Analisis Algoritma *Divide and Conquer* pada Permasalahan Penyusunan Data Secara Terurut

Muhammad Zulfiansyah Bayu Pratama - 13521028
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail : 13521028@std.stei.itb.ac.id

***Abstrak*—Makalah ini menganalisis algoritma Divide and Conquer pada permasalahan penyusunan data secara terurut. Penyusunan data terurut adalah proses penting dalam pengolahan informasi yang melibatkan pengaturan elemen-elemen data dalam urutan teratur dan dapat diprediksi. Algoritma Divide and Conquer merupakan pendekatan efisien untuk menyelesaikan masalah ini dengan membagi permasalahan menjadi submasalah yang lebih kecil, menyelesaikan masing-masing submasalah secara terpisah, dan menggabungkan hasilnya untuk mendapatkan solusi akhir. Makalah ini menjelaskan konsep dasar algoritma Divide and Conquer dan langkah-langkah yang terlibat dalam pendekatan ini. Selanjutnya, dianalisis penerapan algoritma Divide and Conquer pada penyusunan data terurut. Algoritma-algoritma yang dikaji meliputi merge sort, quicksort, dan heapsort. Kelebihan dan kelemahan dari masing-masing algoritma juga dikupas secara detail. Implementasi dari algoritma-algoritma Divide and Conquer dilaporkan dalam makalah ini dengan fokus pada detail teknis implementasi, termasuk struktur data yang digunakan dan langkah-langkah konkret untuk mengimplementasikan algoritma secara efisien. Beberapa contoh kasus uji juga diberikan untuk menguji performa algoritma-implementasi yang dikembangkan. Makalah ini diharapkan memberikan pemahaman yang komprehensif tentang algoritma Divide and Conquer dan aplikasinya dalam penyusunan data terurut. Hal ini akan membantu pembaca dalam memilih algoritma yang paling sesuai untuk kebutuhan mereka, berdasarkan kebutuhan spesifik dan karakteristik data yang dihadapi.**

Kata Kunci—algoritma Divide and Conquer; penyusunan data terurut; *merge sort*; *quicksort*; *heapsort*

I. PENDAHULUAN

Penyusunan data terurut merupakan proses penting dalam pengolahan informasi di berbagai bidang, mulai dari ilmu komputer hingga analisis data. Dalam konteks ini, algoritma *Divide and Conquer* telah terbukti menjadi pendekatan yang efisien dalam menyelesaikan permasalahan tersebut. Pendekatan ini melibatkan pemecahan permasalahan yang kompleks menjadi sub masalah yang lebih kecil, menyelesaikan masing-masing sub masalah secara terpisah, dan menggabungkan hasilnya untuk mendapatkan solusi akhir.

Makalah ini bertujuan untuk menganalisis dan mengimplementasikan algoritma *Divide and Conquer* pada permasalahan penyusunan data secara terurut. Dalam analisis ini, akan dibahas beberapa algoritma terkenal yang digunakan dalam konteks penyusunan data terurut, seperti *merge sort, quicksort,* dan *heapsort*. Setiap algoritma akan dianalisis secara mendalam untuk memahami prinsip kerjanya, kelebihan, dan kelemahannya.

Selain itu, makalah ini juga akan melaporkan implementasi konkret dari algoritma-algoritma Divide and Conquer yang dipilih dalam bahasa pemrograman C. Implementasi ini akan melibatkan pemilihan struktur data yang tepat dan langkah-langkah implementasi yang efisien untuk memastikan kinerja optimal dari algoritma yang diimplementasikan.

Selanjutnya, dalam makalah ini akan disajikan beberapa kasus uji yang digunakan untuk menguji performa algoritma-implementasi yang dikembangkan. Hal ini bertujuan untuk mengevaluasi efektivitas dan efisiensi dari algoritma-algoritma Divide and Conquer dalam menyelesaikan permasalahan penyusunan data secara terurut.

Dengan pemahaman yang mendalam tentang algoritma Divide and Conquer dan implementasinya pada penyusunan data terurut, diharapkan pembaca dapat memperoleh wawasan yang lebih luas tentang pendekatan ini. Hal ini akan membantu pembaca dalam memilih algoritma yang paling sesuai untuk kebutuhan mereka, serta meningkatkan pemahaman tentang analisis dan implementasi algoritma dalam konteks pengolahan data.

Dalam lanjutan makalah ini, akan dibahas secara rinci mengenai konsep dan prinsip kerja algoritma Divide and Conquer serta penerapannya pada penyusunan data terurut menggunakan algoritma-algoritma yang telah disebutkan sebelumnya.

II. DASAR TEORI

A. *Algoritma Divide and Conquer*

Algoritma *Divide and Conquer* adalah pendekatan dalam pemecahan masalah yang kompleks dengan membaginya menjadi sub masalah yang lebih kecil, menyelesaikan masing-masing sub masalah secara terpisah, dan menggabungkan hasilnya untuk mendapatkan solusi akhir. Pendekatan ini terdiri dari tiga langkah utama, yaitu :

1. *Divide* : membagi persoalan menjadi beberapa upa-persoalan yang memiliki kemiripan dengan persoalan

semula namun berukuran lebih kecil (idealnya berukuran hampir sama)

2. *Conquer* : menyelesaikan masing-masing upa-persoalan (secara langsung jika sudah berukuran kecil atau secara rekursif jika masih berukuran besar)
3. *Combine* : menggabungkan solusi masing-masing upa-persoalan sehingga membentuk solusi persoalan semula.[1]

Langkah pembagian memecah masalah menjadi sub masalah yang lebih kecil, langkah penyelesaian menyelesaikan sub masalah secara rekursif, dan langkah penggabungan menggabungkan solusi dari submasalah menjadi solusi akhir. Algoritma *Divide and Conquer* sering digunakan dalam berbagai masalah, termasuk penyusunan data terurut.

B. Data

Data adalah catatan atas kumpulan fakta.[2] Fakta tersebut dapat berupa teks, angka, gambar, suara, atau bentuk lainnya. Data dapat diklasifikasikan menjadi beberapa jenis, termasuk data numerik (misalnya, angka), data kategorikal (misalnya, jenis kelamin atau warna), data ordinal (misalnya, peringkat), dan data spasial (misalnya, koordinat geografis). Pada makalah ini kita fokus terhadap data yang berisi sejumlah bilangan berupa larik *(array)*.

C. Penyusunan Data Terurut

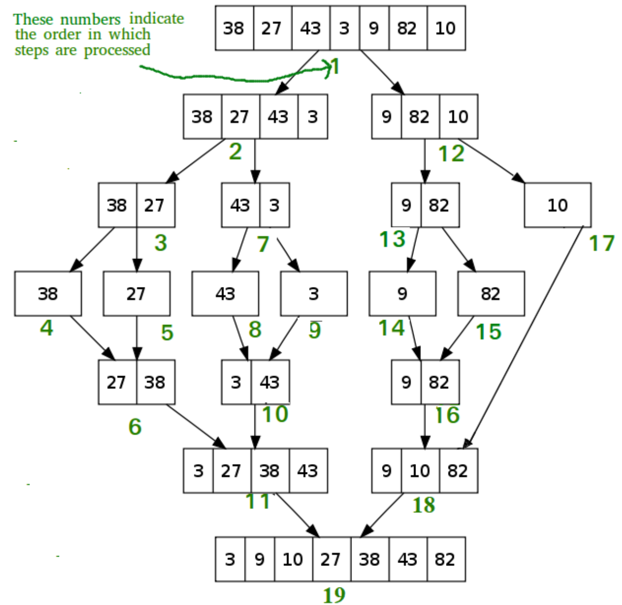
Penyusunan data terurut adalah proses mengatur elemen-elemen data dalam urutan teratur dan dapat diprediksi. Data yang disusun secara terurut memudahkan pencarian, pemrosesan, dan analisis data. Beberapa algoritma yang sering digunakan untuk penyusunan data terurut meliputi *merge sort*, *quicksort*, dan *heapsort*. Tujuan utama dari penyusunan data terurut adalah meningkatkan efisiensi dan keterbacaan data.

D. Merge Sort

Merge sort adalah algoritma *Divide and Conquer* yang efisien untuk menyusun data terurut. Algoritma ini membagi larik menjadi sub larik yang lebih kecil, menyusun masing-masing sub daftar secara terpisah, dan menggabungkan sub larik tersebut secara berurutan. Langkah penggabungan dilakukan dengan membandingkan dan menggabungkan elemen-elemen dari dua sub daftar menjadi satu daftar terurut.

Cara kerja algoritma ini adalah :

1. *Divide* : jika larik yang akan diurutkan lebih dari satu elemen, bagi larik tersebut menjadi dua larik sama besar (atau mendekati sama besar jika panjang larik tersebut ganjil). Kemudian lakukan *merge sort* pada masing-masing sub larik tersebut.
2. *Conquer* : ketika larik berisi hanya satu elemen, tidak perlu lakukan apapun. Larik yang berisi satu elemen sudah pasti terurut
3. *Combine* : jika terdapat dua larik yang terurut, gabungkan keduanya menjadi sebuah larik yang terurut



Gambar 1. Ilustrasi merge sort
Sumber : https://www.geeksforgeeks.org/merge-sort/

E. Quicksort

Quicksort adalah algoritma *Divide and Conquer* yang efisien untuk penyusunan data terurut. Algoritma ini memilih elemen tertentu sebagai pivot, membagi daftar data menjadi dua sub larik berdasarkan pivot, dan menyusun masing-masing subdaftar secara rekursif. Proses pengurutan dilakukan dengan membandingkan dan memindahkan elemen-elemen sehingga elemen-elemen yang lebih kecil dari pivot berada di sebelah kiri pivot, dan elemen-elemen yang lebih besar berada di sebelah kanan pivot.

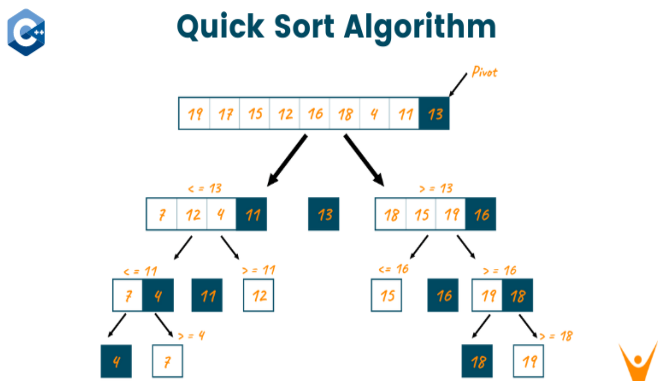
Tahapan dari *quicksort adalah :*

1. *Divide* : pilih suatu elemen yang kita sebut pivot, kemudian kita bagi larik menjadi dua sehingga salah satunya selalu lebih kecil atau sama dengan pivot dan yang sisanya selalu lebih besar dari pivot. Kemudian, lakukan *quicksort* pada masing-masing sub larik tersebut.
2. *Conquer* : ketika larik hanya memiliki satu elemen, larik tersebut sudah terurut.
3. *Combine* : gabungkan sub larik dengan menempelkan hasil *quicksort* bagian kiri dan kanan.

Bagian utama dari algoritma ini adalah proses partisi (bagian divide). Untuk mempartisi larik, ada beberapa algoritma untuk melakukan partisi tanpa menggunakan larik sementara. Salah satunya adalah algoritma partisi Hoare. Cara kerja algoritma ini adalah :

1. Pilih sebuah pivot.

2. Siapkan dua variabel penunjuk, kiri dan kanan dengan masing-masing variabel menunjuk ujung kiri dan ujung kanan dari larik
3. Gerakkan variabel kiri ke arah kanan, sampai elemen yang ditunjuk lebih besar dari pivot.
4. Gerakkan variabel kanan ke arah kiri, sampai elemen yang ditunjuk kurang dari atau sama dengan pivot.
5. Jika kiri kurang dari atau sama dengan kanan, tukar elemen yang ditunjuk kiri dan kanan, kemudian gerakkan kiri ke kanan satu langkah dan kanan ke kiri satu langkah.
6. Jika kiri kurang dari atau sama dengan kanan, maka kembali ke tahap 3. Jika tidak, selesai.



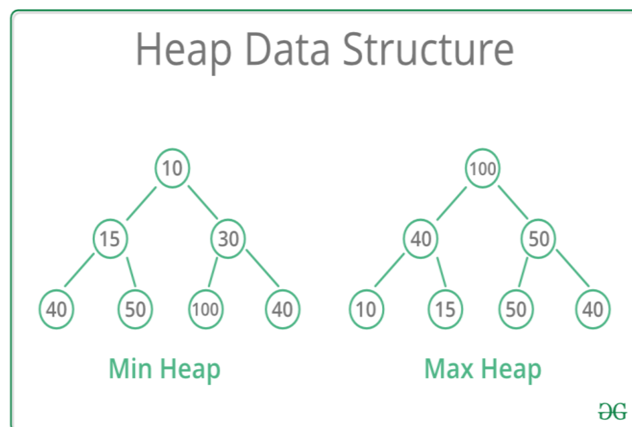
Gambar 2. Ilustrasi Quicksort
Sumber : https://favtutor.com/blogs/quick-sort-cpp

F. *Heapsort*

Heapsort adalah algoritma *Divide and Conquer* yang menggunakan struktur data heap untuk menyusun data terurut. Nama heap sendiri berasal dari Bahasa Inggris, yang berarti "gundukan". Heap merupakan struktur data yang mampu mencari nilai terbesar secara efisien dari sekumpulan elemen yang terus bertambah. Algoritma ini membangun heap dari daftar data, mempertahankan sifat heap, dan secara berulang menghapus elemen terbesar (akar heap) untuk membangun daftar terurut.

Langkah dari *heapsort* adalah :

1. Membangun heap maksimum: Langkah pertama dalam heapsort adalah membangun heap maksimum dari larik angka yang diberikan. Ini dilakukan dengan memperoleh struktur heap biner dari larik, di mana setiap elemen dalam larik mewakili simpul dalam heap. Heap maksimum adalah heap di mana setiap simpul induk memiliki nilai yang lebih besar dari nilai-nilai anak-anaknya. Dimulai dari elemen terakhir dalam larik, lakukan perkolasi ke bawah (percolate down) untuk setiap elemen ke elemen pertama. Hal ini dilakukan dengan membandingkan nilai elemen saat ini dengan nilai anak-anaknya dan menukar posisi jika diperlukan untuk menjaga sifat heap maksimum.
2. Pengurutan: Setelah heap maksimum terbentuk, elemen terbesar (elemen di akar heap) akan berada di posisi akhir larik. Tukar elemen ini dengan elemen pertama dalam larik.
3. Penyusutan heap: Setelah pertukaran elemen, elemen terbesar berada di posisi terakhir larik. Sekarang, larik tidak lagi memenuhi sifat heap maksimum. Untuk memperbaiki ini, lakukan perkolasi ke bawah dari elemen pertama ke elemen terakhir dalam larik. Ini akan memindahkan elemen terbesar dalam larik ke posisi yang benar dalam heap.
4. Ulangi langkah 2 dan 3: Ulangi langkah-langkah kedua dan ketiga sampai seluruh larik terurut. Setiap kali elemen terbesar dipindahkan ke posisi akhir larik, larik yang tersisa untuk diurutkan akan menjadi lebih pendek. Proses ini berlanjut sampai hanya ada satu elemen yang tersisa dalam larik, yang berarti larik sudah terurut.
5. Selesai: Setelah seluruh larik terurut, proses heapsort selesai.



Gambar 2. Ilustrasi Struktur Data Heap
Sumber : https://www.geeksforgeeks.org/heap-data-structure/

III. Analisis dan Pembahasan

A. *Analisis Algoritma Merge Sort*

```
// Fungsi merge sort
void merge(int arr[], int l, int m, 
int r){
    int i, j, k;
    int n1 = m-l+1;
    int n2 = r-m;

    // Membuat array sementara
    int L[n1], R[n2];

    // Memindahkan data ke array 
sementara
    for(i=0; i<n1; i++){
        L[i] = arr[l+i];
    }
    for(j=0; j<n2; j++){
        R[j] = arr[m+1+j];
    }

    // Merge array sementara ke 
array asli
    i=0;
    j=0;
    k=l;
    while(i<n1 && j<n2){
        if(L[i] <= R[j]){
            arr[k] = L[i];
            i++;
        }
        else{
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Memasukkan sisa elemen L 
jika ada
    while(i<n1){
        arr[k] = L[i];
        i++;
        k++;
    }

    // Memasukkan sisa elemen R 
jika ada
    while(j<n2){
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Merge sort memiliki kompleksitas waktu rata-rata O(n log n). Ini berlaku pada sebagian besar kasus, di mana setiap pembagian data menghasilkan subdaftar yang cukup seimbang dalam ukuran. Proses penggabungan membutuhkan langkah-langkah yang sebanding dengan jumlah elemen yang diurutkan. Dalam kasus rata-rata, merge sort menunjukkan kinerja yang baik dalam menyusun data terurut secara efisien.

Dalam kasus terburuk *merge sort*, kompleksitas waktu yang terjadi adalah O(n log n). Hal ini terjadi ketika setiap pembagian data menghasilkan sub larik yang hampir sama besar, dan proses penggabungan memerlukan banyak langkah untuk menyusun daftar terurut secara keseluruhan. Namun, walaupun memiliki kompleksitas waktu yang sama untuk setiap kasus, merge sort masih dianggap efisien karena memiliki kinerja yang konsisten dan terjamin.

Dalam kasus terbaik merge sort, kompleksitas waktu yang terjadi juga O(n log n). Hal ini terjadi ketika setiap pembagian data menghasilkan subdaftar yang sama besar secara tepat, sehingga proses penggabungan berjalan dengan efisien tanpa perlu banyak perbandingan dan perpindahan elemen.

Merge sort membutuhkan ruang memori tambahan untuk menyimpan subdaftar sementara selama proses pembagian dan penggabungan. Ukuran memori tambahan yang diperlukan adalah sebanding dengan ukuran data yang diurutkan. Namun, karena merge sort tidak memerlukan operasi pengurangan dalam pengurutan in-place, ruang memori tambahan yang digunakan biasanya tidak signifikan dan dapat diterima dalam banyak kasus.

B. *Analisis Algoritma Quicksort*

```
// Bagian partisi
int partition(int arr[], int low, int 
high){
    int pivot = arr[high]; // Pivot
    int i = (low-1); // Index dari 
elemen yang lebih kecil

    for(int j=low; j<=high-1; j++){
        // Jika elemen saat ini lebih 
kecil dari pivot
        if(arr[j] < pivot){
            i++; // Menaikkan index 
dari elemen yang lebih kecil
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i+1], &arr[high]);
    return(i+1);
}

// Fungsi untuk melakukan quick sort
void quickSort(int arr[], int low, 
int high){
    if(low < high){
        // pi adalah index partisi, 
arr[pi] sekarang di tempat yang benar
        int pi = partition(arr, low, 
high);

        // Memisahkan elemen sebelum 
dan sesudah partisi
        quickSort(arr, low, pi-1);
        quickSort(arr, pi+1, high);
    }
}
```

Quicksort memiliki kompleksitas waktu rata-rata O(n log n). Pada sebagian besar kasus, quicksort menunjukkan kinerja yang baik dengan membagi data secara seimbang dan melakukan pengurutan dengan efisien. Namun, pemilihan pivot yang tidak optimal dapat mengurangi kinerja quicksort dalam kasus terburuk.

Dalam kasus terburuk quicksort, kompleksitas waktu yang terjadi adalah O(n^2). Hal ini terjadi ketika pemilihan pivot tidak optimal dan setiap pembagian data menghasilkan subdaftar yang sangat tidak seimbang dalam ukuran, seperti saat data yang diurutkan sudah dalam urutan yang terbalik. Dalam kasus ini, proses pengurutan membutuhkan langkah-langkah yang lebih banyak dan kompleksitas waktu

Dalam kasus terbaik quicksort, kompleksitas waktu yang terjadi adalah O(n log n). Hal ini terjadi ketika pemilihan pivot optimal dan setiap pembagian data menghasilkan subdaftar yang seimbang secara merata dalam ukuran. Dalam kasus ini, proses pengurutan berjalan dengan efisien dan membutuhkan jumlah langkah yang lebih sedikit.

Quicksort dapat diimplementasikan dalam dua bentuk: in-place quicksort dan quicksort dengan ruang memori tambahan. Dalam in-place quicksort, tidak ada ruang memori tambahan yang diperlukan selain ruang stack untuk rekursi. Namun, quicksort dengan ruang memori tambahan dapat membutuhkan ruang tambahan untuk menyimpan subdaftar sementara selama proses pembagian dan pengurutan.

C. Analisis Algoritma Heapsort

```
// Fungsi untuk melakukan heap sort
void heapify(int arr[], int n, int
i){
    int largest = i; // Inisialisasi
largest sebagai root
    int l = 2*i+1; // Kiri = 2*i+1
    int r = 2*i+2; // Kanan = 2*i+2

    // Jika anak kiri lebih besar
dari root
    if(l<n && arr[l] > arr[largest]){
        largest = l;
    }

    // Jika anak kanan lebih besar
dari largest sekarang
    if(r<n && arr[r] > arr[largest]){
        largest = r;
    }

    // Jika largest bukan root
    if(largest != i){
        swap(&arr[i], &arr[largest]);

        // Memanggil fungsi heapify
secara rekursif pada sub-tree yang
terpengaruh
        heapify(arr, n, largest);
    }
}
```

```
void heapSort(int arr[], int n){
    // Membuat heap
    for(int i=n/2-1; i>=0; i--){
        heapify(arr, n, i);
    }

    // Mengeluarkan elemen satu per
satu dari heap
    for(int i=n-1; i>0; i--){
        // Memindahkan root ke akhir
        swap(&arr[0], &arr[i]);

        // Memanggil fungsi heapify
pada heap yang dikurangi
        heapify(arr, i, 0);
    }
}
```

Heapsort memiliki kompleksitas waktu rata-rata O(n log n), karena jumlah langkah yang diperlukan untuk membangun heap dan menghapus elemen terbesar tetap proporsional terhadap jumlah elemen yang diurutkan. Algoritma ini menunjukkan kinerja yang stabil dan efisien dalam kasus rata-rata.

Heapsort memiliki kompleksitas waktu yang tetap pada setiap kasus, yaitu O(n log n), termasuk dalam kasus terburuk. Karena sifat struktur data heap yang selalu terjaga, jumlah langkah yang diperlukan untuk menyusun data terurut tetap sama, terlepas dari keadaan data yang diurutkan.

Heapsort tidak memiliki kasus terbaik yang berbeda dari kasus rata-rata, karena jumlah langkah yang diperlukan tetap sama untuk setiap kasus. Sehingga kompleksitas waktu pada kasus terbaik adalah O(n log n).

Heapsort dapat diimplementasikan dalam dua bentuk: *in-place heapsort* dan *heapsort* dengan ruang memori tambahan. Dalam in-place heapsort, tidak ada ruang memori tambahan yang diperlukan selain ruang stack untuk rekursi dan beberapa variabel tambahan. Namun, heapsort dengan ruang memori tambahan dapat membutuhkan ruang tambahan untuk menyimpan heap sementara selama proses pengurutan. Meskipun demikian, kebutuhan memori tambahan biasanya tidak signifikan.

IV. Kesimpulan

Dalam analisis efektivitas waktu dan memori untuk algoritma-algoritma penyusunan data terurut menggunakan pendekatan Divide and Conquer, dapat diambil beberapa kesimpulan:

1. Merge sort adalah algoritma yang memiliki kompleksitas waktu O(n log n) dalam kasus terburuk, terbaik, maupun rata-rata. Algoritma ini menunjukkan kinerja yang konsisten dan terjamin, dengan langkah-langkah yang efisien dalam membagi data menjadi subdaftar yang lebih kecil dan menggabungkannya menjadi satu daftar terurut. Dalam hal memori, merge sort memerlukan ruang memori tambahan yang sebanding dengan ukuran data yang diurutkan, namun

ukuran tersebut biasanya dapat diterima dalam banyak kasus.

2. Quicksort adalah algoritma yang memiliki kompleksitas waktu O(n^2) dalam kasus terburuk jika pemilihan pivot tidak optimal. Namun, dalam kasus terbaik dan rata-rata, kompleksitas waktu quicksort adalah O(n log n). Algoritma ini menunjukkan kinerja yang efisien dalam membagi data secara seimbang dan melakukan pengurutan. Dalam hal memori, quicksort dapat diimplementasikan dalam dua bentuk, yaitu in-place quicksort yang tidak memerlukan ruang memori tambahan, dan quicksort dengan ruang memori tambahan yang memerlukan penyimpanan subdaftar sementara. Pemilihan bentuk implementasi akan mempengaruhi kebutuhan memori tambahan.

3. Heapsort adalah algoritma dengan kompleksitas waktu O(n log n) dalam kasus terburuk, terbaik, maupun rata-rata. Algoritma ini menggunakan struktur data heap untuk menyusun data terurut. Heapsort menunjukkan kinerja yang stabil dan efisien, karena jumlah langkah yang diperlukan tetap sama pada setiap kasus. Dalam hal memori, heapsort dapat diimplementasikan sebagai in-place heapsort yang tidak memerlukan ruang memori tambahan, atau heapsort dengan ruang memori tambahan yang memerlukan penyimpanan heap sementara. Kebutuhan memori tambahan biasanya tidak signifikan.

REFERENSI

[1] Vardiansyah, Dani. Filsafat Ilmu Komunikasi: Suatu Pengantar, Indeks, Jakarta 2008. Hal.3

[2] Munir, Rinaldi "Algoritma Divide And Conquer" https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-(2021)-Bagian1.pdf (diakses pada tanggal 22 Mei 2023

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, 22 Mei 2023

Ttd

Muhammad Zulfiansyah Bayu Pratama

13521028