Kompleksitas Algoritma Pengurutan Selection Sort dan Insertion Sort

Setia Negara B. Tjaru (13508054)

Program Studi Teknik Informatika ITB
Bandung
e-mail: if18054@students.if.itb.ac.id

ABSTRAK

Makalah ini membahas tentang beberapa algoritma pengurutan yang biasa digunakan pada lingkungan akademisi. Pengurutan atau Sorting merupakan suatu proses mengatur susunan data-data menurut syarat tertentu. Meskipun pengurutan ini sepertinya hanya sebuah masalah klasik dalm keinformatikaan, namun perannya tidak dapat dipisahkan terutama dalam pengolahan data. Suatu pengolahan data biasanya akan lebih efisien jika datanya telah terurut, seperti *Binary Search* misalnya. Mengingat pentingnya pengurutan dalam penggunaannya dalam hal keinnformatikaan, maka perlu diketahui algoritma mana yang sebenarnya paling efisien untuk dipakai. Meskipun suatu algoritma pengurutan mempunyai kelebihan dan keterbatasan masing-masing, kompleksitas dan keefisiensiannya tetap harus dipertimbangkan.

Untuk menjelaskan masalah keefisiensian dari suatu algoritma, digunakanlah teori kompleksitas algoritma. Maka muncullah Big-Oh sebagai notasi yang melambangkan suatu nilai keefisiensian suatu algoritma.

Kata kunci: *Pengurutan, Sorting, Algoritma, Insertion Sort, Selection Sort, Kompleksitas, Big-Oh.*

1. PENDAHULUAN

Pengurutan adalah satu hal yang sangat penting dalam dunia keinformatikaan. Terutama dalam pengelolaan data. Sering kali, dengan pengurutan, proses pengelolaan data dapat dilakukan dengan lebih mudah dan efisien. *Binary Search* contohnya, pasti lebih efisien daripada algoritma pencarian biasa yang lebih konvensional. Namun kita dapat melakukan Binary Search jika data yang bersangkutan belum diurut terlebih dahulu.

Ada berbagai macam algoritma pengurutan. Namun hanya beberapa yang dipakai sebagai "pengenalan" terhadap siswa-siswa di suatu institusi. Di antaranya adalah Selection Sort dan Insertion Sort. Kedua algoritma Pengurutan ini lah yang menjadi fokus pembahasan pada makalah ini.

Untuk dapat mengetahui seberapa efisien suatu algoritma, dipakailah teori kompleksitas algoritma sebagai dasar kajian. Kompleksitas terbagi atas dua, yaitu kompleksitas waktu dan kompleksitas ruang. Kompleksitas cenderung tidak dibahas, karena hal tersebut berkenaan dengan struktur data yang digunakan untuk mengimplementasikan algoritma. Sementara topik tersebut di luar kajian Matematika Diskrit[1].

Kompleksitas Waktu, T(n), adalah jumlah operasi yang dilakukan untuk melaksanakan algoritma sebagai fungsidari ukuran masukan n. Maka, dalam mengukur kompleksitas waktu dihitunglah banyaknya operasi yang dilakukan oleh algoritma. Idealnya, kita memang harus menghitung semua operasi yang ada dalam suatu algoritma. Namun, untuk lasan praktis, cukup menghitung jumlah operasi abstrak yang mendasari suatu algoritma. Operasi abstrak ini disebut Operasi Dasar.

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 $n \geq n_0$,
maka kompleksitas dapat dinyatakan dengan

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

Terdapat 2 jenis penggunaan notasi Big O,
yaitu :
1. Infinite asymptotics
2. Infinitesimal asymptotics
Perbedaan kedua jenis penggunaan notasi ini

hanya pada aplikasi. Sebagai contoh, pada infinite asymptotics dengan persamaan

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

Untuk n yang besar, pertumbuhan T(n) akan sebanding dengan n_2 dan dengan mengabaikan suku yang tidak mendominasi kita, maka kita tuliskan

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

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

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

Kesalahannya memiliki selisih

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

2. PEMBAHASAN

2.1. Selection Sort

2.1.1. Konsep Selection Sort

Algoritma pengurutan sederhana salah satunya adalah *Selection Sort*. Ide dasarnya adalah melakukan beberapa kali *pass* untuk melakukan penyeleksian elemen struktur data. Untuk *sorting ascending* (menaik), elemen yang paling kecil di antara elemen-elemen yang belum urut, disimpan indeksnya, kemudian dilakukan pertukaran nilai elemen dengan indeks yang disimpan tersebut dengan elemen yang paling depan yang belum urut. Sebaliknya, untuk *sorting descending* (menurun), elemen yang paling besar yang disimpan indeksnya kemudian ditukar.

Selection Sort diakui karena kesederhanaan algoritmanya dan performanya lebih bagus daripada algoritma lain yang lebih rumit dalam situasi tertentu.

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.

2.1.2. Simulasi Selection Sort

Untuk lebih jelasnya, perhatikanlah simulasi *Selection Sort Ascending* berikut dengan menggunakan larik

5	1	43	27	6	18	33

Dalam satu kali *pass,* ditentukan elemen yang paling kecil di dalam bagian list yang belum urut. Elemen yang paling kecil ini, diwarnai merah . Untuk bagian larik yang telah diurutkan diberi warna biru.

Pass

0		5	1	43	27	6	18	33
1		1	5	43	27	6	18	33
2		1	5	43	27	6	18	33
3		1	5	6	27	43	18	33
4		1	5	6	18	43	27	33
5		1	5	6	18	27	43	33
6		1	5	6	18	27	33	43

Gambar 2. Simulasi Selection Sort

2.1.3. Notasi Algoritmik Selection Sort

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}
```

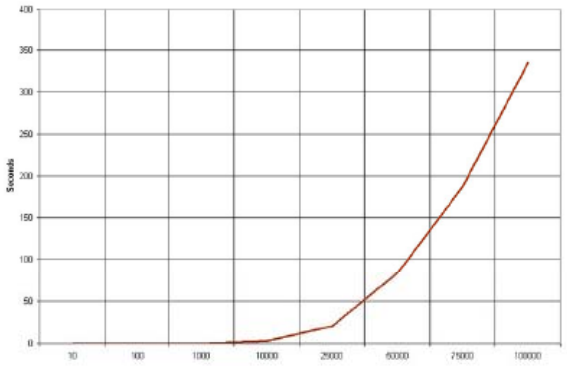
2.1.4. Kompleksitas *Selection Sort*

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:

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

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

Berarti kompleksitasnya secara simptotik adalah $O(n^2)$. Adapun grafik efisiensi selection sort dapat dilihat pada tabel dibawah ini:



Gambar 2. Grafik Kompleksitas Selection Sort

Pengurutan model ini tergolong buruk dan lebih baik dihindari penggunaannya, terutama untuk penanganan tabel dengan lebih dari 1000 elemen. Karena masih ada algoritma lain yang implementasinya sama mudahnya, namun performansinya jauh lebih baik. Algoritma yang dimaksud adalah *insertion sort.*

2.1.5 Stabilitas Selection Sort

Selection Sort ini pada dasarnya merupakan algoritma sorting yang tidak stabil, namun dapat diubah menjadi stabil pada kasus tertentu.

Algoritma sorting yang stabil akan mampu memanfaatkan relatifitas antar record melalui definisi di tiap-tiap keys yang dimiliki oleh record tersebut. Misalkan ada dua record R dan S dengan key yang sama dan dengan ketentuan R muncul sebelum S, maka pada hasil output akan muncul R sebelum S.

Namun ketika terdapat elemen yang sama (tidak dapat dibedakan) pada umumnya terdapat pada tipe data integer, stabilitas akan kembali di utamakan. Misal terdapat pasangan data berikut yang akan diurutkan berdasarkan komponen pertama:

(4, 1) (3, 7) (3, 1) (5, 6)

Pada kasus ini, akan menghasilkan dua output yang berbeda, dimana salah satunya akan memperhatikan key dari data dalam pengurutan dan solusi yang lain tidak.

(3, **7**) (3, **1**) (4, **1**) (5, **6**) (order maintained/ stable)

(3, **1**) (3, **7**) (4, **1**) (5, **6**) (order changed/ unstable)

Algoritma sorting yang **tidak stabil** akan mengubah keterhubungan antar record berdasarkan key yang dimiliki, namun **algoritma stabil** akan mengabaikan key tersebut. Bagaimanapun juga algoritma yang stabil tetap bisa diubah menjadi tidak stabil dengan membandingkan key yang dimilikki tiap-tiap recordnya.

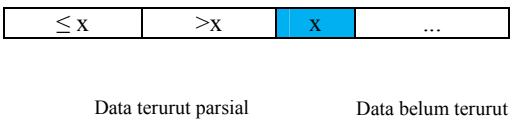
2.2. Insertion Sort

2.2.1. Konsep Insertion Sort

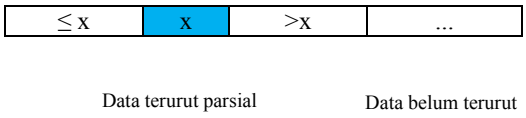
Algoritma *insertion sort* adalah sebuah algoritma sederhana yang cukup efisien untuk mengurutkan sebuah list yang hampir terurut. Algorima ini juga bisa digunakan sebagai bagian dari algoritma yang lebih canggih. Cara kerja algoritma ini adalah dengan mengambil elemen list satu-per-satu dan memasukkannya 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. Seperti sudah dibahas di bagian pendahuluan, salah satu implementasinya pada kehidupan sehari-hari adalah saat kita mengurutkan kartu remi. Kita ambil kartu satu-per-satu lalu membandingkan dengan kartu sebelumnya untuk mencari posisi yang tepat. Variasi pada umunya yang dilakukan terhadap array pada insertion sort adalah sebagai berikut :

- Elemen awal di masukkan sembarang, lalu elemen berikutnya dimasukkan di bagian paling akhir.
- 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.
- Setiap pergeseran akan mengganti nilai elemen berikutnya, namun hal ini tidak menjadi persoalan sebab elemen berikutnya sudah diproses lebih dahulu.

Ilustrasi:
Sebelum di-*insert:*



Setelah di- *insert:*



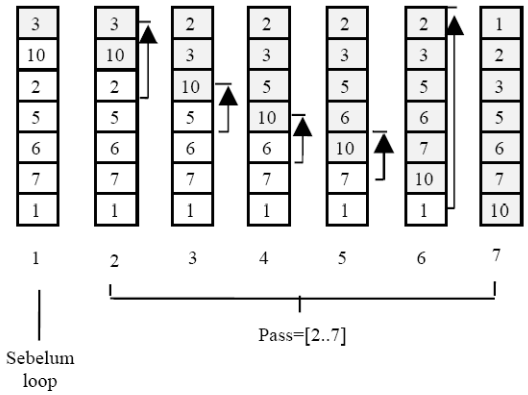
Insertion sort ini memiliki beberapa keuntungan :

- Implementasi yang sederhana
- Paling efisien untuk data berukuran kecil
- Merupakan *online algorithmic*, yang berarti bisa langsung melakukan sort setiap ada data baru
- Proses di tempat (memerlukan O(1) memori tambahan)
- Stabil.

Pada tahun 2004 Bender, Farach-Colton, and Mosteiro menemukan pengembangan baru dari algoritma ini, disebut library sort atau *gapped insertion sort* yang menggunakan beberapa gap kosong di sepanjang array. Dengan algoritma ini, pergeseran elemen dilakukan sampai gap tersebut dicapai. Algoritma ini cukup baik dengan kompleksitas O(n log n).

2.2.2 Simulasi Insertion Sort

Berikut ini adalah contoh dari simulasi Insertion Sort.



Gambar 3. Simulasi Insertion Sort

Setiap satu kali Pass akan ada satu nilai yang disisipkan. Kemudain pada Pass n-1 data akan terurut.

Data yang telah terurut diberi warna abu-abu. Panah menunjukkan perubahan posisi nilai yang akan di-*insert.*

2.2.3 Notasi Algoritmik Insertion Sort

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

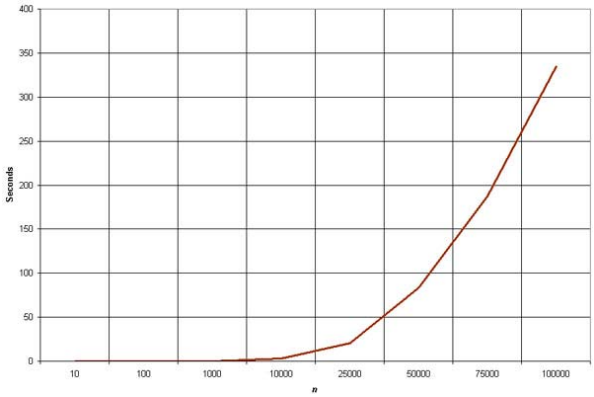
Algoritma:
Pass traversal [2..N]
    Temp ←TPass
    i ← pass - 1
    while (j ≥ 1) and (Ti > Temp) do
        Ti+1 ←Ti
        i ← i-1
depend on (T, i, Temp)
    Temp ≥ Ti : Ti+1 ← Temp
    Temp < Ti : Ti+1 ← Ti
    Ti ← Temp
    {T[1..Pass-1] terurut}

2.2.4 Kompleksitas Algoritma Insertion Sort

Algoritma *Insertion Sort* juga terdiri dari 2 kalang bersarang. Dimana 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.

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

Secara Kompleksitas, selection sort dan insertion sort mempunyai Big-Oh yang sama. Walaupun begitu, *insertion sort* sebenarnya lebih mangkus. Perhatikan gambar berikut:



Gambar 3. Grafik Kompleksitas Insertion Sort

Berdasarkan gambar, *Insertion Sort* 40% lebih cepat daripada *Selection Sort*. Namun, *Insertion Sort mempunyai kekurangan. Insertion Sort* lebih baik tidak digunakan untuk menangani struktur data dengan lebih dari 2000 elemen.

3. KESIMPULAN

Algoritma Pengurutan atau Sorting mempunyai berbagai macam jenis. Tiap-tiap algoritma pengurutan mempunyai kekurangan dan kelebihannya sendiri.

Pada makalah ini telah dibahas algoritma selection sort dan insertion sort. Khusus untuk selection sort dapat disimpulkan bahwa :

a. Kompleksitas *selection sort* relatif lebih kecil.
b. Kompleksitas algoritma selection sort adalah $O(n^2)$.
c. Tidak ada Best Case dan Worst Case karena $O(n^2)$ berlaku sama.
d. Pada dasarnya Selection Sort merupakan algoritma yang tidak stabil.

Untuk Insertion Sort, mempunyai beberapa keuntungan:
Implementasi yang sederhana

a. Paling efisien untuk data berukuran kecil
b. Merupakan *online algorithmic*, yang berarti bisa langsung melakukan sort setiap ada data baru
c. Proses di tempat (memerlukan O(1) memori tambahan)
d. Stabil.

Insertion Sort juga mempunyai $O(n^2)$.

Insertion Sort dan Selection Sort adalah metode pengurutan yang paling sederhana. Meskipun Insertion dan Selection Sort mempunyai Kompleksitas sama dengan $O(n^2)$, namun Insertion Sort lebih efisien jika kita melihat grafik pada pembahasan. Insertion Sort juga sudah memiliki alternatif algoritma yang ber-O(n log n).

REFERENSI

[1] Munir, Rinaldi. 2003. Diktat Kuliah IF2153 Matematika Diskrit. Program Studi Teknik Informatika, Institut Teknologi Bandung.

[2] Liem, Inggriani. 2007. Draft Diktat Kuliah Dasar Pemrograman (Bagian Pemrograman Prosedural).
Program Studi Teknik Informatika, Institut
Teknologi Bandung. hlm. 141-142

[3] Setyo, Danang dkk. 2008. Analisis Algoritma Selection Sort. Institut Teknologi Telkom. Bandung.

[4] Wikipedia, http://wikipedia.org/selection%sort. Diakses tanggal 20 Desember 2009.