Penerapan Kompleksitas Algoritma dalam Pembuatan Game yang Efisien

Lazuardi Firdaus(13515136)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13515136@std.stei.itb.ac.id

***Abstrak*—Salah satu hal yang dipelajari dalam matematika diskrit adalah kompleksitas algoritma. Pada kompleksitas algoritma, mahasiswa belajar tentang bagaimana cara menghitung sumber daya berupa waktu yang digunakan oleh suatu algoritma. Penerapan dari ilmu ini dalam kehidupan sehari-hari salah satunya adalah dalam membuat game dimana game yang dibuat bisa berjalan menggunakan waktu seefisien mungkin sehingga dapat memaksimalkan pengalaman pengguna.**

***Keywords*—Kompleksitas, Algoritma, Game, Efisien**

I. Pendahuluan

Sekarang, video game sudah menjadi salah satu industri hiburan berbasis informatika terbesar di Dunia. Di Amerika Serikat sendiri, konsumen menghabiskan sebanyak 23,5 milyar dollar untuk industri ini sendiri[1]. Walaupun memiliki resikonya sendiri, menjadi developer game adalah salah satu lapangan pekerjaan untuk lulusan teknik informatika.

Dalam membuat video game, khususnya dari segi pembuatan arsitektur game itu sendiri, programmer dituntut untuk membuat arsitektur game yang memungkinkan game untuk menampilkan performa yang maksimal dengan menggunakan sumber daya yang sesedikit mungkin. Sumber daya yang dimaksud bisa berupa waktu dan ruang (memori) dimana dalam melakukan berbagai proses, game dapat berjalan dengan menggunakan waktu dan memori seminimal mungkin. Walaupun kebanyakan komputer yang digunakan pada umumnya pada saat ini sudah dapat memproses data dengan sangat cepat, efisien atau tidaknya keberjalanan suatu game akan tetap terasa mengingat ukuran video game yang semakin besar akhir-akhir ini sehingga membutuhkan pengolah data yang semakin banyak untuk hal-hal seperti mengatur grafik dalam game, berkomunikasi dengan server luar, menerima masukan dari pemain, dan sebagainya. Oleh karena itu, peran programmer sangatlah penting untuk merancang dan membuat arsitektur yang efisien dalam industri video game.

Salah satu faktor yang menentukan apakah suatu arsitektur dapat berjalan dengan efisien dapat dilihat dari algoritmanya. Arsitektur dengan algoritma yang mangkus akan berjalan lebih efisien dibanding dengan arsitektur yang algoritmanya tidak mangkus. Mangkus dan tidaknya suatu algoritma dapat ditinjau dari kompleksitas algoritmanya.

II. Kompleksitas Algoritma

Analisis kompleksitas algoritma mengukur seberapa banyak waktu yang dibutuhkan untuk menjalankan suatu algoritma berdasarkan faktor jumlah suatu data. Walaupun ada cara untuk mengukur seberapa waktu (dan memori) yang dibutuhkan komputer untuk menjalankan suatu program secara nyata menggunakan *profiling*, waktu pemrosesan dapat dipengaruhi faktor seperti hardware yang menjalankan program, compiler yang digunakan untuk membuat program tersebut, dll yang membuat pengukuran kecepatan algoritma dengan waktu jalan kurang akurat. Oleh karena itu analisis kompleksitas algoritma melihat algoritma "mentah"nya secara langsung dan menghitung seberapa banyak pemrosesan data yang berlangsung.

"We already know there are tools to measure how fast a program runs. There are programs called profilers which measure running time in milliseconds and can help us optimize our code by spotting bottlenecks. While this is a useful tool, it isn't really relevant to algorithm complexity. Algorithm complexity is something designed to compare two algorithms at the idea level — ignoring low-level details such as the implementation programming language, the hardware the algorithm runs on, or the instruction set of the given CPU."[2]

A. Menghitung Kompleksitas Suatu Algoritma

Dalam menghitung kompleksitas suatu algoritma kita menghitung jumlah operasi dasar yang bisa dilakukan prosesor kita sebagai satu instruksi yaitu mengisi suatu variabel dengan nilai, mencari nilai suatu elemen pada array, membandingkan dua nilai, menaikkan suatu nilai, dan operasi seperti penjumlahan dan pengurangan.

"We'll assume our processor can execute the following operations as one instruction each:

- *Assigning a value to a variable*
- *Looking up the value of a particular element in an array*
- *Comparing two values*
- *Incrementing a value*
- *Basic arithmetic operations such as addition and multiplication"[3]*

Dengan begini kita bisa mulai menghitung kompleksitas dari suatu algoritma. Misalkan kita ingin menghitung kompleksitas suatu algoritma dimana kita mengisi semua elemen array berukuran N dengan nilai 0 dalam notasi algoritmik.

```
i traversal [1..n]
      array_i ← 0
```

Pada algoritma di atas kita bisa melihat bahwa program akan berjalan. Program akan melakukan suatu operasi yaitu mengisi elemen array ke i dengan nilai 0 sebanyak N kali. Dengan begini kita bisa tahu bahwa lama jalan algoritma adalah T(n) = n. Sekarang mari kita lihat algoritma yang lain.

```
i traversal [1..n]
      j traversal [1..n]
            temp ← 0
            k traversal [1..n]
                  temp ← A1_ik * A2_kj
            A3_ij ← temp
```

Algoritma di atas mengalikan matriks A1 dengan A2 lalu menaruhnya di A3 sesuai dengan aturan perkalian matriks nXn. Bisa dilihat bahwa algoritma memiliki 2 operasi di dalam 2 loop dan 1 operasi di dalam 3 loop sehingga memberikan kita jumlah operasi sebanyak $T(n) = n^3 + 2n^2$.

B. Kompleksitas Waktu Asimtotik

Misalkan kita mendapatkan bahwa jumlah operasi dalam suatu algoritma adalah $T(n) = n^3 + 2n^2$. Dalam analisis kompleksitas algoritma, kita lebih memedulikan apa yang terjadi ketika algoritma diberikan data dalam jumlah besar yaitu ketika n merupakan bilangan yang sangat besar. Oleh karena itu, kita lebih memedulikan bagian n^3 dibandingkan bagian $2n^2$ karena pada bilangan besar, kenaikan nilai yang disebabkan oleh n^3 jauh lebih signifikan daripada yang disebabkan $2n^2$. Maka dalam kompleksitas algoritma kita mengenal notasi big O dimana $T(n) = n^3 + 2n^2 = O(n^3)$.

III. GAME YANG EFISIEN

Dalam menyelesaikan berbagai permasalahan, bisa didapat berbagai macam algoritma yang berbeda tetapi tetap mendapatkan hasil yang sama. Walau begitu, perancangan algoritma harus selalu memikirkan kecepatan dari algoritma yang dibuat. Contohnya adalah berbagai algoritma sorting dimana satu algoritma bisa berjalan lebih cepat dari yang lain tergantung kondisi datanya. Walau perbedaan kecepatan ini tidak terlihat dalam lingkungan praktikum dengan program yang berukuran kecil, dalam program-program yang biasa kita gunakan sehari-hari dan berukuran sangat besar, perbedaan kecepatan ini baru akan terlihat.

Salah satu dari program yang biasa kita gunakan adalah game dimana 2 game yang "sama", menggunakan hardware yang sama, tetapi menggunakan algoritma yang berbeda, bisa berjalan dengan kecepatan yang berbeda. Game-game modern menggunakan berbagai macam algoritma kompleks untuk memproses data yang tidak sedikit. Contoh algoritma yang sering dipakai dalam pembuatan game adalah *pathfinding*. *Pathfinding* adalah algoritma untuk mencari jalur terpendek dari satu titik ke titik yang lain. Algoritma ini dibuat menggunakan prinsip *algoritma Dijkstra*. Sesuai dengan referensi [4], *algoritma Dijkstra* awalnya dibuat untuk mencari lintasan terpendek dari suatu graf berbobot. Sekarang *algoritma Dijkstra* digunakan untuk mendasari berbagai algoritma *pathfinding*. *Pathfinding* digunakan untuk menggerakan karakter menyusuri peta secara alami dimana karakter harus bergerak dari satu tempat ke tempat lain dengan menyusuri berbagai objek dalam peta dan tidak bergerak menembus objek dalam peta. Salah satu algoritma yang biasa dipakai adalah Algoritma *A* Search*.

Seperti yang disebutkan pada referensi [5], penggunaan algoritma seperti *A* Search* menggunakan banyak sumber daya dari CPU. Belum lagi ada kemungkinan bahwa lintasan yang dicari sebenarnya tidak ada sehingga memaksa komputer untuk mencari lintasan pada keseluruhan map yang memakan waktu dan sebenarnya tidak perlu. Oleh karena itu penggunaan algoritma ini perlu dibatasi dengan beberapa cara seperti menggunakan map yang lebih kecil, tidak melakukan *pathfinding* lebih dari 1 setiap waktu, menggunakan unit pencarian yang lebih besar, membuat jalur yang sudah dibuat secara manual, menentukan terlebih dahulu apakah lintasan benar-benar ada, dan menentukan jalan buntu sebelum mencari lintasan. Meminimalisir penggunaan algoritma seperti ini akan membuat game berjalan lebih lancar secara keseluruhan.

Sepeti yang telah kita tinjau pada bab ini, implementasi berbagai algoritma dalam game akan mempengaruhi performa game secara keseluruhan. Kita baru meninjau algoritma yang digunakan untuk menggerakan karakter secara alami sedangkan dalam game yang berukuran besar, ada banyak algoritma yang digunakan untuk menjalankan berbagai fungsi. Sebagian kecil dari algoritma yang digunakan untuk membuat game adalah algoritma untuk menampilkan gambar 3 dimensi pada game, menentukan berbagai status yang dimiliki karakter dalam game, memproses model 3 dimensi dalam game, dan menentukan bentuk dari peta dalam game jika game menggunakan peta yang dibuat secara random. Semua

bagian dari game yang tadi telah disebutkan akan mempengaruhi seberapa baik game akan berjalan. Dalam industri game, dimana untung atau tidaknya perusahaan bergantung pada kepuasan konsumen dan lancar atau tidaknya suatu game bisa menentukan laku atau tidaknya game tersebut, perbedaan kecepatan algoritma memegang peran yang sangat krusial.

IV. Penerapan Kompleksitas Algoritma Dalam Membuat Game yang Efisien

Seperti yang telah dikatakan pada bab sebelumnya bahwa kecepatan eksekusi program memegang peranan yang penting dalam industri video game, dalam membuat game kita perlu memastikan bahwa penggunaan sumber daya perlu ditekan semaksimal mungkin. Dalam menekan sumber daya inilah kita bisa menggunakan analisis kompleksitas algoritma agar setiap algoritma yang digunakan menjadi seefisien mungkin.

Dalam mengetes penggunan kompleksitas algoritma untuk membuat game yang efisien ini penulis menggunakan sebuah game bernama “Brighsouls” yang penulis buat bersama teman-teman penulis untuk tugas mata kuliah Algoritma dan Struktur Data yang penulis ambil untuk semester ini. “Brightsouls” adalah game dimana pemain bisa memainkan seorang karakter yang bisa berjalan menyelusuri sebuah peta dimana pada peta terdapat berbagai macam objek dan karakter lain seperti dinding, obat, dan musuh. Jika karakter berpapasan dengan musuh, karakter akan melawan musuh tersebut. Jika karakter pemain menang, musuh akan hilang dari map. Jika musuh menang, pemain akan melihat kredit dan game akan berhenti.

Salah satu fitur kecil dari game ini adalah mode penjelajahan. Fitur ini memungkinkan pengguna untuk menggerakan karakter dalam game di sekitar peta. Dalam membuat fitur tersebut, tim menggunakan struktur data berbentuk matriks dimana setiap elemen matriks menyimpan variabel apa yang ada pada koordinat pada peta tersebut. Matriks peta tersebut perlu diisi ulang dalam periode tertentu untuk menunjukkan kepada pengguna tentang kondisi dalam game.

Dalam mengetes game ini, penulis akan menggunakan 2 fungsi yaitu mengisi elemen matriks dengan karakter pemain dan mengisi elemen matriks dengan musuh. Setiap fungsi akan dites menggunakan 2 buah algoritma dimana algoritma satu dengan yang lain memiliki kompleksitas yang berbeda. Setiap algoritma akan diimplementasi pada sebuah program dan dihitung waktu yang telah berjalan dari awal sampai akhir algoritma. Setiap algoritma akan dites pada ukuran data yang berbeda-beda berdasarkan suatu variabel N yang mengatur ukuran peta dan banyaknya karakter musuh pada peta. Pengukuran dilakukan menggunakan hardware yang sama dan perbedaan antara program hanya terletak di algoritma yang dituliskan disini saja. Program juga dicompile menggunakan compiler yang sama. Waktu diukur mulai dari tepat sebelum algoritma yang diukur sampai tepat selesainya algoritma diukur sehingga pengukuran tidak mengambil waktu inisialisasi dan sebagainya melainkan hanya algoritma objek dari makalah ini saja. Ini semua dilakukan untuk meminimalisir faktor dari luar yang dapat mempengaruhi hasil dari pengukuran.

A. Algoritma Pengisian Matriks dengan Karakter pemain pada Brightsouls

Dalam mengisi ulang peta, game perlu memerhatikan posisi player yang disimpan dalam bentuk koordinat yaitu absis dan ordinat. Contoh X dan Y. Matriks peta akan disimpan di dalam sebuah matriks bernama M berukuran NxN dimana parameter N bisa diubah-untuk mengetes kinerja algoritma kali ini pada ukuran data-data tertentu. Setiap elemen matriks M perlu diisi dengan ‘-‘ jika pada koordinat tersebut ada jalan (tidak ada apa-apa) dan ‘P’ jika pada koordinat tersebut ada karakter pemain. Pada program yang sesungghunya isi elemen matriks memiliki kemungkinan isi yang lain. Namun agar penggunaan kompleksitas algoritma bisa terlihat lebih jelas, pada kali ini algoritma pengisian peta telah disederhanakan.

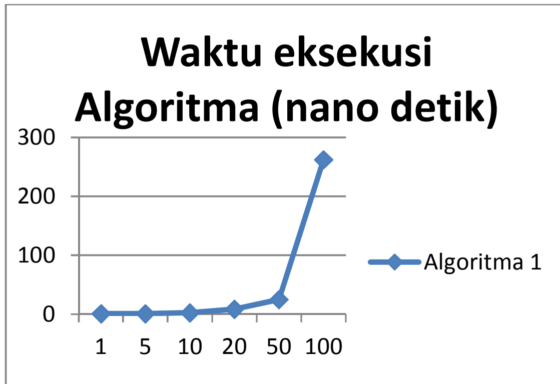
i. Algoritma 1

```
for( i=1 ; i<=N ; i++){
        for(j=1 ; j<=N ; j++){
                if( i==X && j==Y ){
                        Elmt( M , i , j ) = 'P';
                }
                else{
                        Elmt( M , i , j ) = '-';
                }
        }
}
```

Pada algoritma yang pertama, program akan memroses matriks dari baris ke-1 kolom ke-1 sampai baris ke-N kolom ke-N. Program akan memroses menyusuri kolom dari satu baris kemudian pindah ke baris selanjutnya jika sudah selesai. Untuk itu algoritma menggunakan dua buah loop. Loop yang pertama untuk memroses baris dan di dalamnya ada loop kedua untuk memroses kolom. Di dalam loop yang kedua terdapat sebuah pembagian kasus. Di sini program akan mengecek apakah kolom dan baris matriks yang sedang diproses sama dengan koordinat dari karakter pemain. Jika didapat sama, elemen matriks yang sedang diproses akan diisi dengan ‘P’. Jika didapat tidak sama, elemen matriks akan diisi dengan ‘-‘. Proses dilaksanakan sampai seluruh elemen matriks telah selesai diproses. Pada akhir proses, elemen matriks akan berisi ‘-‘ kecuali pada koordinat dimana karakter pemain berada dimana elemen matriks akan berisi ‘P’.

Bisa dilihat bahwa kompleksitas algoritma di atas bernilai $T(N) = 2N^2$. Dalam mengetes algoritma ini penulis memvariasikan nilai N untuk melihat bagaimana pengaruh kenaikan jumlah data terhadap waktu eksekusi

algoritma. Didapat hasil sebagai berikut: pada N=1 waktu eksekusi program 0,001 ms, N=5 0,001 ms, N=10 0,003ms, N=20 0,009 ms, N=50 0,025 ms, N=100 0,262ms. Walau ada perbedaan antara kenaikan nilai N dan kenaikan waktu eksekusi dengan formula T(N) yang sudah kita buat, data ini menunjukkan bahwa jumlah data memang mempengaruhi waktu eksekusi algoritma.

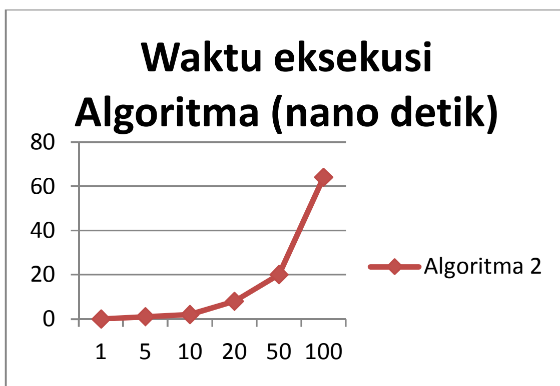


ii. Algoritma 2

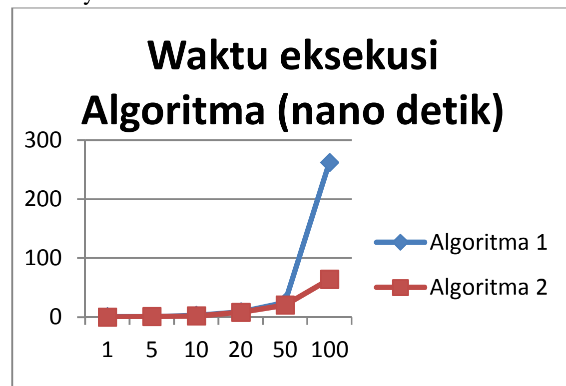
```
for(i=1;i<=N;i++){
            for(j=1;j<=N;j++){
                        Elmt(M,i,j)='-';
            }
}
Elmt(M1,X,Y)='P';
```

Algoritma yang kedua ini mirip dengan algoritma yang pertama. Perbedaannya ada pada pengisian karakter 'P' nya. Pada algoritma kedua ini pengisian karakter 'P' dibedakan dengan pengisian karakter '-'. Dengan begini loop bisa berlangsung dengan lebih cepat walaupun ada operasi tambahan setelah selesai loop.

Kompleksitas algoritma di atas memiliki nilai T(N) = N^2+1. Nilai N divariasi sehingga mendapatkan data sebagai berikut. Pada N=1 waktu eksekusinya 0.000000ms (ini berarti waktu eksekusi sangat cepat sampai tidak bisa terhitung), N=5 0,001ms, N=10 0,002 ms, N=20 0,008 ms, N=50 0,002 ms, N=100 0,064 ms.



Sebenarnya jika kita membandingkan antara kompleksitas algoritma 1 dengan algoritma 2 tidak ada banyak berpedaan karena keduanya memiliki nilai notasi Big O yang sama yaitu O(N^2). Perbedaan kompleksitas algoritma hanya bisa dilihat jika kita melihat formula kompleksitas algoritma murninya yaitu untuk algoritma 1 T(N)=2N^2 dan untuk algoritma 2 T(N)=N^2 +1 dimana terlihat pada nilai N yang besar algoritma 2 lebih cepat dari pada algoritma 1. Jika kita melihat data dari hasil pengukuran dapat dilihat bahwa memang ada perbedaan dalam waktu eksekusi algoritma dimana algoritma 2 dapat menyelesaikan proses lebih cepat dibanding algoritma 1. Bahkan hingga 4 kali lebih cepat pada N=100. Ini membuktikan bahwa algoritma 2 dapat melakukan pekerjaan lebih cepat dibandingkan algoritma 1 dan lebih tepat untuk digunakan dalam game *brightsouls* sesuai dengan hasil formula analisi kompleksitas algoritma sebelumnya.



B. Algoritma Pengisian Matriks dengan musuh pada Brightsouls

Game *brightsouls* juga menyimpan data musuhnya. Ini dilakukan agar ketika pemain memasuki bagian peta yang sudah dilewati sebelumnya, ia akan menemukan bahwa musuh berada di tempat yang sama dengan sebelumnya. Data musuh disimpan dalam sebuah list dan menyimpan data berupa koordinat musuh, id musuh, dan level musuh individu. Misalkan kita ingin mengisi matriks M berukuran NxN dengan data musuh yang berjumlah N pula. Sama dengan sebelumnya, untuk menyederhanakan algoritma, kali ini kita hanya mengisi matriks dengan '-' jika tidak ada musuh dan 'E' jika ada musuh.

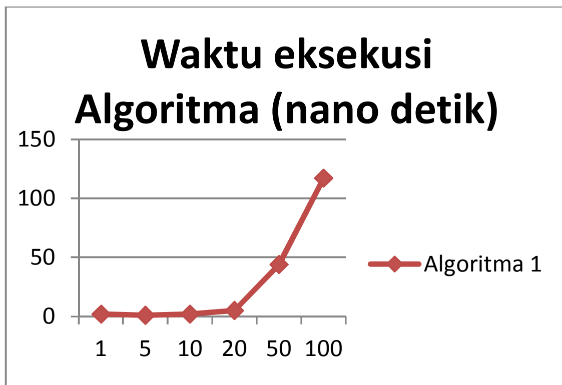
i. Algoritma 1

```
for(i=1;i<=N;i++){
        for(i=j;j<=N;j++){
                Elmt(M,i,j)='-';
                A=First(L);
                while(A!=Nil){
                        if(Absis(Pos(A))==j          &&
Ordinat(Pos(A))==i){
                                Elmt(M,i,j)='E';
```

```
                    }
                    A=Next(A);
            }
    }
}
```

Pada algoritma yang pertama ini program pertama-tama melakukan 2 loop ini dilakukan untuk memroses baris dan kolom matriks secara berurutan. Matriks lalu diisi karakter '-' sebagai dugaan awal. Program lalu akan mengecek list musuh untuk menentukan apakah pada koorsidat yang sedang diproses pada peta terdapat seorang musuh. Proses ini dilakukan menggunakan loop while sehingga loop akan berhenti ketika pointer A sudah berada di ujung list. Ini memungkinkan seluruh isi list bisa diproses. Jika program menemukan musuh yang berkoordinat sama dengan elemen matriks yang sedang diproses, elemen matriks tersebut akan diisi dengan karakter 'E'.

Di awal sudah ditentukan bahwa jumlah musuh sama dengan nilai N, sehingga kita bisa tahu bahwa algoritma di atas memiliki nilai kompleksitas algoritma sekitar $T(N)=2N^3 + 2N^2$. Kali ini nilai N mempengaruhi ukuran peta dan jumlah musuh. Di dapat waktu eksekusi algoritma pada N=1 0,002 ms, N=5 0,001 ms, N=10 0,002 ms, N=20 0,005 ms, N=50 0,044 ms, N=100 0,117 ms. Seperti pada kasus sebelumnya kenaikan waktu eksekusi tidak tepat sama seperti formula tetapi sudah memberikan data bahwa kenaikan waktu eksekusi disebabkan kenaikan jumlah data yang diproses.

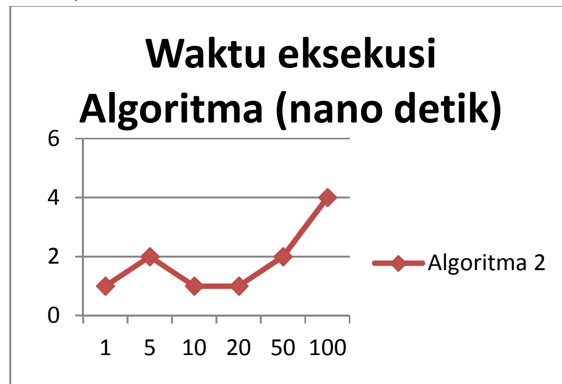


ii. Algoritma 2

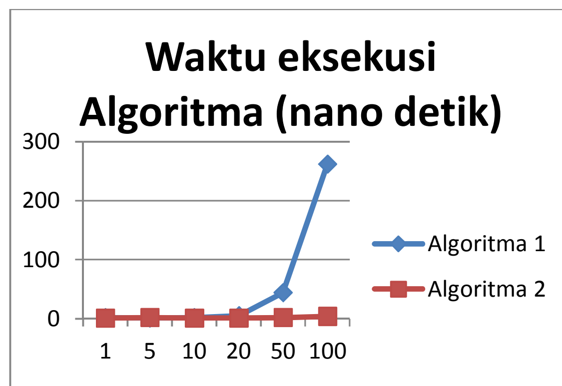
```
for(i=1;i<=N;i++){
        for(i=j;j<=N;j++){
                Elmt(M,i,j)='-';
        }
}
A=First(L);
while(A!=Nil){
        Elmt(M,Absis(Pos(A)), Ordinat(Pos(A)))='E';
        A=Next(A);
}
```

Pada algoritma yang kedua ini pertama-tama seluruh elemen matriks diisi dengan karakter '-'. Lalu kemudian barulah program melihat isi dari list musuh dan kemudian mengisi elemen matriks yang memiliki koordinat yang sama dengan list musuh dengan karakter 'E'.

Algoritma di atas memiliki kompleksitas algoritma bernilai kira-kira $T(N)=N^2+2N+1$. Didapat data melalui menjalankan program menggunakan berbagai variasi N dimana pada N=1 waktu eksekusi 0,001 ms, N=5 0,002 ms, N=10 0,001 ms, N=20 0,001 ms, N=50 0,002 ms, dan N=100 0,004 ms.



Dengan membandingkan data yang sudah kita dapatkan dari kedua algoritma kita bisa membandingkan bagaimana waktu eksekusi kedua algoritma pada program yang berjalan di dunia nyata. Algoritma 1 memiliki kompleksitas $T(N)=2N^3 + 2N^2$ dan algoritma 2 memiliki kompleksitas $T(N)=N^2+2N+1$. Pada nilai N yang besar, algoritma 1 akan memakan waktu yang lebih besar daripada algoritma 2 bisa dilihat dari notasi Big O dimana algoritma 1 memiliki $O(N^3)$ dimana algoritma 2 memiliki $O(N^2)$ sehingga dipresiksi bahwa algoritma 2 akan berjalan lebih cepat. Sesuai dengan data dari kedua algoritma, didapat bahwa algoritma 2 memang berjalan lebih cepat dan memakan waktu yang lebih sedikit dibandingkan dengan algoritma 1. Sehingga terbukti bahwa kompleksitas algoritma berdampak pada waktu eksekusi program.



V. SIMPULAN

Sesuai dengan data yang telah didapatkan, bisa

disimpulkan bahwa kompleksitas suatu algoritma memang memiliki dampak yang nyata terhadap waktu eksekusi program dalam kasus ini, sebuah game. Oleh karena itu, dapat disimpulkan bahwa analisi kompleksitas algoritma dapat digunakan dalam mengembangkan game yang bisa berjalan dengan lebih lancar. Pengembang game baiknya selalu menganalisis kompleksitas dari algoritma yang dipakai dalam gamenya dan mencari apakah ada alternatif yeng memiliki kompleksitas yang berfungsi lebih cepat dalam memroses data dalam jumlah besar.

VI. UCAPAN TERIMA KASIH

Pertama-tama penulis ingin mengucapkan terima kasih kepada Tuhan yang maha Esa karena berkat karunianya saya dapat menyelesaikan makalah ini. Terima kasih sebesar-besarnya juga saya berikan kepada kedua orang tua saya yang sudah menafkahi saya sehingga saya bisa mempelajari informatika di Institut Teknologi Bandung ini. Penulis juga ingin berterima kasih kepada Ibu Harlili S. selaku dosen mata kuliah Matematika Diskrit yang telah dengan sabar membimbing saya sehingga saya bisa memahami segala konsep dari mata kuliah tersebut. Yang terakhir, penulis juga ingin mengapresiasi teman-teman penulis baik dari dalam maupun luar jurusan Informatika ITB yang telah memberikan dukungan sehingga akhirnya makalah ini bisa selesai.

REFERENSI

[1] http://fortune.com/2016/02/16/video-game-industry-revenues-2015/ diakses tanggal 7 Desember 2016 pk. 10.56 WIB
[2] http://discrete.gr/complexity/ diakses tanggal 7 Desember 2016 pk. 12.55
[3] http://discrete.gr/complexity/ diakses tanggal 7 Desember 2016 pk. 12.55
[4] Munir. Rinaldi, *Diktat Kuliah IF2120 Matematika Diskrit*. Bandung. Program Studi Teknik Informatika Sekolah Teknik Elektro dan Informatika Institut Teknologi Bandung, 2016, hal. 45-46
[5] http://www.gamedev.net/page/resources/_/technical/artificial-intelligence/a-pathfinding-for-beginners-r2003 diakses tanggal 9 Desember pk.08.19

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, 8 Desember 2016

ttd

Nama dan NIM