Kompleksitas Algoritma *Brute Force* Dalam Penyelesaian Sudoku

Yusuf Alwansyah Hilmy/13519005[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13519005@itb.ac.id

***Abstract*—Permainan Puzzle selalu menarik untuk dimainkan salah satunya adalah sudoku. Meskipun sudah ditemukan bertahun-tahun lalu sudoku masih tetap menarik. Di zaman modern ini sudoku bukan hanya olahraga otak namun juga bagian dari penelitian. Banyak orang yang berusaha membuat program yang dapat menyelesaikan sudoku. Dari sekian banyak program, algoritma yang paling dikenal adalah *brute force* yang simpel tetapi memakan banyak waktu. Kompleksitas dari brute force harus diketahui untuk memnetukan apakah algoritma ini efektif atau tidak.**

***Keywords*—Algoritma,Kompleksitas,Sudoku.**

I. PENDAHULUAN

Sudoku adalah permainan puzzle yang sudah ada sejak dulu. Sudoku berasal dari kata "*Suji wa dokushin ni kagiru*" yang artinya angkanya tunggal. Sebagai mana asal usul katanya, puzzle ini berasal dari Jepang, walaupun ada berbagai sumber yang mengatakan puzzle ini berasal dari Swiss.

Permainan puzzle sudoku ini cukup sederhana, hanya perlu mengisi kotak 9x9 dengan angka 1-9 dengan syarat setiap baris, kolom, dan kotak 3x3 tidak terdapat angka yang sama.

	6					5		3
	8	1			2			4
4	5			1	6		7	
2		8		6	5	3		
	7	3		2	4		5	
1	4					9		
		7					6	
9		6		4	7	2	8	1
		4			1		3	5

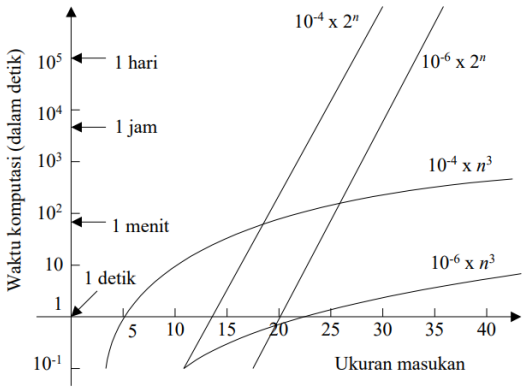
Gambar 1. Contoh puzzle sudoku

Walaupun kebanyakan sudoku memiliki ukuran 9x9, nyatanya banyak juga sudoku dengan ukuran yang lain seperti 6x6 (mempunyai nama lain Rudoku) yang didalamnya dibagi menjadi 2x3 dan/atau 3x2. Lalu ada juga yang berukuran 12x12 yang didalamnya dibagi menjadi 3x4 atau 6x2. Sudoku bisa saja kosong pada awalnya, namun umumnya sudoku sudah diisi sebagian nilainya yang kemudian dilanjutkan pemain.

Puzzle ini awalnya dimainkan di atas kertas. Namun seiring perkembangan zaman, permainan ini juga terdapat di internet dan dalam bentuk aplikasi. Lalu banyak orang yang juga membuat algoritma yang mempermudah dalam menyelesaikan sudoku. Dari berbagai macam pembuatnya, terdapat beberapa algoritma yang umum dipakai untuk menyelesaikan sudoku diantaranya adalah *brute force*,

II. KOMPLEKSITAS ALGORITMA

Dalam melakukan pembuatan program, diperlukan algoritma yang tidak hanya benar sesuai kebutuhan namun juga efisien yaitu tidak memakai banyak memori dan waktu yang sedikit dalam eksekusinya. Penggunaan memori dan waktu bergantung pada input yang dimasukkan. Input yang ada menjadi indikator bagus atau tidaknya suatu algoritma. Semakin sedikit memori dan waktu yang dipakai, maka algoritma itu semakin bagus. Dalam setiap permasalahan selalu memiliki bermacam-macam algoritma, karenanya diperlukan peninjauan kompleksitas algoritma.



Gambar 2. Grafik perbandingan input terhadap waktu (Sumber: http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian1.pdf)

Dari grafik diatas terlihat ukuran masukkan akan sangat mempengaruhi waktu eksekusi, namun perbedaan waktu yang terjadi baru akan terlihat jika ukuran masukan sangat besar. Model yang digunakan dalam perhitungan waktu kompilasi

bukanlah berdasarkan waktu yang dijalankan komputer. Hal ini disebabkan *processor* yang berfungsi menjalankan program tidak selalu sama untuk tiap perangkat yang ada. Karenanya suatu program mungkin berjalan cepat di perangkat canggih, namun lambat jika di perangkat lama. Hal lain yang jadi pertimbangan adalah *compiler* dari setiap bahasa pemrograman yang juga berbeda-beda.

Karena pertimbangan tersebut, diperlukan model abstrak pengukuran waktu/ruang memori yang tidak terikat perangkat yang menjalankannya. Kompleksitas algoritma menjadi besaran untuk menerangkan hal itu. Kompleksitas algoritma ada 2 yaitu kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu dilambangkan T(n) dan Kompleksitas ruang dilambangkan S(n) dengan n adalah ukuran masukan. Yang akan dibahas kali ini hanyalah kompleksitas waktu.

Kompleksitas waktu memperhitungan jumlah komputasi yang dilakukan dalam algoritma yang dipakai. Jumlah komputasi dihitung berdasarkan input masukkan. Terdapat banyak operasi-operasi dalam algoritma yang ada, namun yang perlu dihitung adalah operasi yang khas yaitu yang menjadi dasar suatu algoritma.

Contoh operasi-operasi yang khas:

- Algoritma pencarian
- Algoritma pengurutan
- Algoritma perkalian matriks

Sebagai contoh jika terdapat program (dalam bahasa C) penjumlahan semua array A berukuran n:

```
int sum = 0;
for (int i = 0; i<n; i++):{
    sum += A[i];
}
```

Operasi penjumlahan sum akan tergantung dengan ukuran array-nya yaitu n, karenanya T(n) = n.

Penjumlahan array merupakan algoritma yang selalu berbanding lurus dengan masukan karenanya kompleksitas waktunya hanya T(n). Hal ini berbeda dengan algoritma pencarian karena pencarian akan berhenti jika sudah ketemu. Sehingga algoritma pencarian bergantung pada letak dari yang dicari.

```
int i =  0;
while (A[i] != X && i<n):{
    i++;
}
```

Pada program diatas akan dilakukan pencarian X di dalam array A yang berukuran n. Jika X terletak di akhir atau tidak ada maka jumlah operasi yang dilakukan adalah n. Tapi jika X ditempat lain maka operasinya tidak n, melainkan indeks dari tempat X berada. Karenanya kompleksitas waktu dibagi menjadi 3:

- *Worst Case*: Kemungkinan terburuk terjadinya operasi, waktu yang digunakan maksimum ($T_{max}(n)$)
- *Best Case*: Kemungkinan terbaik terjadinya operasi, waktu yang digunakan minimum ($T_{min}(n)$)
- *Avg Case*: Kemungkinan rata-rata terjadinya operasi, waktu yang digunakan rata-rata ($T_{avg}(n)$)

Pada algoritma pencarian, *Worst Case* memberikan waktu sebanyak n, Best *Case* memberikan waktu sebanyak 1. Perhitungan T(n) untuk algoritma ini adalah melakukan perhitungan rata-rata sebagai berikut:

$$Tavg(n) = \frac{1 + 2 + \cdots + n}{n} = \frac{n-1}{2}$$

Jadi rata-rata akan dilakukan (n-1)/2 operasi pada algoritma pencarian. Namun, perlukah kita menghitung secara presisi? Bagaimana jika algoritma yang dijalankan sangat panjang dan rumit? Dalam komplesitas algoritma, nilai presisi bukanlah hal yang penting karena yang akan menjadi penilaian kompleksitas adalah saat masukan berukuran besar sekali. Karenanya diperlukan notasi untuk merepresentasikan kerja algoritma untuk n yang sangat besar. Notasi itu dinamakan Kompleksitas waktu asimptotik.

Untuk mengetahui kinerja algoritma digunakan notasi Big-O (O(n)) yang berarti O(n) "sebanding" dengan T(n) untuk T(n) = 2n.

Definisi Big-O adalah sebagai berikut:

"T(n) = O(f(n)) berarti T(n) adalah O(f(n)) yang bermakna T(n) memilliki orde terbesar f(n). T(n) = O(f(n)) berlaku jika terdapat konstanta C dan n_0 sehingga $T(n) \leq Cf(n)$ untuk $n \geq n_0$"

Dari definisi tersebut terdapat beberapa teorema:

Teorema 1

"Bila $T(n) = a_m n^m + a_{m-1} n^{m-1} + ... + a_1 n + a_0$ adalah polinom derajat $\leq$ m maka $T(n) = O(n^m)$."

Teorema 2

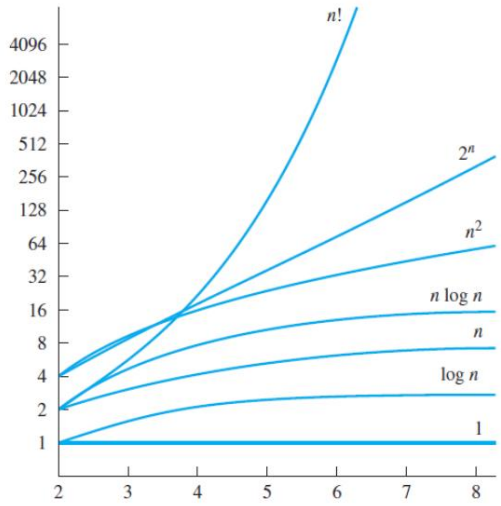
"Misalkan T1 (n) = O(f(n)) dan T2 (n) = O(g(n)), maka:

(a) T1 (n) + T2 (n) = O(f(n)) + O(g(n)) = O(max(f(n), g(n))
(b) T1 (n)T2 (n) = O(f(n))O(g(n)) = O(f(n)g(n))
(c) O(cf(n)) = O(f(n)), c adalah konstanta
(d) f(n) = O(f(n))"

Karena yang terpenting dalam kompleksitas algoritma adalah orde yang paling besar maka terdapat urutan algoritma berdasarkan Big-O (semakin bawah semakin besar)

Kelompok Algoritma	Nama
O(1)	Konstan
O(log n)	Logaritmik
O(n)	Linier
O(n log n)	Linier Logaritmik
$O(n^2)$	Kuadratik
$O(n^3)$	Kubik
$O(2^n)$	Eksponensial
O(n!)	Faktorial

Tabel 1. Kelompok Algoritma dengan urutan naik (Sumber: http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian2.pdf)



Gambar 3. Grafik perbandingan Kelompok Algoritma (Sumber: http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian2.pdf)

Penjelasan dari tiap kelompok algoritma:

- O(1)
  Algoritma ini tidak bergantung pada ukuran masukan sehingga nilainya selalu konstan. Contohnya pada operasi menukar 2 buah angka.
- O(log n)
  Algoritma ini bergantung pada ukuran n namun pertambahannya tidak terlalu banyak. Contohnya pada algoritma binary search.
- O(n)
  Algoritma ini selalu berbanding lurus dengan ukuran masukan. Contohnya algoritma menghitung rata-rata
- O(n log n)
  Algoritma ini menggunakan metode *divide and conquer* yaitu memecah masalah menjadi lebih kecil lalu diselesaikan sendiri-sendiri baru digabungkan.
- $O(n^2)$
  Algoritma ini bertambah sebesar kuadratik sehingga nilainya tidak bisa terlalu besar karena akan tidak efektif. Contohnya adalah beberapa algoritma pengurutan.
- $O(n^3)$
  Algoritma ini mirip sebelumnya namun nilainya semakin besar. Contohnya adalah algoritma perkalian matriks.
- $O(2^n)$
  Algoritma ini sudah cukup buruk karena nilainya sangat besar bahkan untuk ukuran yang kecil. Algoritma *brute force* kebanyakan masuk golongan ini.
- O(n!)
  Algoritma ini menghubungkan masukan dengan masukan yang dikurangi 1. Contohnya adalah *Salesman Traveling Problem.*

Selain notasi Big-O, ada juga notasi yang lain yaitu Big-Omega dan Big Tetha.

Definisi Big-Omega adalah sebagai berikut:

"T(n) = Ω(g(n)) berarti T(n) adalah Ω(g(n)) yang bermakna T(n) memilliki orde terkecil g(n). T(n) = Ω(g(n)) berlaku jika terdapat konstanta C dan n_0 sehingga $T(n) \geq Cg(n)$ untuk $n \geq n_0$"

Definisi Big-Theta adalah sebagai berikut:

"T(n) = Θ(h(n)) berarti T(n) adalah Θ(h(n)) yang bermakna T(n) berorde sama dengan h(n) jika T(n) = O(h(n)) dan T(n) = Ω(h(n))

III. SUDOKU

Sudoku dipercaya berawal dari Euler yang membuat *Latin Square* di 1783. *Latin Square* adalah kotak berukuran nxn yang berisi nilai 1 sampai n yang diurutkan agar tiap baris dan kolom mempunyai tiap angka tepat 1 kali saja. Namun ada pendapat lain dari Howard Garns yang menyatakan Sudoku baru ada pada tahun 1970-an di New York. Saat itu sebuah majalah di New York menerbitkan permainan puzzle yang mirip dengan nama '*Number Place*'.

Sudoku memiliki beberapa istilah sebagai berikut:

- *Bands*: kumpulan kotak dalam vertikal
- *Stack*: kumpulan kotak dalam horizontal
- S: Lambang untuk sudoku keseluruhan
- $S_{a,b}$: Lambang yang menunjukkan kotak 3x3 yang berada di *band* a dan *stack* b (Sebagai contoh, $S_{2,3}$ adalah kotak 3x3 yang berada di paling atas bagian tengah
- $[S_{a,b}]_{i,j}$: Lambang yang menunjukkan baris i dan kolom j dalam kotak 3x3 di *band* a dan *stack* b.
- *Irreducible Sudoku*: Sudoku yang memberikan semua angka penting untuk menciptakan solusi unik dari sudoku tersebut
- *Strongly Completable*: Sudoku yang bisa diselesaikan cukup dengan logika saja
- *Weakly Completable*: Sudoku yang memerlukan coba-coba dalam pengerjaannya
- *Critical Set*: Set terkecil yang meneyediakan solusi unik dalam sudoku

Ukuran dari Critical Set yang diketahui untuk Sudoku 9x9 adalah 17. Selain itu dalam memberikan angka yang diberikan harus memiliki syarat-syarat tertentu:

- Setidaknya terdapat 8 angka dari 1-9 yang tersedia
- Tiap 2 baris di *band* dan Tiap 2 kolom di *stack* setdiaknya memiliki 1 angka yang diberikan
- Setiap baris dan kolom harus memiliki setidaknya 1 angka yang diberikan

IV. ALGORITMA PENYELESAIAN SUDOKU

```
def findNextCellToFill(grid, i, j):
  for x in range(i,9):
     for y in range(j,9):
        if grid[x][y] == 0:
           return x,y
  for x in range(0,9):
     for y in range(0,9):
        if grid[x][y] == 0:
           return x,y
     return -1,-1

def isValid(grid, i, j, e):
  rowOk = all([e != grid[i][x] for x in
range(9)])
  if rowOk:
```

```
    columnOk = all([e != grid[x][j] for
x in range(9)])
    if columnOk:
        # finding the top left x,y co-
ordinates of the section containing the
i,j cell
        secTopX, secTopY = 3 *(i//3), 3
*(j//3) #floored quotient should be used
here.
        for x in range(secTopX,
secTopX+3):
            for y in range(secTopY,
secTopY+3):
                if grid[x][y] == e:
                    return False
                return True
    return False

def solveSudoku(grid, i=0, j=0):
    i,j = findNextCellToFill(grid, i, j)
    if i == -1:
        return True
    for e in range(1,10):
        if isValid(grid,i,j,e):
            grid[i][j] = e
            if solveSudoku(grid, i, j):
                return True
            # Undo the current cell for
backtracking
            grid[i][j] = 0
    return False
# Source code:
https://stackoverflow.com/questions/1697
334/algorithm-for-solving-sudoku
```

Pada algoritma penyelesaian sudoku(dalam bahasa Phyton) ini terdapat tiga fungsi utama:

- findNextCellToFill
  Fungsi ini mencari kotak yang kosong untuk diisi, jika semua kotak sudah terisi akan dikirim nilai -1.
- isValid
  Fungsi ini mengecek apakah pengisian kotak tidak melanggar aturan sudoku
- solveSudoku
  Main program dari penyelsaian sudoku. Pertama program akan mencari koordinat kotak kosong, lalu program mencoba akan mengecek jika nilai dari 1 sampai 9 yang akan dimasukkan benar atau tidak. Jika salah maka sudoku tidak bisa diselesaikan dengan nilai tersebut. Jika benar, maka kotak akan diisi nilai tersebut. Kemudia program akan menjalankan dirinya sendiri hingga ditemukan kesalahan pengisian (yang akan membuatnya melakukan *backtracking*) atau sudoku terselesaikan.

Dalam eksekusi programnya, dapat diambil asumsi kotak yang kosong adalah yang paling banyak yaitu 64 karena *Critical Set* yang paling sedikit adalah 17. Karena itu $T_{min}(n) = 64$ atau $T_{min}(n) = O(n)$.

Worst Case yang ada pada program ini adalah setiap pengisian kotak dicoba satu persatu sehingga setiap kotak dicoba 9 kali. Namun karena terdapat rekursif, maka akan terjadi pengulangan berbentuk eksponensial sehingga kompleksitas dari $T_{max}(n) = O(9^n)$.

V. KESIMPULAN

Penggunaan *brute force* ternyata memakan banyak waktu untuk menyelesaikan sudoku 9x9. Hal ini terlihat dengan *Worst Case* yang bisa melakukan 9^n operasi. Tentunya ada algoritma-algoritma lain yang lebih efektif seperti pendekatan *stochastic optimization*. Namun algoritma seperti itu tidak akan dibahas kesempatan kali ini

VI. UCAPAN TERIMA KASIH

Pertama saya ucapkan syukur kepada Allah swt. yang telah memberikan saya hidup, kemampuan untuk menulis ini, dan banyak nikmat yang tak hingga. Lalu saya juga ucapkan untuk kedua orang tua saya yang telah memfasilitasi belajar saya termasuk perangkat yang saya gunakan untuk menyelesaikan makalah ini. Tak lupa juga saya ucapkan terima kasih pada Pak Rinaldi selaku pengajar mata kuliah Matematika Diskrit ini. Tanpa bapak saya tidak akan bisa paham materi yang sudah bapak ajarkan walaupun ditengah pandemi. Terakhir saya ucapkan terimakasih sedalam-dalamnya pada pihak yang sudah membantu dalam penyelesaiaan makalah ini, baik Google, Youtube dan pihak lain yang tidak bisa saya ucapkan satu persatu

REFERENCES

[1] Munir, Rinaldi. 2015. Slide presentasi materi Kompleksitas Algoritma Bagian 1. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian1.pdf (diakses tanggal 10 Desember 2020).

[2] Munir, Rinaldi. 2015. Slide presentasi materi Kompleksitas Algoritma Bagian 2. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian2.pdf (diakses tanggal 10 Desember 2020).

[3] Sian K. Jones, S. Perkins, and P. A. Roach, “Properties of Sudoku Puzzles”, 2007 https://www.researchgate.net/publication/264572738_Properties_of_Sudoku_Puzzles

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, 11 Desember 2020

Yusuf Alwansyah Hilmy/13519005