Analisis Kompleksitas dan Performa Algoritma *Double Hashing Sort*

Ilham Bintang Nurmansyah - 13520102
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13520102@std.stei.itb.ac.id

***Abstrak*—Mengurutkan sebuah array dengan n elemen merupakan satu dari sekian banyak masalah yang sering dihadapi oleh berbagai bidang dirumpun keinformatikaan seperti basis data, graf, geometri komputasi, dan bioinformatika. Dewasa ini, sebuah algoritma sekuensial bernama *double hashing sort (DHS)* telah ditunjukkan melewati performa dari algoritma *quick sort* sebesar 10-25%. Dalam makalah ini, akan dibahas mengenai analisis kompleksitas dan performa dari algoritma tersebut. Analisis kompleksitas algoritma DHS yang dilakukan berdasarkan pada relasi antar ukuran input dan domain input elemen.**

***Keywords*—Algoritma, Array, D*ouble Hashing Sort*, Kompleksitas Algoritma, Performa Algoritma.**

I. PENDAHULUAN

Sorting atau pengurutan adalah hal yang penting dan mendasar dalam bidang ilmu informatika. Sorting digunakan dalam berbagai hal, seperti database, grafik, bioinformatika, dan lain-lain. Algoritma dari sorting merupakan hal yang penting, karena dengan data yang terurut, akan didapatkan *runtime* yang lebih cepat dan efisien. Sebagai contoh, untuk mencari elemen pada array yang belum terurut, diperlukan waktu O(n), sedangkan untuk array terurut, hanya diperlukan O(log n).

Kemudian, sorting bisa dilakukan dengan berbagai algoritma berbeda dengan berbagai strategi yang ada. Strategi yang dimaksud seperti *brute-force, divide-and-conquer, randomized, advanced data structure,* dan sebagainya. Contohnya, dengan strategi *brute-force,* kita dapat membuat algoritma *insertion sort, selection sort,* atau *bubble sort*. Sedangkan dengan strategi *divide-and-conquer*, kita dapat membuat algoritma *merge sort* dan *quick sort*, dan lain-lain. Strategi tersebut dapat digunakan untuk memilih cara sorting yang tepat. Dengan algoritma yang tepat, maka sorting dapat dilakukan dengan lebih efektif dan efisien.

Banyak orang yang sudah melakukan atau mencari algoritma sorting yang paling efektif dan efisien. Untuk mencarinya, tentu diperlukan parameter tertentu untuk membandingkan keefektivannya. Parameter yang pertama adalah *runtime*. Semakin banyak operasi yang dilakukan oleh algoritma tersebut, maka semakin efektif algoritma yang dibuat. Kedua, banyaknya perbandingan yang dilakukan dalam algoritma sorting tersebut. Kemudian, *data movements,* yaitu banyaknya perubahan urutan seperti penukaran tempat atau pergeseran dalam array. Berikutnya, kebutuhan memori yang digunakan untuk algoritma ini adalah tetap atau konstan. Dan terakhir, algoritma ini harus stabil.

II. LANDASAN TEORI

Algoritma *sorting* adalah algoritma yang digunakan untuk mengurutkan suatu data yang dimiliki dalam suatu keterurutan. Algoritma *sorting* biasanya digunakan untuk mengurutkan sebuah *array* yang ada dalam baris kode. Operasi yang digunakan dalam algoritma *sorting* adalah operasi pembanding (seperti > dan <), dan hasil yang didapatkan akan bergantung pada operator yang digunakan.

Banyak sekali algoritma *sorting* yang ada sekarang, seperti *selection sort, bubble sort, insertion sort, heap sort, quick sort,* dan masih banyak lagi. Kumpulan algoritma sorting ini juga datang dengan kompleksitas yang berbeda beda, mulai dari $O(\log n)$ hingga $O(n^2)$. Kompleksitas dari algoritma *sorting* ini menjadi salah satu hal yang kita pikirkan ketika akan memilih suatu algoritma *sorting* untuk program kita.

Mengapa kompleksitas algoritma ini penting untuk kita pikirkan? Jawabannya sederhana, yaitu karena semakin kompleks program yang kita miliki, semakin buruk performa dari program itu akan berjalan. Kompleksitas algoritma sendiri secara sederhana memiliki arti berapa lama waktu yang dibutuhkan sebuah algoritma untuk menyelesaikan tugasnya bila diberika sebuah input sebesar *n.* Jika suatu algoritma harus di skalakan, ia harus menghitung hasilnya dalam waktu yang terbatas dan praktis, bahkan untuk nilai *n* yang besar. Walaupun biasanya kompleksitas dihitung dalam satuan waktu, kadang kompleksitas juga di analisis dalam satuan ruang, yaitu seberapa besar memori yang digunakan oleh suatu algoritma.

Analisis dari kompleksitas sebuah algoritma sangat membantu ketika ingin membandingkan antar algoritma yang memiliki tujuan yang sama. Biasanya hasil dari analisis kompleksitas algoritma ini berupa sebuah notasi yang melambangkan seberapa kompleks algoritma tersebut. Ada 3 notasi yang digunakan untuk melambangkan kompleksitas algoritma. Pertama adalah notasi Big O. Notasi Big O ini digunakan untuk menggambarkan batas atas dari sebuah algoritma. Lalu yang kedua adalah notasi Big Omega. Sama halnya seperti notasi Big O, notasi ini juga menggambarkan kompleksitas algoritma, namun melihat batas bawahnya. Terakhir adalah notasi Big Theta. Saat menggunakan notasi ini, kita mengatakan bila kita memiliki batasan yang ketat secara

asimtotik terhadap *runtime* algoritmanya. Dikatakan secara asimtotik karena hanya akan berpengaruh ketika nilai n nya besar.

Algoritma DHS (*double sorting algorithm)* adalah salah satu dari banyak algoritma *sorting* yang biasa digunakan oleh para programmer. Algoritma ini pada dasarnya menggunakan 2 fungsi hash untuk membedakan input elemen pada data kedalam 2 buah kelompok. Kelompok yang pertama berisi semua elemen yang memiliki pengulangan (muncul lebih dari 1 kali), sedangkan kelompok kedua berisi semua elemen yang tidak memiliki pengulangan (hanya muncul 1 kali).

Fungsi hashing yang pertama digunakan untuk menghitung jumlah elemen di setiap blok dan untuk menentukan batasan dari setiap blok. Lalu fungsi hashsing yang kedua berguna untuk memberikan elemen elemen tadi sebuah index. Selanjutnya kelompok kedua tadi diurutkan menggunakan algoritma *quick sort.* Algoritma DHS ini terdiri dari 3 fase.

Fase 1 yaitu hashing pertama. Hashing adalah proses untuk mendapatkan indeks dengan cara memotong *key* dan mencampurkannya dalam berbagai cara yang akan terdistribusi secara merata pada *range* dari indeks. Dalam hash yang pertama, algoritma ini mengambil info distribusi dari elemen didalam array lalu membagi *array* tersebut kedalam block dan menghitung jumlah elemen yang ada didalam setiap block. Algoritma ini akan membuat sebuah variable bernama counterArray. Persamaan dibawah menunjukkan bagaimana array disusun kedalam 10 blok:

$$s_b = \lceil \frac{Max - Min + 1}{10} \rceil$$

Lalu algoritma ini akan melakukan perhitungan sebagai berikut pada setiap elemen didalam *array* untuk menghitung blok elemen dan *counter* nya.

$$Block_n = \left\lfloor \frac{elemen}{s_b} \right\rfloor;$$

$$counterArray[Block_n] + +;$$

Selanjutnya di fase kedua yaitu akan dilakukan hashing kedua. Hashing kedua akan bekerja sesuai data yang sudah dikumpulkan selama hashing pertama tadi. Algoritma ini akan mengambil setiap elemen blok dan menghitung nilai maksimum dan minimumnya sebagai berikut:

$$Max = (specifiedBlock + 1) * S_b$$

$$Min = specifiedBlock * S_b$$

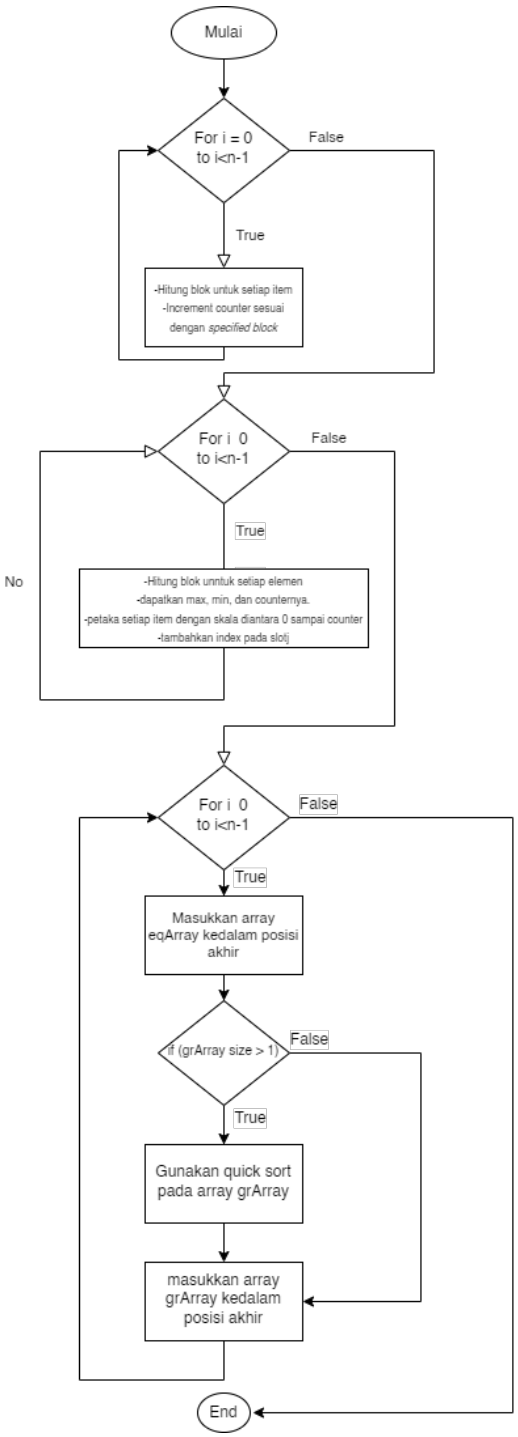
Lalu algoritma ini akan memetakan elemen ke indeks virtual didalam *specified block* tadi sebagai berikut:

$$Float\ index = \frac{el - Min}{Max - Min}$$

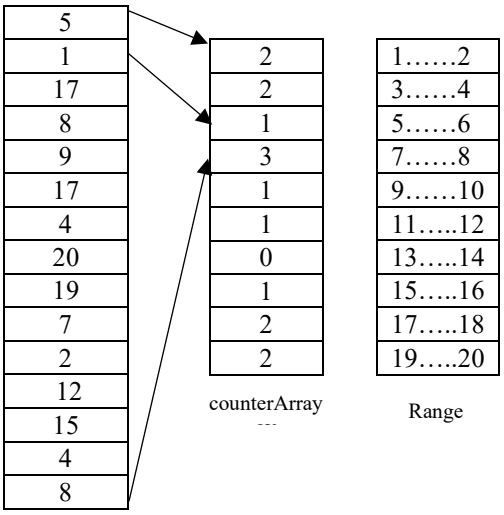
$$Index = index * (counterArray[specifiedBlock])$$

Secara simpel, yang dilakukan algoritma diatas adalah sebagai berikut. Misalkan elemen adalah e, *specified block* adalah b, dan *counterArray[b]* = 5, maka algoritma ini akan memetakan e pada indeks virtualnya sebesar lima elemen. Akhir dari fase kedua algoritma ini akan menghasilkan sebuah array of indeks. Indeks yang dihasilkan bisa berupa bilangan bulat (integer), ataupun pecahan (float).

Terakhir, yaitu fase 3. Pada fase 3 ini, algoritma DHS berisi sekumpulan indeks virtual disetiap blok dan partisi indeks ke slot, dengan masing-masing slot berisi satu atau lebih indeks yang menunjuk ke bagian integer. Lalu setiap slop dibagi menjadi dua array yaitu eqArray yang berisi indeks integer dan grArray yang berisi indeks float. Selanjutnya, array grArray tersebut di sort menggunakan algoritma quick sort. Dibawah ini adalah flowchart dari algoritma DHS.



Berikut adalah contoh visualisasi dari algoritma DHS, dengan array A memiliki 15 elemen. Tabel dibawah



1	1	
2		2
3		
4	4.4	5
5		
6		
7		7
8		
9	8.8	
10		
11		9
12		12
13		
14		15
15		
16		
17		
18	17.17	
19		19
20	20	

Slot eqArray grArray

III. ANALISIS ALGORITMA DHS

Dalam bab ini akan dibahas mengenai analisis kompleksitas dari algoritma *double hashing sort*. Algoritma DHS ini didasarkan pada pembagian elemen *array* kedalam ruang ruang atau slot, yang mana setiap slotnya mengandung rentang yang spesifik. Ada 3 kasus mengenai analisis algoritma DHS ini. Tiga kasus ini didasarkan pada relasi antara ukuran array n, dan domain elemen didalam array (m). Ketiga kasus tersebut yaitu:

Kasus 1: O(m) < O(n). Dalam kasus ini, rentang nilai elemen yang terdapat didalam array lebih kecil dibanding jumlah elemen maksimalnya. Kasus ini bisa dibentuk sebagai A = (a_1, a_2, ..., a_n), yang dalam hal ini $a_i < m$ dan $m<n$. Notasi Big O digunakan untuk mengilustrasikan bahwa perbedaan antara n dan m cukup signifikan.

Kasus 2: O(m) = O(n). Dalam kasus ini, rentang nilai elemen dalam array n besarnya sama seperti jumlah elemen array. Kasus ini bisa dibentuk sebagai A = (a_1, a_2, ..., a_n), yang dalam hal ini $a_i \leq m$, $n \approx m$, dan $m = \alpha\, n \pm \beta$, dimana α dan β adalah konstan.

Kasus3:O(n) < O(m). Untuk kasus yang terakhir ini, rentang nilai elemen dalam array n lebih besar dari jumlah banyaknya elemen. Seperti kasus sebelumnya, kasus ini juga bisa dibentuk sebagai A = (a_1, a_2, ..., a_n), yang dalam hal ini $a_i < m$ dan $m > n$.

Mari kita analisis lebih jauh lagi mengenai ketiga kasus diatas. Pada kasus 1 dimana O(m) < O(n), nilai dari elemen (m) lebih kecil bila dibandingkan dengan jumlah elemennya (n). Artinya, array ini akan mengantung banyak elemen yang berulang. Karena jumlah maksimum slot yang akan didapat juga sebesar m, maka kita tidak perlu memetakan elemen dalam array pada slot. Solusi dari kasus ini lebih effisien bila diselesaikan dengan algoritma *counting sort*. Sedikit penjelasan mengenai *counting sort,* algoritma ini menghitung banyaknya elemen yang kurang dari integer $i \in [1,m]$. Lalu menggunakan nilai yang tadi dihitung, algoritma CS (*counting sort)* untuk mengalokasikan elemen a_j kedalam posisi yang tepat di array A. Algoritma CS memiliki kompleksitas $O(n + m) = O(n)$, karena $O(m) < O(n)$. Berikut adalah contoh penggunaan algoritma CS pada algoritma DHS.

Dalam kasus ini, tidak diperlukan untuk membagi array menjadi 2 array seperti yang dibahas pada bab 2. Misalnya $m = 5$, $n = m^2 = 25$, dan array A berisi seperti berikut:

2	3	3	3	5	1	1	1	1	5	2	3	5	2	2	5	1	2	3	2	5	2	3	2	1

Langkah pertama yang dilakukan adalah membuat array counter C, yang dalam hal ini C[i] merepresentasikan jumlah pengulangan dari integer $i \in [1,m]$ (index i merepresentasikan nilai elemen di array A). Langkah kedua yaitu menghitung prefix-sum dari array prefix-sum adalah mengiterasi sebuah array dan menambahkan setiap elemennya dengan nilai sebelumnya. Misalnya kita memiliki array X sebagai berikut:

10	20	10	5	15

Bila dilakukan prefix-sum terhadap array X tersebut, maka yang terjadi adalah seperti ini:

prefixArr[0] = 10,
prefixArr[1] = prefixArr[0] + arrayX[1] = 30,
prefixArr[2] = prefixArr[1] + arrayX[2] = 40,
....
prefixArr[n] = prefixArr[n-1] + arrayX[n] = ...

Sehingga, bila prefix-sum dilakukan terhadap array C maka akan didapatkan hasil sebagai berikut:

6	8	6	0	5

Array C

6	14	20	20	25

Prefix-sum Array C

Lalu setelah prefix-sum dilakukan, artinya integer "1" berada di posisi 1 sampai 6, integer "2" pada posisi 7-14 dan seterusnya. Bila array ini dicetak, maka akan menghasilkan array seperti berikut:

1	1	1	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3	3	5	5	5	5	5

Selanjutnya untuk kasus 2, yaitu O(m) = O(n), nilai dari m sama dengan atau mendekati nilai dari n. Bila elemen didalam array tersebar dengan merata, maka jumlah pengulangan elemen di dalam array yang terjadi juga akan menjadi konstan. Dalam kasus ini, kita memiliki 2 hal mengenai algoritma DHS. Pertama adalah tidak diperlukannya untuk membuat 2 array yang berbeda, yaitu grArray dan eqArray. Lalu yang kedua yaitu kita tidak memerlukan algoritma *quick sort* karena kita bisa mengurutkannya menggunakan algoritma *counting sort*.

Namun, bila pesebaran elemen dalam array tidak merata, maka jumlah pengulangannya pun akan bermacam-macam. Misalkan jumlah pengulangan dari semua elemen didalam array adalah $\Phi(n)$. Array eqArray mengandung $\Phi(n)$ elemen, dan array grArray mengandung $n - \Phi(n)$. Maka *running time* dari algoritma DHS menjadi $O(n + (n - \Phi(n)) \log (n - \Phi(n)))$, yang dalam hal ini. Dalam *average case,* algoritma DHS ini memiliki pengulangan elemen sebanyak *n/2*, sehingga *running time* nya menjadi $O((n/2) \log (n/2)) = O(n \log n)$. Oleh karena itu algoritma *counting sort* lebih baik dibandingkan dengan DHS pada kasus ini.

Terakhir yaitu kasus ketiga, dimana O(n) < O(m). Pada kasus ini, yang terjadi adalah nilai dari m lebih besar dibanding dari n, sehingga setiap elemen dalam array memiliki kemungkinan untuk berjauhan atau repetisi dari array n akan konstan. Untuk kasus seperti ini, algoritma DHS dan CS tidak cocok digunakan. Ada beberapa alasan mengapa algoritma DHS dan CS tidak cocok digunakan pada kasus seperti ini.

Pertama, algoritma semacam DHS dan CS memerlukan *storage* yang cukup besar untuk memetakan setiap elemennya. Misalnya, jika $m = n^2$ dan $n = 10^6$, maka m akan menjadi 10^{12}, dan bisa dibilang angka tersebut sangat besar.

Kedua, sendainya mesin yang digunakan memiliki memori yang cukup besar, maka *running time* dari algoritma DHS adalah O($n \log n$). Masalahnya, kelemahan utama dari algoritma DHS ini adalah keluaran dari fungsi hashing kedua yang tidak unik, dan persamaan yang digunakan untuk membedakan antara elemen yang berulang dan tidak berulang kurang akurat. Artinya algoritma ini memiliki kemungkinan untuk sebuah elemen dalam arraynya yang tidak memiliki pengulangan dan elemen yang memiliki pengulangan, mereka berdua memiliki indeks visual yang sama yang dibentuk dari persamaan tadi. Oleh karena itu, algoritma *merge sort* atau *quick sort* lebih baik dari DHS dalam kasus ini.

Ketiga, dalam kasus algoritma *counting sort*, algoritmanya akan membaca array berukuran m untuk mengalokasikan elemen pada posisi yang tepat. Karenanya, *running time* dari algoritma ini adalah O(m), yang dalam hal ini O(m) > O(n). Sebagai contohnya, jika $m = n^2$, maka *running time* nya menjadi O(n^2), yang mana lebih besar dari *running time* algoritma *merge sorgi* yaitu O($n \log n$).

IV. UJI PERFORMA

Pada bab ini, akan dilakukan uji performa algoritma DHS dan CS menggunakan implementasi code berbahasa C (karena cenderung lebih cepat dari bahasa lain). Saya mengambil beberapa uji data dari internet. Spesifikasi mesin yang digunakan untuk uji performa algoritma ini memiliki *processor* dengan kecepatan 2.4 GHz dan memori sebesar 16GB. Mesin yang digunakan juga berjalan pada sistem operasi Windows.

Uji performa ini akan dilakukan menggunakan beberapa *benchmark* sebagai berikut:

1. Metode pertama menggunakan array yang berisi elemen elemen yang terdistribusi secara merata. Elemen ini di *generate* menggunakan fungsi random() pada library C.
2. Metode kedua menggunakan array yang berisi elemen elemen yang terdistribusi secara acak.
3. Metode yang ketiga cukup mirip dengan metode kedua, namun arraynya sudah terurut membesar.
4. Metode yang keempat serupa dengan metode ketiga, namun terurut mengecil.
5. Metode kelima mirip seperti metode tiga dan empat, namun array tidak terurut sepenuhnya (hampir terurut)
6. Metode yang terakhir menggunakan array dengan elemen yang di *generate* dengan cara mengambil rata rata dari empat kali (angka dibulatkan agar integer) pemanggilan fungsi random().

Ada tiga parameter yang akan digunakan dalam pengujian performa. Pertama adalah ukuran dari array n dan domain m. Lalu parameter yang kedua adalah distrbusi data, yaitu 6 jenis distribusi data yang sudah disebutkan diatas. Berdasarkan relasi antara n dan m , misal O(m) < O(n), ukuran array n akan disesuaikan dan akan mengambil nilai m,m_i yang berbeda seperti $m_1 = 10^6$, $m_2 = 10^5$, $m_3 = 10^4$, $m_4 = 10^4$, $m_5 = 10^2$. Untuk setiap nilai n dan m_i yang telah dipilih, program akan mengenerate enam array input berbeda berdasarkan enam *benchmark* diatas yang sudah disebutkan. Untuk setiap benchmark, *runnting time* akan dihitung dengan menjalankan program sebanyak 50 kali, dan waktu yang diambil adalah waktu rata rata dari 50 kali program dijalankan. Waktu akan diukur dalam satuan milisekon. Persamaan yang digunakan untuk menghitung *running time* dari sebuah algoritma *Alg* adalah sebagai berikut:

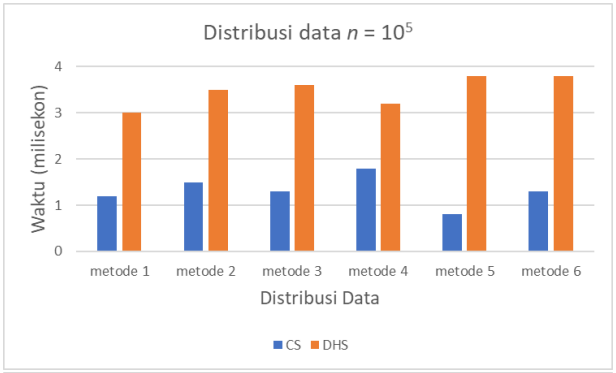
$$\frac{1}{n_m}\sum_{i=1}^{n_m}\left(\frac{1}{50}\sum_{j=1}^{50} t_i(n, m_i, dd, Alg)\right)$$

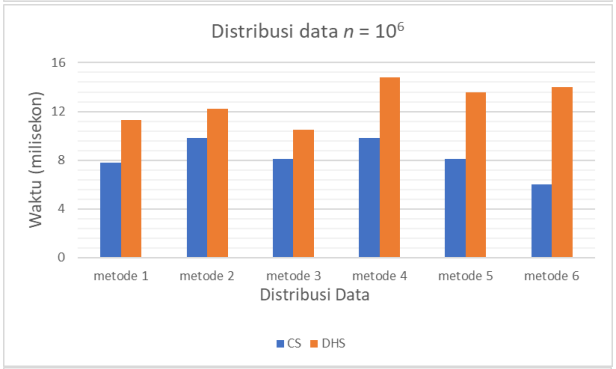
Dalam persamaan diatas, m_i adalah salah satu elemen dalam array, dalam rentang m, dan memenuhi O(m) < O(n) atau O(m) = O(n). Misalnya $n = 10^x$, maka $10^2 \leq m_i \leq 10^{x-2}$. Pada persamaan diatas juga, n_m merupakan jumlah banyakna angka yang berbeda dari m_i, sebagai contoh misal $n = 10^x$, maka $n_m = x\text{-}3$ karena $10^2 \leq m_i \leq 10^{x-2}$. Lalu *dd* merepresentasikan tipe distribusi data, *dd* merupakah salahsatu dari enam *benchmark* yang sudah disebutkan diatas. *Alg* adalah algoritmanya, yaitu algoritma CS atau algoritma DHS yang akan di hitung nanti. Terakhir, t_i adalah *running time* untuk algoritma *Alg* menggunakan parameter yang sudah dijelaskan diatas.

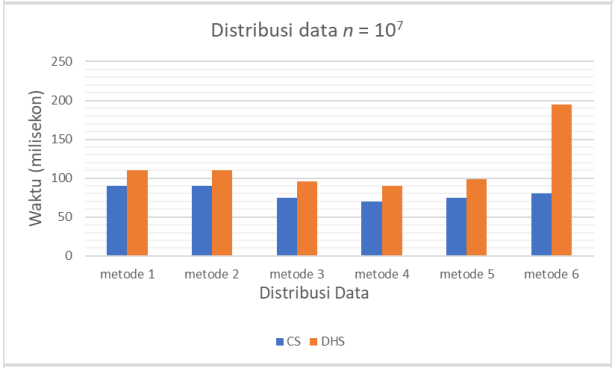
Pengujian akan dilakukan dengan memilih n sebesar 10^8, 10^7, 10^6, dan 10^5, karena *running time* dari algoritma CS dan DHS sangatlah kecil bila n yang diambil dibawah 10^5.

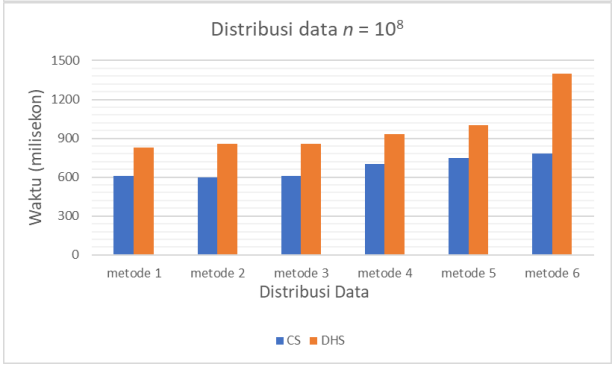
Hasil dari pengujian *running time* algoritma DHS dan CS dengan semua benchmark dan parameter yang mempengaruhi

algoritma tersebut bisa dilihat pada grafik dibawah.

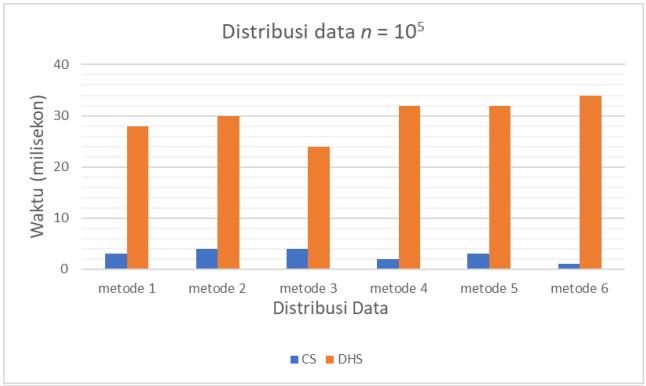


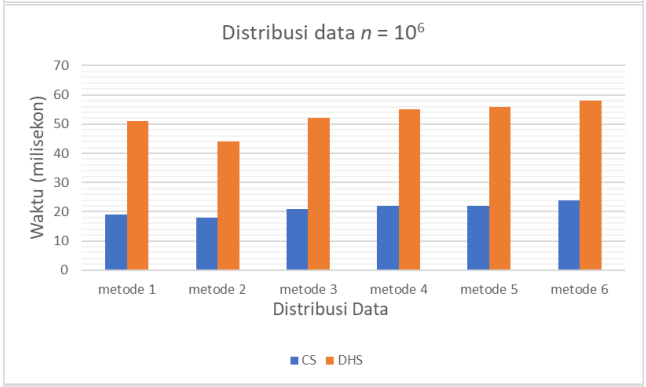


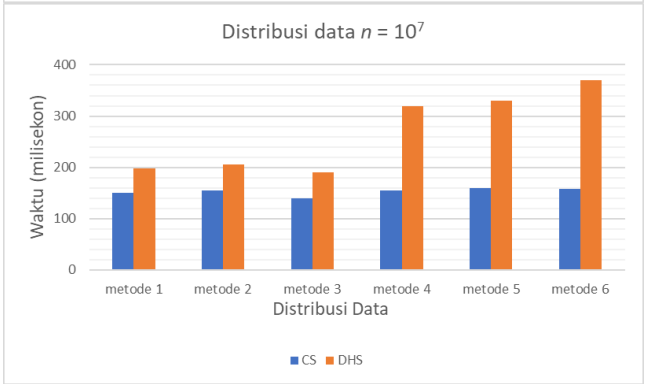


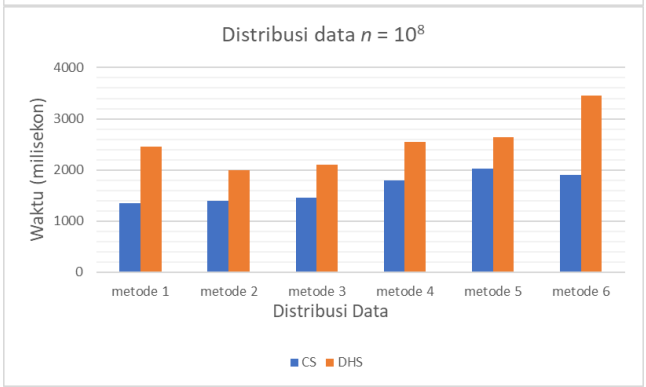


Grafik diatas menunjukkan hasil rata rata dari 50 kali repetisi algoritma CS dan DHS. Grafik diatas menunjukkan data ketika kasus O(m) < O(n) terjadi.









Grafik diatas menunjukkan rata rata dari 50 kali repetisi algoritma CS dan DHS juga. Namun, kali ini dengan kasus O(m) = O(n). Dari hasil pengujian diatas, kita dapat melihat bahwa *running time* algoritma CS bekerja lebih baik daripada algoritma DHS untuk semua tipe distribusi data dan variasi n. Seacara keseluruhan, perbedaan terjauh terletak ketika algoritma menjalankan metode ke 6.

V. KESIMPULAN

Sorting adalah suatu cara untuk menyusun ulang sebuah array dengan ketentuan tertentu. Sorting sangat penting dalam berbagai aplikasi ilmu komputer. Dalam makalah ini, dibahas mengenai algoritma sorting *double hash sort* dan melakukan pengujian performa yang dibandingkan dengan algoritma *counting sort*. Hasilnya ternyata algoritma DHS tidak lebih baik daripada algoritma CS dalam hal *running time*. Hasil ini sekaligus memberikan bukti bahwa algoritma *counting sort* bisa dibilang jauh lebih cepat (dibeberpa kasus) ketimbang algoritma DHS.

VI. UCAPAN TERIMAKASIH

Puji syukur penulis ucapkan kepada Tuhan Yang Maha Esa, karena atas rahmat-Nya penulis dapat menyelesaikan penulisan makalah ini. Penulis juga berterima kasih kepada tim dosen dan asisten yang telah dengan sabar membimbing penulis selama masa perkuliahan IF2120 semester 1 2021/2022. Penulis juga mengucapkan terima kasih kepada segenap teman-teman IF 2020 yang turut membantu penulis dalam perkuliahan dan dalam penyelesaian makalah ini

REFERENSI

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian1.pdf
[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian2.pdf
[3] Y. M. K. Omar, H. Osama and A. Badr, "Double Hashing Sort Algorithm," in Computing in Science & Engineering, vol. 19, no. 2, pp. 63-69, Mar.-Apr. 2017, doi: 10.1109/MCSE.2017.26.
[4] Osama, H., Omar, Y., Badr, A.: Mapping Sorting Algorithm, pp. 48–491. SAI Computing Conference 2016, London (2016)
[5] Hazem M.Bahig, "Complexity analysis and performance of double hashing sort algorithm", in Journal of the Egyptian Mathematical Society, 27 Mar, 2019
[6] https://www.geeksforgeeks.org/prefix-sum-array-implementation-applications-competitive-programming/
[7] https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/
[8] https://www.geeksforgeeks.org/double-hashing/

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, 13 Desember 2020

Ilham Bintang Nurmansyah dan 13520102