanya, kompleksitas waktu suatu algoritma diukur untuk kondisi terbaik (*best case*), kondisi terburuk (*worst case*), dan kondisi rata-rata (*average*).

Kompleksitas algoritma yang lain adalah kompleksitas ruang (memori). Kompleksitas ini diukur dari besarnya memori yang digunakan untuk menjalankan suatu algoritma dengan ukuran masukan tertentu. Kompleksitas ruang algoritma sangat dipengaruhi oleh tipe struktur data yang digunakan pada algoritma tersebut. Notasi untuk menyatakan kompleksitas ruang algoritma adalah S(n).

Dalam makalah ini, kompleksitas algoritma yang akan dibahas hanyalah kompleksitas waktu dari beberapa algoritma *sorting*.

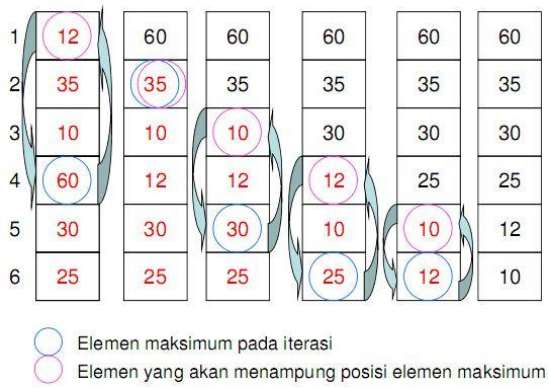
B. Algoritma Sorting

Masalah *sorting* (pengurutan) merupakan salah satu masalah yang sering muncul dan harus dipecahkan dalam dunia pemrograman, misalnya dalam menentukan *highscores* dalam suatu *game*, mengurutkan daftar pemain berdasarkan abjad untuk mempermudah dilakukannya pencarian (*searching*) nama pemain, dan lain sebagainya. Sampai saat ini, sudah ada belasan algoritma untuk menyelesaikan masalah *sorting* ini. Kasus terbaik bagi algoritma *sorting* adalah jika data yang diproses sudah terurut, sedangkan kasus terburuknya adalah data yang akan diproses terurut

secara terbalik. Algoritma *sorting* yang akan dibahas di dalam makalah ini adalah *Selection Sort*, *Insertion Sort*, *Quick Sort*, dan *Bubble Sort*.

1. *Selection Sort*

   Ide dari algoritma ini adalah mencari nilai ekstrim (maksimum atau minimum) dari suatu data, menukar nilai ekstrim tersebut dengan elemen pertama, lalu menandai elemen pertama tersebut sebagai elemen yang sudah terurut. Proses tersebut diulangi dengan meng-*exclude* elemen pertama dari data yang akan diproses hingga semua elemen data terurut (mengecil atau membesar). Untuk lebih jelasnya dapat dilihat dari ilustrasi berikut ini:



Gambar 1 Ilustrasi *Selection Sort*

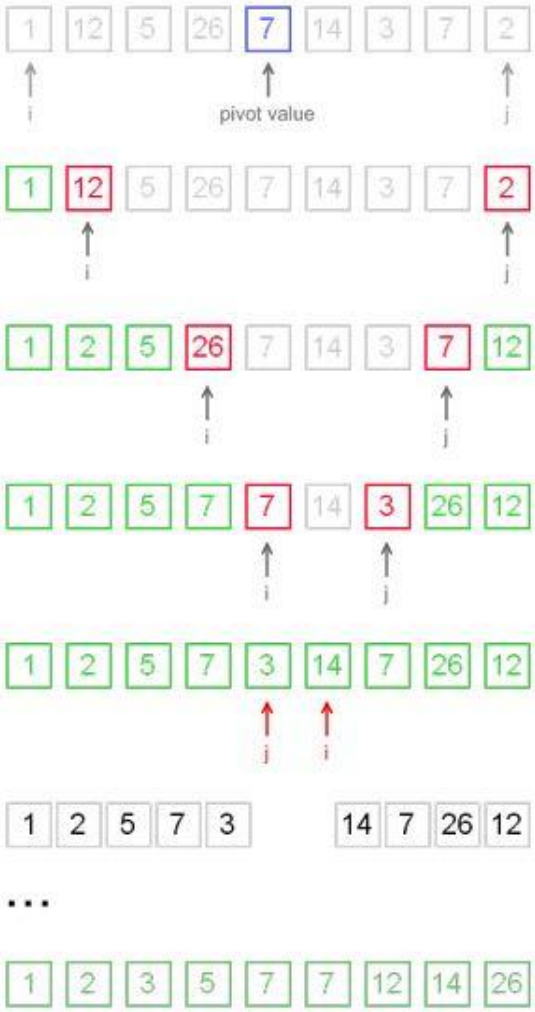
2. *Insertion Sort*

   Pengurutan menggunakan algoritma ini dilakukan dengan pertama-tama menganggap elemen pertama sudah terurut dan elemen-elemen sisanya belum terurut. Selanjutnya, elemen pertama dari elemen-elemen yang belum terurut diproses dengan cara menyisipkannya di tempat yang tepat (pada bagian data yang sudah terurut).

Gambar 2 Ilustrasi *Insertion Sort*

3. *Quick Sort*

   Quick Sort merupakan algoritma *sorting* yang bersifat rekursif. Awalnya, untuk membuat suatu data menjadi terurut membesar, algoritma ini memartisi data menjadi dua bagian dengan cara mengambil suatu nilai pivot (poros). Nantinya, data yang nilainya lebih kecil dari nilai pivot akan diatur sedemikian rupa agar terletak di sebelah kiri nilai pivot sedangkan data yang nilainya lebih besar dari nilai pivot terletak di sebelah kanan nilai pivot. Selanjutnya, lakukan *Quick Sort* pada masing-masing bagian (kanan dan kiri). Langkah rekursif ini terus dilakukan hingga semua data terurut membesar.



Gambar 3 Ilustrasi *Quick Sort*

4. *Bubble Sort*

   Algoritma ini melakukan pengurutan dengan cara setiap kali membandingkan elemen yang ‘bertetangga’ lalu menukar posisinya jika kedua elemen tersebut belum terurut. Setelah sampai di akhir data, algoritma ini akan kembali ke bagian awal data dan melakukan proses *Bubble Sort* lagi jika pada pemrosesan sebelumnya masih terjadi minimal satu kali penukaran elemen. Untuk lebih jelasnya dapat dilihat pada ilustrasi berikut ini:

Gambar 4 Ilustrasi *Bubble Sort*

C. Notasi Big-O

Notasi *Big-O* sering digunakan untuk menyatakan kompleksitas waktu suatu algoritma secara lebih sederhana, yaitu hanya dengan menyatakan ordenya (tanpa menyebutkan koefisiennya). Dengan menggunakan notasi *Big-O* untuk menyatakan kompleksitas waktu suatu algoritma, kita dapat dengan mudah membandingkan berbagai algoritma untuk menyelesaikan permasalahan yang sama dan menentukan algoritma mana yang paling mangkus.

Definisi:
T(n) = O(f(n)) bila terdapat konstanta C dan n_0 sedemikian sehingga T(n)≤C(f(n)) untuk n≥n_0.

Notasi *Big-O* dari T(n) dapat ditentukan hanya dengan melihat orde tertinggi dari T(n). Misal, notasi *Big-O* untuk T(n) = $2n^2$ + n -2 adalah O(n^2).

Berikut ini adalah tabel pengelompokan algoritma berdasarkan notasi *Big-O*:

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 *Big-O*

III. KOMPLEKSITAS WAKTU BEBERAPA ALGORITMA *SORTING*

A. Selection Sort

Contoh implementasi algoritma *Selection Sort* dengan bahasa C++ adalah sebagai berikut:

```
void selectionSort(int arr[], int n)
{
    int i, j, minIndex, tmp;
    for (i = 1; i < n; i++) {
        minIndex = i;
        for (j = i + 1; j <= n;
j++)
            if (arr[j] <
arr[minIndex])
                minIndex = j;
        if (minIndex != i) {
            tmp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = tmp;
        }
    }
}
```

Saat data yang diproses sudah terurut sesuai spesifikasi, algoritma ini hanya melakukan operasi pembandingan (tidak melakukan penukaran elemen). Banyaknya operasi pembandingan pada algoritma ini:

T(n) = (n-1) + (n-2) + … + 1 (deret aritmatika dengan beda=1, banyaknya suku = (n-1))

T(n) = n(n-1)/2

T(n) = O(n^2)

Sedangkan untuk kasus terburuk, yaitu saat data yang akan diproses terurut secara terbalik, maka selain melakukan operasi pembandingan, algoritma ini juga melakukan operasi penukaran. Banyaknya operasi penukaran pada algoritma ini:

T(n) = n-1 (kasus terburuk)

T(n) = O(n)

Jadi, kompleksitas waktu algoritma *Selection Sort* (untuk kasus terburuk) adalah:

O(T(n)) = maksimum(O(n^2),O(n))

O(T(n)) = O(n^2)

B. Insertion Sort

Contoh implementasi algoritma *Insertion Sort* dengan bahasa C++ adalah sebagai berikut:

```
void insertionSort(int arr[], int
length) {
    int i, j, tmp;
    for (i = 2; i <= length; i++) {
          j = i;
          while (j > 1 && arr[j - 1] >
arr[j]) {
                tmp = arr[j];
                arr[j] = arr[j - 1];
                arr[j - 1] = tmp;
                j--;
          }
    }
}
```

Algoritma ini mengasumsikan elemen pertama dari data yang diproses adalah elemen yang sudah terurut. Kemudian algoritma ini memproses elemen kedua, yaitu dengan terlebih dahulu membandingkan elemen tersebut dengan elemen-elemen yang sudah terurut lalu meletakkannya di tempat yang tepat. Jika data yang diproses sudah terurut sesuai dengan spesifikasi, maka algoritma ini hanya menjalankan operasi pembandingan. Sedangkan untuk data yang terurut secara terbalik operasi pembandingan dan penukaran dilakukan secara berselang-seling (penukaran dilakukan tepat setelah dua elemen dibandingkan hingga elemen yang diproses menemukan tempat yang tepat). Jadi, banyaknya operasi pembandingan dan penukaran pada algoritma ini sama, yaitu:

$T(n) = 1 + 2 + \ldots + (n-1)$ (deret aritmatika dengan beda=1, banyaknya suku = (n-1))

$T(n) = n(n-1)/2$

$T(n) = O(n^2)$

C. Quick Sort

Contoh implementasi algoritma *Quick Sort* dengan bahasa C++ adalah sebagai berikut:

```
void quickSort(int arr[], int left, int
right) {
      int i = left, j = right;
      int tmp;
      int pivot = arr[(left + right) /
2];
      /* partition */
      while (i <= j) {
            while (arr[i] < pivot)
                  i++;
            while (arr[j] > pivot)
                  j--;
            if (i <= j) {
                  tmp = arr[i];
                  arr[i] = arr[j];
                  arr[j] = tmp;
                  i++;
                  j--;
            }
      };
      /* recursion */
      if (left < j)
            quickSort(arr, left, j);
      if (i < right)
            quickSort(arr, i, right);
}
```

Algoritma ini adalah algoritma *sorting* rekursif yang menggunakan konsep *divide and conquer* (dalam hal ini membagi data yang akan di-*sorting* menjadi dua bagian, empat bagian, delapan bagian, dan seterusnya secara rekursif). Untuk kasus data yang terurut secara terbalik, algoritma *Quick Sort* ini bisa dikatakan lebih mangkus daripada *Selection Sort* dan *Insertion Sort*. Hal tersebut disebabkan oleh pemakaian metode *divide and conquer* pada algoritma *Quick Sort* ini. Kompleksitas waktu algoritma *Quick Sort* untuk kasus pembalikan urutan data adalah:

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

D. Bubble Sort

Contoh implementasi algoritma *Bubble Sort* dengan bahasa C++ adalah sebagai berikut:

```
void bubbleSort(int arr[], int n) {
  bool swapped = true;
  int j = 0;
  int tmp;
  while (swapped) {
        swapped = false;
        j++;
        for (int i = 0; i < n - j; i++)
{
              if (arr[i] > arr[i + 1])
{
                    tmp = arr[i];
                    arr[i] = arr[i + 1];
                    arr[i + 1] = tmp;
                    swapped = true;
              }
        }
    }
}
```

Untuk kasus terburuk, algoritma *Bubble Sort* melakukan operasi pembandingan dan penukaran sebanyak (n-1)+(n-2)+…+1. Maka, kompleksitas waktu untuk kasus terburuk algoritma ini adalah:

$T(n) = n(n-1)/2$

$T(n) = O(n^2)$

IV. KESIMPULAN

Dari keempat algoritma *sorting* yang dibahas di dalam makalah ini, dapat disimpulkan bahwa algoritma *Quick Sort* merupakan algoritma yang paling mangkus (dilihat dari kompleksitas waktunya) untuk menangani masalah pengurutan data. Secara lebih spesifik, algoritma *Quick Sort* lebih mangkus dibanding algoritma *sorting* yang lain jika digunakan untuk membalik urutan suatu data. Hal tersebut dapat dilihat dari notasi *Big-O* dari setiap algoritma *sorting* yang dibahas di sini:

Algoritma	Notasi *Big-O*
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Quick Sort	O(n log n)
Bubble Sort	$O(n^2)$

Tabel 2 Notasi *Big-O* untuk beberapa algoritma *sorting*

REFERENSI

Rosen, Kenneth H., Discrete Mathematics and Its Application, Sixth Edition, McGraw-Hill International 2007.

Munir, Rinaldi. Matematika Diskrit. Bandung, Penerbit Informatika Bandung, 2010.

Loudon, Kyle, Mastering Algorithms with C, O'Reilly & Associates, Inc.,1999.

Sedgewick, Robert, Algorithms (Second Edition), Addison Wesley Publishing Company, 1989.

http://www.algolist.net/Algorithms/Sorting/Selection_sort , 15 Desember 2013, 09:59 WIB.

http://www.algolist.net/Algorithms/Sorting/Insertion_sort , 15 Desember 2013, 09:59 WIB.

http://www.algolist.net/Algorithms/Sorting/Quicksort , 15 Desember 2013, 09:59 WIB.

http://www.algolist.net/Algorithms/Sorting/Bubble_sort , 15 Desember 2013, 14:57 WIB.

http://bigocheatsheet.com/ , 15 Desember 2013, 17:01 WIB.

Slide Kuliah KU1071: Array (Tabel) Pengurutan Nilai (Sorting), Semester I, 2012/2013.

PERNYATAAN

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

Bandung, 16 Desember 2013

Hayyu' Luthfi Hanifah
13512080