Analisis Kompleksitas Algoritma Pengurutan Terburuk, *BogoSort*

Muhammad Roihan - 13522152
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*13522152@std.stei.itb.ac.id*

***Abstract*—*BogoSort* adalah sebuah algortima pengurutan yang tidak efisien dan tidak praktis, yang mengandalkan keberuntungan acak untuk mengurutkan sebuah daftar elemen. Metode ini secara sederhana mengacak ulang elemen-elemen dalam daftar dan memeriksa apakah urutannya sudah benar. Jika tidak, proses pengacakan diulang. *BogoSort* tidak memiliki kinerja yang dapat diprediksi dan cenderung membutuhkan waktu yang sangat lama untuk mengurutkan daftar elemen yang signifikan. Meskipun *BogoSort* menawarkan pendekatan yang unik terhadap masalah pengurutan, kekurangan efisiensi dan ketidakpastian waktu eksekusi membuatnya tidak cocok untuk penggunaan praktis dan lebih sering digunakan sebagai lelucon dalam dunia komputasi.**

***Keywords*—Analisis, *BogoSort,* Kompleksitas, Pengurutan.**

I. PENDAHULUAN

Algoritma pengurutan adalah suatu proses pengaturan elemen-elemen dalam suatu urutan tertentu agar sesuai dengan aturan tertentu. Dalam dunia pemrograman, ada berbagai algoritma pengurutan yang dikembangkan dengan tujuan untuk menciptakan algortima yang sangkil (*efficient*). Kesangkilan algoritma diukur dari waktu (*time*) yang diperlukan untuk menjalankan algoritma dan ruang (*space*) memori yang dibutuhkan oleh algoritma tersebut. Namun, terdapat sebuah anomali dalam dunia algoritma pengurutan dimana terdapat algoritma yang seharusnya diciptakan agar mencapai kinerja yang optimal, justru sebaliknya, yaitu *BogoSort*.

BogoSort juga dikenal sebagai *permutation sort, stupid sort, slow sort, shotgun sort* atau *monkey sort* adalah sebuah algortima paling tidak efektif yang pernah dibayangkan. Selain itu, nama "*bogosort*" sendiri merupakan gabungan kata dari "*bogus*" dan "*sort*," yang dengan akurat mencerminkan sifat algoritma ini. Karena bergantung pada pengacakan acak dan tidak menjamin terminasi, kompleksitas waktu rata-rata dan terburuk dari *BogoSort* adalah tidak terbatas. Algoritma ini akan melakukan permutasi untuk setiap elemen sampai kombinasi urutan yang sesuai dengan aturan tertentu.

Makalah ini bertujuan untuk menyelidiki dan menganalisis kompleksitas algoritma *BogoSort,* yang terkenal karena keunikan pendekatannya yang sangat tidak efisien. Analisis akan melibatkan pemahaman tentang cara kerja *BogoSort*, serta eksplorasi terhadap kinerja dan sifat algoritma tersebut dalam konteks pengurutan. Melalui analisis kompleksitas algoritma *BogoSort*, diharapkan makalah ini dapat memberikan pemahaman yang lebih baik tentang sifat-sifat unik algoritma ini, memberikan wawasan tentang perbandingannya dengan algoritma pengurutan lainnya, dan menyoroti pentingnya pemahaman kompleksitas algoritma dalam merancang solusi efisien.

II. LANDASAN TEORI

A. Kompleksitas Algoritma

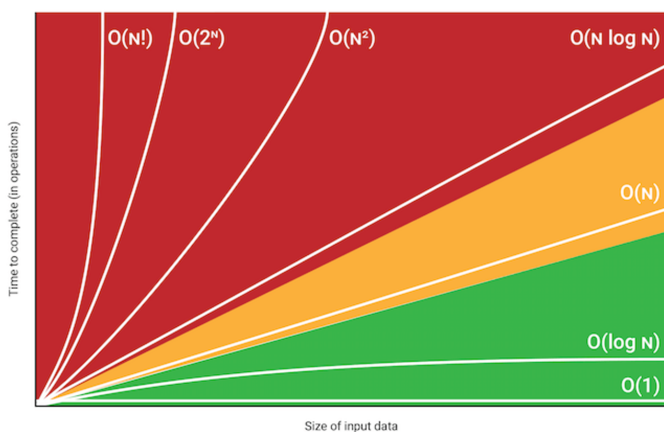
Dalam ranah pemrograman, suatu algoritma tidak hanya harus benar secara logika, melainkan juga harus optimal dalam efisiensinya. Tingkat efisiensi algoritma diukur melalui waktu yang diperlukan untuk menjalankannya dan jumlah ruang memori yang diperlukan oleh algoritma tersebut. Sebuah algoritma dikatakan efisien jika mampu meminimalkan kebutuhan waktu dan ruang memori. Untuk menilai seberapa efisien suatu algoritma dapat beroperasi, diperlukan perhitungan terkait waktu dan penggunaan memori.

Kompleksitas algoritma menjadi suatu metrik penting dalam mengevaluasi kinerja algoritma, khususnya terkait dengan kemampuannya menyelesaikan tugas dengan efisiensi maksimal. Analisis kompleksitas algoritma melibatkan dua dimensi utama, yaitu kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu mengukur durasi waktu yang dibutuhkan oleh algoritma untuk menyelesaikan suatu tugas, sedangkan kompleksitas ruang mengukur seberapa besar alokasi memori yang diperlukan oleh algoritma. Dengan memahami dan menilai kedua aspek ini, kita dapat mengukur dan membandingkan efisiensi relatif dari berbagai algoritma dalam menangani berbagai skenario.

Kompleksitas waktu menjadi penentu utama seberapa cepat algoritma dapat menyelesaikan tugasnya dan diukur melalui waktu eksekusi riilnya, yang direpresentasikan dalam satuan detik, ketika algoritma dijalankan oleh komputer. Saat program merepresentasikan suatu algoritma, waktu eksekusi riilnya menjadi parameter kritis untuk mengevaluasi sejauh mana algoritma tersebut efisien dalam menangani masukan tertentu.

Sementara itu, kompleksitas ruang mengukur seberapa efisien algoritma menggunakan memori selama operasinya. Hal ini mengacu pada berapa banyak alokasi memori yang diperlukan oleh algoritma untuk menyimpan variabel, data, dan struktur lainnya selama berjalannya program. Dalam konteks ini, efisiensi memori menjadi faktor penting untuk memastikan bahwa algoritma tidak hanya cepat tetapi juga hemat dalam penggunaan sumber daya.

Dalam konteks analisis kompleksitas algoritma, penggunaan notasi Big O menjadi landasan kritis untuk mengukur kinerja dan efisiensi suatu algoritma. Notasi Big O menyajikan cara yang ringkas dan informatif untuk menyatakan sejauh mana waktu eksekusi atau penggunaan memori suatu algoritma meningkat seiring dengan pertambahan ukuran masukan. Notasi Big O adalah suatu notasi matematis yang digunakan untuk menyatakan batasan atas pertumbuhan algoritma. Secara khusus, Big O menyajikan estimasi terburuk dari kinerja algoritma saat ukuran masukan mendekati tak terbatas. Notasi ini berguna untuk memberikan gambaran tentang tingkat pertumbuhan relatif dari algoritma tanpa memperhatikan faktor konstan. Berikut ilustrasi beberapa jenis kompleksitas menggunakan notasi Big O.



Gambar 2.1 Ilustrasi Jenis Kompleksitas Menggunakan Notasi Big O
Sumber : https://towardsai.net/p/programming/big-o-notation-what-is-it

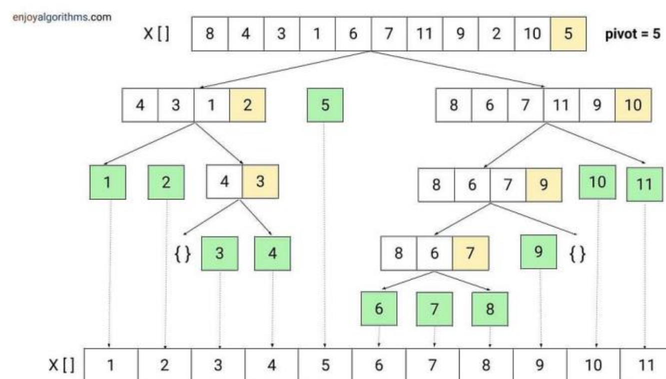
Berdasarkan ilustrasi di atas dapat dilihat bahwa jenis kompleksitas tercepat adalah O(1). Notasi O(1) menunjukkan bahwa kompleksitas waktu suatu algoritma konstan, artinya waktu eksekusi tidak bergantung pada ukuran input n. Algoritma dengan kompleksitas O(1) memiliki kinerja yang sangat efisien, seperti mengakses elemen di dalam array atau variabel. Selanjutnya ada algoritma dengan kompleksitas O(log n) menunjukkan bahwa waktu eksekusi tumbuh secara logaritmik seiring dengan peningkatan ukuran input n. Contoh algoritma dengan kompleksitas logaritmik adalah *binary search*, di mana setiap iterasi mengurangi setengah elemen yang perlu diperiksa.

Notasi O(n) menunjukkan bahwa kompleksitas waktu suatu algoritma tumbuh secara linier dengan ukuran input n. Algoritma ini memiliki proporsi waktu eksekusi yang sebanding dengan ukuran inputnya. Contoh algoritma O(n) adalah iterasi linear melalui elemen-elemen array. Selanjutnya kompleksitas waktu O(n log n) umumnya ditemukan dalam algoritma pengurutan efisien, seperti *mergesort* dan *heapsort*. Algoritma dengan kompleksitas ini memiliki pertumbuhan waktu yang lebih cepat daripada O(n) tetapi lebih lambat daripada O(n^2).

Berikutnya algoritma dengan kompleksitas O(2^n) memiliki pertumbuhan waktu yang eksponensial, artinya waktu eksekusinya tumbuh sangat cepat seiring dengan peningkatan ukuran input n. Algoritma ini sering ditemukan dalam permasalahan pemecahan kombinatorial, seperti permutasi atau subset. Terakhir adalah Notasi O(n!) menunjukkan bahwa kompleksitas waktu suatu algoritma tumbuh faktorial dengan ukuran input n. Algoritma dengan kompleksitas ini memiliki pertumbuhan waktu yang sangat cepat dan sangat tidak efisien. Contoh kasus ini sering ditemui dalam permasalahan kombinatorial yang melibatkan permutasi penuh.

B. Algoritma Pengurutan

Algoritma pengurutan adalah suatu metode atau langkah-langkah yang ditetapkan secara sistematis untuk menyusun kembali elemen-elemen dalam suatu struktur data sehingga mengikuti urutan tertentu. Algoritma pengurutan paling umum dilakukan untuk mengurutkan elemen-elemen dalam urutan numerik atau *leksikografis*, baik secara *ascending* ataupun *descending*. Pemilihan algoritma pengurutan yang baik seringkali mampu mengurangi kompleksitas suatu masalah. Oleh karena itu, algoritma pengurutan memiliki peran yang sangat penting dalam ilmu komputer.



Gambar 2.2 Ilustrasi Algoritma Quicksort
Sumber : https://www.enjoyalgorithms.com/blog/quick-sort-algorithm

Beberapa algoritma pengurutan umum telah menjadi penentu kritis dalam dunia pemrosesan data. Salah satunya adalah *quicksort*, yang mengadopsi pendekatan rekursif dengan memilih elemen pivot dan mempartisi array menjadi dua subarray. *Quicksort* dikenal memiliki kompleksitas waktu O(n log n) untuk kasus terbaiknya, namun dapat mencapai O(n^2) dalam kasus terburuk. Dalam hal kompleksitas ruang, *quicksort* memiliki kompleksitas sebesar O(log n).

Selain *quicksort*, terdapat beragam algoritma pengurutan lainnya, seperti *mergesort*, *heapsort*, *bubblesort*, dan sebagainya, yang masing-masing menawarkan pendekatan yang unik untuk menangani tugas pengurutan. *Mergesort*, misalnya, memisahkan data menjadi dua bagian, mengurutkan setiap bagian secara terpisah, dan kemudian menggabungkan hasilnya. *Heapsort*, di sisi lain, menggunakan struktur data heap untuk mengurutkan elemen. Kemudian, *bubblesort* menjadi algoritma yang sederhana tetapi kurang efisien dengan mengulang pembandingan dan pertukaran elemen secara berulang.

Pentingnya memilih algoritma pengurutan yang cocok dengan tugas tertentu menunjukkan bahwa kita perlu benar-

benar mengerti bagaimana setiap algoritma bekerja dan seberapa sulit atau mudahnya mereka dalam menangani tugas khusus. Dengan melakukan penelitian dan pemahaman yang baik tentang algoritma pengurutan, kita dapat menentukan algoritma yang paling efisien untuk menyelesaikan masalah pengurutan yang kita hadapi. Jadi, pengetahuan yang baik tentang algoritma membantu kita membuat pilihan yang tepat sesuai dengan kebutuhan tugas tertentu.

C. BogoSort

Bogosort merupakan algoritma pengurutan yang tidak efisien dan tidak praktis. Cara kerjanya sangat unik karena mengandalkan keberuntungan acak untuk menyusun daftar elemen. Ide dasarnya adalah dengan secara sederhana mengacak-acak elemen dalam daftar dan kemudian memeriksa apakah urutannya sudah benar. Jika belum, proses pengacakan diulang. Untuk memberikan gambaran lebih jelas, mari kita lihat contohnya dengan sebuah array yang berisi 5 elemen.

4	12	3	32	16
-3	32	4	12	16
16	32	12	4	-3
-3	4	12	16	32
-3	4	12	16	32

Gambar 2.3 Ilustrasi Pengurutan Array Menggunakan BogoSort
Sumber : https://chercher.tech/kotlin/bogosort-kotlin

Berdasarkan ilustrasi tersebut, dapat diamati bahwa pertama-tama terdapat sebuah *array* yang terdiri dari lima elemen, yaitu 4, 12, -3, 32, 16, dan ingin diurutkan secara menaik (*ascending*). Awalnya, elemen-elemen dalam *array* diacak sehingga menghasilkan urutan -3, 32, 4, 12, 16. Karena urutan tersebut masih belum sesuai dengan yang diinginkan, *array* tersebut kembali diacak. Terlihat bahwa proses pengacakan ini terulang sebanyak empat kali sebelum akhirnya *array* berhasil diurutkan secara menaik. Ilustrasi ini mencerminkan cara kerja *BogoSort*, di mana pengurutan dilakukan dengan cara yang sangat tidak efisien dan tidak terencana..

D. Analisis Kompleksitas

Analisis kompleksitas adalah proses penilaian dan pemahaman tentang seberapa efisien suatu algoritma dalam menangani masukan data tertentu. Analisis ini melibatkan evaluasi kinerja algoritma berdasarkan faktor-faktor seperti waktu eksekusi dan penggunaan memori.

Analisis kompleksitas algoritma sangat penting dalam pengembangan perangkat lunak karena membantu dalam pemilihan algoritma yang tepat untuk penyelesaian suatu masalah. Pemahaman tentang kompleksitas algoritma dapat membantu untuk memprediksi bagaimana kinerja suatu algoritma akan berubah seiring dengan peningkatan ukuran masukan. Dengan kata lain, kompleksitas algoritma memberikan gambaran tentang seberapa baik algoritma dapat menangani skenario penggunaan yang berbeda.

Pentingnya analisis kompleksitas algoritma juga terletak pada kemampuannya untuk membandingkan dan memilih solusi yang paling efisien di antara beberapa alternatif. Dalam dunia yang terus berkembang dengan ukuran data yang semakin besar, memilih algoritma dengan kompleksitas yang sesuai dapat membuat perbedaan signifikan dalam kinerja aplikasi atau sistem.

III. Analisa Kompleksitas Pengurutan Menggunakan Bogosort

A. Implementasi BogoSort

Berikut implementasi algoritma *BogoSort* dalam bahasa C++.

```cpp
// Cek apakah array telah terurut
bool isSorted(int a[], int n)
{
    while (--n > 0)
        if (a[n] < a[n - 1])
            return false;
    return true;
}

// Mengacak elemen array
void shuffle(int a[], int n)
{
    for (int i = 0; i < n; i++)
        swap(a[i], a[rand() % n]);
}

// Mengurutkan array a[0..n-1]
// menggunakan BogoSort
void bogosort(int a[], int n)
{
    // Jika array belum terurut lakukan
    // pengurutan ulang
    while (!isSorted(a, n))
        shuffle(a, n);
}
```

Gambar 3.1 Implementasi BogoSort Dalam Bahasa C++

Mekanisme *BogoSort* melibatkan penyusunan elemen-elemen dalam suatu *array* secara acak hingga mencapai urutan yang diinginkan. Dalam implementasi di atas, terdapat tiga prosedur utama yang digunakan. Proses pertama merupakan prosedur bogosort() yang memiliki dua buah parameter yaitu sebuah *array* yang ingin diurutkan dan panjang dari *array* itu sendiri. Awalnya, prosedur ini memeriksa apakah elemen-elemen dalam *array* telah terurut sesuai dengan urutan yang ditentukan dengan menggunakan prosedur isSorted(), dalam hal

ini, urutan secara menaik. Jika belum terurut, prosedur akan memanggil prosedur shuffle() untuk mengacak elemen-elemen dalam array.

Pada prosedur shuffle, fungsi bawaan dari C++, yaitu swap(), digunakan untuk menukar posisi dua elemen dalam array secara acak. Demi pengacakan yang lebih efektif, fungsi rand() juga digunakan untuk menghasilkan angka acak yang mendukung proses pengacakan. Dengan pendekatan ini, implementasi *BogoSort* mencoba menyimulasikan pengurutan elemen dengan cara yang sepenuhnya tergantung pada keberuntungan acak, menciptakan pendekatan yang sangat tidak efisien namun unik dalam menyusun suatu *array*.

B. Uji Coba Program

Untuk Pengujian program penulis menggunakan 3 buah *test case* yaitu *array* berisi 5, 10 dan 100 elemen. Berikut hasilnya.

```
Sorted array :
1 35 42 68 70
Time taken by function: 9 microseconds
```

Gambar 3.2 Uji coba BogoSort menggunakan 5 buah elemen

Dapat diperhatikan bahwa dalam uji coba yang melibatkan 5 elemen, waktu eksekusi program masih berada dalam kisaran yang wajar, yakni sekitar 9 μs.

```
Sorted array :
1 25 35 42 59 63 65 68 70 79
Time taken by function: 1862565 microseconds
```

Gambar 3.3 Uji coba BogoSort menggunakan 10 buah elemen

Dapat diamati bahwa dalam uji coba yang melibatkan 10 elemen, terjadi perbedaan yang signifikan dalam waktu eksekusi program jika dibandingkan dengan waktu eksekusi pada 5 elemen. Awalnya, waktu eksekusi untuk 5 elemen hanya sekitar 9 μs, namun menjadi 1862565 μs pada uji coba dengan 10 elemen. Perbedaan yang mencolok ini menunjukkan peningkatan drastis dalam kompleksitas waktu program seiring dengan pertambahan jumlah elemen dalam proses uji coba. Sebagai perbandingan, penulis juga menjalankan eksperimen pengurutan terhadap 10 elemen menggunakan algoritma *Bubble Sort*. Berikut adalah hasil dari percobaan tersebut.

```
Sorted array :
1 25 35 42 59 63 65 68 70 79
Time taken by function: 0 microseconds
```

Gambar 3.4 Uji coba BubbleSort menggunakan 10 buah elemen

Dapat diperhatikan bahwa hasil pengujian yang mencakup 10 elemen menggunakan algoritma *BubbleSort* menunjukkan waktu eksekusi yang signifikan lebih cepat dibandingkan dengan penggunaan *BogoSort*. Dalam uji coba dengan *BubbleSort*, waktu eksekusi mencapai nilai 0 μs, menandakan bahwa waktu eksekusinya jauh lebih singkat daripada satuan mikrodetik. Perbandingan ini menggambarkan perbedaan kinerja yang mencolok antara *BubbleSort* dan *BogoSort* dalam konteks pengurutan elemen.

Pengujian yang melibatkan 100 elemen tidak dapat disajikan dalam makalah ini karena waktu eksekusi yang belum diketahui. Penulis memulai pengujian pada tanggal 9 Desember 2023, pukul 16:00 WIB, namun pada tanggal 10 Desember 2023, pukul 13:40 WIB, program masih berjalan tanpa menunjukkan hasil akhir dari pengujian tersebut. Situasi ini menggambarkan bahwa pengujian dengan jumlah elemen yang signifikan memerlukan waktu eksekusi yang cukup lama, sehingga hasilnya belum dapat dipresentasikan pada tahap tertentu.

C. Kompleksitas Waktu

Analisis kompleksitas waktu pada *BogoSort* adalah sebuah tantangan karena pendekatannya yang sangat bergantung pada faktor keberuntungan acak. Pada dasarnya, *BogoSort* mencoba menyusun suatu *array* dengan mengacak elemen-elemen secara acak, dan di kondisi paling ideal, algoritma ini memiliki potensi menjadi salah satu algoritma pengurutan paling cepat dengan kompleksitas waktu O(1). Selanjutnya akan ditentukan kompleksitas rata-rata dari *BogoSort.* Untuk menentukan kompleksitas rata-rata kita perlu menjabarkan beberapa hal.

Pertama *BogoSor*t bekerja dengan mengacak ulang elemen array dan memeriksa apakah urutannya sudah sesuai. Dengan n elemen, ada n! cara yang berbeda untuk mengatur elemen tersebut. Pada setiap iterasi, *BogoSort* melakukan pengacakan dan memeriksa apakah array sudah terurut. Dalam kasus rata-rata, kita dapat berasumsi bahwa setiap iterasi akan menghasilkan pengurutan yang benar dengan kemungkinan 1/n!, karena hanya satu cara yang benar dari n! cara yang mungkin. Dengan demikian, kompleksitas waktu rata-rata *BogoSor*t menjadi O(n*n!) dikarenakan kita melakukan n! rata-rata permutasi acak dan masing-masing membutuhkan waktu O(n). Meskipun terdapat variasi dalam penilaian ini tergantung pada asumsi yang dibuat tentang distribusi acak, nilai eksaknya dapat sangat bervariasi. Namun, nilai O(n*n!) memberikan gambaran tentang tingginya kompleksitas rata-rata dan ketidakpastian dalam kinerja algoritma ini.

Bagaimana jika kita bandingkan O(n*n!) dengan kompleksitas terburuk dari *bubblesort* yaitu $O(n^2)$. Dalam notasi O(n*n!), n adalah ukuran input. O(n * n!) menunjukkan bahwa waktu eksekusi algoritma akan tumbuh dengan cepat seiring dengan peningkatan ukuran input. Contohnya, jika n = 3, kompleksitas waktu akan menjadi 3 * 3! = 18, dan jika n = 4, kompleksitas waktu akan menjadi 4 * 4! = 96. Jadi, dengan peningkatan kecil pada n, kompleksitas waktu dapat meledak secara signifikan.

$O(n^2)$ adalah kompleksitas waktu yang lebih rendah dibandingkan dengan O(n * n!). Dalam notasi ini, n adalah ukuran input. $O(n^2)$ menunjukkan bahwa waktu eksekusi algoritma tumbuh secara kuadrat dengan peningkatan ukuran input. Contohnya, jika n = 3, kompleksitas waktu akan menjadi $3^2 = 9$, dan jika n = 4, kompleksitas waktu akan menjadi $4^2 = 16$. Meskipun pertumbuhannya lebih cepat daripada O(n), namun tidak secepat O(n * n!). Dengan demikian, $O(n^2)$ jauh lebih

efisien daripada O(n * n!) karena pertumbuhannya lebih lambat, meskipun keduanya masih termasuk dalam kompleksitas tinggi dan perlu diperhatikan dalam pemilihan algoritma.

Kompleksitas waktu terburuk pada algoritma *BogoSort* sangat sulit ditentukan karena sifatnya yang sangat tidak efisien dan tergantung pada faktor keberuntungan acak. Dalam kasus terburuk, *BogoSort* dapat menghabiskan waktu yang sangat lama atau bahkan tidak pernah berhasil menyusun suatu array dengan benar. Oleh karena itu, kompleksitas waktu terburuk *BogoSort* dapat dianggap sebagai tidak terbatas atau O(∞).

D. Kompleksitas Ruang

Berdasarkan impelementasi pada subbab sebelumnya, kompleksitas ruang pada algoritma *BogoSort* adalah O(1). Penyebabnya adalah *BogoSort* hanya menggunakan array yang sudah diberikan sebagai input untuk diurutkan. Algoritma ini tidak menggunakan struktur data tambahan, tidak melakukan alokasi memori dinamis, dan tidak menyimpan informasi tambahan yang berkaitan dengan ukuran array atau proses pengurutan. Oleh karena itu, penggunaan memori *BogoSort* tetap konstan, tidak bergantung pada ukuran masukan, dan dapat dianggap sebagai O(1).

E. Pengaruh Fungsi rand() C++

Fungsi rand() dalam bahasa pemrograman C++ berfungsi untuk menciptakan nilai-nilai acak dalam program. Dalam situasi penggunaan algoritma *BogoSort*, di mana pengacakan elemen array secara acak adalah bagian pokok dari algoritma, penggunaan fungsi rand() memiliki dampak yang besar terhadap kinerja dan efisiensi algoritma tersebut. Namun, perlu diketahui bahwa fungsi rand() pada C++ dapat menghasilkan pola tertentu setiap kali program dijalankan, seperti yang terlihat pada hasil di bawah ini..

```
g++ -o random .\randomTest.cp
.\random.exe
Random Number :
42 68 35 1 70 25 79 59 63 65
.\random.exe
Random Number :
42 68 35 1 70 25 79 59 63 65
g++ -o random .\randomTest.cp
.\random.exe
Random Number :
42 68 35 1 70 25 79 59 63 65
```

Gambar 3.5 Uji coba fungsi rand() C++

Keadaan ini mengakibatkan jika kita menguji algoritma dua kali dengan menggunakan *array* yang sama, kita akan mendapatkan waktu eksekusi yang hampir sama. Hal ini seharusnya tidak terjadi, mengingat algoritma *BogoSort* seharusnya sulit diprediksi karena sangat bergantung pada faktor keberuntungan acak.

```
.\bogo3.exe
Sorted array :
1 25 35 42 59 63 65 68 70 79
Time taken by function: 1881030 microseconds
.\bogo3.exe
Sorted array :
1 25 35 42 59 63 65 68 70 79
Time taken by function: 1883146 microseconds
```

Gambar 3.6 Uji coba BogoSort dengan menggunakan array yang sama

IV. KESIMPULAN

Makalah ini membahas secara mendalam algoritma pengurutan yang sangat tidak efisien, yaitu *BogoSort*. *BogoSort* menghadirkan pendekatan unik dalam menyusun elemen-elemen array dengan cara yang sepenuhnya bergantung pada keberuntungan acak. Analisis kompleksitas *BogoSort* mengungkapkan bahwa algoritma ini memiliki kompleksitas waktu dan ruang yang sangat tinggi, membuatnya tidak praktis untuk digunakan dalam pengurutan data pada skenario dunia nyata.

BogoSort, dengan pendekatannya yang bergantung pada pengacakan elemen secara acak, menghasilkan kompleksitas waktu yang tidak terbatas atau sangat tinggi pada kasus terburuknya. Keberuntungan acak memainkan peran kritis dalam menentukan seberapa efisien algoritma ini. Walaupun *BogoSort* memiliki kompleksitas waktu yang sangat tinggi, kompleksitas ruangnya dapat dianggap relatif efisien karena algoritma ini hanya menggunakan array yang sudah ada sebagai input tanpa memerlukan alokasi memori tambahan atau struktur data yang kompleks. Meskipun *BogoSort* memberikan perspektif unik mengenai kompleksitas algoritma, tidak terdapat keunggulan praktis dalam pengaplikasiannya. Kemungkinan waktu eksekusi yang sangat lama atau bahkan tak terbatas menjadikannya tidak cocok untuk pengurutan data dalam konteks aplikasi dunia nyata.

Dalam makalah ini, juga dilakukan perbandingan antara algoritma *BogoSort* dan *BubbleSort*. Pada perbandingan yang melibatkan 10 elemen, terdapat perbedaan yang mencolok dalam waktu eksekusi program. *BubbleSort* menunjukkan kinerja yang jauh lebih baik dengan waktu eksekusi 0 µs, sedangkan *BogoSort* memerlukan waktu yang signifikan lebih lama, yakni 1862565 µs. Perbandingan ini menegaskan bahwa *BubbleSort* secara konsisten memberikan hasil yang lebih baik dalam konteks pengurutan elemen dibandingkan dengan *BogoSort*, mempertegas kelemahan praktis terkait dengan *BogoSort*.

Dalam makalah ini, juga dilakukan uji coba terkait pengaruh fungsi rand() C++ pada *BogoSort*. Didapatkan hasil bahwa jika dilakukan pengujian dengan dua buah array yang sama akan menghasilkan waktu yang eksekusi yang hampir sama. Hal ini dikarenakan fungsi rand() yang ada di C++ menghasilkan pola tertentu setiap kali program dijalankan.

Dengan demikian, kesimpulan makalah ini menekankan bahwa sementara *BogoSort* mungkin menarik sebagai eksperimen konseptual, dalam praktiknya, algoritma ini tidak

memberikan solusi yang efisien atau dapat diandalkan untuk masalah pengurutan data.

V. UCAPAN TERIMA KASIH

Dengan penuh rasa syukur, penulis ingin mengucapkan puji dan syukur kepada Tuhan Yang Maha Esa atas segala rahmat, karunia, serta taufik dan hidayah-Nya, yang telah memandu dan memberikan keberkahan sehingga penulis berhasil menyelesaikan makalah berjudul "Analisis Kompleksitas Algoritma Pengurutan Terburuk, *BogoSort*". Makalah ini disusun sebagai bagian dari tugas mata kuliah Matematika Diskrit IF2120.

Penulis juga mengucapkan terima kasih yang setinggi-tingginya kepada Bapak Dr. Ir. Rinaldi, M.T. dan Bapak Monterico Adrian, S.T., M.T., sebagai dosen pengajar dalam mata kuliah Matematika Diskrit IF2120 Kelas K3. Terima kasih atas dedikasi, bimbingan, dan pengajaran yang telah diberikan selama satu semester ini, memberikan kontribusi besar dalam pembentukan pemahaman kami sebagai mahasiswa. Penulis juga beterimakasih kepada seluruh pihak yang telah berkontribusi baik secara langsung maupun tidak langsung terhadap kelancaran penulisan makalah ini yang namanya tidak bisa disebutkan satu persatu. Penulis juga ingin menyampaikan permohonan maaf apabila terdapat kesalahan dalam penulisan makalah ini. Semoga makalah ini dapat bermanfaat dan memberikan sumbangan positif dalam pemahaman mata kuliah Matematika Diskrit IF2120.

REFERENCES

[1] Rinaldi Munir, “Kompleksitas Algoritma bagian 1”. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/24-Kompleksitas-Algoritma-Bagian1-2023.pdf diakses pada tanggal 9 Desember 2023.

[2] https://www.geeksforgeeks.org/bogosort-permutation-sort/ diakses pada tanggal 9 Desember 2023

[3] Ayman Patil, “What is Bogosort?”.https://aymanpatil.medium.com/what-is-bogosort-65f19a793c3b diakses pada tanggal 9 Desember 2023

[4] Adolfo Neto, “Bogosort: The Stupid Sorting Algoritm”.https://dev.to/adolfont/bogosort-the-stupid-sorting-algorithm-168f diakses pada tanggal 9 Desember 2023

[5] Jay Wengrow, “A Common-Sense Guide to Data Structures and Algorithms, Second Edition Level Up Your Core Programming Skills,”.

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, 10 Desember 2023

Muhammad Roihan, 13522152