Perbandingan Algoritma Pengurutan *Merge Sort, Quick Sort* dan *Heap Sort* Dilihat dari Kompleksitasnya

Made Edwin Wira Putra (13508010)

Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Jl. Ganesha no. 10, Bandung, Jawa Barat
e-mail : if18010@students.if.itb.ac.id

ABSTRAK

Makalah ini membahas tentang analisa perbandingan kompleksitas dari algoritma *Merge Sort, Quick Sort* dan *Heap Sort*. Algoritma-algoritma tersebut merupakan algoritma pengurutan. Secara umum, *Quick Sort* yang memiliki kompleksitas pembandingan elemen rata-rata $A(n) = 1{,}38n \lg n$ lebih banyak digunakan dibanding *Merge Sort* yang lebih mudah dimengerti dengan kompleksitas pembandingan elemen rata-rata $A(n) = n \lg n$. *Heap Sort* memiliki kompleksitas pembandingan elemen terendah di antara ketiga algoritma tersebut, yaitu $A(n) = n \lg n$. *Linked implementation* terhadap *Merge Sort* bisa menghilangkan *disadvantages* dalam penggunaan *Merge Sort,* kecuali masalah penggunaaan memori tambahan.

Kata kunci: *Heap Sort, Merge Sort, Quick Sort, Kompleksitas Algoritma.*

1. PENDAHULUAN

Analisa algoritma bertujuan untuk mengetahui efisiensi algoritma. Dalam makalah ini yang akan dibahas adalah algoritma sorting *Merge Sort*, *Quick Sort* dan *Heap Sort*. Analisis dilakukan dengan membandingkan *Assignment Records* antara algoritma *Merge Sort, Quick Sort* dan *Heap Sort* dan jumlah pembandingan elemen data antara *Merge Sort, Quick Sort* dan *Heap Sort*. Hasil analisis yang dapat adalah kompleksitas *Assignment Records* dan jumlah pembandingan data untuk masing-masing algoritma.

2. ALGORITMA *MERGE SORT*

2.1 Konsep Algoritma *Merge Sort*

Secara konseptual, untuk sebuah *array* berukuran *n*, Merge *Sort* bekerja sebagai berikut:

1. Jika bernilai 0 atau 1, maka *array* sudah terurut. Sebaliknya:
2. Bagi *array* yang tidak terurut menjadi dua sub-*array*, masing-masing berukuran *n*/2.
3. Urutkan setiap sub-*array*. Jika sub-*array* tidak cukup kecil, lakukan rekursif langkah 2 terhadap sub-*array*.
4. Menggabungkan dua sub-*array* kembali menjadi satu *array* yang terurut.

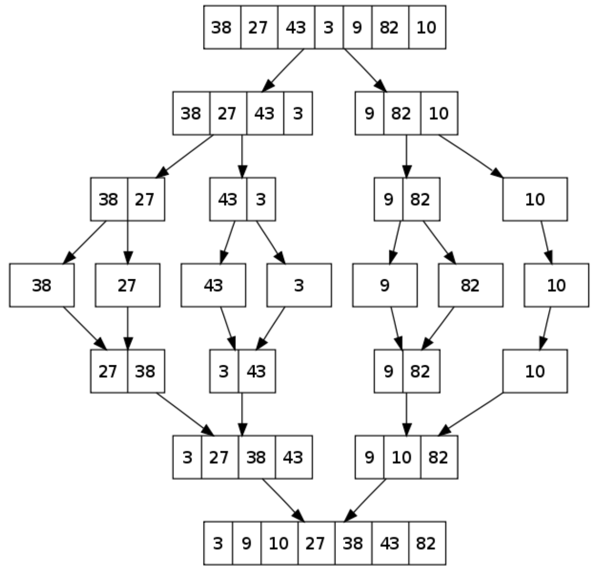
Merge sort menggabungkan dua ide utama untuk meningkatkan *runtime*nya:

1. *Array* kecil akan mengambil langkah-langkah untuk menyortir lebih sedikit dari *array* besar.
2. Lebih sedikit langkah yang diperlukan untuk membangun sebuah *array* terurut dari dua buah *array* terurut daripada dari dua buah *array* tak terurut.

Dalam bentuk pseudocode, algoritma *Merge Sort* dapat terlihat seperti ini:

```
procedure mergesort(input n : integer,
input/output S : array[1..n] of
keytype)
{Mengurutkan array sebesar n dengan
urutan takmenurun
I.S. : n bilangan bulat positif, S
terdefinisi berindex dari 1 sampai n
F.S. : array S berisi elemen dengan
urutan takmenurun
}
const
     h = n div 2
     m = n - h
var
     U : array [1..h] of keytype
     V : array [1..m] of keytype
begin
     if n > 1 then
       copy dari S[1] hingga S[h] ke U
       copy dari S[h+1] hingga S[n] ke V
       mergesort(h,U)
       mergesort(m,V)
       merge(h,m,U,V,S)
     end
end
```

```
procedure merge(input h,m : integer, U
: array [1..h] of keytype, V : array
[1..m] of keytype, input/output S :
array [1..h+m] of keytype)
{Menggabungkan dua array terurut
menjadi satu array terurut
I.S. : h dan m bilangan bulat positif,
array terurut U berindeks dari 1 sampai
h, array terurut V berindeks dari 1
sampai m
F.S. : array S berindex dari 1 sampai
h+m berisi elemen U dan V yang telah
terurut
}
var
i,j,k : indeks;
begin
     i = 1
     j = 1
     k = 1
     while i<=h and j<=m do
       if U[i]<V[j] then
            S[k]=U[i]
            i = i + 1
       else
            S[k] = V[j]
            j = j+1
       k = k + 1
     end
     if i>h then
       copy V[j] sampai V[m] ke S[k]
sampai S[h+m]
     else
       copy V[i] sampai U[h] ke S[k]
sampai S[h+m]
     end
end
```



Gambar 1 : Langkah-langkah ketika mengurutkan menggunakan *Merge Sort*

2.2. Kompleksitas *Merge Sort*

Dalam algoritma ini, jumlah perbandingan yang terjadi bergantung pada h dan m. Kondisi terburuk terjadi ketika perulangan berhenti, karena salah satu indeks, sebut saja i, telah mencapai titik berhentinya dimana indeks lain j telah mencapai m – 1, lebih rendah 1 dari titik berhentinya. Sehingga,

$$W(h,m) = h + m - 1$$

Jumlah keseluruhan perbandingan adalah jumlah banyaknya perbandingan dalam pemanggilan rekursif *merge sort* dimana U sebagai *input*, banyaknya perbandingan dalam pemanggilan rekursif *merge* sort dimana V sebagai *input*, dan banyaknya perbandingan di *top-level* pemanggilan *merge.* Sehingga,

$$W(n) = \underbrace{W(h)}_{\text{Waktu untuk mengurutkan U}} + \underbrace{W(m)}_{\text{Waktu untuk mengurutkan V}} + \underbrace{h + m - 1}_{\text{Waktu untuk bergabung}}$$

Pertama, kita menganalisa kasus diaman n adalah eksponen dari 2. Dalam kasus ini,

Ekspresi untuk W(n) menjadi

Ketika besar input adalah 1, kondisi pemberhentian terpenuhi dan tak ada penggabungan. Sehingga, $W(1)$ adalah 0.

Solusi dari rekurens tersebut adalah

Merge Sort akan selalu membagi dua tiap sub-*array*nya hingga mencapai basis, sehingga kompleksitas dari algoritma *Merge Sort*,   berlaku untuk semua kasus (*Worst Case* = *Best Case* = *Average Case*).

3. ALGORITMA *QUICK SORT*

3.1 Konsep Algoritma *Quick Sort*

Quick Sort mengurutkan menggunakan berbasiskan strategi *Divide and Conquer* untuk membagi *array* menjadi dua sub-*array*.

Langkah-langkahnya :

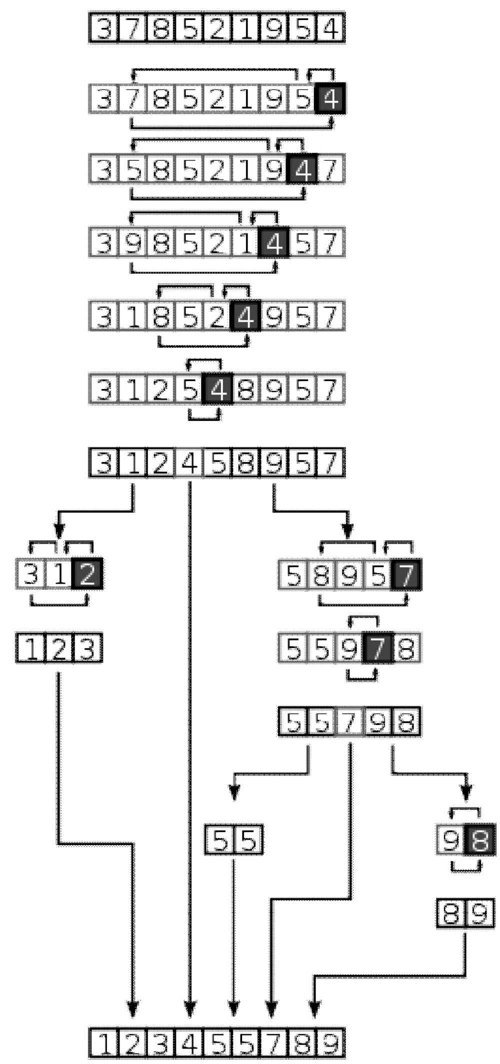
1. Ambil sebuah elemen dari *array*, beri nama pivot.
2. Urutkan kembali *array* sehingga elemen yang lebih kecil dari pivot berada sebelum pivot dan elemen yang lebih besar berada setelah pivot. Langkah ini disebut *partition*.
3. Secara rekursif, urutkan kembali sub-*array* elemen yang lebih kecil dan sub-*array* elemen yang lebih besar

Basis dari rekursif adalah besarnya *array* 1 atau 0, dimana menunjukkan *array* telah terurut.

Dalam bentuk pseudocode, algoritma *Quick Sort* dapat terlihat seperti ini:

```
procedure quicksort(input low,high : indeks)
{Mengurutkan n elemen secara tak menurun
I.S. : n bilangan bulat positif, S terdefinisi berindeks dari 1 sampai n
F.S. : array S berisi elemen dengan urutan takmenurun
}
var
     pivot : indeks
begin
     if high > low then
       partition(low,high,pivotpoint)
       quicksort(low,pivotpoint-1)
       quicksort(pivotpoint+1,high)
     end
end
```

```
procedure partition(input low,high: indeks, input/output pivotpoint : indeks)
{Mempartisi array S untuk Quick Sort
I.S. : Dua indeks, low dan high, dan sub-array S terdefinisi berindeks low sampai high
F.S. : pivotpoint, pivot point dari sub-array terindeks antara low sampai high}
Var
     i,j : indeks
     pivotitem : keytype
begin
     pivotitem = S[low]
     j = low
     i traversal low+1..high
       if S[i] < pivotitem then
             j = j + 1
             swap S[i] dan S[j]
       end
     end
     pivotpoint = j
     swap S[low] dan S[pivotpoint]
end
```



Gambar 2 : Langkah-langkah ketika mengurutkan menggunakan *Quick Sort*

3.2. Kompleksitas *Quick Sort*

Kasus terburuk untuk *Quick Sort* terjadi jika *array* dalam keadaaan telah terurut takmenurun. Bila *array* telah terurut, maka tak ada elemen lebih kecil dari pivot yang dipindahkan ke sebelah pivot. Sehingga,

$$T(n) = \underbrace{T(0)}_{\substack{\text{Waktu untuk}\\ \text{mengurutkan}\\ \text{sub-}array\text{ kiri}}} + \underbrace{T(n-1)}_{\substack{\text{Waktu untuk}\\ \text{mengurutkan}\\ \text{sub-}array\text{ kiri}}} + \underbrace{n-1}_{\substack{\text{Waktu untuk}\\ \text{mempartisi}}}$$

Karena T(0) = 0, maka rekurensnya,

$$\boxed{\begin{array}{l} T(n) = T(n-1) + n - 1 \qquad untuk\ n > 0 \\ T(0) = 0 \end{array}}$$

Solusi dari rekurens tersebut adalah $T(n) = \frac{n(n-1)}{2}$, sehingga kompleksitas kasus terburuk *Quick Sort* adalah $W(n) = \frac{n(n-1)}{2} \in \theta(n^2)$.

Waktu kompleksitas untuk kasus rata-rata dapat ditentukan dengan menyelesaikan rekurens ini :

$$A(n) = \sum_{p=1}^{n} \underbrace{\frac{1}{n}[A(p-1) + A(n-p)]}_{\substack{\text{Waktu rata-rata untuk}\\ \text{mengurutkan sub-}array\\ \text{ketika pivotpoint = p}}} + \underbrace{(n-1)}_{\substack{\text{Waktu untuk}\\ \text{mempartisi}}}$$

$$\sum_{p=1}^{n} \frac{1}{n}[A(p-1) + A(n-p)] + (n-1) = 2\sum_{p=1}^{n} A(p-1)$$

Sehingga,

$$A(n) = \frac{2}{n}\sum_{p=1}^{n} A(p-1)$$

$$nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1)$$

Bila n adalah n-1, maka

$$(n-1)A(n-1) = 2\sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2)$$

Dengan men-substrak dua persamaan tersebut, didapat

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

Misal $a_n = \frac{A(n)}{n+1}$, maka akan didapat rekurens,

$$\boxed{\begin{array}{l} a_n = a_{n-1} + \dfrac{2(n-1)}{n(n+1)} \qquad untuk\ n > 0 \\ a_{0=}0 \end{array}}$$

Solusi dari rekurens tersebut adalah $a_n \approx 2\ln n$ yang mengakibatkan

$$A(n) \approx (n+1)2\ln n = (n+1)2(\ln 2)(\lg n)$$
$$\approx 1.38(n+1)\ \lg n\ \in \theta(n\ \lg n)$$

4. ALGORITMA *HEAP SORT*

4.1 Konsep Algoritma *Heap Sort*

Binary heap digunakan sebagai struktur data dalam algoritma *Heap-Sort*. Sebagaimana diketahui, ketika suatu *Heap* dibangun maka kunci utamanya adalah: *node* atas selalu mengandung elemen lebih besar dari kedua *node* dibawahnya. Apabila elemen berikutnya ternyata lebih besar dari elemen *root*, maka harus di *swap* dan lakukan: proses *heapify* lagi. *Root* dari suatu *Heap Sort* mengandung elemen terbesar dari semua elemen dalam *Heap*.

Proses *Heap Sort* dapat dijelaskan sebagai berikut:

1. Representasikan *Heap* dengan n elemen dalam sebuah *array* $A[n]$
2. Elemen *root* tempatkan pada $A[1]$
3. Elemen $A[2i]$ adalah node kiri dibawah $A[i]$
4. Elemen $A[2i+1]$ adalah node kanan dibawah $A[i]$
5. Ambil nilai *root* (terbesar) $A[1..n-1]$ dan pertukarkan dengan elemen terakhir dalam *array*, $A[n]$
6. Bentuk *Heap* dari (n-1) elemen, dari $A[1]$ hingga $A[n-1]$
7. *Ulangi langkah 5 dimana indeks terakhir berkurang setiap langkah.*

Dalam bentuk pseudocode, algoritma *Heap Sort* dapat terlihat seperti ini:

```
procedure siftdown(input/output : H :
heap)
var
    parent,largerchild : node
begin
    parent = rootdari H
    largerchild = anak dari parent
yang menampung elemen terbesar
    while elemen di parent lebih kecil
dari elemen di largerchild do
      tukar elemen di parent dengan
elemen di larger child
      parent = largerchild
      largerchild = anak dari parent
yang menampung elemen terbesar
    end
end
```

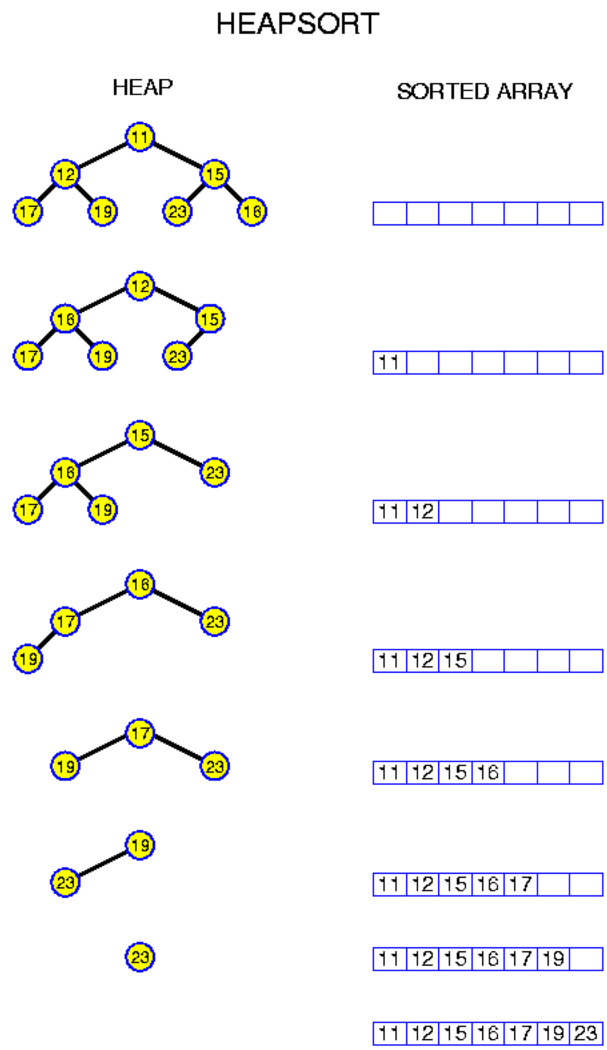
```
function root(input/output H :
heap):keytype
var
    keyout : keytype
begin
    keyout = elemen di akar
    pindah elemen di node bawah ke
akar
    delete node bawah
    siftdown(H)
```

```
     root = keyout
end
```

```
procedure removekeys(input n : integer , H : heap, input/output S : array [1..n] of keytype)
var
     i : indeks
begin
     i traversal n..1
       S[i] = root(H)
     end
end
```

```
procedure makeheap(input n : integer, input/output H : heap)
var
     i : indeks
     Hsub: heap
begin
     i traversal d-1..0
       untuk semua subtrees yang akar mempunyai kedalaman i do
             siftdown(Hsub)
       end
     end
end
```

```
procedure heapsort(input n : integer, H : heap, input/output S : array [1..n] of keytype)
begin
     makeheap(n,H)
     removekeys(n,H,S)
end
```



Gambar 3 : Langkah-langkah ketika mengurutkan menggunakan *Heap Sort*

4.2. Kompleksitas *Heap Sort*

Dengan menganalisa *makeheap*, maka didapat banyak pembandingan elemen yang terjadi oleh *makeheap* paling banyak $2(n-1)$. Selanjutnya dengan menganalisa *removekeys*, maka banyaknya pembandingan elemen yang dilakukan oleh *removekeys* paling banyak adalah

$$2\sum_{j=1}^{d-1} j2^j = 2(d2^d - 2^{d+1} + 2) = 2n \lg n - 4n + 4$$

Dengan mengkombinasikan analisis dari *makeheap* dan *removekeys*, didapat banyaknya pembandingan elemen di *Heap Sort* ketika n merupakan eksponen dari 2 paling banyak adalah

$$2(n-1) + 2n \lg n - 4n + 4 = 2(n \lg n - n + 1) \approx 2n \lg n$$

Sehingga untuk n merupakan eksponen dari 2,

$$W(n) \approx 2n \lg n \in \theta(n \lg n)$$

Sulit untuk menganalisa kompleksitas kasus rata-rata *Heap Sort* secara analitis. Namun, studi empiristelah menunjukkan bahwa kompleksitas untuk kasus rata-ratanya tidak lebih baik dari kasus terburuknya. Ini menunjukkan bahwa kompleksitas untuk kasus rata-rata *Heap Sort* adalah

$$A(n) \approx 2n \lg n \in \theta(n \lg n)$$

5. ANALISIS DAN PERBANDINGAN

Merge Sort, *Quick Sort* dan *Heap Sort* mempunyai batasan yang sama $\theta(n \lg n)$, tetapi dengan basis log yang berbeda. Khusus *Quick Sort* memiliki kompleksitas $\theta(n^2)$ untuk kasus terburuk. Di antara ketiganya, *Heap Sort* memiliki kompleksitas terendah. Pada *Quick Sort*, kasus rata-rata memiliki kompleksitas $1{,}38n \lg n$.

Heap Sort merupakan sorting yang tidak stabil. Suatu *sorting* yang stabil akan menangani permintaan relatif dari *record* dengan kunci yang setara. Algoritma *sorting* yang tidak stabil dapat mengubah susunan relatif dari kunci yang sama/setara. *Sorting* yang tidak stabil dapat diimplementasikan menjadi stabil dengan cara memperlebar kunci perbandingan.. *Merge Sort* sendiri memiliki beberapa keungulan dari *Heap Sort* :

- *Merge Sort* mempertimbangkan performa *data cache* yang lebih baik dari *Heap Sort*, karena akses yang terurut tidak acak
- Mudah dimengerti
- Stabil

Namun, karena *Merge Sort* melakukan tiga kali lebih banyak *assignment records* dibanding *Quick Sort* secara rata-ratanya, maka *Quick Sort* lebih dipilih dibanding *Merge Sort*, meskipun *Quick Sort* melakukan pembandingan elemen yang lebih banyak dalam kasus rata-ratanya.

Tabel 1 Analisis Rangkuman untuk $\theta(n\ lg\ n)$ Algoritma Pengurutan *Merge Sort, Quick Sort* dan *Heap Sort*

Algoritma	Pembandingan Elemen
Merge Sort	$W(n) = n \lg n$ $A(n) = n \lg n$
Quick Sort	$W(n) = n^2/2$ $A(n) = 1{,}38n \lg n$
Heap Sort	$W(n) = 2n \lg n$ $A(n) = 2n \lg n$
Algoritma	*Assignment Records*
Merge Sort	$T(n) = 2n \lg n$
Quick Sort	$A(n) = 0{,}69n \lg n$
Heap Sort	$W(n) = n \lg n$ $A(n) = n \lg n$

6. KESIMPULAN

Secara keseluruhan, *Quick Sort* dipilih karena *Assignment Records*nya yang paling sedikit di antara 3 algoritma pengurutan tersebut, meskipun pembandingan elemen yang dilakukan *Quick Sort* lebih banyak dibandingkan *Merge Sort*.

Heap Sort memiliki kompleksitas terendah di antara 3 algoritma tersebut. Pembandingan elemen yang dilakukan oleh *Heap Sort* lebih banyak dibanding *Merge Sort*, dan *Merge Sort* melakukan *Assignment Records* lebih banyak dibanding *Heap Sort*. Namun, *Merge Sort* memiliki kelebihan-kelebihan lain yang membuatnya lebih unggul dibandingkan *Heap Sort*.

Merge Sort sendiri, jika mampu dikembangkan dengan mengimplementasikan linked list, maka keburukan dari *Merge Sort* dapat dihilangkan. Satu-satunya keburukan yang tersisa adalah ruang tambahan yang digunakan untuk link ekstra $\theta(n)$.

REFERENSI

[1] http://en.wikipedia.org/wiki/Heapsort
Waktu Akses : 20 Desember 2009, Pukul 10.00
[2] http://en.wikipedia.org/wiki/Merge_sort
Waktu Akses : 20 Desember 2009, Pukul 10.00
[3] http://en.wikipedia.org/wiki/Quicksort
Waktu Akses : 20 Desember 2009, Pukul 10.00
[4] http://www.ilmu-komputer.net/algorithms/sorting-algoritm-analysis/
Waktu Akses : 20 Desember 2009, Pukul 10.00
[5] Munir, Rinaldi. (2008). Diktat Kuliah IF2091 Struktur Diskrit, Departemen Teknik Informatika, Institut Teknologi Bandung.
[6] Neapolitan, Richard E. (1996). *Foundation of Algorithms*, D. C. Heath and Company. Toronto

.