Analisis Kemangkusan Algoritma Pengurutan *Quicksort*

Ahmad 13512033
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13512033@std.stei.itb.ac.id

***Abstrak*— Algoritma pengurutan merupakan salah satu permasalahan yang sering dibahas oleh para ilmuwan komputer. Terdapat sangat banyak algoritma yang dapat mengatasi masalah pengurutan ini, tetapi masing-masing algoritma memberikan waktu eksekusi yang berbeda. *Quicksort* yang memiliki nilai O(n logn) ternyata memiliki waktu eksekusi yang jauh lebih cepat dari algoritma yang memiliki nilai big-O yang sama. Hal ini disebabkan algoritma pada *quicksort* merupakan algoritma yang mangkus, dipandang dari kompleksitas waktu dan kompleksitas ruang.**

***Kata Kunci*— *Quicksort*, penyususunan, kompleksitas algoritma, kompleksitas waktu.**

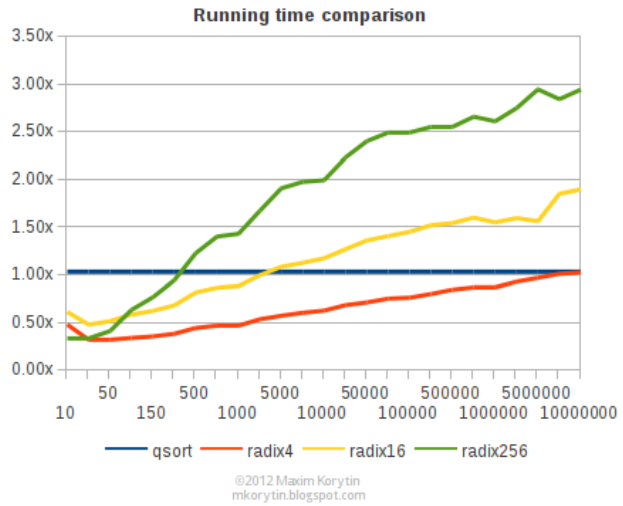
I. PENDAHULUAN

Dalam suatu basis data, terkadang pengguna membutuhkan data yang terurut berdasarkan identitas tertentu dari suatu basis data yang acak. Programmer mengatasi masalah ini dengan menggunakan algoritma pengurutan, dimana algoritma ini akan menghasilkan data yang terurut membesar ataupun terurut mengecil, tergantung dengan kebutuhan pengguna. Semakin besar basis data yang ada, semakin lama pula waktu untuk melakukan penyusunan terurut . Oleh karena itu, para programmer terus mencoba untuk menghasilkan algoritma yang mangkus, sehingga waktu yang dibutuhkan untuk melakukan penyusunan terurut makin kecil dengan tidak tergantung hanya kepada kapasitas performansi komputer yang ada. Persoalan mengenai pengurutan ini menghasilkan berbagai macam tipe pengurutan, diantaranya adalah *bubble sort*, *insertion sort*, *selection sort*, *merge sort*, *heap sort*, dan *quicksort*.

Programmer berlomba-lomba untuk menghasilkan algoritma yang lebih efektif dan efisien sehingga waktu komputasi yang diperlukan lebih sedikit. Kecepatan komputasi suatu algoritma dapat dihitung menggunakan kompleksitas waktu, dimana kompleksitas waktu akan memberikan jumlah waktu yang diperlukan suatu program saat dijalankan. Kompleksitas waktu ini bergantung dari berbagai macam bentuk algoritma yang dibuat. Semakin baik algoritma yang dimiliki, maka kompleksitas waktu atau T(n) akan semakin kecil nilainya.

Untuk membandingkan suatu algoritma, juga dapat dilihat dari kompleksitas ruangannya. Kompleksitas ruang atau S(n) diukur dari memori yang digunakan oleh strukur data yang terdapat didalam algoritma sebagai fungsi dari ukuran masukan n. Kompleksitas memori juga selain bergantung kepada kualitas memori yang dimiliki, juga bergantung kepada algoritma yang dibuat oleh para programmer. Karena saat suatu program dijalankan, komputasi dari program tersebut akan memengaruhi alokasi memori dalam komputer, sehingga semakin baik aturan alokasi maupun aturan dealokasi yang dijalankan oleh suatu program, maka nilai dari S(n) akan semakin kecil.

Alasan perhitungan kebutuhan waktu dihitung menggunakan T(n) maupun S(n) dibandingkan dengan menggunakan ukuran waktu yang sesungguhnya(detik) adalah karena setiap komputer dengan arsitektur berbeda mempunyai bahasa mesin yang berbeda, sehingga waktu setiap operasi antara satu komputer dengan komputer lain tidaklah sama. Selain itu, kompiler bahasa pemrograman yang berbeda menghasilkan kode mesin yang berbeda, yang berakibat waktu setiap operasi antara kompiler dengan kompiler lain tidak sama.



Gambar 1.1. Grafik Perbandingan Kecepatan Algoritma Pengurutan

Gambar 1.1. diatas menunjukkan sebuah grafik yang membandingkan kecepatan dari empat algoritma pengurutan yaitu *quicksort*, *radix4 sort*, *radix16 sort*, *radix256 sort*. Perbandingan algoritma pengurutan tersebut dibandingkan menggunakan banyaknya masukan atau data yang akan dipengurutan. Dari grafik diatas, dapat dilihat *quicksort*, yang dikembangkan oleh Tony Hoare pada tahun 1986, merupakan tipe pengurutan yang paling stabil kecepatannya dengan jumlah input yang turut membesar dibandingkan ketiga jenis pengurutan lainnya.

II. TEORI

2.1 Definisi Pengurutan

Algoritma pengurutan adalah suatu algoritma yang menaruh elemen dari suatu list dalam satu aturan tertentu. Pengurutan yang efektif dan mangkus sangat penting dalam mengoptimalkan kegunaan dari algoritma yang lainnya, seperti searching dan merging. Adapun data yang seringkali diproses adalah dalam bentuk larik, yang memungkinkan akses data secara random, tidak seperti list yang harus secara sekuensial.

Semenjak awal dari zaman komputasi, masalah pengurutan sudah menjadi utama yang menarik perhatian para peneliti dan para programmer. Hal ini disebabkan banyaknya cara kompleks tetapi efisien yang dapat dilakukan untuk memecahkan masalah pengurutan yang sebenarnya sangat sederhana ini.

2.2 Klasifikasi Pengurutan

Dibawah ini akan dipaparkan mengenai pengklasifikasian dari algoritma pengurutan.

2.2.1Kompleksitas Komputasional berdasarkan ukuran Data.

Secara umum, algoritma pengurutan yang baik adalah ketika memiliki batas atas atau O(n logn) , nilai n menyatakan jumlah data yang di susun secara terurut, dan algoritma pengurutan yang buruk adalah ketika memiliki algoritma dengan batas atasnya $O(n^2)$. Untuk algoritma pengurutan yang ideal adalah ketika batas atasnya O(n), tetapi algoritam pengurutan yang memiliki kompleksitas sseperti ini sangat sulit terjadi pada banyak kasus.

2.2.2 Kompleksitas Komputasional Berdasarkan Banyaknya Pertukaran Data

Pengklasifikasian dari algoritma pengurutan juga melihat dari banyaknya pertukaran yang terjadi saat melakukan pengurutan. Pertukaran yang dimaksud adalah dalam proses pengurutan suatu data, baik mengecil atau membesar, dibutuhkan proses penukaran posisi elemen dari posisi yang acak menjadi yang terurut. Semakin banyak pertukaran yang dibutuhkan maka semakin lama pengurutan yang terjadi.

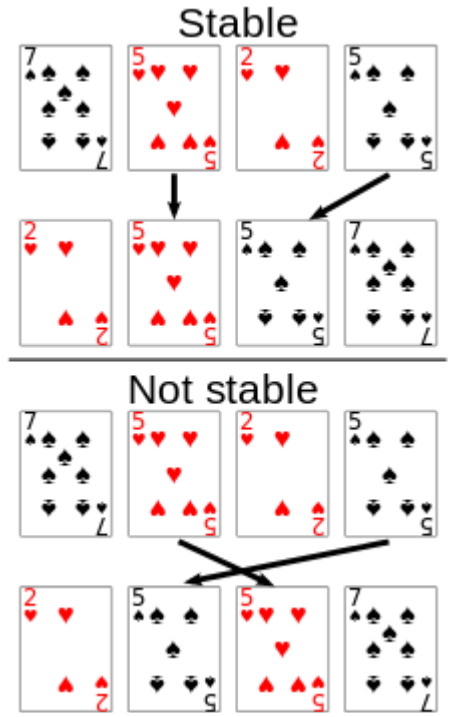
2.2.3 Penggunaan Memori

Penggunaan memori dalam proses penyusunan suatu data melibatkan proses perpindahan data dari masing-masing alamat memori ke alamat memori yang lain. Saat proses penyusunan dilakukan, komputer akan menyusuri alamat-alamat pada memori dan melakukan penusunan data sesuai dengan perintah yang diberikan oleh algoritma. Algoritma penyusunan yang baik akan mengakibatkan penggunaan memori yang seminimal mungkin. Sehingga komputer tidak perlu membutuhkan waktu yang lama dalam melakukan akses ke masing-masing alamat pada data yang akan disusun. Algoritma yang baik, berpengaruh terhadap lokalisasi memori sehingga pengaksesan data akan lebih cepat apabila lokalisasi yang diterapkan oleh algoritma merupakan lokalisasi yang baik.

2.2.4 Rekursi

Beberapa algoritma penyusunan ada yang merupakan algoritma rekursi dan non-rekursi. Bahkan ada algoritma penyusunan yang juga merupakan algoritma rekursi dan non-rekursi, yaitu merge sort.

2.2.5 Kestabilan



Gambar 2.1 Perbandingan Pengurutan Stabil dengan Pengurutan Tidak Stabil

Gambar 2.1 adalah gambar yang mengilustrasikan antara penyusunan yang stabil dan penyusunan yang tidak stabil. Terdapat empat buah kartu yang ingin disusun berdasarkan nilai yang ada pada kartu dengan urutan penyusunan membesar. Pada kasus tersebut, terdapat dua buah kartu yang memiliki nilai yang

sama, yaitu kartu dengan nilai lima yang terdapat pada lima hati dan lima sekop. Algoritma yang tidak stabil akan menghasilkan hasil penyusunan dengan kartu lima sekop akan diletakkan di urutan ke dua, sedangkan kartu lima hati diletakkan di urutan ke tiga. Algoritma yang stabil menghasilkan penyusunan kartu dengan nilai lima hati tetap berada di urutan kedua dan kartu lima sekop berada di urutan ke tiga. Algoritma penyusunan yang baik adalah algoritma yang menghasilkan jumlah pertukaran yang paling sedikit, atau dengan kata lain data yang pada awalnya berada di urutan ke-i, dimana nilai i menyatakn indeks urutan data, dan pada akhir penysunan berada di urutan ke-i, maka algoritma tidak akan menukar posisinya sama sekali saat proses pertukaran data dalam penyusunan data dilakukan. Algoritma yang demikian dinamakan algoritma yang stabil.

2.3 Jenis-jenis Pengurutan

Dibawah ini akan dipaparkan mengenai berbagai jenis pengurutan yang sering digunakan dalam kehidupan sehari-hari.

2.3.1 *Insertion sort*

Insertion sort adalah algoritma penyusunan yang simpel dan cukup efisien untuk ukuran data yang relafi kecil. *Insertion sort* sendiri biasa digunakan sebagai bagian dari algoritma yang cukup kompleks. Cara kerja dari *insertion sort* adalah dengan mengambil elemen dari data satu persatu, dan meletakannya di posisi yang tepat pada data baru yang telah terurut. Dibawah ini merupakan implementasi dari algoritma *insertion sort* dalam bahasa C.

```
void InsertionSort()
{
int i, j, x;
for(i=1; i<Max; i++){
x = Data[i];
j = i - 1;
while (x < Data[j]){
Data[j+1] = Data[j];
j--;
}
Data[j+1] = x;
                }
}
```

2.3.2 *Selection sort*

Metode seleksi atau *selection sort* melakukan pengurutan dengan cara mencari data yang terkecil, kemudian menukarkan data terkecil tersebut dengan data yang digunakan sebagai acuan(pivot). Proses pengurutan dengan metode seleksi dimulai dengan mencari data terkecil dari data pertama sampai dengan data terakhir. Kemudian data terkecil tersebut ditukar dengan data pertama. Kemudian mulai dicari data terkecil dari data kedua sampai data terakhir, data terkecil kedua ditukar dengan data kedua. Proses ini dilakukan sampai semua elemen telah dalam keadaan terurut. Di bawah ini diimplementasikan *selection sort* dalam bahasa C.

```
void SelectionSort()
{
int i, j, k;
for(i=0; i<Max-1;i++){
     k = i;
    for (j=i+1; j<Max; j++)
         if(Data[k] > Data[j])
               k = j;
        Tukar(&Data[i], &Data[k]);
         }
}
```

2.3.3 *Bubble sort*

Metode gelembung (*bubble sort)* adalah metode pengurutan data dengan cara memmbandingkan masing-masing elemen, kemudin melakukan penukaran bila diperlukan. Metode ini merupakan metode yang pada umumnya paling mudah untuk dipahami dan diimplementasikan pada suatu program, tetapi metode ini merupakana metode pengurutan yang paling tidak efisien.

Metode *bubble sort* menggunakan dua kali pengulangan. Pengulangan pertama yaitu melakukan pengulangan dari elemen ke 2 sampai dengan elemen ke N-1(misal variabel i) . Pengulangan kedua yaitu pengulangan menurun dari elemen ke N sampai elemen ke i(misal variabel j). Pada tiap-tiap pengulangan, elemen ke j-1 dibandingkan dengan elemen ke –j, jika data ke j-1 lebih besar dibandingkan data ke-j, maka dilakukan proses penukaran. Dibawah ini diimplementasikan bubble sort dalam bahasa C.

```
void BubbleSort()
{
     int i, j;
    for(i=1; i<Max-1; i++)
       for(j=Max-1; j>=i; j--)
          if(Data[j-1] > Data[j])
             Tukar(&Data[j-1], &Data[j]);
}
```

2.3.4 *Merge sort*

Merge sort biasa diimplementasikan dalam pengurutan data dari sekumpulan banyak data. Prinsip dari merge sort adalah mula-mulai terdapat dua kumpulan data yang sudah berada dalam keadaan terurut. Kedua kumpulan daa

tersebut akan digabungkan sehingga menjadi satu data baru dengan keadaan yang juga terurut. Proses penggabungan dimulai dengan mula-mulai mengambil data pertama dari data satu dan mengambil data pertama dari data dua. Kedua data pertama ini dibandingkan dan yang memiliki nilai yang lebih kecil akan diletakkan sebagai data pertama pada data gabungan. Kemudian data pertama yang lebih besar akan dibandingkan dengan data kedua dari data pertama yang lebih kecil. Data yang lebih kecil akan diletakkan sebagai data kedua pada data hasil gabungan. Proses ini dilakukan sampai semua data yang berada di data satu dan data dua telah diletakkan di data hasil yang baru secara terurut. Di bawah ini merupakan implementasi dari merge sort pada bahasa C.

```
void MergeSort(int T1[] ,int T2[], int J1, int J2,
int T3[], int *J3)
{
int i=0, j=0;
int t=0;
while ((i<J1)||(j<J2)){
if(T1[i]<T2[j]){
  T3[t] = T1[i];
i++;
}
else{
T3[t] = T2[j];
j++;
        }
        t++;
}

if(i>J1)
       for(i=j; i<J2; i++){
T3[t] = T2[i];
t++;
}

if(j>J2)
       for(j=i; j<J1; j++){
T3[t] = T1[j];
t++;
          }
*J3 = t;
}
```

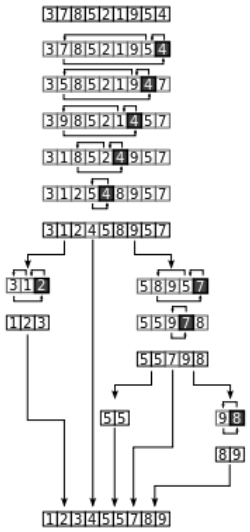
2.3.5 Radix sort

Algoritma penyusunan radix aldalah algoritma yang mengurutkan sekumpulan angka dengan cara memproses masing-masing digit dari angka tersebut. Misalkan terdapat n buah angka dengan masing-masing angka memilik k buah digit, maka sekumpulan angka tersebut akan dapat disusun selama O(n.k). Penyusunan secara radix ini akan memproses tiap digit dari masing-masing angka bisa berdasarkan nilai digit terkecil (LSD) atau bisa juga berdasarkan nilai digit terbesar(MSD).
Untuk penyusunan radix berdasarkan digit terkecil(LSD), algoritma ini pertama kali akan mengurutkan data berdasarkan digit terkecil, kemudian akan disusun diurutkan lagi berdasarkan digit setelahnya, proses ini akan dilakukan terus menerus sampai digit terbesar telah diurutkan. Dibawah ini diimplementasikan algoritma penyusunan radix dalam bahasa C.

```
  radix_sort(int arr[], int n)
{
 int bucket[10][5],buck[10],b[10];
 int i,j,k,l,num,div,large,passes;

 div=1;
 num=0;
 large=arr[0];

 for(i=0 ; i<n ; i++)
 {
  if(arr[i] > large)
   {
    large = arr[i];
   }

 while(large > 0)
 {
  num++;
  large = large/10;
 }

  for(passes=0 ; passes<num ; passes++)
 {
  for(k=0 ; k<10 ; k++)
  {
   buck[k] = 0;
  }
  for(i=0 ; i<n  ;i++)
  {
   l = ((arr[i]/div)%10);
   bucket[l][buck[l]++] = arr[i];
  }

  i=0;
  for(k=0 ; k<10 ; k++)
  {
   for(j=0 ; j<buck[k] ; j++)
   {
    arr[i++] = bucket[k][j];
   }
  }
  div*=10;
 }
}
}
```

2.4 *Quicksort*

Algoritma penyusunan *quick* atau biasa disebut dengan algoritma pertukaran partisi adalah algortima penyusunan yang dikembangkan oleh Tony Hoare. Algoritma ini memiliki O(n log n) untuk menyusun n buah data. Pada kasus terburuknya algoritma ini bisa mencapai $O(n^2)$, dimana kasus seperti ini sulit untuk terjadi. *Quicksort* seringkali memiliki waktu komputasi yang lebih cepat dibandingkan jenis algoritma penyusunan yang memiliki nilai O(n logn) lainnya. Selain itu, lokalisasi memori yang diakibatkan oleh algoritma *quicksort* juga menghasilkan referensi data yang bekerja dengan baik pada *cache*.



Gambar 2.2 Ilustrasi Algoritma *Quicksort*

Quicksort pada awalnya membagi data yang besar menjadi dua subdata, dimana terdapat subdata dengan nilai elemen yang kecil dan subdata dengan nilai elemen yang besar. Kemudian *quicksort* secara rekursif akan menyusun subdata tersebut. Langkah dari *quicksort* secara bertahap adalah, pertama tentukan element sebagai pembanding(pivot) dari data besar. Susun ulang data tersebut sehingga semua elemen yang bernilai lebih kecil dari pivot terletak sebelum pivot pada data yang sedang diurutkan, dan semua data yang lebih besar dari pivot diletakan setelah pivot pada data yang sedang diurutkan. Tahap ini dinamakan dengan tahap partisi. Setelah tahap partisi untuk pivot telah selesai, secara rekursif ulangi tahap untuk sub-data dengan nilai lebih kecil dan sub-data dengan nilai lebih besar. Pada proses rekursi ini basis dari rekursi adalah saat ukuran data adalah nol, dimana data tidak perlu disusun sama sekali. Dibawah ini adalah impelementasi dari algoritma *quicksort* dalam bahasa c.

```
void quicksort(int x[10],int first,int last){
   int pivot,j,temp,i;

   if(first<last){
      pivot=first;
      i=first;
      j=last;
      while(i<j){
         while(x[i]<=x[pivot]&&i<last)
            i++;
         while(x[j]>x[pivot])
            j--;
         if(i<j){
            temp=x[i];
             x[i]=x[j];
             x[j]=temp;
         }
      }

      temp=x[pivot];
      x[pivot]=x[j];
      x[j]=temp;
      quicksort(x,first,j-1);
      quicksort(x,j+1,last);
   }
}
```

2.5 Kompleksitas Algoritma

Dalam pembuatan suatu algoritma, selain tujuan dari algoritma harus dicapai, para programmer juga mengutamakan kemangkusan dari algoritma itu sendiri. Adapun kemangkusan dari algoritma dapat diukur berdasarkan jumlah waktu dn ruang memori yang dibutuhkan saat program hasil algoritma dijalankan. Kemangkusan berarti minimnya kebutuhan akan waktu dan ruang memori.

Kebutuhan waktu dan ruang suatu algoritma juga bergantu pada ukuran data yang akan diproses. Kemangkusan dibutuhkan karena dalam suatu algoritma dengan algoritma yang lain memiliki kebutuhan waktu yang berbeda-beda. Apabila suatu algoritma merupakan algoritma yang mangkus, maka algoritma tersebut dapat dijalankan oleh komputer secara cepat.

2.5 Kompleksitas Waktu

Kompleksitas waktu adalah suatu ukuran yang menghitung jumla dari waktu yang dibutuhkan untuk mengeksekusi suatu algoritma. Untuk kebanyakan algoritma, membandingkan kompleksitas waktu lebih menarik dibandingkan membandingkan komplekitas ruang.

Dalam menganalissi kompleksitas waktu faktor-faktor yang tidak dapat mempengaruhi nilai kompleksitas waktu antara lain, bahasa pemrograman yang dipilih untuk melaksanakan suatu algoritma, kualitas dari kompiler, kecepatan dari komputer dimana algoritma itu dijalankan. Adapun tujuan dari analisis kompleksitas waktu adalah menentukan kemangkusan suatu algoritma dengan mengestimasi nilai dari batas atas (big-o) dari jumlah waktu yang dibutuhkan pada suatu algoritma. Tujuan lainnya yaitu membandingkan berbagai macam algoritma sebelum menentukan algoritma mana yang lebih baik untuk dijalankan.

Analisis pada kompleksitas waktu bergantung pada

jumlah proses khusus yang dilaksanakan pada suatu algoritma. Kompleksitas waktu menggambarkan hubungan antara ukuran data yang diproses dengan waktu berjalannya suatu algoritma. Analisis perhitungan kompleksitas waktu ini dapat disederhanakan dengan memperhitungkan banyaknya operasi aritmatika yang dilakukan, banyaknya operasi perbandingan, banyaknya pengulangan, banyaknya jumlah larik yang diakses, dsbg.

Name	Best	Average	Worst	Memory	Stable
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends	Yes
In-place Merge sort	—	—	$n (\log n)^2$	1	Yes
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Insertion sort	n	n^2	n^2	1	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No
Selection sort	n^2	n^2	n^2	1	Depends
Timsort	n	$n \log n$	$n \log n$	n	Yes
Shell sort	n	$n(\log n)^2$	$O(n \log^2 n)$	1	No
Bubble sort	n	n^2	n^2	1	Yes
Binary tree sort	n	$n \log n$	$n \log n$	n	Yes
Cycle sort	—	n^2	n^2	1	No
Library sort	—	$n \log n$	n^2	n	Yes
Patience sorting	—	—	$n \log n$	n	No
Smoothsort	n	$n \log n$	$n \log n$	1	No
Strand sort	n	n^2	n^2	n	Yes
Tournament sort	—	$n \log n$	$n \log n$		
Cocktail sort	n	n^2	n^2	1	Yes
Comb sort	—	—	n^2	1	No
Gnome sort	n	n^2	n^2	1	Yes
Bogosort	n	$n \cdot n!$	$n \cdot n! \to \infty$	1	No

Gambar 2.2 Tabel Kompleksitas Waktu dan Kompleksitas Ruang pada Algoritma Pengurutan

Gambar 2.2 memberikan informasi mengenai berbagai macam pengurutan dengan kemungkinan waktu terbaik, kemungkinan waktu terburuk, rata-rata waktu yang dibutuhkan, ukuran memori yang dibutuhkan, serta kestabilan dari masing-masing algortima pengurutan.

III. ANALISIS TERHADAP ALGORITMA *QUICKSORT*

Gambar 3.1 Grafik Ukuran Larik Terhadap Waktu Pada Algoritma Penyusunan

Grafik yang diberikan pada Gambar 3.1 menunjukkan waktu dari eksekusi yang dibutuhkan dari ketiga algoritma yaitu *quicksort*, *heap sort*, *merge sort* terhadap ukuran dari larik. Pada gambar 2.2 diketahui ketiga jenis pengurutan ini memiliki nilai big-o yang sama yaitu O(n logn). Walaupun ketiga jenis pengurutan ini memiliki nilai big-o yang sama tetapi *quicksort* menunjukkan peningkatan waktu yang jauh lebih rendah dibandingkan kedua pengurutan lainnya yang memiliki nilai big-o yang sama. Apakah yang menyebabkan hal ini dapat terjadi?

Secara singkat, *quicksort* diimplementasikan seperti pengurutan yang memiliki dua pointer yang saling menunjuk data yang berbeda, sehingga waktu yang diperlukan untuk menjalankan *quicksort* jauh lebih singkat dibandingkan jenis pengurutan lainnnya. Don Knuth menghitung rata-rata waktu yang diperlukan dari berbagai tipe algortima secara eksak dari komputer buatan Don Knuth pada bukunya "The Art of Computer Programming". Hasil dari penelitiannya memberikan data sebagai berikut:

a. *Quicksort*: $11.667(n+1)\ln(n) - 1.74n - 18.74$
b. *Merge sort*: $12.5n \ln(n)$
c. *Heap sort*: $16n\ln(n) + 0.01n$
d. *Insertion sort*: $2.25n^2 + 7.75n - 3\ln(n)$

Analisis atas *quicksort* dapat dilihat dari melihat jumlah operasi abstrak yang dilakukan pada algoritma ini. Dalam buku "Algorithms" karya Robert Sedgewick, dipaparkan pendekatan perhitungan akan jumlah pertukaran dan perbandingan elemen pada kasus pengurutan. Hasil yang diberikan oleh Robert Sedgewick adalah sebagai berikut:

a. *Quicksort*: Jumlah perbandingan $2n \ln(n)$
Jumlah pertukaran $0.33n \ln(n)$
b. *Merge sort*: Jumlah perbandingan $1.44n \ln(n)$
Jumlah pengaksesan larik $8.66n \ln(n)$
c. *Insertion sort*: Jumlah perbandingan $0.25n^2$
Jumlah pertukaran $0.25n^2$

Dengan bukti perhitungan yang dipaparkan diatas, dapat dilihat ternyata jumlah T(n) dari *quicksort* masih jauh lebih kecil dibandingkan T(n) pada algoritma pengurutan lainnya.

Hal lain yang menjelaskan mengapa *quicksort* pada banyak kondisi jauh lebih cepat dari jenis algoritma penyusunan yang lainnya adalah karena algoritma pada *quicksort* jauh lebih mangkus pada kerja *cache*. Waktu dari berjalannya pengurutan ini sesungguhnya $O(n/B \log(n/B))$, dimana B menyatakan ukuran blok memori. Dibandingkan dengan algoritma pengurutan lain, misalnya heapsort, kemangkusan *cache* yang diberikan oleh *quicksort* jauh lebih baik.

Waktu yang dibutuhkan pada pelaksanaan pengurutan yang dilakukan oleh *quicksort* jauh lebih cepat juga disebabkan oleh proses pengulangan yang diberikan oleh algoritmanya. Pengulangan yang diberkan oleh algoritma ini dapat dengan mudah dan efisien diimplementasikan di berbagai macam rancangan komputer dan juga data asli yang biasa dihadapi dalam kehidupan sehari-hari. Adapun, *quicksort* cenderung menghasilkan penggunaan hirarki memori yang sangat baik. Secara maksimal mengambil keuntunganyang diberikan dari memori virtual serta *cache* yang tersedia.

Walaupun dapat dibilang *quicksort* merupakan

algoritma dengan tingkat eksekusi waktu yang paling cepat, tetapi *quicksort* masih memiliki kekurangan. Kekurangan yang dimiliki oleh *quicksort* adalah, di kasus terburuk yang mungkin terjadi, performansi yang diberikan dapat setingkat dengan performansi dari algoritma bubblesort.

IV. KESIMPULAN

Quicksort merupakan algoritma pengurutan dengan O(n logn) yang memberikan eksekusi waktu tercepat dibandingkan algoritma pengurutan lainnya dengan nilai big-O yang sama. Hal ini disebabkan algoritma yang mangkus sehingga lokalisasi dan hirarki memori yang berjalan dengan baik, sehingga eksekusi berjalan dengan cepat. Perbandingan waktu terhadap ukuran data yang diberikan oleh *quicksort* menawarkan algoritma pengurutan ini sebagai pilihan utama dalam mengatasi database yang berukuran sangat besar.

V. UCAPAN TERIMA KASIH

Puji syukur penulis panjatkan kepada Tuhan YME karena atas izin-Nya makalah ini dapat selesai. Penulis mengucapkan terima kasih kepada bapak Rinaldi Munir atas bimbingannya selaku dosen matematika diskrit, serta kepada ilmuwan ilmu komputer yang telah memberikan kerja keras terbaiknya sehingga karya yang mereka ciptakan sangat membantu dalam menyelesaikan makalah ini.

REFERENCES

[1] Demuth, H. Electronic Data Pengurutan. PhD thesis, Stanford University, 1956.

[2] Sedgewick, R. (1978). "Implementing *Quicksort* programs". *Comm. ACM*.

[3] D. E. Knuth, *The Art of Computer Programming*, *Volume 3: Pengurutan and Searching*.

[4] A. LaMarca and R. E. Ladner. "The Influence of *Cache*s on the Performance of Pengurutan." Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 1997.

[5] Munir,Rinaldi. *Diktat Kuliah IF2091 Matematika Diskrit.* Bandung: Penerbit Informatika. 2008.

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, 15 Desember 2013

Ahmad 13512033