Implementasi Algoritma Pengurutan *General Purpose* dan Berbasis Komparasi untuk Data Berkategori dalam Waktu Linier Tanpa Paralelisasi

Malik Akbar Hashemi Rafsanjani - 13520105
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13520105@std.stei.itb.ac.id

***Abstract*—Data berkategori merupakan jenis data yang banyak sekali digunakan, mulai dari kehidupan sehari-hari, analisis data, dan juga data kecerdasan buatan. Namun, seringkali jumlah elemen data sangat banyak sehingga diperlukan algoritma pengurutan yang efektif dan efisien. Pada makalah ini, penulis berusaha mendesain algoritma pengurutan yang bersifat *general purpose* dan berbasis komparasi yang memiliki kompleksitas waktu linier tanpa perlu paralelisasi. Berdasarkan data hasil pengujian, algoritma ini dapat dibilang memiliki performa setara dengan algoritma pengurutan efektif pada umumnya untuk data dengan jumlah kategori menyamai jumlah elemen. Namun, performa dari algoritma ini relatif lebih baik dibanding algoritma lain untuk jumlah kategori lebih kecil daripada jumlah elemen dan jauh lebih baik jika kunci komparasi merupakan tipe data kompleks.**

Keywords—algoritma pengurutan, data berkategori, pengurutan tanpa paralelisasi

I. PENDAHULUAN

Data berkategori merupakan koleksi data yang dibagi menjadi beberapa grup berdasarkan kategori masing-masing. Jenis data ini sangat banyak digunakan pada kehidupan sehari-hari, mulai dari pelabelan data berdasarkan kategori tertentu, analisis data yang dapat dikelompokkan ke dalam beberapa grup sampai dengan data pelatihan dan pengetesan pada model kecerdasan buatan. Data jenis ini biasanya dikumpulkan dengan jumlah yang sangat besar.

Seringkali pada awalnya, data tercecer dan tidak terurut berdasarkan kategorinya, atau data perlu diganti pengurutan kategorinya. Hal tersebut memerlukan waktu yang cukup lama untuk melakukan pengurutan jika jumlah datanya sangat banyak. Padahal, keperluan pengurutan merupakan hal yang sangat penting, baik untuk pembacaan yang *user-friendly*, analisis dan pelatihan model kecerdasan buatan yang lebih tepat.

Oleh karena itu, diperlukan algoritma pengurutan yang efisien dan efektif. Pada umumnya, algoritmanya pengurutan memiliki kompleksitas O(n logn). Kompleksitas tersebut cukup cepat, tetapi jika data cukup besar, dalam order jutaan, maka kecepatan pengurutan akan tetaplah lama. Maka dari itu, pada makalah ini, penulis ingin mendesain suatu algoritma pengurutan baru, yang dapat mengurutkan data berkategori dengan lebih cepat. Namun, algoritma ini tetaplah perlu bersifat *general purpose* dan berbasis komparasi. Hal ini dikarenakan data berkategori datang dengan bentuk yang sangat bervariasi sehingga algoritma ini perlu dapat menyelesaikan seluruh bentuk-bentuk data yang bervariatif tersebut.

Algoritma ini juga perlu didesain agar dapat menyelesaikan permasalahan pengurutan untuk data berkategori, tetapi semua atau hampir semua kategori bersifat unik. Data jenis tersebut dapat dibilang sama seperti data pada umumnya. Algoritma ini diharapkan memiliki kompleksitas komputasi yang linier jika jumlah kategori yang jauh lebih kecil daripada jumlah elemen dan memiliki kompleksitas komputasi yang sama seperti algoritma pengurutan efektif lainnya (O(n logn)) untuk jumlah kategori menyamai jumlah elemen. Semua ini dicapai tanpa perlu melakukan paralelisasi terhadap keberjalanan algoritma.

II. TEORI DASAR

A. Algoritma Pengurutan

Algoritma pengurutan merupakan algoritma yang digunakan untuk menempatkan koleksi data dalam urutan. Koleksi data ini dapat berupa banyak bentuk, seperti array ataupun list. Kebanyakan urutan yang dipakai adalah urutan numerik dan urutan leksikografis, baik terurut menaik ataupun teurut menurun. Pengurutan sering berfungsi untuk menghasilkan keluaran yang mudah dibaca manusia. Selain itu, pengurutan juga banyak dipakai untuk mengubah representasi data menjadi representasi standard ataupun representasi yang lebih mudah diolah. Hal ini dikarenakan cukup banyak algoritma lain yang mengharuskan data terurut terlebih dahulu sebelum algoritma tersebut dapat digunakan.

Algoritma pengurutan dapat diklasifikasikan berdasarkan kompleksitas komputasinya. Pada umumnya, algoritma pengurutan tanpa paralelisasi disebut baik jika kompleksitas komputasinya O(n logn) dan kurang baik jika kompleksitasnya $O(n^2)$. Namun, dengan paralesisasi, kompleksitas pengurutan dapat diturunkan menjadi $O(\log^2 n)$. Namun, paralelisasi memerlukan sumber daya yang lebih dibanding tanpa paralelisasi.

Salah satu algoritma pengurutan yang baik dan banyak dipakai ialah algoritma *merge sort*. Algoritma ini merupakan algoritma pengurutan yang efisien, *general purpose*, dan berbasis komparasi. Algoritma ini merupakan algoritma *divide and conquer* yang membagi input menjadi beberapa bagian sampai titik tertentu, lalu mengurutkan bagian-bagian tersebut, serta menggabungkan bagian-bagian yang sudah terurut kembali dengan menjaga keterurutan

B. Data Berkategori

Data berkategori merupakan data yang memiliki label kategori pada setiap instantiasi data. Data berkategori merupakan jenis bentuk data yang banyak sekali dipakai dalam *data science*. Data ini biasanya disajikan dalam jumlah yang sangat banyak

C. Algoritma Divide and Conquer

Algoritma *divide and conquer* adalah paradigma desain algoritma. Sebuah algoritma *divide and conquer* secara rekursif memecah masalah menjadi dua atau lebih sub-masalah dari jenis yang sama atau terkait, sampai bagian tersebut menjadi cukup sederhana untuk diselesaikan secara langsung. Solusi untuk sub-masalah kemudian digabungkan untuk memberikan solusi untuk masalah asli.

Teknik *divide and conquer* adalah dasar dari algoritma yang efisien untuk banyak masalah, seperti pengurutan (misalnya, *quicksort*, *merge sort*), mengalikan bilangan besar (misalnya, algoritma Karatsuba), menemukan pasangan titik terdekat, analisis sintaksis (misalnya, parser *top-down*), dan menghitung transformasi Fourier diskrit (FFT).

Merancang algoritma *divide and conquer* yang efisien bisa jadi sulit. Seperti dalam induksi matematika, seringkali perlu untuk menggeneralisasi masalah agar dapat diterima untuk solusi rekursif. Kebenaran dari algoritma *divide and conquer* biasanya dibuktikan dengan induksi matematika, dan biaya komputasinya sering ditentukan dengan menyelesaikan hubungan perulangan.

D. Binary Search Tree

Binary Search Tree (BST) adalah struktur data pohon biner yang node internalnya masing-masing menyimpan kunci lebih besar dari semua kunci di subpohon kiri node dan lebih kecil dari yang ada di subpohon kanannya. Kompleksitas waktu operasi pada pohon pencarian biner berbanding lurus dengan tinggi pohon.

BST memungkinkan pencarian biner untuk pencarian cepat, penambahan, dan penghapusan item data. Karena node dalam BST ditata sedemikian rupa sehingga setiap perbandingan melompati sekitar setengah dari pohon yang tersisa, kinerja pencarian sebanding dengan logaritma biner.

Kinerja BST tergantung pada urutan penyisipan simpul ke dalam pohon karena penyisipan sewenang-wenang dapat menyebabkan degenerasi. Operasi dasar meliputi: pencarian, traversal, *insert* dan *delete*. BST dengan kompleksitas kasus terburuk yang dijamin berkinerja lebih baik daripada array yang tidak disortir, yang akan membutuhkan waktu pencarian linier. Analisis kompleksitas BST menunjukkan bahwa, rata-rata, penyisipan, penghapusan, dan pencarian membutuhkan O(log n) untuk n simpul. Dalam kasus terburuk, performa BST menjadi O(n).

E. Linked List

Linked list adalah kumpulan linier elemen data yang urutannya tidak ditentukan oleh penempatan fisiknya dalam memori. Sebaliknya, setiap elemen menunjuk ke yang berikutnya. Ini adalah struktur data yang terdiri dari kumpulan simpul yang bersama-sama mewakili urutan. *Linked List* adalah salah satu struktur data paling sederhana dan paling umum. Struktur data ini dapat digunakan untuk mengimplementasikan beberapa tipe data abstrak umum lainnya. Minimalnya, setiap simpul berisi data dan referensi ke simpul berikutnya dalam urutan.

Struktur ini memungkinkan penyisipan atau penghapusan elemen secara efisien dari posisi manapun. Manfaat utama dari *linked list* dibandingkan array biasa adalah bahwa elemen *linked list* dapat dengan mudah dimasukkan atau dihapus tanpa realokasi atau reorganisasi seluruh struktur karena item data tidak perlu disimpan secara berurutan di memori atau di disk. *Linked list* memungkinkan penyisipan dan penghapusan simpul pada titik mana pun, dan memungkinkan melakukannya dengan jumlah operasi yang konstan dengan menjaga referensi simpul sebelum referensi ditambahkan atau dihapus dalam memori selama penelusuran *list*. Kelemahan dari linked list adalah waktu aksesnya linier. Akses yang lebih cepat, seperti akses acak, tidak mungkin dilakukan. Selain itu, array memiliki lokalitas cache yang lebih baik dibandingkan dengan *linked list*.

F. Hash Table

Hash table merupakan struktur data yang dapat memetakan kunci (*key*) ke nilai (*value*). *Hash table* menggunakan fungsi *hash* untuk mengkomputasikan indeks (*hash code*) dari suatu elemen dalam pencarian, penambahan, dan penghapusan elemen. *Hash table* menyimpan larik dari slot di mana nilai atau elemen dapat ditemukan. Selama pencarian, kunci akan di-*hash* dan indeks hasil *hashing* menunjukkan dimana elemen disimpan. Oleh karena itu, struktur data ini memiliki kompleksitas komputasi konstan untuk pencarian dan penambahan elemen karena hanya bergantung pada fungsi *hash* untuk mencari tahu dimana elemen disimpan.

Idealnya, fungsi *hash* hanya akan menetapkan setiap kunci ke slot yang unik, tetapi kebanyakan *hash table* diimplementasikan menggunakan fungsi *hash* yang tidak sempurna. Hal ini dapat menyebabkan *collision* di mana fungsi *hash* menghasilkan indeks yang sama untuk kunci yang berbeda. Namun, *collision* ini memiliki banyak metode penanganan.

Penanganan *collision* yang banyak dipakai ialah *chaining* dan *probing*. Pada metode *chaining*, nilai yang disimpan pada *hash table* bukanlah nilai elemen, melainkan *linked list*. Hal ini bertujuan, jika terdapat *collision*, maka elemen-elemen yang terjadi *collision* ditempatkan pada *linked list* tersebut. Namun, ketika terjadi banyak *collision*, maka kita perlu

mengiterasi *linked list* yang didapat untuk menemukan elemen yang kita cari.

Metode penanganan *collision* selain *chaining* ialah *probing*. Ketika terjadi *collision*, maka indeks akan dikalkulasikan kembali (*probe*) menurut aturan tertentu, sampai ditemukan slot yang masih kosong. Metode *probing* juga dibagi menjadi beberapa jenis, dengan jenis-jenis yang terkenal ialah *linear probing*, *quadratic probing*, dan *double hashing*. *Linear probing* memiliki interval antar *probe* tetap (biasanya satu) dan *quadratic probing* memiliki interval antar *probe* ditingkatkan setiap iterasi dengan fungsi polynomial kuadrat. Sedangkan *double hashing* menghitung indeks ulang menggunakan fungsi *hash* sekunder.

III. Implementasi dan Analisis Kompleksitas

A. Implementasi

Pada makalah ini, penulis membuat 2 alternatif algoritma pengurutan. Pada algoritma ini, digunakan beberapa struktur data, yaitu *pair*, *linked list*, *binary search tree*, dan juga *hash table.* Namun, *hash table* hanya akan dipakai untuk alternatif kedua. Struktur data pair digunakan untuk menyimpan data yang berpasangan. Struktur data *linked list* dipilih penulis karena struktur data ini merupakan struktur data yang dapat menyimpan beberapa data secara linier dan memiliki kompleksitas komputasi penambahan elemen yang konstan. Selain itu, struktur data *linked list* tidak perlu mengalokasikan ukuran di muka, sesuai dengan permasalahan ini di mana penulis menyimpan data masing-masing kategori pada suatu linked list dan kita tidak tahu berapa banyak jumlah data setiap kategori. Struktur data *binary search tree* dipilih karena struktur data ini memiliki kompleksitas komputasi penambahan elemen yang logaritmik dengan menjaga keterurutan serta dapat diiterasi secara terurut. Terakhir, struktur data *Hash Table* dipilih karena struktur data ini memiliki kompleksitas komputasi penambahan elemen dan pencarian elemen yang konstan. Namun, struktur data ini memiliki *overhead* pada proses *hashing* dan juga adanya potensi *collision* sehingga penulis memberikan 2 alternatif dengan dan tanpa memakai *hash table*.

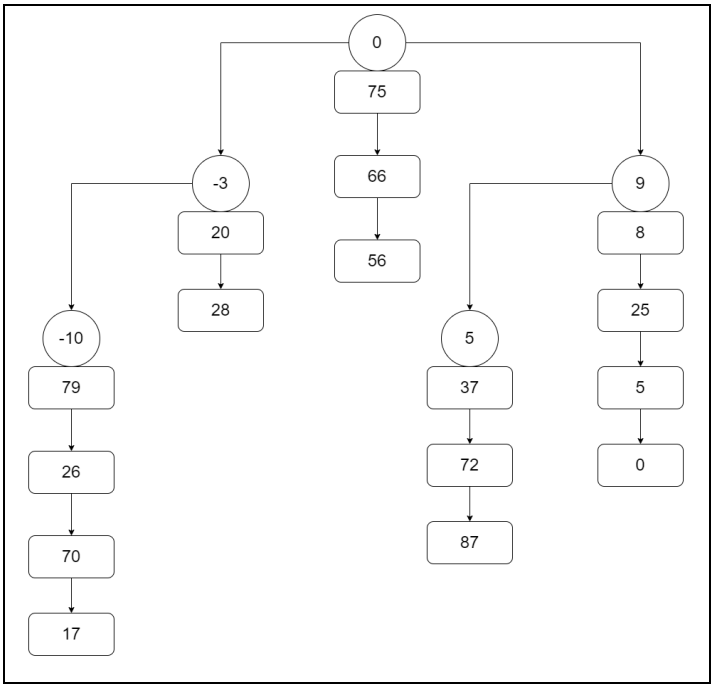
Pada alternatif pertama, program akan menginisialisasi sturktur data yang diperlukan serta meminta masukan larik bertipe elemen apapun, banyak elemen, serta fungsi untuk mendapatkan kunci komparasi dari tipe elemen tersebut. Setiap elemen pada larik dimasukkan ke dalam struktur data *binary search tree* yang simpulnya berisi nilai kunci komparasi / kategori serta *linked list* berisi elemen-elemen yang berkategori sesuai kunci. Elemen dimasukkan ke dalam *linked list* dengan dimasukkan di awal dengan kompleksitas komputasi konstan. Setelah semua elemen larik disimpan di *linked list* sesuai dengan kategori masing-masing, *binary search tree* diiterasi secara terurut masing-masing simpulnya, dan setiap elemen pada *linked list* di setiap simpul dimasukkan kembali ke dalam larik dengan menjaga keterurutannya.

Pada alternatif kedua, terdapat sedikit perbedaan pada penambahan elemen pada *binary search tree*. Pada alternatif ini, *hash table* berisi pasangan kunci dari kategori dan nilai dari referensi simpul *binary search tree*. Ketika penambahan elemen, maka akan dicek terlebih dahulu apakah di *hash table*, terdapat kategori yang bersesuaian dengan elemen yang akan dimasukkan. Jika tidak ada, maka lakukan penambahan elemen sesuai alternatif satu dan simpan kategori dari data tersebut dan referensi simpul *binary search tree*. Jika sudah ada pada *hash table*, maka elemen larik akan secara langsung dimasukkan ke dalam *linked list* pada simpul kategori dari data yang bersesuaian.

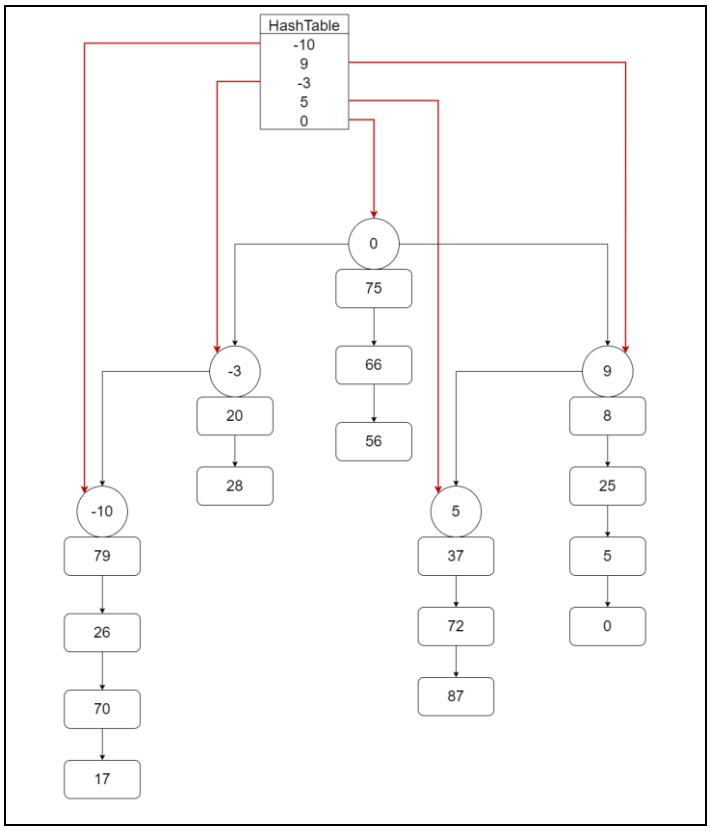
Misalkan terdapat data sebagai berikut:

[<-10,17>, <-10,70>, <9,0>, <-3,28>, <9,5>, <-10,26>, <5,87>, <9,25>, <0,56>, <5,72>, <-3,20>, <0,66>, <-10,79>, <0,75>, <9,8>, <5,37>].

Ilustrasi dari hasil pengisian pada struktur data *binary search tree* menggunakan kedua alternatif dari algoritma yang didesain dapat dilihat sebagai berikut.



Gambar 1. Ilustrasi pengisian binary search tree untuk alternatif 1



Gambar 2. Ilustrasi pengisian binary search tree untuk alternatif 2

Berikut merupakan pseudocode dari implementasi program algoritma yang didesain.

Algorithm 1 Categorical Data Sort

```
template <class T, class U >
procedure SORT(T *arr, int N, U (*getKey)(const T))
    BinarySearchTree <T, U>bst(getKey)
    for i traverse 0..N-1 do
        bst.insert(arr[i])
    end for
    bts.fill(arr)
end procedure

template <class T, class U >
procedure OPTIMALIZEDSORT(T *arr, int N, U (*getKey)(const T))
    BinarySearchTree <T, U>bst(getKey)
    HashTable <U, BSTNode*>hashTable(N)
    for i traverse 0..N-1 do
        U key ← getKey(arr[i])
        BSTNode *node ← hashtable.get(key)

        if node is NULL then
            bst.insert(arr[i])
            hashtable.insert(key, bst.getAdded())
        else
            bst.directInsert(arr[i], node)
        end if

    end for
end procedure
```

Gambar 3. Pseudocode algoritma pengurutan

B. Analisis Kompleksitas

Pada alternatif pertama, kita dapat meninjaunya ke dalam 2 bagian yaitu pada pengisian ke *binary search tree* dan pengisian kembali ke larik. Misalkan jumlah elemen larik n dan jumlah kategori k. Pada pengisian ke *binary search tree*, dilakukan iterasi setiap elemen (O(n)), dan setiap elemen dilakukan pencarian simpul yang bersesuaian dengan kategorinya (O(logk)) dan setelah ditemukan, maka elemen dimasukkan ke *linked list* (O(1)). Maka total kompleksitas komputasi pengisian ke *binary search tree* ialah O(n) * (O(logk) + O(1)) = O(n logk). Pada pengisian kembali ke larik, dilakukan traversal setiap simpul pada *binary search tree* dan linked list pada setiap simpul semua elemen ke dalam larik. Misalkan jumlah elemen pada *linked list* di simpul ke-i adalah e_i, maka kompleksitasnya dapat dinyatakan sebagai berikut

$$\sum_{i=1}^{k} e_i = n$$

Karena kita melakukan traversal dari 1 sampai k dengan setiap iterasi, kompleksitasnya ialah ($O(e_i)$), maka total kompleksitasnya adalah O(n). Oleh karena itu total kompleksitas dari alternatif pertama ialah O(n logk) + O(n) = O(n logk).

Pada alternatif kedua, perbedaan hanya terdapat pada pengisian *binary search tree* sehingga kompleksitas pengisian kembali ke larik keduanya sama, yaitu O(n). Pengisian *binary search tree* dapat dibagi menjadi 2 kondisi, yaitu ketika kategori elemen yang akan dimasukkan tidak ada di *hash table* dan ada di *hash table*. Kondisi pertama terjadi k kali, sedangkan kondisi kedua terjadi n-k kali. Kompleksitas kondisi pertama sama dengan alternatif satu. Sedangkan pada kondisi kedua, kita dapat langsung memasukkan elemen ke dalam *linked list* dari simpul dengan kategori yang bersesuaian, sehingga kompleksitas untuk satu pengisian elemen adalah O(1). Oleh karena itu, total kompleksitas komputasi pengisian *binary search tree* pada alternatif kedua dapat dinyatakan sebagai berikut.

$$k * \big(O(logk) + O(1)\big) + (n-k) * \big(O(1)\big) = O(k\,logk) + O(n-k)$$
$$= O(n + k\log k)$$

Dengan diasumsikan jumlah kategori dari data (k) jauh lebih kecil daripada jumlah elemen (n), maka didapatkan kompleksitas komputasi dari alternatif kedua ialah O(n). Dengan kata lain, algoritma alternatif kedua ini akan berjalan secara linier jika syarat tersebut terpenuhi. Namun, jika jumlah kategori menyamai jumlah elemen, maka kompleksitas kedua alternatif ini akan setara dengan algoritma pengurutan efektif pada umumnya, yaitu O(n logn).

IV. PENGUJIAN DAN ANALISIS HASIL

A. Pengujian

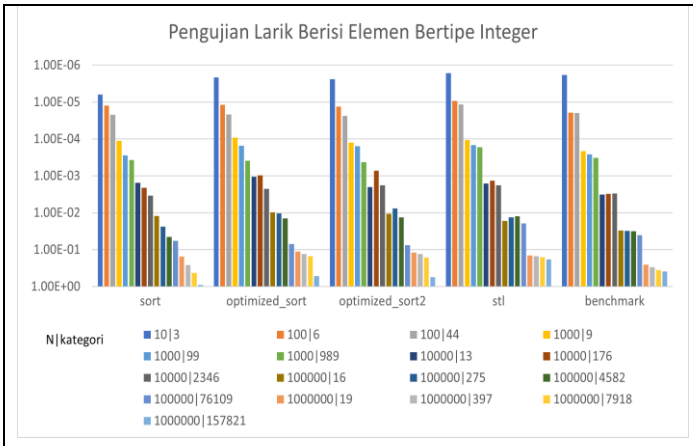
Program ini akan diuji dengan 3 jenis tes, yaitu tes pengurutan larik berisi elemen bertipe integer, pengurutan larik berisi elemen bertipe suatu kelas yang memiliki kunci komparasi bertipe integer, dan pengurutan dataset berisi lima elemen bertipe integer dan satu elemen bertipe string sebagai

kunci komparasi. Selain itu, untuk alternatif kedua, karena implementasi jenis hash table cukup berpengaruh terhadap performansi, maka penulis membuat 2 algoritma untuk alternatif kedua, menggunakan *hash table linear probing* dan *hash table chaining*. Di samping menguji performansi algoritma yang diimplementasikan penulis, dipakai juga algoritma pembanding yaitu fungsi *sort* dari *library* C++ dan juga algoritma *merge sort* yang penulis implementasikan juga. Dipakainya algoritma *merge sort* dikarenakan kita tidak tahu secara langsung implementasi dari algoritma *sort* dari stl, apakaj ia menggunakan *multithreading*, paralelisasi, ataupun teknik lainnya.

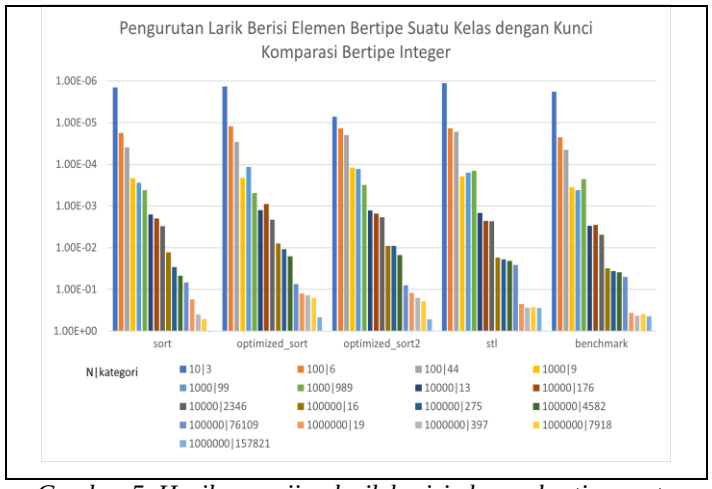
Hasil pengurutan akan diuji keterurutannya setiap kali selesai mengurutkan. Pengujian juga dilakukan dengan berbagai kombinasi antara jumlah elemen larik (n) dan jumlah kategori dari data (k). Sesuai dengan kompleksitas komputasi yang diperkirakan, jika k jauh lebih kecil dari pada n, maka algoritma yang diimplementasikan akan bertindak sebagai algoritma yang berjalan secara linier terutama pada algoritma alternatif kedua. Hasil pengujian disimpan ke dalam file csv yang akan dianalisis lebih lanjut.

B. *Analisis Hasil*

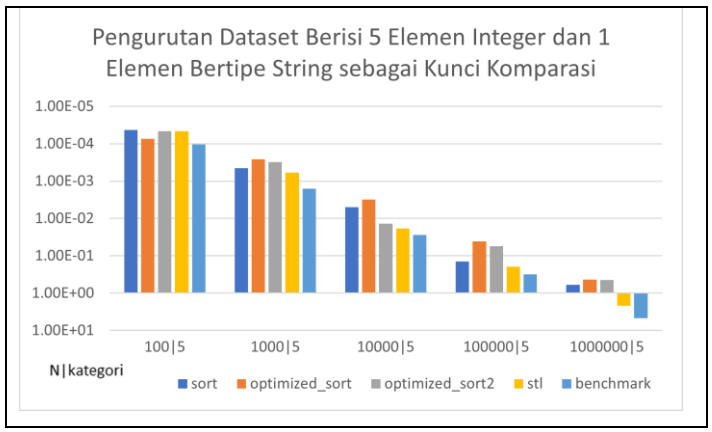
Visualisasi hasil pengujian untuk masing-masing jenis tes dipisahkan untuk mempermudah pembacaan dengan algoritma alternatif pertama berlabel **sort**, alternatif kedua dengan *hash table linear probing* berlabel **optimized_sort,** alternatif kedua dengan *hash table chaining* berlabel **optimized_sort2**. Selain itu, karena orde jumlah elemen dan jumlah kategori sangat berbeda satu sama lain sehingga waktu pengurutannya pun sangat berbeda satu sama lain, maka data divisualisasikan dengan skala logaritmik. Untuk memudahkan pembacaan juga, skala juga dibalik agar pembacaan grafik dapat dari atas ke bawah, sehingga semakin tinggi grafik, maka performansi semakin bagus. Namun, karena data divisualisasikan dengan skala logaritmik, maka perbedaan sedikit saja di ketinggian grafik dapat cukup berpengaruh terhadap perbedaan performansi. Hasil visualisasi disajikan sebagai berikut.



Gambar 4. Hasil pengujian larik berisi elemen bertipe integer



Gambar 5. Hasil pengujian larik berisi elemen bertipe suatu kelas dengan kunci komparasi bertipe integer



Gambar 4. Hasil pengujian dataset berisi 5 elemen bertipe integer dan 1 elemen bertipe string sebagai kunci komparasi

Dapat dilihat pada grafik visualisasi pertama dan kedua, perbedaan terhadap banyak kategori, cukup berdampak pada algoritma alternatif pertama dan sedikit berdampak pada algoritma alternatif kedua. Hal ini sesuai dengan perkiraan kompleksitas komputasi keduanya yaitu O(n logk) dan O(n+k(logk-1)). Sedangkan untuk stl dan *merge sort*, penambahan kategori tidak terlalu berdampak terhadap performansi, yang sesuai juga dengan kompleksitasnya yang O(n logn).

Dari sisi performansi, dapat dilihat pada ketiga grafik pengujian, performa ketiga algoritma dari penulis lebih baik daripada algoritma *merge sort*, terlihat dari grafik bahwa batang ketiga algoritma tersebut lebih tinggi, terutama jika rasio jumlah elemen dan jumlah kategori cukup besar. Sedangkan, dibandingkan dengan stl, algoritma *sort* alternatif pertama cenderung lebih buruk, tetapi masih satu orde dengan algoritma alternatif kedua. Bahkan algoritma optimized_sort dan optimized_sort2 terkadang lebih baik dari stl ketika rasio jumlah elemen dan jumlah kategori cukup besar.

Untuk pengujian dataset, algoritma buatan penulis dapat dikatakan menang telak atas implementasi stl dan *merge sort* untuk rasio jumlah elemen dan jumlah kategori cukup besar. Bahkan untuk jumlah elemen 1,000,000 dan jumlah kategori 5, algoritma optimized_sort 10,65 kali lebih cepat daripada algoritma *merge sort* dan 4,9 kali lebih cepat daripada algoritma stl. Hal ini mungkin dikarenakan komparasi dari tipe string yang lebih mahal daripada komparasi dari tipe integer. Algoritma yang penulis buat memiliki jumlah komparasi yang jauh lebih sedikit tetapi dengan biaya terhadap menginisialisasi banyak struktur data.

V. KESIMPULAN

Algoritma pengurutan yang penulis buat memiliki kompleksitas komputasi O(n logk) dan versi optimalisasinya O(n + k logk). Performa dari algoritma ini lebih bagus daripada algoritma *merge sort* biasa, dan terkadang lebih baik dari algoritma *sort* dari *library* stl terutama untuk rasio jumlah elemen dengan jumlah kategori cukup besar. Untuk dataset dengan rasio jumlah elemen dengan jumlah kategori besar dan tipe komparator string, algoritma penulis dapat 10,65 kali lebih cepat daripada algoritma *merge sort* biasa dan 4,9 kali lebih cepat daripada algoritma *sort* dari *library* stl. Hal ini menunjukkan bahwa algoritma ini efektif untuk mengurutkan data berkategori, terutama untuk data berkategori dengan rasio jumlah elemen dengan jumlah kategori besar dan tipe komparator yang kompleks.

VIDEO LINK AT YOUTUBE

https://youtu.be/GbWmZC76u6s

UCAPAN TERIMA KASIH

Puji syukur penulis panjatkan kepada Tuhan Yang Maha Esa karena dengan rahmat dan berkah-Nya, makalah ini dapat diselesaikan tepat waktu. Penulis juga mengucapkan terima kasih kepada kedua orangtua, serta teman-teman yang telah memberikan dukungan selama pengerjaan makalah ini. Tidak lupa rasa hormat dan terima kasih kepada dosen-dosen pengampu mata kuliah IF2211 Strategi Algoritma, terutama Pak Rinaldi selaku dosen pengampu pada kelas penulis, yang telah memberi materi kepada penulis pada semester keempat Tahun Ajar 2020/2021. Penulis meminta maaf jika terdapat kesalahan kata-kata dalam makalah ini, penulis berharap makalah ini dapat digunakan sebaik-baiknya.

REFERENCES

[1] M. Rinaldi, M. Nur Ulfa, K. Masayu Leylia. 2021. Algoritma Divide and Conquer. Diakses pada laman https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-(2021)-Bagian1.pdf pada tanggal 21 Mei 2022.

[2] Cole, R. (1988). Parallel merge sort. *SIAM Journal on Computing*, 17(4), 770-785.

[3] Agresti, A. (2018). *An introduction to categorical data analysis*. John Wiley & Sons.

[4] Wengrow, J. (2020*). A Common-Sense Guide to Data Structures and Algorithms*. Pragmatic Bookshelf.

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, 23 Mei 2022

Malik Akbar Hashemi Rafsanjani
13520105