Kompleksitas Algoritma *Sorting* yang Populer Dipakai

Wahyu Fahmy Wisudawan [1)]
1) Program Studi Teknik Informatika ITB, Bandung 40135,
email: mailto:al_izzatusysyifa@students.itb.ac.id

Abstract – *Makalah ini secara khusus membahas kompleksitas lima algoritma sorting yang populer dipakai di dunia informatika. Lima algoritma tersebut adalah Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, dan Quick Sort. Pertama-tama, makalah ini akan membahas definisi dari algoritma, sorting, dan istilah lain yang digunakan di dalam makalah. Setelah itu, baru menginjak pembahasan algoritma sorting. Hal-hal yang dibahas di antaranya adalah konsep algoritma, ide dasar, simulasi, dan kompleksitasnya. Kemudian digambarkan grafik yang menunjukkan seberapa besar peningkatan waktu yang dibutuhkan terhadap panjang tabel yang diurutkan.*

Kata Kunci*: algoritma sorting, kompleksitas algoritma, bubble sort, selection sort, insertion sort, merge sort, quick sort, big O notation, pseudocode, efisiensi.*

1. PENDAHULUAN

1.1. Definisi Algoritma

Algoritma adalah kumpulan langkah sistematis untuk memperoleh hasil yang diinginkan[1]. Sebelum sebuah algoritma dijalankan, biasanya ada suatu kondisi awal (*initial state*) yang harus dipenuhi. Kemudian, langkah-langkah ini diproses hingga mencapai suatu kondisi akhir (*final state*).

Salah satu contoh dari algoritma adalah *Sorting* (pengurutan).

1.2 Definisi Sorting

Sorting didefinisikan sebagai pengurutan sejumlah data berdasarkan nilai kunci tertentu. Pengurutan dapat dilakukan dari nilai terkecil ke nilai terbesar (*ascending*) atau sebaliknya (*descending*).

Algoritma *Sorting* termasuk salah satu contoh yang kaya akan solusi. Dalam makalah ini, hanya akan dibahas lima algoritma *sorting* yang populer dipakai di dunia informatika. Lima algoritma tersebut adalah:

1. *Bubble Sort*
2. *Selection Sort*
3. *Insertion Sort*
4. *Merge Sort*, dan
5. *Quick Sort*.

Masing-masing algoritma memiliki kelebihan dan kekurangannya tersendiri. Karena itulah ada kecenderungan algoritma yang satu lebih disukai dan lebih sering dipakai daripada algoritma yang lain.

Hal-hal yang menyebabkan suatu algoritma sering digunakan adalah kestabilan, kesesuaian dengan kebutuhan, kesesuaian dengan struktur data yang dipakai, kenaturalan, dan kemangkusan (*efficiency*). Ukuran untuk menyatakan kemangkusan algoritma tersebut dapat dinyatakan dengan kompleksitas algoritma.

1.3. Kompleksitas Algoritma

Kompleksitas dari suatu algoritma merupakan ukuran seberapa banyak komputasi yang dibutuhkan algoritma tersebut untuk mendapatkan hasil yang diinginkan. Algoritma yang dapat memperoleh hasil yang diinginkan dalam waktu yang singkat memiliki kompleksitas yang rendah, sementara algoritma yang membutuhkan waktu yang lama untuk memperoleh hasil tersebut mempunyai kompleksitas yang tinggi. Biasanya kompleksitas algoritma dinyatakan secara *asimptotik* dengan notasi big-O.

Jika kompleksitas waktu untuk menjalankan suatu algoritma dinyatakan dengan $T(n)$, dan memenuhi

$$T(n) \leq C(f(n))$$

untuk $n \geq n_0$
maka kompleksitas dapat dinyatakan dengan

$$T(n) = O(f(n)).$$

2. PEMBAHASAN

2.1. Bubble Sort

2.1.1. Konsep Bubble Sort

Bubble Sort merupakan cara pengurutan yang sederhana. Konsep dari ide dasarnya adalah seperti "gelembung air" untuk elemen struktur data yang semestinya berada pada posisi awal. Cara kerjanya adalah dengan berulang-ulang melakukan traversal (proses *looping*) terhadap elemen-elemen struktur data yang belum diurutkan. Di dalam traversal tersebut, nilai dari dua elemen struktur data dibandingkan. Jika ternyata urutannya tidak sesuai dengan "pesanan", maka dilakukan pertukaran (*swap*). Algoritma sorting ini disebut juga dengan *comparison sort* dikarenakan hanya mengandalkan perbandingan nilai elemen untuk mengoperasikan elemennya.

[1] Rangkuman dari dua sumber: Munir, Rinaldi. (2006). Matematika Diskrit. Teknik Informatika ITB. dan Paul E. Black, di *Dictionary of Algorithms and Data Structures* [online], U.S. *National Institute of Standards and Technology*. Tersedia dari: http://www.nist.gov/dads/HTML/algorithm.html

2.1.2. Simulasi Bubble Sort

Berikut ini adalah contoh simulasi algoritma sorting terhadap larik (*array*) dengan panjang 5 elemen.

Misalnya untuk larik dengan elemen

7	3	4	1	6

akan diurutkan dengan algoritma *bubble sort* secara terurut menaik (*ascending*).

Proses pembandingan di dalam setiap *looping* digambarkan dengan warna kuning. Jika ternyata diperlukan pertukaran nilai, maka akan ditulis "tukar" di bawah elemen larik yang dibandingkan tersebut. Hasil penukaran digambarkan di tabel simulasi berikutnya.
Untuk membedakan dengan elemen tabel yang lain, bagian tabel yang telah diurutkan digambarkan dengan warna biru muda.

Perhatikan bahwa nilai '1' yang merupakan nilai terkecil di dalam larik seolah-olah mengapung mengikuti proses *looping*. Pengapungan ini terus berlangsung hingga menemukan elemen tabel yang nilainya lebih kecil lagi dari elemen tabel tersebut. Inilah yang dimaksud dengan "efek gelembung" di dalam *bubble sort.*

Looping pertama:

7	3	4	1	6

7	3	4	1	6

tukar

7	3	1	4	6

tukar

7	1	3	4	6

tukar

1	7	3	4	6

Looping kedua:

1	7	3	4	6

1	7	3	4	6

1	7	3	4	6

tukar

1	3	7	4	6

Looping ketiga:

1	3	7	4	6

1	3	7	4	6

tukar

1	3	4	7	6

Looping keempat:

1	3	4	7	6

tukar

1	3	4	6	7

2.1.3. Algoritma Bubble Sort

Algoritma *bubble sort* dapat diringkas sebagai berikut, jika N adalah panjang elemen struktur data, dengan elemen-elemennya adalah T_1, T_2, T_3, …, T_{N-1}, T_N, maka:

1. Lakukan *traversal* untuk membandingkan dua elemen berdekatan. Traversal ini dilakukan dari belakang.
2. Jika elemen pada $T_{N-1} > T_N$, maka lakukan pertukaran (*swap*). Jika tidak, lanjutkan ke proses *traversal* berikutnya sampai bertemu dengan bagian struktur data yang telah diurutkan.
3. Ulangi langkah di atas untuk struktur data yang tersisa.

Pseudocode dari algoritma ini untuk struktur data larik (*array*) *one-base* (struktur data yang memiliki indeks dasar satu) dapat ditulis sebagai berikut:

```
Procedure BubbleSort (Input/Output
T: TabInt, Input N: integer)
{mengurut tabel integer [1 .. N]
dengan Bubble Sort}

Kamus:
  i: integer
  Pass: integer
  Temp: integer

Algoritma:
  Pass traversal [1..N-1]
    i traversal [N..Pass+1]
      if (Ti < Ti-1) then
        Temp ← Ti
        Ti ← Ti-1
        Ti-1 ← Temp
    {T[1..Pass] terurut}
```

Versi algoritma ini adalah algoritma *bubble sort* yang paling alami dan sederhana.

2.1.4. Kompleksitas Bubble Sort

Algoritma di dalam *bubble sort* terdiri dari 2 kalang (*loop*) bertingkat. Kalang pertama berlangsung selama *N*-1 kali. Indeks untuk kalang pertama adalah *Pass*. Kemudian kalang tingkat kedua berlangsung dari *N* sampai dengan *Pass*+1.

Dengan demikian, proses *compare* yang terjadi sebanyak:

$$T(n) = (N-1) + (N-2) + \ldots + 2 + 1$$
$$= \sum_{i=1}^{N-1} N - i = \frac{N(N-1)}{2}$$
$$T(n) = \frac{N(N-1)}{2} = O(n^2)$$

$T(n)$ ini merupakan kompleksitas untuk kasus terbaik ataupun terburuk. Hanya saja pada kasus terbaik, yaitu pada kasus jika struktur data telah terurut sesuai perintah, proses *Pass* terjadi tanpa adanya *assignment*.

Hal ini jelas menunjukkan bahwa algoritma akan berjalan lebih cepat. Hanya saja tidak terlalu signifikan.

Menurut Wikipedia[2], algoritma *bubble sort,* jika dikembangkan, kompleksitasnya mencapai $\Omega(n)$. Hal ini hanya terjadi jika memanfaatkan tipe data *boolean* untuk menyatakan bahwa proses penukaran tak perlu dilakukan lagi. Dengan demikian, kompleksitas $\Omega(n)$ tidak berlaku bagi *pseudocode* di atas. Untuk itu perlu dilakukan perubahan. Perubahan yang terjadi pada *pseudocode*-nya dapat diperhatikan sebagai berikut:

```
Procedure BubbleSort (Input/Output
T: TabInt, Input N: integer)
{mengurut tabel integer [1 .. N]
dengan Bubble Sort dengan memanfaat-
kan boolean}

Kamus:
  i: integer
  Pass: integer
  Temp: integer
  Tukar: boolean

Algoritma:
  Pass ← 1
  Tukar ← true
  while (Pass ≤ N-1) and (Tukar) do
    Tukar ← false
    i traversal [N..Pass+1]
      if (Ti < Ti-1) then
        Temp ← Ti
        Ti ← Ti-1
        Ti-1 ← Temp
        Tukar ← true
  {T[1..Pass] terurut}
```

Dengan demikian ketika *Pass* terjadi sekali melalui semua bagian struktur data, kemudian ditemukan bahwa tidak diperlukan lagi proses penukaran elemen, maka proses *pass* akan berhenti. *Bubble sort* hanya melakukan N-1 perbandingan di dalam algoritmanya dan menetapkan bahwa struktur data telah terurut.

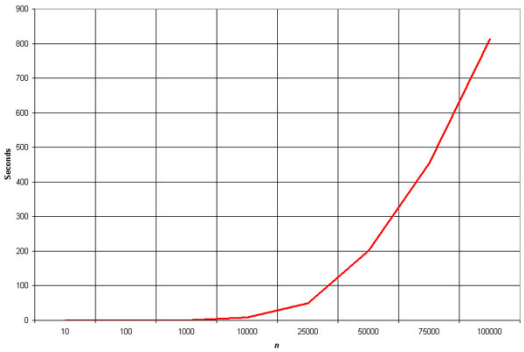
Dengan demikian,

$$T_{\min}(n) = N = O(n)$$

Sementara $T_{\max}(n)$ nya tetap sama.

Bubble sort tergolong algoritma yang paling tidak efisien di antara algoritma *sorting* dengan kompleksitas $O(n^2)$.

Berikut ini adalah data analisis empiris efisiensi *Bubble Sort*.



Gambar 1. Grafik Efisiensi *Bubble Sort*

Berdasarkan tabel, tidak ada perbedaan performansi secara signifikan untuk pengurutan terhadap 100 item atau kurang. Namun, *bubble sort* tidak disarankan untuk pengurutan yang terus berulang, atau pengurutan yang menangani lebih dari 200 item.

2.2. Selection Sort

2.2.1. Konsep Selection Sort

Algoritma *sorting* sederhana yang lain adalah *Selection Sort*. Ide dasarnya adalah melakukan beberapa kali *pass* untuk melakukan penyeleksian elemen struktur data. Untuk *sorting ascending* (menaik), elemen yang paling kecil di antara elemen-elemen yang belum urut, disimpan indeksnya, kemudian dilakukan pertukaran nilai elemen dengan indeks yang disimpan tersebut dengan elemen yang paling depan yang belum urut. Sebaliknya, untuk *sorting descending* (menurun), elemen yang paling

[2] Wikipedia, *the free encyclopedia*, dengan alamat: en.wikipedia.org

besar yang disimpan indeksnya kemudian ditukar.

2.2.2. Simulasi Selection Sort

Untuk lebih jelasnya, perhatikanlah simulasi *Selection Sort Ascending* berikut dengan menggunakan larik

33	41	4	16	21	9	13

Dalam satu kali *pass,* ditentukan elemen yang paling kecil di dalam bagian list yang belum urut. Elemen yang paling kecil ini, diwarnai merah muda. Untuk bagian larik yang telah diurutkan diberi warna biru muda.

1	2	3	4	5	6	7
33	41	4	16	21	9	13
4	41	33	16	21	9	13
4	9	33	16	21	41	13
4	9	13	16	21	41	33
4	9	13	16	21	41	33
4	9	13	16	21	41	33
4	9	13	16	21	33	41

2.2.3. Algoritma Selection Sort

Algoritma *selection sort* dapat dirangkum sebagai berikut:

1 Temukan nilai yang paling minimum (atau sesuai keinginan) di dalam struktur data. Jika *ascending,* maka yang harus ditemukan adalah nilai yang paling minimum. Jika *descending,* maka temukan nilai yang paling maksimum.

2 Tukar nilai tersebut dengan nilai pada posisi pertama di bagian struktur data yang belum diurutkan.

3. Ulangi langkah di atas untuk bagian struktur data yang tersisa.

Pseudocode untuk algoritma *ascending selection sort* untuk struktur data larik basis satu dapat ditulis sebagai berikut:

```
Procedure SelectionSort
(Input/Output T: TabInt, Input N:
integer)
{mengurut tabel integer [1 .. N]
dengan Selection Sort secara
ascending}

Kamus:
  i: integer
  Pass: integer
  min: integer
  Temp: integer

Algoritma:
  Pass traversal [1..N-1]
    min ← Pass
    i traversal [Pass+1..N]
      if (Ti < Tmin) then
        min ← j
    Temp ← TPass
    TPass ← Tmin
    Tmin ← Temp
    {T[1..Pass] terurut}
```

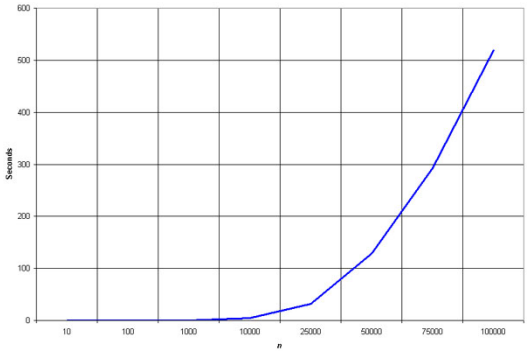
2.2.4. Kompleksitas *Selection Sort*

Algoritma di dalam *Selection Sort* terdiri dari kalang bersarang. Dimana kalang tingkat pertama (disebut *pass*) berlangsung N-1 kali. Di dalam kalang kedua, dicari elemen dengan nilai terkecil. Jika didapat, indeks yang didapat ditimpakan ke variabel `min`. Lalu dilakukan proses penukaran. Begitu seterusnya untuk setiap *Pass*. Berdasarkan operasi perbandingan elemennya:

$$T(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \sum_{i=1}^{n-1} n - i$$

$$= \frac{n(n-1)}{2} = O(n^2)$$

Perhitungan ini mirip dengan perhitungan kompleksitas *bubble sort* sebelumnya. Namun, melalui uji empiris, didapatkan bahwa *Selection Sort* lebih efisien daripada *Bubble Sort.* Grafiknya dapat dilihat berikut ini:



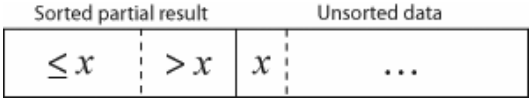
Gambar 2. Grafik Efisiensi *Selection Sort*

Peningkatan performansi yang diberikan 60% lebih baik daripada *bubble sort.* Namun, *sorting* model ini tergolong buruk dan lebih baik dihindari penggunaannya, terutama untuk penanganan tabel dengan lebih dari 1000 elemen. Karena masih ada algoritma lain yang implementasinya sama mudahnya, namun performansinya jauh lebih baik. Algoritma yang dimaksud adalah *insertion sort* yang akan dibahas di bawah ini.

2.3. Insertion Sort

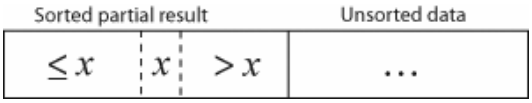
2.3.1. Konsep Insertion Sort

Cara kerja *insertion sort* sebagaimana namanya. Pertama-tama, dilakukan iterasi, dimana di setiap iterasi *insertion sort* memindahkan nilai elemen, kemudian menyisipkannya berulang-ulang sampai ke tempat yang tepat. Begitu seterusnya dilakukan. Dari proses iterasi, seperti biasa, terbentuklah bagian yang telah di-*sorting* dan bagian yang belum. Perhatikan gambar:



Gambar 3. Sebelum penyisipan

Dalam gambar, *insertion sort* sedang menangani elemen tabel yang ditandai *x*. Dengan satu kali *pass,* dilakukan penyisipan berulang-ulang dengan elemen-elemen sebelumnya sampai ditemukan elemen dengan nilai lebih kecil atau sama dengan *x*. Perhatikan gambar:



Gambar 4. Sesudah penyisipan

2.3.2. Algoritma *Insertion Sort*

Algoritma *Insertion Sort* dapat dirangkum sebagai berikut:

1. Simpan nilai T_i kedalam variabel sementara, dengan i = 1.
2. Bandingkan nilainya dengan elemen sebelumnya.
3. Jika elemen sebelumnya (T_{i-1}) lebih besar nilainya daripada T_i, maka tindih nilai T_i dengan nilai T_{i-1} tersebut. *Decrement* i (kurangi nilainya dengan 1).
4. Lakukan terus poin ke-tiga, sampai $T_{i-1} \leq T_i$.
5. Jika $T_{i-1} \leq T_i$ terpenuhi, tindih nilai di T_i dengan variabel sementara yang disimpan sebelumnya.
6. Ulangi langkah dari poin 1 di atas dengan i di-*increment* (ditambah satu).

Pseudocode untuk *Insertion Sorting* secara *Ascending* dapat dituliskan sebagai berikut:

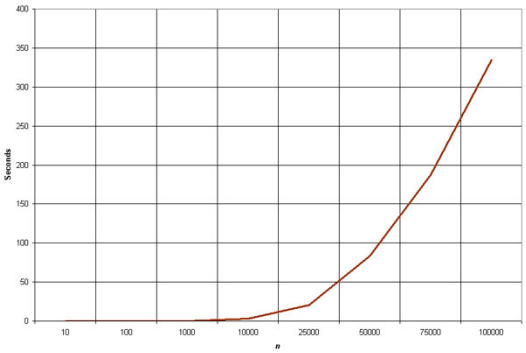
```
Procedure InsertionSort
(Input/Output T: TabInt, Input N:
integer)
{mengurut tabel integer [1 .. N]
dengan Insertion Sort secara
ascending}

Kamus:
  i: integer
  Pass: integer
  Temp: integer

Algoritma:
  Pass traversal [2..N]
    Temp ← TPass
    i ← i-1
    while (j ≥ 1) and (Ti > Temp) do
      Ti+1 ← Ti
      i ← i-1
    Ti+1 ← Temp
    {T[1..Pass-1] terurut}
```

2.3.3. Kompleksitas *Insertion Sort*

Algoritma *Insertion Sort* juga terdiri dari 2 kalang bersarang. Dimana terjadi N-1 *Pass* (dengan N adalah banyak elemen struktur data), dengan masing-masing *Pass* terjadi *i* kali operasi perbandingan. *i* tersebut bernilai 1 untuk *Pass* pertama, bernilai 2 untuk *Pass* kedua, begitu seterusnya hingga *Pass* ke N-1.

$$T(n) = 1 + 2 + .. + n - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

Walaupun sama seperti dua algoritma *sorting* sebelumnya, *insertion sort* lebih mangkus. Perhatikan gambar berikut:



Gambar 5. Grafik Efisiensi *Insertion Sort*

Berdasarkan gambar, *Insertion Sort* lebih dari dua kali lebih mangkus daripada *Bubble Sort* dan 40% lebih cepat daripada *Selection Sort*. *Insertion Sort* lebih baik tidak digunakan untuk menangani struktur data dengan lebih dari 2000 elemen.

2.4. Merge Sort

2.4.1. Sekilas Tentang Merge Sort

Algoritma *Merge Sort* ditemukan oleh John von Neumann di tahun 1945. *Merge Sort* termasuk paradigma algoritma *divide and conquer* (kurang lebih berarti: bagi dan atasi). Hal ini dikarenakan algoritma ini melakukan pembagian struktur data sebelum kemudian dioperasi satu per satu. Intinya, algoritma ini menggunakan dua ide utama sebagai berikut,

1. Sebuah list yang kecil membutuhkan langkah yang lebih sedikit untuk pengurutan daripada sebuah list yang besar.
2. Untuk membentuk sebuah list terurut dari dua buah list terurut membutuhkan langkah yang

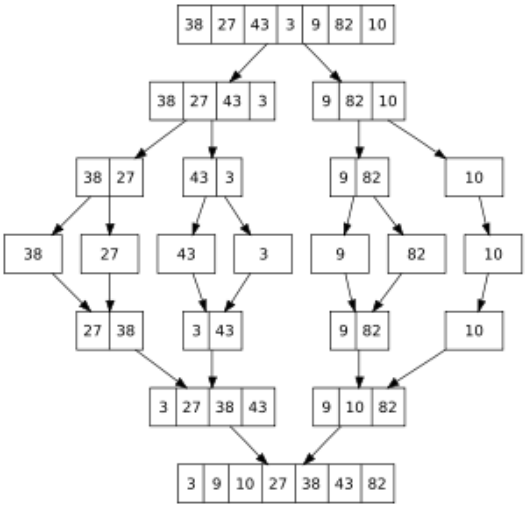
lebih sedikit daripada membentuk sebuah list terurut dari dua buah list tak terurut. Contoh: hanya diperlukan satu kali *traversal* untuk masing-masing list jika keduanya sudah terurut.

2.4.2. Algoritma *Merge Sort*

Algoritma *Merge Sort* sederhananya, dapat ditulis berikut:

1. Bagi list yang tak terurut menjadi dua sama panjang atau salah satunya lebih panjang satu elemen.
2. Bagi masing-masing dari 2 sub-list secara rekursif sampai didapatkan list dengan ukuran 1.
3. Gabung 2 sublist kembali menjadi satu list terurut.

Untuk lebih jelasnya, perhatikan gambar berikut:



Gambar 6. Skema ini menggambarkan *Merge Sort* dalam menangani pengurutan tabel dengan 7 elemen.

Pseudocode Algoritma *Merge Sort* dapat ditulis sebagai berikut (Tabel yang digunakan berbasis 0):

```
Function mergesort(num,temp: TabInt, N: integer)→ TabInt
{mengurut tabel integer [0 .. N-1] dengan Merge Sort}

Kamus:

Algoritma:
  → m_sort(num, temp, 0, N-1)


Function  m_sort(num,temp:  TabInt, left,right: integer)→ TabInt

Kamus:
  left,right,result: TabInt
  mid: integer

Algoritma:
  if (right > left)
    mid ← (right + left) / 2
    m_sort(num,temp,left,mid)
    m_sort(num,temp,mid+1,right)
    → merge(num,temp,left,mid+1,
right)
Function  merge  (num,temp:  TabInt,
left,mid,right: integer)→ TabInt

Kamus:
  i,left_end: integer
  num_el,tmp_pos: integer

Algoritma:
  left_end ← mid-1
  tmp_pos ← left
  num_el ← right-left+1
  while (left ≤ left_end) and (mid ≤
right) do
    if (num[left] ≤ num[mid]) then
      temp[tmp_pos] ← num[left]
      tmp_pos ← tmp_pos + 1
      left ← left +1
    else
      temp[tmp_pos] ← num[mid]
      tmp_pos ← tmp_pos + 1
      mid ← mid + 1
  while (left ≤ left_end) do
    temp[tmp_pos] ← num[left]
    left ← left + 1
    tmp_pos ← tmp_pos + 1
  while (mid ≤ right) do
    temp[tmp_pos] ← num [mid]
    mid ← mid + 1
    tmp_pos ← tmp_pos + 1
  i traversal [0..num_el]
    num[right] ← temp[right]
    right ← right – 1
  → num
```

2.4.3. Kompleksitas *Merge Sort*

Untuk pengurutan n elemen, *merge sort* memiliki kompleksitas $O(n \log n)$. Kompleksitas ini, juga berlaku untuk kasus terburuk.

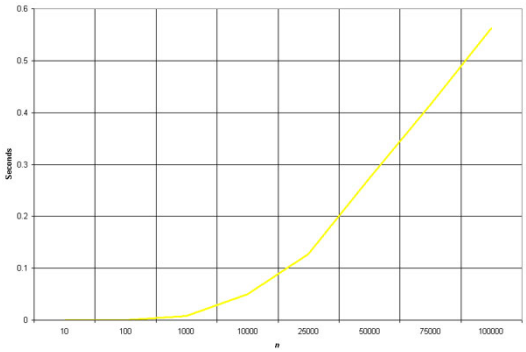
Jika *running time* untuk *merge sort* terhadap struktur data dengan banyak elemen n adalah $T(n)$, maka *recurrence*-nya $T(n) = 2T(n/2) + n$ berdasarkan definisi dari algoritma.

Algoritma ini jelas lebih mangkus daripada algoritma-algoritma sebelumnya.

Algoritma ini rekursif, sehingga algoritma ini menjadi pilihan buruk jika ingin dijalankan di mesin komputer dengan memori yang terbatas.

Perhatikan gambar! Bandingkan dengan algoritma-algoritma dengan kompleksitas $O(n^2)$ sebelumnya yang ketika diimplementasikan umumnya membutuhkan waktu di atas 100 detik ketika menangani tabel dengan ribuan elemen.

Perhatikan gambar di bawah ini:



Gambar 7. Grafik Efisiensi *Merge Sort*

2.5. Quick Sort

2.5.1. Sekilas Tentang Quick Sort

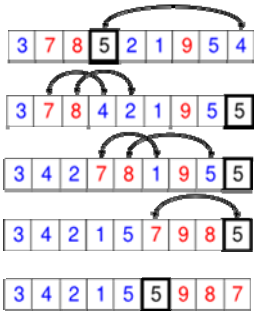
Quick Sort adalah algoritma *sorting* yang terkenal yang dirancang oleh C.A.R. Hoare pada tahun 1960 ketika bekerja untuk perusahaan manufaktur komputer saintifik kecil, *Elliott Brothers*. Algoritma ini rekursif, dan termasuk paradigma algoritma *divide and conquer*.

2.5.2. Algoritma Quick Sort

Algoritma ini terdiri dari 4 langkah utama:

1. Jika struktur data terdiri dari 1 atau 0 elemen yang harus diurutkan, kembalikan struktur data itu apa adanya.
2. Ambil sebuah elemen yang akan digunakan sebagai *pivot point* (poin poros). (Biasanya elemen yang paling kiri.)
3. Bagi struktur data menjadi dua bagian – satu dengan elemen-elemen yang lebih besar daripada *pivot* point, dan yang lainnya dengan elemen-elemen yang lebih kecil daripada *pivot* point.
4. Ulangi algoritma secara rekursif terhadap kedua paruh struktur data.

Perhatikan gambar simulasi berikut. Kotak yang ditebalkan adalah *pivot*. Angka yang berwarna merah bernilai ≤ *pivot,* sementara yang berwarna biru sebaliknya. Simulasi ini menggambarkan proses penukaran yang terjadi supaya terpenuhi kondisi yang dimaksud algoritma no 3:



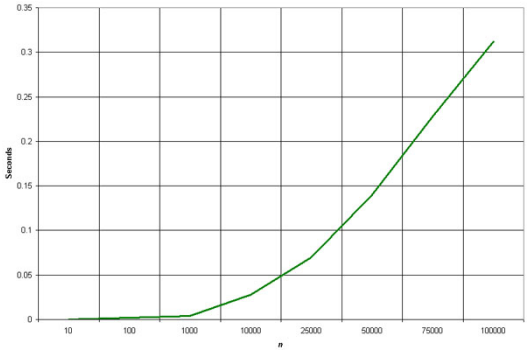
Gambar 8. Simulasi untuk algoritma poin ke 3.

Versi sederhana *pseudocode* untuk algoritma ini dapat ditulis berikut:

```
Function quicksort(num: TabInt,
left,right: integer)→ TabInt
{mengurut tabel integer [1 .. N]
dengan Quick Sort}

Kamus:
  pivIdx: integer {indeks pivot}
  pivNewIdx: integer {indeks pivot
yang baru}

Algoritma:
  if (right > left) then
    pivIdx ← left
    {pengambilan pivot point bisa
apa saja / random}
    pivNewIdx ← partition
(num,left,right,pivIdx)
    quicksort(num,left,pivNewIdx-1)
    quicksort(num,pivNewIdx+1,right)
  → num


Function partition(num: TabInt,
left,right,pivIdx: integer)→ integer

Kamus:
  pivVal: integer {penyimpan
sementara nilai pivot}
  storeIdx,i: integer

Algoritma:
  pivVal ← num[pivIdx]
  swap(num[pivIdx],num[right])
   {memindahkan pivot ke akhir}
  storeIdx ← left
  i traversal [left..right]
    {left ≤ i < right}
    if (num[i] ≤ pivVal)
      swap (num[i],num[storeIdx])
      storeIdx ← storeIdx + 1
  swap (num[storeIdx],num[right])
  → storeIdx
```

2.5.3. Kompleksitas Quick Sort

Efisiensi algoritma ini sangat dipengaruhi elemen mana yang dipilih sebagai *pivot point*. Kasus terburuk memiliki efisiensi $O(n^2)$, terjadi jika struktur data telah terurut, sementara elemen yang paling kiri dipilih sebagai *pivot point*. Pemilihan *pivot point* secara *random* (acak) disarankan jika data yang akan diurut-kan tidak *random*. Selama *pivot point* dipilih secara acak, *Quick Sort* memiliki kompleksitas algoritma $O(n \log n)$. Perhatikan gambar!



Gambar 9. Grafik Efisiensi *Quick Sort*

3. KESIMPULAN

Algoritma yang mudah dalam hal implementasi adalah *Bubble Sort, Selection Sort, dan Insertion Sort*. Ketiganya memiliki kompleksitas $O(n^2)$. Di antara algoritma ini, yang paling effisien adalah *Insertion Sort*. Algoritma yang lebih mangkus adalah *Merge Sort* dan *Quick Sort* dengan kompleksitasnya adalah *O(n log n)*. Adapun yang paling mangkus dari lima algoritma ini adalah *Quick Sort*.

DAFTAR REFERENSI

[1] Munir, Rinaldi. 2003. Diktat Kuliah IF2153 Matematika Diskrit. Program Studi Teknik Informatika, Institut Teknologi Bandung. hlm. V-1 dan X-1 s.d. X-20

[2] Liem, Inggriani. 2007. Draft Diktat Kuliah Dasar Pemrograman (Bagian Pemrograman Prosedural). Program Studi Teknik Informatika, Institut Teknologi Bandung. hlm. 141-142

[3] Wikipedia, *the free encyclopedia*. 2007. Tanggal akses: 26 Desember 2007 pukul 23.00 WIB dan 31 Desember 2007 pukul 22.00 WIB http://en.wikipedia.org/

[4] Michael's Homepage. Tanggal akses: 31 Desember 2007 pukul 22.00 WIB dan 2 Januari 2008 pukul 12.50 WIB. http://linux.wku.edu/~lamonml/

[5] *Dictionary of Algorithms and Data Structures* [online]. 2007. Tanggal akses: 31 Desember 2007 pukul 22.00 WIB, 1 Januari 2008 pukul 13.30 WIB. http://www.nist.gov/dads/HTML/

[6] Cprogramming.com. 2007. Tanggal akses: 31 Desember 2007 pukul 22.00 WIB. http://www.cprogramming.com/tutorial/computer sciencetheory/

[7] J.E.N.I. Tanggal akses: 1 Desember 2007 pukul 11.41 WIB hlm. 1-3. http://dahlan.unimal.ac.id/files/java2/JENI-Intro2-Bab06-Algoritma%20Sorting.pdf .

[8] Ngoen, Thompson Susabna. Tanpa tahun. Institut Bisnis dan Informatika Indonesia. hlm. 1. http://anthonysteven.wordpress.com/2007/08/26/algoritma-bubble-sort/