BAB III

ANALISA ALGORITMA PROSEDUR QUICKSORT BERREKURSIF

3.1. Running Time Program

Dalam penyelesaian suatu masalah, biasanya terdapat beberapa pilihan algoritma untuk menyelesaikan masalah tersebut. Atas dasar apakah sebuah algoritma dipilih dalam penyelesaian suatu masalah ?. Ada dua hal mendasar dari algoritma-algoritma yang sering saling bertentangan tujuan yaitu :

1. Algoritma yang mudah untuk dimengerti, mudah dalam pengkodean dan debugging (pengecekan kesalahan).
2. Algoritma yang efisien dalam penggunaan komputer , yang paling istimewa, terutama dapat dijalankan secepat mungkin.

Dasar yang pertama cocok jika program bertujuan untuk dijalankan hanya sekali serta program tersebut simpel sedangkan dasar yang ke dua cocok jika program bertujuan untuk dijalankan lebih dari satu kali serta program-program yang rumit. Berdasar kedua hal di atas maka ditentukan kompleksitas waktu dari eksekusi program, dengan diketahui kompleksitas waktu untuk jumlah input tertentu, maka efisiensi program dalam waktu dapat dibedakan menurut Running Timenya.

Definisi 3.1

Running Time adalah ukuran waktu dalam fungsi besar data / ukuran data untuk melaksanakan suatu program sehingga menghasilkan output pada suatu kompiler dan mesin eksekusi tertentu.

Penjelasan :

Dalam analisa algoritma, Running Time dinyatakan dalam simbol T(n). Running Time sebuah program tergantung beberapa faktor seperti :

1. Input program
2. Kualitas/kemampuan dari kompiler yang digunakan dalam melakukan kompilasi pada program.
3. Kemampuan dan kecepatan mesin yang digunakan untuk mengeksekusi program
4. Kompleksitas waktu program

3.2. Menghitung Running Time Program

Menghitung Running Time program, dalam prakteknya biasanya tidak mengalami kesulitan, karena hanya digunakan sedikit prinsip dasar. Sebelum disampaikan prinsip-prinsip ini akan kita pelajari tentang fungsi pertumbuhan yang merupakan dasar dari Running Time serta bagaimana untuk menjumlahkan dan mengalikan dalam notasi "big-oh".

3.2.1. Fungsi Pertumbuhan

Diberikan sebuah program komputer dengan input n bilangan bulat. Satu pertimbangan terpenting yang berkaitan dengan kegunaan program tersebut adalah berapa lama sebuah komputer menyelesaikan problem tersebut. Untuk menganalisa kegunaan pada program, dibutuhkan pengertian bagaimana kecepatan fungsi pertumbuhan pada n pertumbuhan. Notasi yang sering digunakan untuk menganalisa fungsi pertumbuhan disebut notasi **big-O.**

3.2.1.1. Notasi big-O

Fungsi pertumbuhan seringkali dideskripsikan dengan sebuah notasi khusus. Dengan mengikuti definisi dideskripsikanlah notasi tersebut.

Definisi 3.2

Diberikan f dan g suatu fungsi dari himpunan bilangan bilangan bulat atau himpunan bilangan real pada suatu himpunan bilangan real. Dikatakan f(x) adalah O(g(x)) jika terdapat sebuah konstanta C dan k sedemikian sehingga :

$$| f(x) | \leq C | g(x) |$$

dengan x > k. Dibaca f(x) adalah "**big-oh**" pada g(x).

Penjelasan :

Untuk menunjukkan f(x) adalah O(g(x)), cukup dengan menemukan satu pasangan konstanta C dan k sedemikian sehingga $| f(x) | \leq C | g(x) |$ jika x>k. Pasangan C, k yang memenuhi definisi tidak pernah tunggal. Selanjutnya, jika satu pasangan ada, maka terdapat tak terbatas pasangan yang lain. Sebuah cara sederhana untuk melihat hal tersebut adalah, jika C, k adalah satu pasangan, pasangan yang lain C′, k′ dengan C < C′ dan k < k′ juga memenuhi definisi, jika $| f(x) | \leq C | g(x) | \leq C' | g(x) |$ dengan x > k′ > k.

Contoh 1:

Tunjukkan bahwa $f(x) = x^2 + 2x + 1$ adalah $O(x^2)$.

Penyelesaian :

Jika $0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$ dengan x > 1, maka f(x) adalah $O(x^2)$.

Untuk menerapkan definisi pada notasi big-O, ambil C = 4 dan k=1. Pada permasalahan

ini tidak perlu menggunakan nilai absolut jika semua fungsi pada persamaan ini adalah positif ketika x bernilai positif.

Pendekatan yang lain adalah jika $x > 2$, maka $2x \leq x^2$. Akibatnya, jika $x>2$, terlihat bahwa : $0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2$. Dari definisi didapat $C = 3$ dan $k=2$.

Pengamatan pada relasi f(x) adalah $O(x^2)$, x^2 dapat diganti dengan sebarang fungsi yang lain dengan nilai yang lebih besar dari x^2 , misalnya f(x) adalah $O(x^3)$, f(x) adalah $O(x^2+2x+7)$ dan seterusnya. Hal ini juga benar bahwa x^2 adalah $O(x^2+2x+1)$, jika $x^2 < x^2+2x+1$ dengan $x > 1$. ■

Dari contoh tersebut diperoleh dua fungsi, $f(x) = x^2 + 2x + 1$ dan $g(x) = x^2$, sedemikian sehingga f(x) adalah O(g(x)) dan g(x) adalah O(f(x)) - pernyataan terakhir dari pertidaksamaan $x^2 \leq x^2 + 2x + 1$, untuk semua x bilangan real tidak negatif. Dikatakan bahwa fungsi f(x) dan g(x) sesuai dengan relasi big-O dengan order yang sama.

Penjelasan :

Kenyataan bahwa f(x) adalah g(x) kadang - kadang ditulis f(x) = O(g(x)). Bagaimanapun, tanda sama dengan pada notasi tersebut tidak dapat direpresentasikan dengan sebuah persamaan sesungguhnya. Selanjutnya, notasi tersebut menggambarkan bahwa pertidaksamaan yang diperoleh menghubungkan nilai pada fungsi f dan g untuk bilangan yang cukup besar pada domain fungsi tersebut.

Notasi big-O digunakan pada bidang matematika, khususnya pada ilmu

komputer untuk analisa algoritma. Matematikawan Jerman yang memperkenalkan pertama kali notasi big-O pada tahun 1892 pada sebuah buku tentang teori Bilangan. Notasi big-O kadang - kadang disebut juga dengan **Simbol Landau**.

Ketika f(x) adalah O(g(x)), dan h(x) adalah sebuah fungsi yang mempunyai nilai absolut lebih besar daripada g(x) untuk bilangan x yang cukup besar, mengikuti definisi di atas maka f(x) adalah O(h(x)). Dengan kata lain, fungsi g(x) pada relasi f(x) adalah O(g(x)) dapat direpresentasikan dengan sebuah fungsi dengan nilai absolute yang lebih besar.

$| f(x) | \leq C | g(x) |$ jika $x > k$

dan jika $| h(x) | > | g(x) |$ untuk semua $x > k$, maka

$| f(x) | \leq C | h(x) |$ jika $x > k$.

Oleh karena itu f(x) adalah O(h(x)).

Ketika notasi big-O digunakan, fungsi g pada relasi f(x) adalah O(g(x)) dipilih dari nilai terkecil yang mungkin. Kadang - kadang dari sebuah himpunan fungsi tertentu, misalnya fungsi tersebut pada bentuk x^n , dengan n adalah bilangan bilangan bulat positif.

Contoh 2:

Tunjukkan bahwa $7 x^2$ adalah $O(x^3)$.

Penyelesaian :

Pertidaksamaan $7 x^2 < x^3$ dengan $x > 7$, akan terlihat bila dilakukan pembagian kedua sisi pertidaksamaan dengan x^2. Oleh karena itu, $7 x^2$ adalah $O(x^3)$, dengan mengambil $C = 1$ dan $k = 7$ pada definisi notasi big-O. ■

Contoh 3:

Pada contoh 2 ditunjukkan bahwa $7x^2$ adalah $O(x^3)$. Apakah juga benar bahwa x^3 adalah $O(7x^2)$?

Penyelesaian :

Untuk menentukan apakah x^3 adalah $O(7x^2)$, perlu ditentukan apakah terdapat konstanta C dan k sedemikian sehingga $x^3 \leq C(7x^2)$ dengan $x > k$. Pertidaksamaan tersebut ekuivalen dengan pertidaksamaan $x < 7C$, yang dihasilkan dari pembagian kedua sisi dengan x^2. Tidak ada C jika x dibuat untuk sebarang bilangan yang lebih besar. Oleh karena itu x^3 bukan $O(7x^2)$. ■

Teorema 3.1

Diberikan $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, dengan $a_0 , a_1 , \ldots , a_{n-1} , a_n$ adalah bilangan real. Maka f(x) adalah $O(x^n)$.

Bukti :

Dengan menggunakan pertidaksamaan segitiga, jika $x > 1$, diperoleh :

$$\begin{aligned} |f(x)| &= |a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0| \\ &\leq |a_n| x^n + |a_{n-1}| x^{n-1} + \ldots + |a_1| x + |a_0| \\ &= x^n (|a_n| + |a_{n-1}| / x + \ldots + |a_1| / x^{n-1} + |a_0| / x^n) \\ &\leq x^n (|a_n| + |a_{n-1}| + \ldots + |a_1| + |a_0|). \end{aligned}$$

Terlihat bahwa

$$|f(x)| \leq C x^n$$

dengan $C = |a_n| + |a_{n-1}| + \ldots + |a_1| + |a_0|$ untuk $x > 1$. Oleh karena itu, f(x) adalah $O(x^n)$. ■

3.2.1.2. Kombinasi Fungsi Pertumbuhan

Banyak algoritma dibuat pada dua atau lebih sub prosedur. Jumlah langkah yang digunakan oleh komputer untuk menyelesaikan sebuah permasalahan dengan input / masukkan tertentu pada sebuah algoritma adalah jumlahan setiap langkah yang digunakan oleh sub prosedur tersebut. Untuk mendapatkan estimasi big-O pada jumlah langkah yang dibutuhkan, sangat perlu untuk menemukan estimasi big-O pada jumlah langkah yang digunakan pada setiap sub prosedur dan selanjutnya estimasi tersebut dikombinasikan.

Estimasi big-O pada kombinasi fungsi dapat diberikan jika akan mengambil ketelitian estimasi ketika estimasi big-O yang berbeda akan digabungkan. Khususnya, sangat perlu untuk mengestimasi pertumbuhan pada jumlahan dan perkalian dua fungsi.

Teorema 3.2

Diberikan $f_1(x)$ adalah $O(g_1(x))$ dan $f_2(x)$ adalah $O(g_2(x))$. Maka $(f_1 + f_2)(x)$ adalah $O(\max(g_1(x), g_2(x)))$. Selanjutnya disebut juga dengan **Maximum rule.**

Bukti :

Misalkan $f_1(x)$ adalah $O(g_1(x))$ dan $f_2(x)$ adalah $O(g_2(x))$, dari definisi 3.2, maka terdapat konstanta C_1 , C_2 , k_1 dan k_2 sedemikian sehingga :

$$| f_1(x) | \leq C_1 | g_1(x) |$$

ketika $x > k_1$ dan

$$| f_2(x) | \leq C_2 | g_2(x) |$$

ketika $x > k_2$. Untuk mengestimasi jumlahan pada $f_1(x)$ dan $f_2(x)$, perhatikan bahwa

$$| (f_1 + f_2)(x) | = | f_1(x) + f_2(x) |$$

$$\leq | f_1(x) | + | f_2(x) |$$

(dengan menggunakan pertidaksamaan segitiga $| a + b | \leq | a | + | b |$).

Ketika x lebih besar dari k_1 dan k_2, dengan mengikuti pertidaksamaan untuk $| f_1(x) |$ dan $| f_2(x) |$ maka :

$$| f_1(x) | + | f_2(x) | < C_1 | g_1(x) | + C_2 | g_2(x) |$$

$$\leq C_1 | g(x) | + C_2 | g(x) |$$

$$= (C_1 + C_2) | g(x) |$$

$$= C | g(x) |$$

dengan $C = C_1 + C_2$ dan $g(x) = \max(| g_1(x) | , | g_2(x) |)$.

maka , $| (f_1 + f_2)(x) | = | f_1(x) + f_2(x) |$

$$\leq | f_1(x) | + | f_2(x) |$$

$$\leq C | g(x) |$$

$$\leq C \max(| g_1(x) | , | g_2(x) |)$$

dengan $C = C_1 + C_2$ dan $g(x) = \max(| g_1(x) | , | g_2(x) |)$ serta $x > k$,

untuk $k = \max(k_1 , k_2)$.

Dari definisi 3.2 maka $(f_1 + f_2)(x)$ adalah $O(\max(g_1(x) , g_2(x)))$ ■

Seringkali dipunyai estimasi big-O untuk f_1 dan f_2 pada hubungan tersebut mempunyai fungsi yang sama yaitu g. Pada situasi ini, teorema 3.2 dapat digunakan

untuk menunjukkan bahwa ($f_1 + f_2$) (x) adalah juga O(g(x)), jika max (g(x) , g(x)) = g(x). Hasil tersebut ditetapkan sebagai akibat.

Akibat teorema 3.2

Diberikan $f_1(x)$ dan $f_2(x)$ keduanya adalah O(g(x)). Maka ($f_1 + f_2$) (x) adalah O(g(x)).

Teorema 3.3

Diberikan $f_1(x)$ adalah $O(g_1(x))$ dan $f_2(x)$ adalah $O(g_2(x))$. Maka (f_1 f_2) (x) adalah $O(g_1(x)g_2(x))$.

Bukti :

Dengan cara yang sama estimasi big-O dapat diperoleh dengan perkalian fungsi f_1 dan f_2 . Ketika x lebih besar daripada max (k_1 , k_2), maka :

$$
\begin{aligned}
| (f_1 f_2) (x) | &= | f_1(x) | \, | f_2(x) | \\
&\le C_1 \, | g_1(x) | \; C_2 \, | g_2(x) | \\
&\le C_1 \; C_2 \, | (g_1 \; g_2) (x) | \\
&\le C \, | (g_1 \; g_2) (x) |
\end{aligned}
$$

dengan $C = C_1 \; C_2$ dan k=max(k_1 , k_2), untuk x > k, dengan mengikuti definisi maka (f_1 f_2)(x) adalah $O(g_1 (x)g_2(x))$. ■

Tujuan menggunakan notasi big-O pada fungsi estimasi adalah untuk memilih sebuah fungsi g(x) pada pertumbuhan relatif terendah dengan f(x) adalah O(g(x)).

Contoh 7 :

Dapatkan estimasi big-O untuk $f(x) = (x + 1) \log(x^2 + 1) + 3 x^2$

Penyelesaian :

Pertama, estimasi big-O untuk $(x + 1) \log(x^2 + 1)$ akan diperoleh. Catatan ($x + 1$) adalah $O(x)$. selanjutnya, $x^2 + 1 \leq 2 x^2$ ketika $x > 1$. Sehingga,

$\log(x^2 + 1) \leq \log(2 x^2) = \log 2 + \log x^2 = \log 2 + 2 \log x \leq 3 \log x$, jika $x > 2$.

Ini terlihat bahwa $\log(x^2 + 1)$ adalah $O(\log x)$.

Dari teorema 3.3 bahwa $(x + 1) \log(x^2 + 1)$ adalah $O(x \log x)$. Untuk $3 x^2$ adalah $O(x^2)$, teorema 3.2 menunjukkan bahwa $f(x)$ adalah $\max(O(x \log x , x^2))$. Padahal $x \log x \leq x^2$, untuk $x > 1$, sehingga $f(x)$ adalah $O(x^2)$.

3.3. Kompleksitas Algoritma

Salah satu alat yang digunakan untuk mengetahui efisiensi sebuah algoritma adalah waktu yang digunakan oleh komputer untuk menyelesaikan sebuah permasalahan dengan menggunakan algoritma tersebut, ketika sebuah nilai input ukurannya telah dispesifikasikan.

Pernyataan tentang hal tersebut di atas termasuk dalam **Kompleksitas perhitungan** sebuah algoritma. Sebuah analisa waktu yang diinginkan untuk menyelesaikan sebuah permasalahan pada ukuran tertentu termasuk dalam **Kompleksitas waktu** sebuah algoritma. Sebuah analisa memory yang diinginkan termasuk dalam **Kompleksitas ruang** pada algoritma. Pertimbangan kompleksitas waktu dan ruang pada sebuah algoritma adalah penting ketika sebuah algoritma akan diimplementasikan. Jelas sekali, penting untuk diketahui ketika sebuah algoritma akan menghasilkan jawaban dalam mikrosecond, menit atau jutaan tahun. Demikian juga, memory yang diinginkan harus tersedia untuk menyelesaikan suatu permasalahan,

sehingga kompleksitas ruang harus dapat digunakan.

Kompleksitas waktu pada sebuah algoritma dapat diekspresikan pada jumlah operasi yang digunakan oleh algoritma ketika input mempunyai ukuran tertentu. Operasi - operasi yang digunakan untuk mengukur kompleksitas waktu dapat berupa perbandingan bilangan bulat, penambahan bilangan bulat, perkalian bilangan bulat, pembagian bilangan bulat ataupun sebarang operasi dasar lain.

Contoh :

Diberikan algoritma untuk mencari elemen maksimum sebagai berikut :

procedure $max(a_1, a_2, \ldots, a_n : bilangan\ bulat)$

$max := a_1$

for $i := 2$ **to** n

if $max < a_i$ **then** $max := a_i$

Penyelesaian :

Bilangan yang dibandingkan akan digunakan untuk mengukur kompleksitas waktu algoritma, karena perbandingan adalah operasi dasar yang digunakan.

Untuk menemukan elemen maksimum pada sebuah himpunan dengan n elemen, daftar mempunyai sebarang order, kadang - kadang elemen maksimum ada pada himpunan pertama sama dengan inisial pada daftar. Maka, setelah dibandingkan akan diperoleh bahwa akhir daftar bukan daerah jangkauan, kadang - kadang maksimum dan suku kedua dibandingkan, selanjutnya nilai maksimum pada elemen kedua jika ini yang terbesar. Prosedur ini bersambung, menggunakan dua penambahan perbandingan untuk setiap term pada daftar. Jika dua perbandingan digunakan untuk setiap selesai langkah

kedua pada n elemen dan satu lagi perbandingan digunakan untuk keluar dari loop ketika i = n + 1, tepat 2 (n - 1) + 1 = 2n - 1 perbandingan digunakan pada aplikasi algoritma tersebut. Sehingga, algoritma untuk menemukan elemen maksimum pada himpunan dengan n elemen mempunyai kompleksitas waktu O(n), yang merupakan ukuran pada term - term perbandingan bilangan yang digunakan.

Tabel 1 menunjukkan beberapa terminologi yang digunakan untuk mendeskripsikan kompleksitas waktu algoritma. Sebuah algoritma dikatakan mempunyai **Kompleksitas ekponensial** jika mempunyai waktu kompleksitas $O(b^n)$, dimana $b > 1$. Dengan cara yang sama, sebuah algoritma dengan waktu kompleksitas $O(n^b)$ dikatakan mempunyai **kompleksitas polynomial.**

Tabel 1

Terminologi yang digunakan untuk Kompleksitas Algoritma

Kompleksitas	Terminologi
O(1)	Kompleksitas Konstanta
O(log n)	Kompleksitas Logaritma
O(n)	Kompleksitas Linier
O(n log n)	Kompleksitas n log n
$O(n^b)$	Kompleksitas Polynomial
$O(b^n)$, dimana $b > 1$	Kompleksitas Eksponensial
O(n!)	Kompleksitas Faktorial

Diambil dari buku "Discrete Mathematics and its Applications", Kenneth H. Rossen, edisi tiga halaman 108

Sebuah estimasi big-O pada kompleksitas waktu algoritma menyatakan bagaimana waktu yang diinginkan untuk menyelesaikan permasalahan berubah sesuai dengan pertambahan ukuran input. Dalam prakteknya, estimasi terbaik yang digunakan (dengan fungsi referensi terkecil) dapat ditunjukkan. Estimasi big-O pada kompleksitas waktu tidak dapat diterjemahkan secara langsung ke dalam penjumlahan aktual waktu yang digunakan komputer. Satu alasan adalah bahwa estimasi big-O f(n) adalah O(g(n)), dengan f(n) adalah kompleksitas waktu sebuah algoritma dan g(n) adalah fungsi referensi, sehingga $f(n) \leq C\ g(n)$ ketika $n > k$, dengan C dan k adalah konstanta. Sehingga tanpa diketahui konstanta C dan k pada pertidaksamaan, estimasi ini tidak dapat digunakan untuk menentukan batas atas jumlah operasi yang digunakan. Selanjutnya, waktu yang diinginkan untuk suatu operasi tergantung pada tipe operasi dan komputer yang digunakan.

3.4. Analisa Algoritma

Jika ditemukan beberapa algoritma yang berbeda dalam menyelesaikan permasalahan yang sama, maka harus dipilih salah satu yang sesuai dengan kebutuhan. Alat yang paling utama untuk tujuan tersebut adalah *Analisa Algoritma*. Analisa algoritma dapat digunakan untuk menentukan efisiensi dari beberapa algoritma tersebut, dengan demikian dapat diambil keputusan terbaik untuk menggunakan suatu algoritma.

3.4.1. Analisa Struktur Kontrol

Analisa algoritma biasanya proses berjalannya secara terbalik. Pertama, menetapkan waktu yang diperlukan oleh instruksi tunggal (seringkali dibatasi dengan

konstanta), kemudian menggabungkan waktu tersebut berdasarkan struktur kontrol selanjutnya menggabungkan instruksi - instruksi pada program.

3.4.1.1. Barisan *(Sequencing)*.

Diberikan P_1 dan P_2 adalah dua penggalan pada sebuah algoritma. Mungkin merupakan instruksi tunggal atau sub algoritma yang rumit susunannya. Diberikan t_1 dan t_2 adalah waktu yang berturut - turut dimiliki oleh P_1 dan P_2. Waktu tersebut akan tergantung pada bermacam - macam parameter, seperti ukuran kejadian. **Aturan Sequencing** mengatakan bahwa waktu yang dibutuhkan untuk menghitung *"P_1 dan P_2"*, adalah jumlah waktu dari P_1 dan selanjutnya P_2 , secara sederhana $t_1 + t_2$. Dengan menggunakan aturan maksimum *(maximum rule)* , waktu tersebut adalah $O(\max(t_1,t_2))$.

3.4.1.2. Loop *"For"*

Loop **For** adalah loop termudah untuk dianalisa, dibandingkan dengan loop - loop yang lain.

for i ← 1 **to** m **do** P(i)

Misalkan loop tersebut bagian dari sebuah algoritma yang besar, bekerja dengan ukuran kejadian n. Terdapat P(i) yang merupakan statemen atau blok statemen yang berada di dalam loop. Kasus termudah adalah ketika waktu yang dibutuhkan oleh P(i) tidak tergantung pada i, meskipun dapat tergantung pada ukuran kejadian atau secara umum pada kejadiannya sendiri. Diberikan t yang merupakan waktu yang dibutuhkan untuk menghitung P(i). Pada kasus ini, jelas bahwa pada loop P(i) digunakan m kali, setiap waktu berharga t, dan total waktu yang dibutuhkan oleh loop adalah $t = m\ t$.

Loop **For** adalah yang terpendek dibandingkan dengan loop **While.** Bila loop for di atas diubah ke loop while adalah sebagai berikut :

$i \leftarrow 1$

while $i \leq m$ **do**

P(i)

$i \leftarrow i + 1$

Pada banyak keadaan, biasanya suatu harga unit test $i \leq m$, instruksi $i \leftarrow 1$ dan $i \leftarrow i+1$, dan operasi berantai (**go to**) sudah termasuk dalam loop **while**. Diberikan c adalah batas atas waktu yang dibutuhkan oleh setiap operasi. Waktu *t* yang digunakan oleh loop dengan batas di atas adalah

$$
\begin{aligned}
t \leq\ & c && \text{untuk } i \leftarrow 1 \\
& + (m+1)c && \text{untuk test } i \leq m \\
& + mt && \text{untuk eksekusi pada P(i)} \\
& + mc && \text{untuk eksekusi pada } i \leftarrow i+1 \\
& + mc && \text{untuk operasi berantai} \\
\leq\ & (t+3c)m + 2c.
\end{aligned}
$$

waktu *t* tersebut dengan jelas mempunyai batas bawah mt. Jika c diabaikan untuk dibandingkan dengan t, perkiraan sebelumnya bahwa *t* kira - kira sama dengan mt adalah bernilai benar, kecuali pada satu kasus tertentu : $t \approx mt$ adalah kesalahan kompleks ketika $m = 0$ (lebih buruk lagi jika m adalah negatif).

Analisa loop **for** akan lebih menarik ketika waktu yang dibutuhkan pada P(i) berubah - ubah berdasar fungsi pada i. Diberikan waktu tersebut adalah t(i). Secara

umum, waktu yang dibutuhkan P(i) tidak hanya tergantung pada i tetapi juga pada ukuran kejadian n. Jika waktu dalam loop kontrol diabaikan, maka waktu yang diperlukan loop **for** di atas sama dengan hasil penjumlahan dari waktu-waktu (t(1),t(2)..... t(m)) yang diambil oleh P(i) (P(1),P(2), ... , P(m)), sehingga dapat ditulis sebagai berikut :

$$\sum_{i=1}^{m} t(i)$$

3.4.1.3. Prosedur Rekursif

Analisa algoritma rekursif biasanya langsung, langkah demi langkah sampai akhir.

```
function Fibrec(n)
    if n < 2 then return n
    else return Fibrec(n-1)+Fibrec(n-2)
```

Diberikan T(n) adalah waktu yang diperlukan untuk memanggil Fibrec(n). Jika n < 2, algoritma dengan sederhana kembali ke n, dan memberikan waktu suatu konstanta a. Sebaliknya, jika n ≥ 2 sebagian besar pekerjaan berupa pemanggilan dua rekursif, dengan waktu berturut - turut T(n-1) dan T(n-2). Selain itu, juga satu penambahan yang melibatkan f_{n-1} dan f_{n-2}, juga kontrol pada rekursi serta test "**if** n < 2". Diberikan h(n) untuk menunjukkan kerja yang dilibatkan pada penambahan dan kontrol tersebut, dengan mengabaikan waktu yang dibutuhkan untuk memanggil Fibrec(n) itu sendiri dan waktu yang dipakai untuk memanggil dua rekursif . Dari ketentuan T(n) dan h(n) diperoleh

rekurensi:

$$T(n) \leq \begin{cases} a & \text{jika } n = 0 \text{ atau } n = 1 \\ T(n-1) + T(n-2) + h(n) & \text{yang lain} \end{cases}$$

3.4.1.4. Loop "While" dan "Repeat"

Loop **While** dan **Repeat** biasanya lebih sulit untuk dianalisa daripada loop **For** karena tidak ada langkah pasti untuk mengetahui berapa besar waktu yang dibutuhkan untuk melalui loop tersebut. Ada beberapa cara untuk menganalisa loop while maupun loop repeat, salah satu cara yang sederhana adalah dengan memperlakukan atau menganggap loop tersebut sebagai suatu algoritma rekursif dan bukan iteratif.

Akan diperlihatkan *binary search*. Yang selanjutnya akan dianalisa struktur kontrol loop **While.** Tujuan dari *binary search* adalah menemukan sebuah elemen x pada array T[1..n] mempunyai order tetap.

function Binary_search(T[1..n],x)

$i \leftarrow 1 ; j \leftarrow n$

while $i < j$ **do**

$\{T[i] \leq x \leq T[j]\}$

$k \leftarrow (i + j) / 2$

case $x < T[k] : j \leftarrow k - 1$

$x = T[k] : i, j \leftarrow k$ {**return** k}

$x > T[k] : i \leftarrow k + 1$

return i

Dengan mengimplementasikan / memperlakukan while loop sebagai suatu algoritma rekursif (bukan iteratif), masing-masing waktu putar loop direduksi ke range kemungkinan lokasi x pada array. Diberikan t(d) waktu maksimum yang dibutuhkan untuk memberhentikan loop while dengan $j - i + 1 \leq d$. Dapat dilihat bahwa nilai $j - i + 1$ adalah lebih kecil dari waktu yang diperlukan loop. Dalam suku-suku rekursif, dinyatakan bahwa t(d) adalah constanta waktu b yang diambil untuk menjalankan putaran yang pertama, ditambah waktu $t(d \div 2)$ yang dibutuhkan untuk menjalankan putaran selanjutnya sehingga cukup untuk memeberhentikan loop. Dengan mengambil konstanta waktu c untuk menunjukkan bahwa loop akan berhenti disaat $d = 1$, kita dapatkan persamaan rekurensi sebagai berikut :

$$T(d) \leq \begin{cases} c & \text{jika } d = 1 \\ b + T(d \div 2) & \text{yang lain} \end{cases}$$

3.4.2. Aturan Umum Analisa Algoritma

Secara umum, Running Time pada sebuah statemen atau kelompok statemen mungkin terparameterisasi dengan ukuran input dan atau dengan banyak variabel. Parameter yang diperbolehkan untuk Running Time pada keseluruhan program adalah n ukuran input. Aturan umum untuk analisa algoritma adalah sebagai berikut :

1. Running time setiap **assigment** (tugas), baca (**read**) dan statemen **Write** besarnya dapat diambil O(1).
2. Running Time pada barisan statemen ditentukan dengan aturan jumlahan yaitu bahwa Running Time pada barisan adalah tidak melebihi dari sebuah faktor konstan yang merupakan Running Time terbesar pada beberapa statement barisan.

3. Running Time pada statemen **if** adalah harga pada kondisi statemen eksekusi. Waktu menghitung kondisi secara normal adalah O(1). Waktu untuk **if then else** adalah waktu menghitung kondisi ditambah waktu terbesar yang dibutuhkan untuk statemen eksekusi jika kondisinya false (salah).
4. Waktu eksekusi adalah jumlah semua waktu sekitar loop, waktu eksekusi sekumpulan statemen dan waktu mengevaluasi kondisi untuk penghentian (biasanya yang terakhir adalah O(1)).

Untuk selanjutnya f(x) disebut pula dengan T(n) atau Running Time.

Contoh Analisa Algoritma Sorting, diberikan prosedur program sebagai berikut:

```
procedure select(T[1..n])
    for i ← 1 to n - 1 do
        minj ← i ; minx ← T[i]
        for j ← i + 1 to n do
            if T[j] < minx then minj ← j
                                minx ← T[j]
        T[minj] ← T[i]
        T[i] ← minx
```

Meskipun waktu yang diperlukan oleh setiap perputaran di dalam loop bukan konstanta, waktu tersebut adalah dibatasi oleh suatu konstanta c. Untuk setiap nilai i, instruksi-instruksi dalam loop terdalam dieksekusi sebanyak n - (i + 1) + 1 = n - i kali, dan selanjutnya waktu yang diberikan oleh loop terdalam adalah t(i) ≤ (n -

i) c . Waktu yang yang diberikan oleh putaran ke-i pada loop terluar dibatasi oleh b + t(i) untuk sebuah konstanta b memberikan jumlahan pada oprerasi - operasi dasar sebelum dan sesudah loop terdalam dan loop kontrol untuk loop terluar. Selanjutnya, total waktu yang diperlukan oleh algoritma di atas adalah

$$\sum_{i=1}^{n-1}(b+(n-i)\,c) = \sum_{i=1}^{n-1}(b+cn) - c\sum_{i=1}^{n-1} i$$

$$=(n-1)(b+cn)-cn(n-1)/2$$

$$= \frac{1}{2}cn^2 + \left(b-\frac{1}{2}c\right)n - b$$

adalah $O(n^2)$.

3.4.4. Algoritma Program Quicksort Berrekursif

Quicksort adalah suatu metode rekursif untuk mengurutkan suatu array A[1], A[2],.....,A[N] sehingga pada partisi pertama memenuhi kondisi berikut ini :

1. Kunci v berada pada posisi terakhir dalam array (jika v adalah data terkecil ke-j, maka ia berada dalam A[j]).
2. Semua elemen sebelah kiri A[j] adalah lebih kecil atau sama dengan A[j]. (Elemen-elemen ini yaitu A[1], A[2],.....,A[j-1] disebut "left subfile")
3. Semua elemen sebelah kanan A[j] adalah lebih besar atau sama dengan A[j]. (Elemen-elemen ini yaitu A[j+1], A[j+2],.....,A[N] disebut "right subfile")

Setelah partisi, masalah yang semula dari pengurutan array diubah menjadi masalah pengurutan left subfile dan right subfile.

Program berikut ini adalah algoritma dari metode Quicksort berrekursif, dengan proses partisi yang mudah dipahami dalam *pseudo lenguage*.

Program 1

```
procedure quicksort(integer value l,r);
(1)   comment Sort A[ l : r ] where A[ r + 1] >= A[ l ], ..... , A[ r ];
(2)   if r > l then
(3)       i := l ; j := r + 1; v := A[ l ]
(4)       loop;
(5)           loop: i := i + 1; while A[ i ] < v repeat;
(6)           loop: j := j - 1; while A[ j ] > v repeat;
(7)       until j < i;
(8)           A[ i ] :=: A[ j ];
(9)       repeat;
(10)      A[ l ] :=: A[ j ];
(11)      quicksort( l , j - 1);
(12)      quicksort( i , r);
(13)  endif;
```

penjelasan. :

Program ini menggunakan operator pertukaran atau swep :=:, dan statemen kontrol LOOP .....REPEAT dan IF..... ENDIF. Statemen yang terletak diantara LOOP dan REPEAT diiterasi: saat kondisi WHILE tidak dipenuhi, (atau kondisi

UNTIL dipenuhi) maka akan segera keluar dari **LOOP**. Kata "**REPEAT**" bisa diartikan sebagai " eksekusi kode yang diawali dari **LOOP** lagi", untuk contoh, "**UNTIL** j<i" bisa dibaca sebagai "jika j<i maka keluar dari **LOOP**".

Proses partisi mungkin paling mudah dimengerti dengan asumsi pertama bahwa kunci-kunci A[1],A[2],....., A[N] berbeda. Program dimulai dengan mengambil elemen paling kiri sebagai elemen partisi. Kemudian sisanya dibagi dengan mencari dari kiri ke kanan untuk elemen > v, dari kanan ke kiri untuk elemen < v, kemudian menukarkan elemen-elemen tersebut dan proses dilanjutkan sampai pointer-pointer saling berseberangan atau bersilangan. Loop berhenti pada saat j + 1= i, dengan diketahui bahwa A[*l*+1],.....,A[j] < v dan A[j+1],.....,A[r] > v, serta pertukaran A[*l*]:=:A[j] melengkapi proses dari partisi A[*l*],.....,A[r]. Kondisi bahwa A[r+1] harus lebih besar atau sama dengan semua kunci A[*l*],.....,A[r] dimasukkan untuk menghentikan pointer pada kasus v adalah yang terbesar. Prosedur memanggil Quikcsort (1,N) akan mengurutkan A[1],.....,A[N]. Jika A[N+1] diinisialisasi ke beberapa nilai paling tidak sebesar kunci lain (Biasanya ditulis dengan notasi A[N+1] := ~).

Jika kunci-kunci yang sama berada diantara A[1],.....,A[N], maka program (1) masih tetap berjalan dengan benar dan efisien, tetapi tidak persis dengan penjelasan di atas. Jika beberapa kunci sama dengan v sudah berada pada posisinya, maka pointer dapat mencari dan berhenti dengan i = j, sehingga setelah satu kali lagi melalui loop, akan berhenti dengan j+2 = i. Tetapi pada keadaan ini diketahui tidak hanya A[*l*+1],.....,A[j] <= v dan A[j+2],.....,A[r]>=v, tetapi juga diketahui A[j+1] = v.

Setelah pertukaran A[*l*] :=: A[j], didapatkan dua elemen pada tempat terakhir dalam array (A[j] dan A[j+1]), dan subfile yang diurutkan dengan rekursif.

3.4.5. Analisa Algoritma Program Quicksort Berrekursif

Berdasar pada algoritma Quicksort berrekursif beserta penjelasannya maka Running time program Quicksort dapat ditentukan sebagai berikut :

- Pada baris (1), adalah hanya berupa komentar yang menyatakan bahwa prosedur Quicksort akan mengurutkan data dari A[*l*] sampai dengan A[r], dengan A[r+1] >= A[*l*], A[*l*+1], ... , A[r]. Karena hanya berupa komentar maka baris (1) tidak dieksekusi, sehingga tidak mempengaruhi analisa algoritma program tersebut.
- Pada baris (2), terdapat statemen kondisi **if** , maka Running Timenya adalah harga pada kondisi statemen eksekusi. Jadi jika statemen kontrol terpenuhi maka Running Timenya adalah Running Time statemen atau blok statemen setelah **Then** dan jika statemen kontrol tidak terpenuhi maka Running Timenya adalah Running Time dari pengujian statemen kontrol yaitu O(1). Jika pengujian statemen kontrol terpenuhi maka program akan mengeksekusi baris (3) sampai dengan baris (12).
- Pada baris (3), hanya terdapat pemberian harga pada suatu variabel, yaitu pemberian harga i dengan *l*, j dengan r+1 dan v dengan A[*l*] maka Running Time baris (3) adalah O(1).

$$T(n) = O(\max(1,1,1)) = O(1)$$

- Pada baris (4), sesuai dengan *program1* dan *penjelasannya*, **LOOP** adalah awal dari suatu statemen kontrol yang diakhiri dengan **REPEAT**. Statemen **LOOP** pada

baris (4) mempunyai pasangan **REPEAT** pada baris (9), dan eksekusi akan keluar dari statemen kontrol tersebut pada saat **UNTIL** j < i (baris (7)) terpenuhi. Analisa pada **LOOP .. REPEAT** di atas akan tergantung pada statemen penguji dari loop tersebut yaitu **UNTIL** j < i, dari statemen penguji tersebut maka proses dalam loop akan berlangsung sebanyak j - i . Pada blok **LOOP .. REPEAT** baris (4) dan (9) terdapat operator pertukaran ":=:" pada baris (8), sehingga baris (8) mempunyai Running time O(1) karena pada operator pertukaran tersebut hanya terdapat pemberian nilai pada suatu variabel. Pada blok **LOOP .. REPEAT** baris (4) dan (9) juga terdapat dua **LOOP .. REPEAT** yaitu pada baris (5) dan (6) yang masing-masing mempunyai statemen penguji **WHILE** A[i] < v dan **WHILE** A[j] > v. Untuk kedua **LOOP .. REPEAT** tersebut sifatnya adalah sama dengan statemen kondisi **If** , yaitu jika statemen penguji terpenuhi maka akan menjalankan **LOOP .. REPEAT** dan jika tidak terpenuhi maka tidak akan menjalankan **LOOP .. REPEAT** atau keluar dari **LOOP .. REPEAT.** Karena hal tersebut maka untuk menghitung Running Time loop tersebut sama dengan menghitung Running Time pada statemen kondisi **If** , yaitu sebesar harga pada kondisi statemen eksekusi. Karena statemen eksekusinya hanya pemberian harga pada variabel i := i + 1 dan j := j - 1 sehingga banyaknya eksekusi pada kedua loop tersebut adalah j - i. Jadi Running Time **LOOP .. REPEAT** pada baris (5) dan (6) adalah :

$$T(n) = \sum_{j<i} O(1) = \sum_{j<i} c = c(j - i) = c(r - l)$$

dengan menambahkan suatu constanta c sebagai waktu untuk operator pertukaran pada baris (8) dan pengujian pada **UNTIL** j < i maka Running Time **LOOP .. REPEAT** pada baris (4) dan (9) adalah :

$$T(n) = c(r - l) + c = c(r - l + 1)$$

- Pada baris (10) terdapat operator pertukaran " :=: ", maka Running timenya adalah O(1), karena pada operator tersebut hanya terdapat pemberian nilai pada suatu variabel.
- Pada baris (11) dan (12), terdapat pemanggilan prosedur *Quicksort* (prosedur rekursif), yaitu Quicksort(*l*,j-1) dan Quicksort(i,r).

Diberikan T(n) adalah waktu yang diperlukan untuk memanggil Quicksort(1,n). Jika n = 1, algoritma dengan sederhana akan keluar dari prosedur tersebut, dan memberikan waktu suatu konstanta c_1. Sebaliknya, jika n > 1 sebagian besar pekerjaan berupa pemanggilan dua rekursif pada baris (11) dan (12), dimisalkan $1 \leq i < n$, dengan i = j - 1, maka waktu yang diambil oleh kedua rekursif di atas adalah T(i) dan T(n-i). Diberikan $c_2 n$ adalah waktu yang diperlukan diluar prosedur rekursif pada eksekusi yang pertama sebelum masuk ke eksekusi pada kedua rekursif tersebut, dengan mengabaikan waktu yang dibutuhkan untuk memanggil Quicksort(1,n) itu sendiri dan waktu yang dipakai untuk memanggil dua rekursif . Dari ketentuan di atas diperoleh rekurensi:

$$T(n) \leq \begin{cases} c_1 & \text{jika } n = 1 \\ T(i) + T(n - i) + c_2 n & n > 1 \end{cases} \qquad (3,1)$$

dengan c_1 adalah waktu yang diperlukan untuk n = 1

$c_2 n$ adalah waktu yang diperlukan diluar prosedur rekursif untuk n > 1 pada eksekusi yang pertama sebelum masuk ke eksekusi pada kedua rekursif.

Persamaan rekurensi di atas dapat diselesaikan sebagai berikut :

Dengan sejumlah n data, prosedur *Quicksort* akan mengurutkan dari data ke-1 sampai data ke-n, maka pada setiap pemanggilan prosedur *Quicksort* ditemukan $i \leq n/2$, jadi persamaan rekurensi dapat ditulis sebagai berikut :

$$T(n) \leq \begin{cases} c_1 & \text{jika } n = 1 \\ 2T(n/2) + c_2 n & \text{yang lain} \end{cases} \qquad (3,2)$$

Dari persamaan rekurensi di atas dapat dibentuk ke persamaan karakteristik berikut :

$$T(n) = 2T(n/2) + c_2 n \qquad (3,3)$$

$$T(n) - 2T(n/2) = c_2 n \qquad (3,4)$$

dimisalkan $n = 2^k$, maka $k = {}^2\log n$ dan $T_k = T(n)$, jadi persamaan karakteristik di atas dapat ditulis :

$$T_k - 2T_{k-1} = c_2\, 2^k \qquad (3,5)$$

bila kedua sisi dikalikan 2 maka persamaan menjadi :

$$2T_k - 4T_{k-1} = c_2\, 2^{k+1} \qquad (3,6)$$

dengan mengganti k menjadi k-1, maka didapatkan persamaan sebagai berikut:

$$2T_{k-1} - 4T_{k-2} = c_2\, 2^k \qquad (3,7)$$

dari persamaan (3,5) dan (3,7) maka

$$T_k - 2T_{k-1} = 2T_{k-1} - 4T_{k-2}$$

$$T_k - (2+2)T_{k-1} + 4T_{k-2} = 0$$

$$T_k - 4T_{k-1} + 4T_{k-2} = 0 \qquad (3,8)$$

dengan $T(n) = T(2^k) = T_k = x^n$ untuk k adalah derajat tertinggi, maka persamaan (3,8) menjadi :

$$x^2 - 4x + 4 = 0 \qquad (3,9)$$

$$(x - 2)(x - 2) = 0$$

dengan r_1 dan r_2 masing-masing adalah 2, merupakan akar-akar persamaan (3,9)

dengan rumus umum

$$T(n) = \sum_{i=1}^{t} \sum_{j=0}^{m_i - 1} c_{ij} n^j r_i^n \qquad (3,10)$$

dari persamaan (3,9) dan (3,10) maka

$$T(n) = T(2^k) = T_k = c_1 r_1^k + c_2 k r_2^k$$

$$= c_1 2^k + c_2 k 2^k \qquad \text{dengan } k = {}^2\log n$$

$$= c_1 n + c_2 n {}^2\log n$$

$$= c_1 n + (c_2 / \log 2) n \log n$$

dengan mengambil $c_1 = a$ dan $c_2 / \log 2 = b$ maka

$$= an + bn \log n$$

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