Kompleksitas Algoritma *Quick Sort*

Fachrie Lantera – NIM: 13506099

Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Jln. Ganesha 10, Bandung
E-mail : if16099@students.if.itb.ac.id

Abstract *– Makalah ini membahas kompleksitas algoritma dari Quick Sort yang merupakan algoritma pengurutan. Dalam sebuah permasalahan dapat mempunyai banyak algoritma penyelesaian. Algoritma penyelesaian tersebut tidak harus benar, tetapi juga harus mangkus (efisien). Kemangkusan algoritma diukur dari waktu eksekusi dan kebutuhan ruang memori yang digunakan. Algoritma yang efisien adalah algoritma yang meminimumkakan waktu eksekusi dan kebutuhan ruang memori.*

Kata Kunci : *efisien, kompleksitas algoritma, kompleksitas waktu asimptotik.*

1. Pendahuluan

1.1 Kompleksitas Algoritma

Kita sering bertanya mengenai algortima mana yang lebih baik dalam menyelesaikan masalah tertentu. Untuk menjawab masalah di atas tentunya ada hal yang harus diukur supaya kita bisa menilai apakah algoritma tersebut lebih baik atau tidak.

Jika kita mencoba mengeksekusi program dengan algoritma A pada komputer C dan program pada algoritma B pada komputer D. Kita tidak dapat mengatakan algoritma A lebih baik dibandingkan dengan algoritma B hanya karena program dengan algoritma A jauh lebih cepat dieksekusi.

Komputer-komputer yang kita gunakan tidak semua memiliki arsitektur yang sama. Sehingga waktu komputasinya pun juga berbeda. Compiler bahasa pemrograman pun juga berbeda-beda dalam menghasilkan kode mesin. Sehingga pada kasus di atas, bisa jadi program dengan algoritma A jauh lebih cepat dieksekusi dikarenakan arsitektur komputer C lebih baik dibandingkan arsitektur komputer D.

Oleh karena itu kita memerlukan model abstrak pengukuran waktu/ruang yang bebas dari pertimbangan arsitektur komputer dan compiler bahasa pemrograman. Besaran yang dipakai untuk menerangkan model abstrak pengukuran waktu/ruang ini adalah *kompleksitas algoritma*.

Ada dua macam kompleksitas algoritma, yaitu : kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu disimbolkan dengan *T(n)* dan kompleksitas ruang *S(n)*. Kompleksitas waktu, *T(n)*, diukur dari jumlah tahapan komputasi yang dibutuhkan untuk menjalankkan algoritma sebagai fungsi dari ukuran masukan *n*. Komplesitas ruang, *S(n),* diukur dari memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari ukuran masukan *n*.

Dengan menggunakan besaran kompleksitas waktu/ruang algoritma, kita dapat menentukan *laju* peningkatan waktu (ruang) yang diperlukan algoritma dengan meningkatnya ukuran masukan *n*.[1]

Di dalam sebuah algoritma terdapat bermacam jenis operasi:

- Operasi baca/tulis,
- Operasi aritmetika (+, -, *, /)
- Operasi pengisian nilai (assignment)
- Operasi pengaksesan elemen larik
- Operasi pemanggilan fungsi/prosedur
- dll

Dalam praktek, kita hanya menghitung jumlah operasi khas (tipikal) yang mendasari suatu algoritma.

1.2 Kompleksitas Waktu Asimptotik

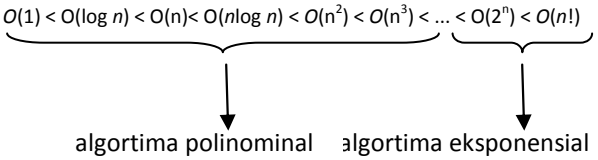
Terkadang kita tidak terlalu membutuhkan kompleksitas waktu yang detil dari sebuah algoritma. Yang kita butuhkan terkadang adalah besaran kompleksitas waktu yang menghampiri kompleksitas waktu yang sebenarnya. Kompleksitas waktu yang demikian disebut *kompleksitas waktu asimptotik* yang dinotasikan dengan "*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.

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

Urutan spektrum kompleksitas waktu algoritma adalah :



2 Algoritma Quick Sort

Quick sort juga disebut juga dengan partition Exchange sort. Disebut Quick Sort, karena terbukti mempunya 'average behaviour' yang terbaik di antara metoda pengurutan yang ada. Disebut Partition Exchange Sort karena konsepnya membuat partisi-partisi, dan sort dilakukan per partisi.

Teknik mempartisi tabel:

(i) pilih $x \in \{a1, a2, ..., an\}$ sebagai elemen pivot,
(ii) pindai (scan) tabel dari kiri sampai ditemukan elemen $a_p \geq x$
(iii) pindai tabel dari kanan sampai ditemukan elemen $a_q \leq x$
(iv) pertukarkan a_p <-> a_q
(v) ulangi (ii) dari posisi p+1, dan (iii) dari posisi q-1, sampai kedua pemindaian bertemu di tengah tabel.

Dalam algoritma quick sort , pemilihan pivot adalah hal yang menentukan apakah algoritma quick sort tersebut akan memberikan performa terbaik atau terburuk. Berikut beberapa cara pemilihan pivot :

1. Pivot = elemen pertama, elemen terakhir, atau elemen tengah tabel. Cara ini hanya bagus jika elemen tabel tersusun secara acak, tetapi tidak bagus jika elemen tabel semula sudah terurut. Misalnya, jika elemen tabel semula menurun, maka semua elemen tabel akan terkumpul di upa-tabel kana.
2. Pivot dipilih secara acak dari salah satu elemen tabel. Cara ini baik, tetapi mahal, sebab memerlukan biaya (cost) untuk pembangkitan prosedur acak. Lagi pula, ita tidak mengurangi kompleksitas waktu algoritma.
3. Pivot = elemen median tabel. Cara ini paling bagus, karena hasil partisi menghasilkan dua bagian tabel yang berukuran seimbang (masing-masing ≈ n/2 elemen). Cara ini memberkan kompleksitas waktu yang minimum. Masalahnya, mencari median dari elemen tabel yang belum terurut adalah persoalan tersendiri.

Berikut Pseudo-code Quick Sort :

```
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 upatabel a[i..q] dan a[q+1..j]
Keluaran upatabel a[i..q] dan upatabel 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+j) div 2] { pivot = elemen tengah }

  p <- i
  q <- j
```

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

    if  (p ≤ q) then
          { pertukarkan a[p] dengan a[q]}
       temp <- a[p]
       a[p] <- a[q]
       a[q] <- temp

{ tentukan awal pemindaian berikutnya}
       p <- p+ 1
       q <- q - 1
   endif

until p > q
```

Misalkan tabel yang akan diurut adalah berikut :

12	5	56	16	4	6	33

Langkah-langkah pempartisian tabel.

(i).

12 p	5	56	**16** pivot	4	6	33 q

(ii) & (iii)

12	5	56 p	**16** pivot	4	6 q	33

(iv)

12	5	6 p	**16**	4	56 q	33

(ii) & (iii)

12	5	6	**16** p	4 q	56	33

(iv)

12	5	6	**4** p	16 q	56	33

(ii) & (iii)

12	5	6	**4** q	16 p	56	33

(q<p, berhenti).

Hasil partisi pertama adalah :

Kiri : (< 16)

12	5	6	4

Kanan (≥ 16)

16	56	33

Jika dilanjutkan. Maka prosesnya adalah sebagai berikut:

- Rekursif untuk tabel partisi kiri a

12	5	6	4

Langkah-langkah pempartisian tabel

(i)

12 p	**5** pivot	6	4 q

(ii) & (iii)

12	**5** pivot	6	4 q

(iv)

4 p	**5** pivot	6	12 q

(ii) & (iii)

4 q	**5** pivot	6 p	12

(q<p, berhenti).

Kiri (< 5)

4

Kanan (≥ 5)

5	6	12

Rekursif untuk tabel partisi kanan a.1

(i)

(ii) & (iii)

←		→
5 p	6 q	12

(q<p, berhenti).

Kiri (<6)

5

Kanan (≥ 56)

6	12

Rekursif untuk tabel partisi kanan a.1.1

(i)

6 Pivot p	12 q

(ii) & (iii)

←	→
6 Pivot q	12 p

(q<p, berhenti).

6	12

- Rekursif untuk tabel partisi kanan b

(i)

16 p	56 pivot	33 q

(ii) & (iii)

→		←
16	56 p	33 q

(iv)

	→	←
16	33 p	56 q

(ii) & (iii)

5 p	6 pivot	12 q

1 6	3 3 q	5 6 p

(q<p, berhenti).

Kiri (< 5)

16

Kanan (≥ 56)

33	56

Rekursif Untuk Tabel Partisi Kanan b.1

Rekursif untuk tabel partisi kanan a.1.1

(i)

33 Pivot p	56 q

(ii) & (iii)

←	→
33 Pivot q	56 p

(q<p, berhenti).

33	56

4	5	6	12	16	33	56

Tabel telah terurut menaik.

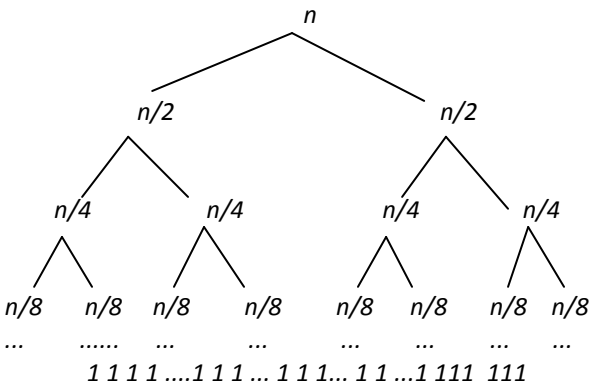
2.1 Kompleksitas Algoritma Quick Sort

Terdapat 3 kemungkinan kasus dari performa algoritma quick sort ini yaitu terbaik (best case), terburuk (worst case), dan rata-rata (average case).

2.1.1 Kasus Terbaik (Best Case)

Kasus terbaik terjadi bila pivot adalah elemen median sedemikian sehingga kedua upa-tabel berukuran relatif sama setiap kali pempartisian. Menentukan median tabel adalah persoalan tersendiri, sebab kita harus menentukan median dari tabel yang belum terurut.

Pohon berikut menggambarkan upa-tabel kiri dan upa-tabel kanan setiap kali pempartisian sampai menghasilkan tabel terurut :



Kompleksitas waktu pengurutan dihitung dari jumlah perbandingan elemen-elemen tabel :
T_{min}(n) = waktu partisi + waktu pemanggilan rekurens (Quick Sort untuk dua bagian tabel hasil partisi)

Kompleksitas prosedur partisi adalah t(n) = cn = *O*(*n*), sehingga kompleksitas algoritma quick sort menjadi (dalam bentuk relasi rekurens) :

$$T(n) = \begin{cases} a & ,n=1 \\ 2T\left(\frac{n}{2}\right)+cn & ,n>1 \end{cases}$$

Penyelesaian persamaan rekurens :

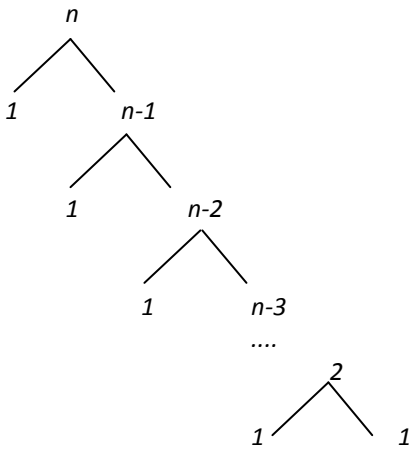
T(n) = 2T(n/2) + cn
= 2(2T(n/4) + cn/2) + cn = 4(T(n/4) + 2cn
= 4(2(T(n/8) + cn/4) + 2cn = 8T(n/8) + 3cn
= ..
= 2^k(T(n/2^k) + kcn

Persamaan terakhir dapat diselesaikan karena basis rekursif adalah ketika ukuran tabel = 1,
n/2^k = 1 → k = ^{2} log n
sehingga

T(n) = nT(1) + cn ^{2} log n
= na + c n ^{2}log n
= *O*(*n* ^{2}*log n*)

2.1.2 Kasus Terburuk (Worst Case)

Kasus ini terjadi bila pada setiap partisi pivot selalu elemen masksimum (atau elemen minimum) tabel. Hal ini menyebabkan pembagian menghasilkan upatabel kiri (atau kanan) berukuran satu elemen dan upatabel kanan (atau kiri) berukuran n – 1 elemen. Keadaan kedua upa tabel ini digambarkan sebagai pohon berikut:



Kompleksitas waktu pengurutan :

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

T(n) = cn + T(n-1)

= bn + { b.(n-1) + T(n-2) }
= bn + b(n-1) + {b(n-2) + T(n-3)}
=...
= b(n+(n-1)+(n-2)..+2) +a
=b{(n-1)(n+2)/2} + a
= bn^2/2 + bn/2 + ((a-b)
= *O*(*n*^{2})

2.1.3 Kasus Rata-Rata

Kasus ini terjadi jika pivot dipilih secara acak dari elemen tabel, dan peluang setiap elemen dipilih menjadi *pivot* adalah sama. Kompleksitas waktunya adalah $T_{avg}(n) = O(n^2 log\ n)$. Pembuktiannya lebih rumit.

3 Kesimpulan

Algoritma Quick Sort dapat kita ketahui sebagai algoritma yang handal dalam melakukan pengurutannya dari besarnya waktu asimptotik yang diperlukan apabila diberikan n buah masukan. Dimana kompleksitas algoritma dari algoritma ini memiliki 3 kasus yaitu *O*(*n* ^{2}*log n*) untuk kasus terbaik, *O*(*n*^{2}) untuk kasus terburuk, dan *O*(*n* ^{2}*log n*) untuk kasus

rata-rata. Kompleksitas tersebut dipengaruhi karena pemilihan pivot. Oleh karena itu proses pemilihan pivot perlu dipertimbangkan. Pivot yang terkadang digunakan yaitu elemen tengah dari tabel yang akan diurut.

DAFTAR PUSTAKA

[1] Munir, Rinaldi. (2006). Diktat Kuliah IF2153 Matematika Diskrit Edisi Keempat. Departemen Teknik Informatika, Institut Teknologi Bandung.

[2] Munir, Rinaldi. (2007). Diktat Kuliah IF2251 Strategi Algoritmik, Departemen Teknik Informatika, Institut Teknologi Bandung.

[3] Moh. Sjukani. (2007). Struktur Data (Algoritma & Struktur Data 2) dengan C, C++,Mitra Wacana Media. Jakarta