Kompleksitas dari Variasi Algoritma Pengurutan Data dalam Larik

Muhammad Hendry Prasetya (13517105)
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517105@std.stei.itb.ac.id

Abstract*—Suatu algoritma dapat dikatakan baik apabila memenuhi kriteria-kriteria tertentu, salah satunya kemangkusan. Kemangkusan algoritma diukur dari jumlah waktu dan ruang yang digunakan. Algoritma yang mangkus adalah algoritma yang kebutuhan waktu dan ruangnya minimum. Untuk mengetahui kemangkusan suatu algoritma, kita dapat menghitung kompleksitas algoritma tersebut dengan menghitung banyaknya eksekusi program. Pada makalah ini, algoritma yang akan dibahas yaitu algoritma pengurutan *bubble sort, counting sort, heap sort, radix sort, merge sort,* dan *quick sort*. Dari algoritma-algoritma tersebut, *heap sort* memiliki kompleksitas waktu asimptotik dan kompleksitas ruang paling kecil kemudian diikuti oleh *merge sort, quick sort, counting sort, radix sort* dan *bubble sort.

***Keywords*—Kompleksitas, Algoritma, Sort, Data**

I. PENDAHULUAN

Algoritma adalah urutan atau langkah-langkah sistematis untuk menyelesaikan suatu masalah. Untuk menyelesaikan suatu masalah tertentu, banyak algoritma yang dapat digunakan. Kompleksitas algoritma dapat membantu pemrogram untuk memilih algoritma mana yang paling mangkus dan paling baik digunakan dalam kasus tertentu. Tiap algoritma memiliki kemangkusan yang dapat diukur dengan menghitung kompleksitas algoritma. Algoritma yang berbeda satu sama lain apabila diterapkan pada suatu data tertentu akan membutuhkan waktu eksekusi dan ruang memori yang berbeda.

Pada dunia pemrograman, ada banyak alternatif solusi untuk mengurutkan suatu data. Beberapa algoritma pengurutan hanya bisa diterapkan pada data tertentu, misalnya algoritma *counting sort* dan *radix sort* yang hanya bisa diterapkan pada data kuantitatif. Algoritma pengurutan memiliki kelebihan dan kekurangan masing-masing. Kelebihan dan kekurangan yang paling sering dibandingkan antar algoritma pengurutan adalah waktu eksekusi algoritma tersebut.

II. TEORI DASAR KOMPLEKSITAS ALGORITMA

Dalam praktiknya, kita tidak dapat mengetahui secara akurat waktu yang digunakan. Komputer dengan arsitektur berbeda dapat memiliki waktu eksekusi yang berbeda pula dalam program yang sama. Oleh karena itu, kita membutuhkan model abstrak yang tidak berbeda-beda nilainya pada komputer dan *compiler* apapun dalam pengukuran waktu dan ruang yang digunakan oleh suatu algoritma.Besaran yang digunakan dalam model abstrak tersebut adalah kompleksitas algoritma.

Kompleksitas algoritma yang akan dibahas yaitu kompleksitas waktu. Kompleksitas waktu suatu algoritma berisi banyaknya langkah eksekusi yang dibutuhkan dalam memproses masukan. Kompleksitas waktu dinyatakan dalam fungsi T(n) dengan n adalah banyaknya masukan. Kompleksitas waktu dihitung berdasarkan jumlah operasi abstrak.

Kompleksitas waktu dibedakan menjadi tiga macam:

1. $T_{max}(n)$ yaitu kompleksitas waktu untuk kasus terburuk (*worst case).*
2. $T_{min}(n)$ yaitu kompleksitas waktu untuk kasus terbaik (*best case*).
3. $T_{avg}(n)$ yaitu kompleksitas waktu untuk kasus rata-rata (*average case*).

Apabila kita meninjau kompleksitas waktu T(n) suatu algoritma dengan fungsi pangkat berderajat m, pertumbuhan jumlah eksekusi akan sembanding dengan n^m. Oleh karena itu, fungsi T(n) berderajat m relatif sama dengan n^m. Kita dapat menyatakan kompleksitas waktu fungsi tersebut dalam notasi $O(n^m)$. Notasi ini disebut notasi O-Besar yang merupakan notasi kompleksitas waktu asimptotik.

Kompleksitas waktu asimptotik dibedakan menjadi tiga macam:

1. $\Omega(n)$ yaitu kompleksitas waktu asimptotik untuk kasus terbaik (*best case*).
2. $\theta(n)$ yaitu kompleksitas waktu asimptotik untuk kasus rata-rata (*average case*).
3. O(n) yaitu kompleksitas waktu aimptotik untuk kasus terburuk (*worst case*).

- Notasi Omega

  $T(n) = \Omega(h(n))$ jika $T(n) = O(h(n))$ dan $T(n) = \Omega(g(n))$
- Notasi Theta

  $T(n) = \theta(f(n))$ jika terdapat konstanta C dan n_0 sedemikian sehingga $|T(n)| \geq c|f(n)|$, $\forall\ n \geq n_0$
- Notasi O-Besar

  $T(n) = O(f(n))$ jika terdapat konstanta C dan n_0 sedemikian sehingga

  $T(n) \leq C(f(n))$, $\forall\ n \geq n_0$

Teorema: misal $T(n) = a_n n^m + ... + a_1 n + a_0$ adalah suatu polinom derajat m. Maka T(n) berorde n^m.

Kecepatan penyelesaian sebuah algoritma biasanya diukur dengan acuan keadaan terburuk (*worst case*) yang dinyatakan

dengan notasi O-Besar. Notasi O-Besar memiliki pengelompokkan algoritma tertentu untuk T(n) yang berbeda-beda. Pengelompokkan tersebut berdasarkan kompleksitas waktu T(n).

Pengelompokkan algoritma dengan O-Besar terurut membesar dari atas ke bawah berdasarkan spektrum kompleksitas waktu algoritma:

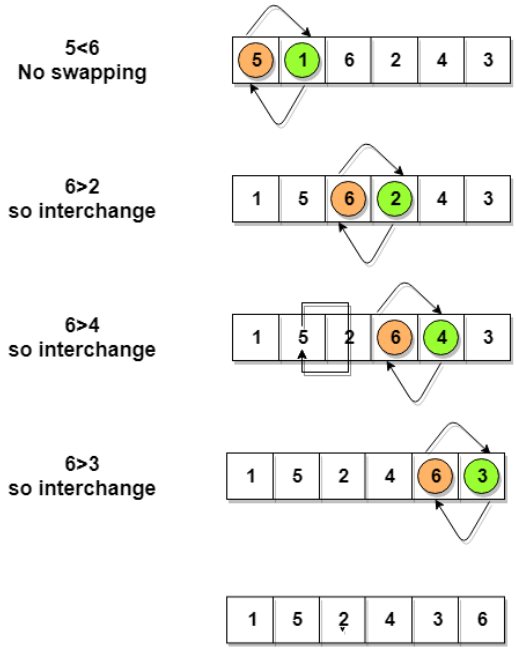
Kelompok Algoritma	Nama
O(1)	konstan
O(log n)	logaritmik
O(n)	Lanjar
O(n log n)	n log n
$O(n^2)$	kuadratik
$O(n^3)$	kubik
$O(2^n)$	eksponensial
O(n!)	Faktorial

Tabel 2.1 Kelompok Algoritma berdasarkan Kompleksitas Waktu Asimptotik pada Kasus Terburuk..

III. PEMBAHASAN

A. Kompleksitas Algoritma Bubble Sort

Algoritma *bubble sort* adalah algoritma pengurutan yang membandingkan tiap elemen dengan seluruh elemen lainnya kemudian menukarnya sampai seluruh elemen terurut. Algoritma *bubble sort* adalah algoritma yang paling mudah dipahami oleh pemrogram pemula.



Gambar 3.1 Sketsa Algoritma Bubble Sort. Sumber: www.studytonight.com/data-structures/bubble-sort

Langkah-langkah yang dilakukan dalam algoritma *bubble sort* dalam mengurutkan n data dalam larik:

1. Meninjau elemen pertama dan membandingkannya dengan elemen kedua.
2. Menukar elemen pertama dengan elemen kedua jika elemen pertama lebih besar daripada elemen kedua. Sebaliknya, jika elemen pertama tidak lebih besar daripada elemen kedua, program tidak perlu menukar elemen manapun.
3. Tinjau elemen berikutnya dan lakukan langkah-langkah sebelumnya hingga seluruh elemen ditinjau.
4. Jika seluruh elemen telah ditinjau, ulangi langkah 1 s.d. 3 untuk elemen ke-2 hingga ke-n.

```
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

       // Last i elements are already in place
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}
```

Gambar 3.2 Potongan Algoritma Bubble Sort. Sumber: https://www.geeksforgeeks.org/bubble-sort.

Algoritma *bubble sort* dapat dikatakan sebagai algoritma pengurutan dengan waktu eksekusi paling lambar di antara algoritma pengurutan populer lainnya. Algoritma ini memiliki kompleksitas waktu asimptotik pada kasus terburuk, rata-rata, dan terbaik berturut-turut adalah $O(n^2)$, $\theta(n^2)$, dan $\Omega(n)$. Untuk kompleksitas ruang, algoritma *bubble sort* membutuhkan ruang sebesar O(log n) pada kasus terburuknya.

Algoritma *bubble sort* tergolong algoritma yang paling mudah dipahami karena hanya memerlukan pemahaman tentang larik dan skema traversal perulangan bersarang. Selain itu, sketsa algoritma ini relatif mudah dipahami.

B. Kompleksitas Algoritma Counting Sort

Algoritma *counting sort* adalah algoritma dengan teknik pengurutan data berdasarkan kunci dalam rentang tertentu. Cara kerja algoritma *counting sort* yaitu dengan menghitung jumlah data yang memiliki nilai kunci yang berbeda. Kemudian, algoritma ini dilakukan operasi pada baris aritmatika sehingga mendapatkan suatu larik berisi data terurut.

```
Counting – Sort (A, B, k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
▷ C[i] now contains the number of elements equal to i.
for i ← 2 to k
    do C[i] ← C[i] + C[i – 1]
▷ C[i] now contains the number of elements less than or equal to i.
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]]-1
```

Gambar 3.3 Sketsa Algoritma Counting Sort. Sumber: www.myassignmenthelp.net/counting-sort-assignment-help

Langkah-langkah yang dilakukan dalam algoritma *counting sort*:

1. Semua elemen larik C diisi dengan nilai 0.
2. Melakukan skema traversal pada larik A berisi data sebelum

diurutkan dari elemen ke-1 sampai elemen terakhir.
3. Jika pada larik A ke-i berisi nilai x, nilai pada larik C ke-x ditambah satu.
4. Melakukan skema traversal pada larik C dari elemen ke-2 sampai elemen terakhir.
5. Mengubah nilai elemen larik C sehingga C ke-i bernilai C ke-i ditambah elemen tepat sebelumnya.
6. Melakukan skema traversal dari panjang larik A sampai satu. Mengisi nilai larik B dengan larik A sesuai larik C.

Dari sketsa di atas, kita dapat menentukan kompleksitas waktu dari algoritma *counting sort*. Jika panjang seluruh larik adalah n, T(n) = 5n - 1. Ambil koefisien C sebesar 6 sedemikian sehingga O-Besar bernilai O(n). Dengan demikian, algoritma *counting sort* memungkinkan kita untuk mengurutkan data dengan kompleksitas waktu asimptotik O(n). Namun, rentang kunci dari *counting sort* tidak selalu bernilai n. Jika k adalah rentang kunci dari *counting sort*, kompleksitas waktu asimptotik algoritma *counting sort* pada kasus terburuk, rata-rata, dan terbaik berturut-turut adalah O(n+k), θ(n+k), dan Ω(n+k).

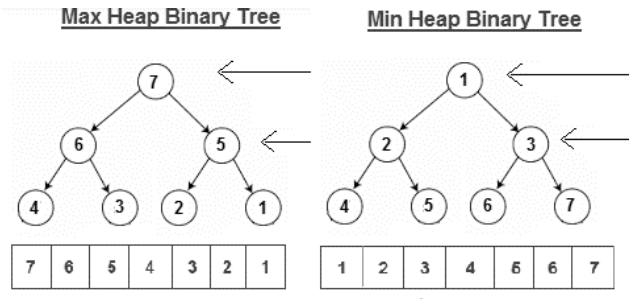
Algoritma *counting sort* hanya dapat dilakukan untuk data kuantitatif serta data tersebut harus merupakan bilangan bulat. Hal ini disebabkan oleh kunci atau index larik harus bernilai bilangan bulat. Algoritma ini pun akan memiliki waktu eksekusi yang besar jika rentang nilai data cukup besar.

C. Kompleksitas Algoritma Heap Sort

Algotirma *heap sort* adalah algoritma pengurutan yang menggunakan teknik perbandingan berdasarkan data struktur tumpukan biner (*Binary Heap*). Tumpukan biner adalah pohon biner lengkap yang simpul dan daunnya ditempatkan dengan aturan tertentu. Pohon biner sendiri merupakan sebuah pohon struktur data yang setiap simpulnya memiliki paling banyak dua anak.

Tumpukan membesar dari bawah disebut *max heap* dan tumpukan mengecil dari bawah disebut *min heap*. Tumpukan tersebut bisa direpresentasikan dalam dua bentuk yaitu bentuk pohon biner dan bentuk larik. Pada makalah ini, bentuk yang akan dibahas adalah bentuk larik.

Untuk merepresentasikan pohon biner dalam larik, kita dapat melakukan pemberian indeks tertentu pada elemen larik. Indeks pertama yang ditandai dengan indeks ke-i merupakan simpul paling atas dari pohon. Selanjutnya, larik dengan indeks 2 kali i ditambah 1 ditandai sebagai anak kiri pohon dan larik dengan indeks 2 kali i ditambah dua ditandai sebagai anak kanan pohon. Indeks larik dimulai dari nol.



Gambar 3.4 Max and Min Heap. Sumber: javabypatel.blogspot.com/2015/11/heap-sort-algorithm.html

Langkah-langkah yang dilakukan dalam menyusun algoritma *heap sort*:
1. Membuat sketsa dari pohon biner berdasarkan larik
2. Memeriksa elemen larik satu per satu dengan skema bawah ke atas (*bottom up*).
3. Pada *max heap*, nilai elemen larik yang menjadi orangtua (*parent*) selalu lebih besar daripada anaknya (*child*). Apabila nilai elemen larik tersebut lebih kecil daripada anaknya, tukar elemen tersebut dengan anaknya yang memiliki nilai lebih besar. Jika elemen telah ditukar tetapi ada anak elemen tersebut yang nilainya lebih besar, tukar elemen tersebut dengan anaknya yang memiliki nilai lebih besar. Dengan demikian, konfigurasi larik pada keluarga (*family*) tersebut telah benar. Proses ini disebut dengan *heapify*.
4. Pada *min heap*, nilai elemen larik yang menjadi orangtua (*parent*) selalu lebih kecil daripada anaknya (*child*). Lakukan proses seperti pada *max heap*, tetapi dengan konfigurasi sesuai dengan *min heap*.
5. Bagian keluarga (family) pohon yang telah diproses tidak perlu diproses kembali, sehingga dapat dihapus dari sketsa pohon.
6. Lakukan proses konfigurasi tersebut hingga pohon teratas memiliki konfigurasi yang sesuai dengan *max/min heap*.

```
// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

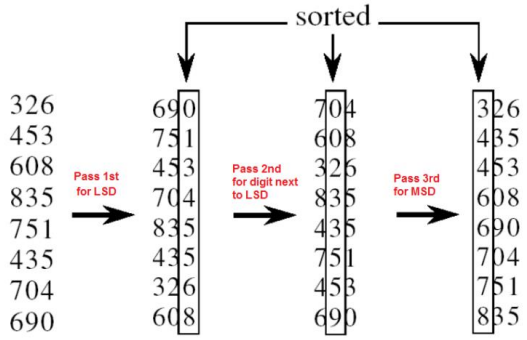
Gambar 3.5 Potongan Algoritma Max Heap Sort. Sumber: https://www.geeksforgeeks.org/heap-sort

Algoritma *heap sort* membutuhkan pemahaman tentang pohon biner dalam penyusunannya. Algoritma *heap sort* memiliki kompleksitas waktu asimptotik yang sama dengan algoritma terkait pohon biner lainnya yaitu sebesar O(n log n), θ(n log n), dan Ω(n log n). Kompleksitas waktu asimptotik tersebut selalu sama di setiap kasus: terbaik, rata-rata, dan terburuk. Selain itu, algoritma ini memiliki kompleksitas ruang yang kecil yaitu O(1).

D. Kompleksitas Algoritma Radix Sort

Algoritma *radix sort* merupakan salah satu algoritma yang membandingkan batas bawah satu dengan lainnya terlebih dahulu dalam pengurutan (*lower bound for comparison based*). Cara kerja algoritma *radix sort* yaitu dengan membandingkan nilai dari digit paling tidak signifikan setiap elemen data. Setelah digit paling tidak signifikan dibandingkan, tinjau digit paling tidak signifikan berikutnya. Hal ini dilakukan berulang hingga

data dengan digit terbanyak diproses.



Gambar 3.6 Sketsa Algoritma Radix Sort. Sumber: www.codingeek.com/algorithms

Langkah-langkah yang dilakukan dalam algoritma *radix sort*:

1. Melakukan skema traversal pada larik A untuk mencari elemen dengan nilai terbesar yang kemudian disimpan ke variabel m.
2. Melakukan skema pengulangan pada larik A untuk mengurutkan digit paling tidak signifikan di setiap elemen hingga digit paling signifikan dengan *counting sort*.

```
// Radix Sort
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

Gambar 3.7 Potongan Algoritma Radix Sort. Sumber: www.geeksforgeeks.org/radix-sort

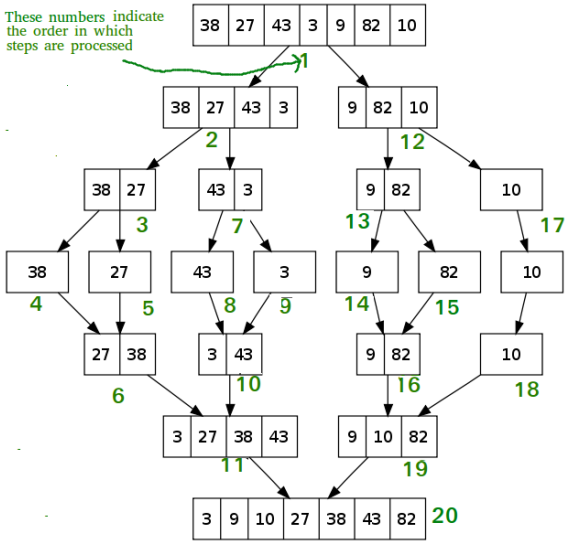
Algoritma *radix sort* memanfaatkan algoritma pengurutan lain yaitu *counting sort*. Sehingga, algoritma ini memiliki batasan-batasan yang mirip dengan algoritma *counting sort*. Algoritma *radix sort* hanya bisa mengurutkan data bilangan bulat karena algoritma ini menggunakan algoritma *counting sort* yang pada larik-lariknya hanya terdapat kunci/indeks bilangan bulat. Algoritma *radix sort* juga memiliki kompleksitas waktu yang bergantung pada rentang data yang dimiliki, misalnya k. Sehingga, kompleksitas waktu asimptotik algoritma *radix sort* pada kasus terburuk, rata-rata, dan terbaik berturut-turut adalah O(nk), θ(nk), dan Ω(nk). Kompleksitas ruang algoritma *radix sort* yaitu O(n+k).

E. Kompleksitas Algoritma Merge Sort

Algoritma *merge sort* adalah algoritma pengurutan dengan memecah data menjadi subdata dan mengurutkannya. Pengurutan subdata juga dilakukan dengan cara memecah subdata tersebut menjadi subdata yang lebih kecil lagi hingga tidak bisa dipecah. Teknik yang digunakan dalam algoritma ini adalah teknik *divide and conquer*.

Teknik *divide and conquer* pada suatu data memiliki tiga langkah utama. Pertama, data dipecah menjadi subdata dengan tipe yang sama. Kemudian, subdata yang telah dipecah akan disusun sesuai tujuan awal program. Misalnya jika kita ingin program mengurutkan data, subdata yang dipecah akan diurutkan. Terakhir, algoritma ini menggabungkan seluruh subdata menjadi data hasil olahan. Teknik *divide and conquer* sangat populer di dunia pemrograman. Beberapa algoritma yang menerapkan teknik *divide and conquer* adalah algoritma pencarian biner, *merge sort*, *quick sort*, dan *closest pair of points*.

Selain pemahaman tentang teknik *divide and conquer*, penggunaan algoritma ini juga perlu pemahaman tentang pemrograman dengan perilaku rekursif. Rekursi dalam pemrograman adalah proses pengulangan perintah dengan cara menjalankan prosedur/fungsi dalam prosedur/fungsi itu sendiri. Dalam *merge sort*, pemrograman rekursif tersebut menjalakan perintah berulang untuk memecah data hingga tidak dapat dipecah lagi kemudian memprosesnya. Kondisi data yang tidak dapat dipecah disebut kondisi basis. Setelah kondisi basis tercapai, program akan mulai memroses subdata demi subdata.



Gambar 3.8 Sketsa Algoritma Merge Sort. Sumber: https://www.geeksforgeeks.org/merge-sort

Dalam penerapan algoritma *merge sort* pada larik berukuran n, larik dibagi menjadi dua buah larik dengan ukuran lebih kecil. Apabila n bernilai genap, larik akan dibagi dua dengan ukuran yang sama. Sedangkan apabila n bernilai ganjil, larik akan dibagi dua dengan perbedaan banyak elemen maksimal sebesar satu elemen.

Langkah-langkah yang dilakukan dalam algoritma *merge sort*:

1. Memecah larik menjadi dua sublarik, kemudian membagi sublariknya berulang hingga tidak bisa dipecah lagi.
2. Membandingkan hasil pecahan satu sama lain dan menggabungkannya.
3. Menggabungkan seluruh sublarik hingga menghasilkan larik dengan ukuran sama dengan larik awal.

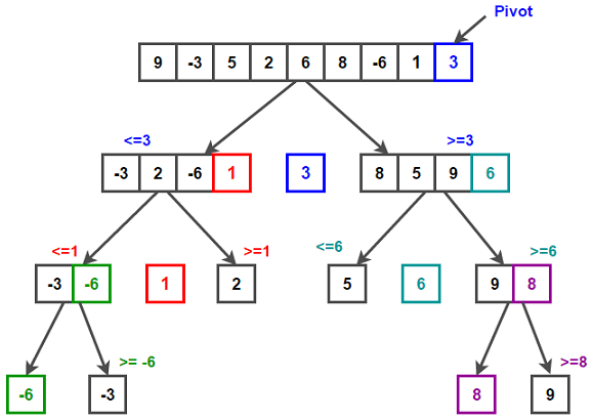
```
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}
```

Gambar 3.9 Potongan Algoritma Utama Merge Sort. Sumber: www.geeksforgeeks.org/merge-sort

Algoritma *merge sort* memiliki kecepatan eksekusi yang cukup cepat jika dibanding beberapa algoritma yang telah dibahas sebelumnya. Algoritma ini memiliki kompleksitas waktu asimptotik yang sama dengan algoritma terkait pohon biner lainnya yaitu sebesar O(n log n), θ(n log n), dan Ω(n log n). Jika dibandingkan dengan algoritma *heap sort*, algoritma ini memiliki kompleksitas waktu asimptotik yang sama. Akan tetapi, kompleksitas ruang algoritma ini berbeda dengan algoritma *heap sort* yaitu sebesar O(n).

F. Kompleksitas Algoritma Quicksort

Algoritma *quick sort* adalah algoritma pengurutan dengan konsep *divide and conquer* serta elemen pivot untuk membantu pengurutan data. Pivot adalah elemen acuan dalam meninjau langkah-langkah pengurutan. Pemilihan pivot memiliki beberapa variasi yaitu: memilih elemen pertama sebagai pivot, memilih elemen terakhir sebagai pivot, memilih elemen acak sebagai pivot, dan memilih median sebagai pivot. Pemilihan pivot harus konsisten dari awal hingga akhir program.



Gambar 3.11 Sketsa Algoritma Quick Sort dalam Bentuk Pohon. Sumber: www.techiedelight.com/quicksort

Pada makalah ini, variasi algoritma *quick sort* yang akan dibahas adalah algoritma *quick sort* dengan pemilihan pivot dari elemen terakhir.Variasi ini mengambil elemen terakhir sebagai pivot dan membandingkannya dengan elemen-elemen yang lain.

Langkah-langkah yang dilakukan dalam algoritma *merge sort* dengan variasi elemen terakhir sebagai pivot:

1. Memilih elemen terakhir sebagai pivot.
2. Melakukan skema traversal untuk menentukan elemen yang lebih besar dan lebih kecil daripada elemen pivot.
3. Mengatur susunan data dari larik sehingga elemen yang lebih kecil dari pivot terletak di sebelah kiri pivot dan elemen yang lebih besar dari pivot terletak di sebelah kanan pivot.
4. Membuat dua partisi baru yaitu partisi dengan elemen lebih kecil daripada pivot dan partisi dengan elemen lebih bisa daripada pivot. Elemen pivot berada pada posisi final.
5. Mengulangi langkah-langkah sebelumnya pada partisi-partisi yang telah dibuat secara rekursif sampai seluruh elemen larik terurut membesar.

```
// Quicksort routine
void QuickSort(int a[] ,int start, int end)
{
    // base condition
    if (start >= end)
        return;

    // rearrange the elements across pivot
    int pivot = Partition(a, start, end);

    // recurse on sub-array containing elements that are less than pivot
    QuickSort(a, start, pivot - 1);

    // recurse on sub-array containing elements that are more than pivot
    QuickSort(a, pivot + 1, end);
}
```

Gambar 3.10 Potongan Algoritma Utama Quick Sort. Sumber: www.techiedelight.com/quicksort

Algoritma *quick sort* adalah algoritma yang populer karena kecepatan eksekusi yang lebih besar dibandingkan sebagian besar algoritma pengurutan yang lain. Algoritma ini cocok untuk mengurutkan data dengan jumlah yang besar. Kompleksitas waktu asimptotik yang dimiliki algoritma pada kasus terburuk, terbaik, dan rata-rata berturut-turut sebesar $O(n^2)$, θ(n log n), dan Ω(n log n). Jika kita meninjau kasus terburuk algoritma *quick sort* dalam mengurutkan data sebanyak n, eksekusi yang dibutuhkan dapat mencapai n^2. Hal ini menyebabkan algoritma *quick sort* dapat menghabiskan waktu sama dengan algoritma *bubble sort*. Meski demikian, kasus terburuk dalam pengurutan data dengan *quick sort* jarang sekali terjadi pada praktiknya.

Untuk menyusun algoritma *quick sort*, pemrogram perlu memiliki pemahaman tentang pohon biner dan teknik *divide and conquer*. Hal ini tentu cukup sulit dipahami oleh pemula. Namun, penerapan algoritma ini membuahkan hasil yang mumpuni.

IV. ANALISIS

Setelah meninjau seluruh algoritma pengurutan yang telah dibahas, kita dapat menganilisis algoritma mana yang paling mangkus dalam mengurutkan data.

Nama Algoritma	Kompleksitas waktu		
	Terbaik	Rata-rata	Terburuk
Bubble Sort	n	n2	n2
Counting Sort	n+k	n+k	n+k
Radix Sort	nk	nk	nk
Heap Sort	n log n	n log n	n log n
Merge Sort	n log n	n log n	n log n
Quick Sort	n log n	n log n	n^2

Tabel 4.1 Tabel Kompleksitas Waktu Algoritma Pengurutan

Berdasarkan tabel 2.1 dan 4.1, algoritma dengan waktu eksekusi tersingkat dalam pengurutan data adalah algoritma dengan kompleksitas waktu asimptot pada kasus terburuk sebesar O(n+k). Algoritma tersebut ialah algoritma *counting sort*. Akan tetapi, algoritma *counting sort* memiliki batasan-batasan, yakni hanya bisa melakukan pengurutan bilangan bulat dan kompleksitas bergantung pada rentang yang digunakan. Sehingga, algoritma ini kurang efektif untuk mengurutkan data dalam jumlah besar dan tidak dapat digunakan untuk mengurutkan data selain bilangan bulat.

Jika diambil perbandingan algoritma yang bisa menangani banyak tipe data, algoritma dengan waktu eksekusi tercepat adalah algoritma dengan kompleksitas waktu asimptot pada kasus terburuk sebesar O(n log n). Algoritma tersebut adalah *heap sort* dan *merge sort*.

Nama Algoritma	Kompleksitas Ruang
Bubble Sort	O(1)
Counting Sort	O(k)
Radix Sort	O(n+k)
Heap Sort	O(1)
Merge Sort	O(n)
Quick Sort	O(log(n))

Tabel 4.2 Tabel Kompleksitas Ruang Algoritma Pengurutan

Berdasarkan tabel 4.2, algoritma dengan kompleksitas ruang paling kecil adalah algoritma dengan kompleksitas ruang sebesar O(1). Algoritma tersebut adalah *heap sort* dan *bubble sort*.

V. KESIMPULAN

Setelah dilakukan analisis kepada setiap algoritma yang telah dibahas, kita dapat menarik kesimpulan bahwa algoritma yang terbaik untuk melakukan pengurutan secara umum adalah *heap sort* kemudian diikuti dengan algoritma *merge sort*, *quick sort*, *counting sort*, *radix sort*, dan *bubble sort*. Algoritma *heap sort* konsisten dengan kompleksitas waktu sebesar O(n log n) pada setiap kasusnya dan kompleksitas ruang sebesar O(1). Algoritma *heap sort* dapat dinyatakan sebagai kompetitor terkuat dari algoritma *quick sort* yang termasuk algoritma pengurutan populer.

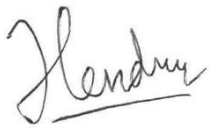
REFERENSI

[1] Donald Knuth. The Art of Computer Programming Vol 1. Fundamental Algorithms, 3rd Edition.
[2] Matematika Diskrit_ Ed. 3 - Rinaldi Munir
[3] http://www.geeksforgeeks.com. Diakses pada 9 Desember 2018, 13.23.
[4] http://bigocheatsheet.com. Diakses pada 9 Desember 2018, 13.26.
[5] https://www.hackerearth.com/practice/algorithms/sorting/heap-sort/tutorial/. Diakses pada 9 Desember 2018, 18.33.
[6] https://www.techiedelight.com/quicksort/. Diakses pada 10 Desember 2018, 01.01.
[7] https://www.studytonight.com/data-structures/ . Diakses pada 10 Desember 2018, 01.40.

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, 9 Desember 2018



Muhammad Hendry Prasetya