PERBANDINGAN KOMPLEKSITAS ALGORITMA PENCARIAN BINER DAN ALGORITMA PENCARIAN BERUNTUN

Yudhistira – NIM 13508105

Mahasiswa Program Studi Teknik Informatika ITB
Jalan Ganesha No.10 Bandung
e-mail: if18105@students.if.itb.ac.id

ABSTRAK

Makalah ini membahas tentang kompleksitas algoritma pencarian beruntun dan algoritma pencarian biner. Di sini akan dibahas kompleksitas dari kedua algoritma tersebut dari berbagai aspek. Di antaranya adalah dengan menggunakan kompleksitas waktu biasa dan kompleksitas waktu asimptotik. Di dalam makalah ini, dijelaskan mengenai kemangkusan kedua algoritma pencarian tersebut. Dalam makalah ini, penulis mencoba mencaritahu algoritma mana yang paling mangkus untuk melakukan pencarian.

Kata kunci: algoritma, pencarian, biner, beruntun,

1. PENDAHULUAN

1.1 Kompleksitas Algoritma

Mudah sekali seseorang membuat sebuah algoritma. Misal, untuk membuat algoritma penambahan satu (n+1) dengan menulis sebagaimana mestinya. Tapi bisa juga orang tersebut menbuliskannya dengan (n*n)-(n*n)+n+1. Dari dua persamaan tersebut, mana yang lebih singkat waktu eksekusinya? Semua orang pasti tahu, tentu saja algoritma pertama. Pertanyaannya, bagaimana kalau yang dipertanyakan adalah algoritma yang sulit untuk ditebak? Jawabannya adalah, menggunakan kompleksitas algoritma.

Tidak semua komputer di dunia memiliki kecepatan yang sama, sehingga tiap komputer membutuhkan waktu yang berbeda untuk mengeksekusi suatu algoritma. Bagaimana jika suatu komputer membutuhkan satu tahun untuk menyelesaikan suatu persoalan? Tentu saja hal ini sangat menjengkelkan dan sangat membuang-buang waktu. Bagaimana supaya kita bisa membuat operasi menjadi lebih cepat tanpa meng-*upgrade* komputer? Kita membutuhkan algoritma yang lebih mangkus.

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

Di dalam bahasan kali ini, penulis hanya akan menekankan pada kompleksitas waktu saja. Simak algoritma hitung rerata di bawah ini:

```
jumlah ← 0
i ← 1
while i ≤ n do
   jumlah ← jumlah + a_i
   i ← i + 1
endwhile
r ← jumlah/n
```

Mari kita hitung kompleksitas waktunya.

```
jumlah ← 0                      1 kali
i ← 1                           1 kali
jumlah ← jumlah + a_i           n kali
i ← i + 1                       n kali
r ← jumlah/n                    1 kali
```

Maka kompleksitas waktunya adalah penjumlahan dari kelima waktu eksekusi di atas, yaitu `2n + 3`.

Kompleksitas waktu dibedakan atas tiga macam:

a. $T_{maks}(n)$: kompleksitas waktu untuk kasus terburuk (*worst case*).
b. $T_{min}(n)$: kompleksitas waktu untuk kasusterbaik (*best case*).
c. $T_{avg}(n)$: kompleksitas waktu untuk kasus rata-rata (*average case)*

.

Ketiga kompleksitas tersebut akan dipakai untuk membandingkan kemangkusan 2 algoritma pencarian di atas.

1.2 Kompleksitas Waktu Asimptotik

Pada pembahasan di atas, sudah disinggung tentang kompleksitas waktu. Pada pembahasan kali ini, penulis ingin mencoba untuk membahas kompleksitas waktu asimptotik. Kompleksitas wakti asimptotik adalah kompleksitas waktu yang hanya memedulikan suku-suku yang paling memengaruhi saja. Misalkan, diketahui suatu kompleksitas waktu $T(n) = 2n^2 + 5n + 2$. Untuk fungsi tersebut, $T(n)$ tumbuh seperti n^2 tumbuh. Maka kita bisa mengabaikan suku-suku yang tidak mendominasi perhitungan pada rumus $T(n)$, sehingga kompleksitas waktu $T(n)$ adalah $2n^2 + suku - suku\ lainnya$. Dengan mengabaikan elemen 2 pada $2n^2$, kita melihat $T(n)$ tumbuh seperti n^2 dan kita tuliskan

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

Yang dibaca "$T(n)$ adalah O dari n^2". Jadi, kita telah mengganti ekspresi seperti $T(n) = 2n^2 + 5n + 2$ dengan ekspresi yang lebih sederhana seperti n^2 yang tumbuh pada kecepatan yang sama dengan $T(n)$. Notasi O disebut notasi "O-Besar" (*Big O)* yang merupakan salah satu dari tiga notasi *kompleksitas waktu asimptotik.*

Tabel Algoritma berasarkan notasi *O*-besar

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

Dari tabel di atas, kita dapat membuat urutan spektrum kompleksitas waktu algoritmanya:

$$\underbrace{O(1) < O(\log n) < O(n) < O(n.\log n) < O(n^2) < O(n^3) < \cdots}_{\text{Algoritma polinomial}} < \underbrace{O(2^n) < n!}_{\text{Algoritma eksponensial}}$$

Urutan spektrum di atas mengurut dari yang paling cepat sampai yang paling lambat.

2. METODE

Di sini, penulis menggunakan metode perbandingan kompleksitas algoritma, perbandingan kompleksitas waktu, dan perbandingan kompleksitas waktu asimptotik. Metode ini dapat pula digunakan untuk membandingkan kemangkusan algoritma yang lain, seperti algoritma pencarian, dan algoritma mencari jarak terpendek. Penulis memilih kompleksitas waktu, karena penulis yakin bahwa waktu merupakan aspek terpenting dalam kemangkusan suatu algoritma.

2.1. Algoritma Pencarian Beruntun

Algoritma pencarian beruntun merupakan algoritma pencarian yang cukup mudah dimengerti dan sudah lazim dipakai oleh berbagai kalangan. Pada dasarnya, algoritma pencarian beruntun adalah proses membandingkan setiap elemen larik satu per satu secara beruntun, mulai dari elemen pertama, sampai elemen yang dicari ditemukan, atau seluruh elemen sudah diperiksa.

Contoh. Perhatikan larik L di bawah ini dengan $n = 6$ elemen:

19	10	14	23	6	8

Misalkan nilai yang dicari adalah: $x = 23$. Elemen yang dibandingkan (berturut-turut): 19, 10, 14, 23. Indeks larik yang dikembalikan: *idx* = 4.

Berikut adalah *pseudo-code* dari algoritma pencarian beruntun:

```
procedure PencarianBeruntun(input a1, a2, ..., 
an : integer, x : integer,
                              output idx : 
integer)
Deklarasi
  k : integer
  ketemu : boolean   { bernilai true jika x 
ditemukan atau false jika x tidak ditemukan }

Algoritma:
  k←1
  ketemu ← false
  while (k ≤ n) and (not ketemu) do
    if ak = x then
      ketemu←true
    else
      k ← k + 1
    endif
  endwhile
  { k > n or ketemu }

  if ketemu then   { x ditemukan }
     idx←k
  else
     idx← 0        { x tidak ditemukan }
  endif
```

2.2. Algoritma Pencarian Biner

Prinsip utama pencarian biner adalah dengan membagi data atas dua bagian. Data yang disimpan harus sudah terurut. Misalkan elemen larik sudah terurut menurun. Dalam proses pencarian, kita memerlukan dua buah indeks larik, yaitu indeks terkecil dan indeks terbesar. Kita menyebut indeks terkecil sebagai indeks ujung kiri larik dan indeks terbesar sebagai indeks ujung kanan larik.

Misalkan indeks kiri adalah *i* dan indeks kanan adalah *j*. Pada mulanya, kita inisialisasi i dengan 1 dan j dengan n.

Langkah pertama, bagi dua elemen larik pada elemen tengah. Elemen tengah adalah elemen dengan indeks $k = (i + j) div\ 2$.

Langkah kedua, periksa apakah $L[k]$ = x, pencarian selesai apabila *x* ditemukan. Tetapi, jika $L[k] \neq x$, harus ditentukan apakah pencarian akan dilakukan di larik bagian kiri atau kanan. Jika $L[k] < x$, maka pencarian dilakukan lagi pada larik bagian kiri. Sebaliknya, jika $L[k] > x$, pencarian dilakukan lagi pada larik bagian kanan.

Langkah ketiga, ulangi langkah pertama hingga *x* ditemukan atau $i > j$ (ukuran larik sudah nol).

Sebagai contoh, misalkan diberikan larik L dengan delapan buah elemen yang sudah terurut menurun seperti di bawah ini:

67	54	51	43	31	29	13	9

Misalkan, kita ingin mencari nilai *x* = 31 dalam larik tersebut.

Langkah pertama,
$i = 1$ dan $j = 8$
Indeks elemen tengah k = (1 + 8) div 2 = 4

67	54	51	43	31	29	13	9

Kiri Kanan

Langkah kedua, apakah $L[4] = 31$? Ternyata tidak. Maka harus diputuskan apakah pencarian akan dilakukan di bagian kiri atau bagian kanan dengan pemeriksaan sebagai berikut:

Disimpulkan bahwa $L[4] > 31$. Maka lakukan pencarian pada larik bagian kanan dengan $i = k + 1 = 5$ dan $j = 8$ (tetap).

31	29	13	9

Kembali lagi ke langkah pertama, kita tentukan indeks tengah $(5 + 8)\ div\ 2 = 6$

31	29	13	9

Kiri' Kanan'

Lalu ke langkah 2 lagi, apakah $L[6] = 31$? Karena tidak, putuskan mau dicari di sebelah kanan atau kiri. Karena $L[6] < 31$, maka dicari pada larik bagian kiri dengan $i = 5$ dan $j = k - 1 = 5$.

31

Kembali lagi ke langkah 1, indeks elemen tengah $k = (5 + 5)\ div\ 2 = 5$

31

Karena $L[5] = 31$, maka *x* ditemukan, dan proses pencarian selesai.

Proses yang terhitung cukup panjang di atas, dapat disederhanakan dalam bentuk pseudo-code algoritmik sebagai berikut:

```
procedure PencarianBiner(input a1, a2, ..., an :
integer, x : integer,
                         output idx : integer)
Deklarasi
   i, j, mid : integer
   ketemu : boolean

Algoritma
   i←1
   j←n
   ketemu←false
   while (not ketemu) and ( i ≤ j) do
      mid ← (i+j) div 2
      if amid = x then
          ketemu ← true
      else
         if amid < x then     {cari di kanan}
            i←mid + 1
         else                 {cari di kiri}
            j←mid - 1;
         endif
      endif
   endwhile
   {ketemu or i > j }

   if ketemu then
      idx←mid
   else
      idx←0
   endif
```

2.3.Kompleksitas Algoritma Pencarian Beruntun

Algoritma pencarian beruntun membandingkan setiap elemen larik dengan *x*, mulai dari elemen pertama sampai *x* ditemukan atau sampai elemen terakhir. Kita akan menghitung jumlah operasi perbandingan elemen larik yang terjadi selama pencarain ($a_k = x$). Operasi perbandingan yang lain, seperti $i \leq n$ tidak dihitung. Operasi perbandingan elemen-elemen larik adalah operasi abstrak yang mendasari algoritma pencarian.

a. Kasus terbaik
Operasi perbandingan elemen (a_k = x) hanya dilakukan satu kali, maka

$$T_{min}(n) = 1$$

b. Kasus terburuk
Bila $a_n = x$ atau *x* tidak ditemukan.
Seluruh elemen larik dibandingkan, maka jumlah perbandingan elemen larik (a_k = x) adalah

$$T_{max}(n) = \text{n}$$

c. Kasus rata-rata
Jika *x* ditemukan pada posisi ke-*j*, maka operasi perbandingan (a_k = x) dilakukan sebanyak *j* kali. Jadi, kebutuhan waktu rata-rata algoritma pencarian beruntun adalah

$$T_{avg}(n) = \frac{n+1}{2}$$

2.4.Kompleksitas Algoritma Pencarian Biner

Algoritma pencarian biner membagi larik di pertengahan menjadi dua bagian yang berukuran sama (*n*/2 bagian), bagian kiri dan bagian kanan. Jika elemen pertengahan tidak sama dengan *x*, keputusan dibuat untuk melakukan pencarian pada bagian kiri atau bagian kanan. Proses bagi-dua dilakukan lagi pada bagian yang dipilih. Perhatikanlah bahwa setiap kali memasuki *while-do* maka ukuran larik yang ditelusuri berkurang menjadi setengah kali ukuran semula: *n, n/2, n/4, ...*

Kita akan menghitung jumlah operasi perbandingan elemen dengan x yang terjadi selama pencarian ($a_{mid} = x$). Operasi perbandingan yang lain, seperti $i \leq$ j dan $a_{mid} < x$ tidak akan dihitung. Untuk penyederhanaan, asumsikan ukuran larik adalah perpangkatan dari 2, yaitu $n = 2^k$.

a. Kasus terbaik:
Kasus terbaik adalah bila *x* ditemukan pada elemen pertengahan (a_{mid}), dan operasi perbandingan elemen (a_{mid} = x) yang dilakukan hanya satu kali. Pada kasus ini:

$$T_{min}(n) = 1$$

b. Kasus terburuk:
Pada kasus terburuk, elemen *x* ditemukan ketika ukuran larik = 1. Pada kasus terburuk ini, ukuran larik setiap kali memasuki *while-do adalah:*

$$n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots, 1 \quad \text{(sebanyak } \log_2 n \text{ kali)}$$

Artinya, kalang *while-do* dikerjakan sebanyak $\log_2 n$ kali.
Untuk n = 128 → 64 → 32 → 16 → 8 → 4 → 2 → 1 (sebanyak 7 kali pembagian)

Jumlah operasi perbandingan elemen ($a_{mid = x)}$ adalah:

$$T_{max}(n) = \log_2 n$$

c. Kasus rata-rata:
Kompleksitas waktu rata-rata algoritma pencarian biner relatif lebih sulit dilakukan dibandingkan dengan kompleksitas waktu rata-rata algoritma pencarian beruntun.

2.5.Kompleksitas Waktu Asimptotik untuk Algoritma Pencarian Beruntun

Jika kompleksitas waktu *T(n)* dari algoritma sudah dihitung, maka kompleksitas waktu asimptotiknya dapat langsung ditemukan dengan mengambil suku yang mendominasi fungsi *T* dan menghilangkan elemennya.

Pada algoritma pencarian beruntun, telah didapat hasil kompleksitas waktu minimal, maksimal, dan rata-rata sebagai berikut:

$$T_{min}(n) = 1$$
$$T_{max}(n) = \text{n}$$
$$T_{avg}(n) = \frac{n+1}{2}$$

Maka dengan mudah kita dapat menentukan kompleksitas wakti asimptotik dari algoritma tersebut.

$$T_{min}(n) = 1 = O(1)$$
$$T_{max}(n) = n = O(n)$$
$$T_{avg}(n) = \frac{n+1}{2} = O(n)$$

2.6.Kompleksitas Waktu Asimptotik untuk Algoritma Pencarian Biner

Sama halnya dengan algoritma pencarian beruntun, apabila kita sudah mengetahui kompleksitas waktu *T(n)* dari algoritma pencarian biner, kita bisa dengan mudah mendapatkan *O-besar* atau juga disebut kompleksitas waktu asimptotik.

Berikut adalah data yang sudah terlebih dahulu kita ketahui:

$$T_{min}(n) = 1$$
$$T_{max}(n) = \log_2 n$$

kita dapat dengan mudah menyatakan persamaan di atas ke dalam bentuk *O*-besar. Hasilnya pasti akan seperti sebagai berikut:

$T_{min}(n) = O(1)$
$T_{max}(n) = \log_2 n = O(\log_2 n)$

Dari persamaan-persamaan yang sudah ada, kita sudah semakin dekat untuk membandingkan, apakah algoritma pencarian beruntun, ataukah algoritma pencarian biner yang akan lebih mangkus dari yang lainnya.

2.7. Perbandingan antara Kompleksitas Algoritma Pencarian Beruntun dan Algoritma Pencarian Biner

Setelah memulai dengan dasar-dasar teori, akhirnya kita sampai pada fasa memperbandingkan. Sekadar untuk mengingat, di pembahasan awal, kita telah mengupas dua buah algoritma pencarian, yaitu Algoritma Pencarian Beruntun dan Algoritma Pencarian Biner. Setiap algoritma telah dibahas, kini saatnya membandingkan kedua algoritma tersebut.

Data-data yang didapat dari kedua algoritma adalah sebagai berikut:

Tabel Perbandingan Algoritma Pencarian Beruntun dan Algoritma Pencarian Biner

Aspek yang dibahas	Algoritma Pencarian Beruntun	Algoritma Pencarian Biner
T_{min}	1	1
T_{max}	n	$\log_2 n$
T_{avg}	$\frac{n+1}{2}$	*--sulit dicari--*
O_{min}	1	1
O_{max}	n	$\log_2 n$
O_{avg}	n	*--sulit dicari--*

Untuk aspek kompleksitas waktu asimptotik minimum, atau O_{min}, bisa disimpulkan bahwa kedua algoritma memiliki kompleksitas yang sama. Kasus terbaik untuk kedua algoritma adalah:

- Untuk Algoritma Pencarian Beruntun, ketika operasi perbandingan elemen hanya dilakukan satu kali, yaitu ketika elemen *x* yang dicari berada pada indeks *i* pertama.

- Untuk algoritma Pencarian Biner, ketika elemen *x* yang dicari ditemukan pada elemen pertengahan (a_{mid}), dan operasi yang dilakukan hanya satu kali. Di mana $a_{mid} = i + j$ div *2.*

Untuk kompleksitas waktu asimptotik maksimim, atau O_{max}, terdapat perbedaan hasil. Pada Algoritma Pencarian Beruntun, O_{max}-nya adalah n, sedangkan untuk Algoritma Pencarian Biner, O_{max}-nya adalah $\log_2 n$. Yang manakah yang lebih mangkus? Dari skema yang telah kita ketahui sebelumnya, yaitu:

$$O(1) < O(\log n) < O(n) < O(n.\log n) < O(n^2) < O(n^3) < \cdots < O(2^n) < n!$$

Algoritma polinomial — Algoritma eksponensial

Dapat kita simpulkan bahwa untuk O_{max}, Algoritma Pencarian Biner terbukti lebih cepat dari Algoritma Pencarian Beruntun.

Jadi, dapat disimpulkan bahwa Algoritma Pencarian Biner lebih mangkus daripada Algoritma Pencarian Beruntun.

III. KESIMPULAN

Dari hasil studi literatur dan penyelesaian beberapa persamaan, terbukti bahwa Algoritma Pencarian Biner lebih cepat daripada Algoritma Pencarian Beruntun. Karena itu, Algoritma Pencarian Biner lebih mangkus daripada Algoritma Pencarian Beruntun.

Saat dibandingkan *best case* atau kasus terbaik masing-masing algoritma, kedua algoritma masih sama mangkusnya. Tapi ketika dibandingkan *worst case* atau kasis terburuknya, terbukti bahwa Algoritma Pencarian Biner jauh lebih cepat daripada Algoritma Pencarian Beruntun. Adapun *best case* kompleksitas waktu asimptotik untuk Algoritma Pencarian Biner adalah $\log_2 n$. Sedangkan *best case* untuk Algoritma Pencarian Beruntun adalah n.

Kesimpulan ini berlaku pada sekumpulan data yang sudah terurut, baik terurut menaik, maupun terurut menurun. Apabila datanya belum terurut, maka harus dilakukan pengurutan terlebih dahulu agar kesimpulan ini bisa berlaku.

REFERENSI

[1] Munir, Rinaldi, "Matematika Diskrit Edisi Ketiga", Penerbit Informatika, 2005.

[2] Munir, Rinaldi, "Algoritma dan Pemrograman", Penerbit Informatika, 2007.

[3] http://www.google.com. Diakses pada tanggal 20 Desember pukul 07.35-09.00