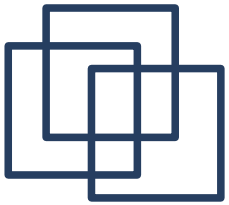
Analisis dan Implementasi Algoritma Sorting Untuk Ukuran Data Melebihi Kapasitas Memori

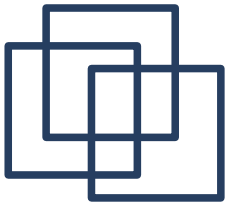
M. Shulhan

2008



Tujuan

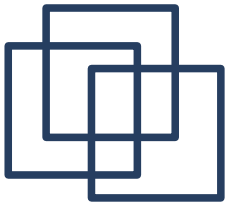
Membandingkan beberapa variasi dari metoda sorting untuk menemukan algoritma yang efisien (dalam waktu atau kecepatan pemprosesan) untuk mengurutkan data dalam jumlah yang besar yang melebihi kapasitas memori internal komputer.



Identifikasi Masalah

Diberikan sebuah file dengan 5000 record yang akan diurutkan, setiap record memiliki 20 field/kolom, dimana satu atau beberapa field adalah *key* yang akan diurutkan.

Jika hanya 1000 record yang dapat disimpan dalam memori internal komputer, apa yang dapat kita lakukan ?



Algoritma Sorting Yang Digunakan

- Merge Sort + Multiway Merge
- Binary Sort + Multiway Merge
- Quicksort + Multiway Merge
- Bucket Sort

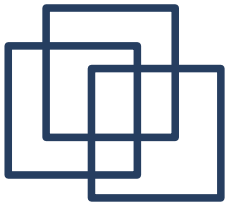
Metoda

Pengujian algoritma dilakukan dengan mengimplementasikan algoritma *sorting* tersebut ke dalam sebuah program.

Pengujian program dilakukan terhadap tiga tipe input data :

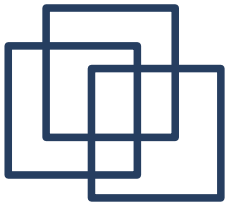
- Data yang tidak terurut (random)
- Data yang telah terurut ascending
- Data yang telah terurut descending

Pembandingan algoritma sorting dilakukan dengan melihat *running time* dari tiap-tiap algoritma terhadap tiga tipe input data tersebut.



Analisis

- Analisis kompleksitas algoritma.
- Analisis frekuensi baca/tulis pada algoritma.
- Analisis jumlah pembandingan (yang dilakukan) pada algoritma.

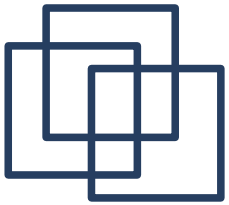


Analisis Kompleksitas Algoritma

Menghitung *running time* dari algoritma.
Running time sebuah algoritma terhadap sebuah input adalah jumlah operasi primitif atau *step* yang dieksekusi.

Asumsi :

- instruksi dieksekusi satu persatu, tanpa ada yang berjalan paralel.
- instruksi/step seperti aritmatika (+, -, /, *, div, mod, round, floor), perpindahan data (read, write, copy), dan kontrol (*conditional*, *function call* atau *function return*) dihitung dalam sebuah konstanta, *cost*.



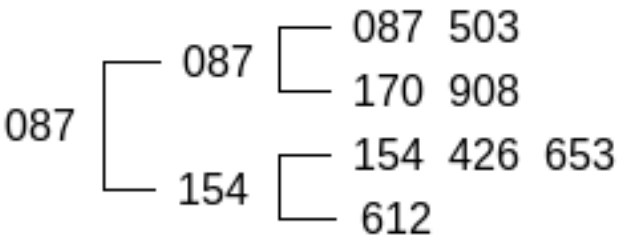
Multiway Merge

Yaitu proses penggabungan beberapa input data menjadi satu output data.

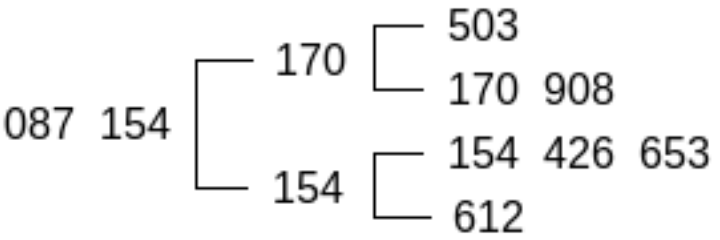
Syarat : seluruh input data haruslah dalam satu urutan yang sama, terurut *ascending* atau *descending*.

Multiway Merge

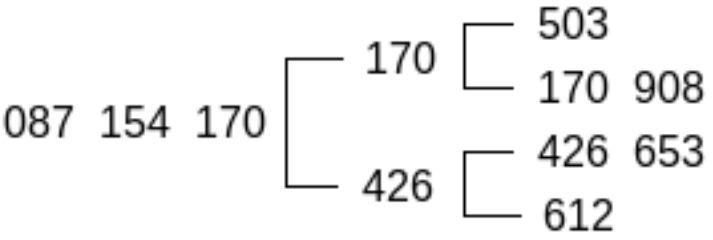
Langkah 1.



Langkah 2.



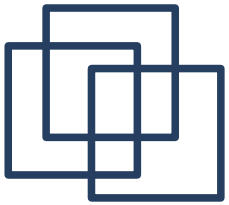
Langkah 3.



...

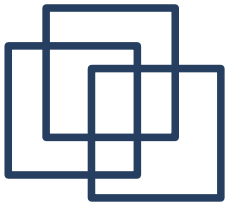
Langkah 9. 087 154 170 426 503 612 653 908

Worst-case running Time : $\Theta(n)$



Multiway Merge

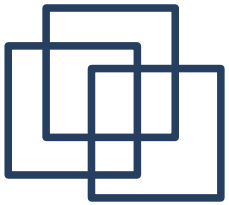
(1) Baca record pertama dari setiap input data.
(2) Bandingkan seluruh record pertama tersebut untuk mendapatkan record dengan key terkecil.
(3) Record terkecil tersebut kemudian disimpan ke output, dan dihapus dari data input.
(4) Baca record selanjutnya dari media input dimana record terkecil berada. Jika input data masih ada kembali ke langkah nomor 2, jika tidak bandingkan sisa record yang ada memori dan simpan ke media output.



Analisis Multiway Merge

multiway_merge(list_input, OUT)	Cost	Times
while (list_input <> NIL) do	c_1	s
R ← read(list_input);	c_2	s - 1
tree ← tree_insert(tree, R);	c_3	s - 1
list_input ← list_input.next;	c_4	s - 1
while (tree <> NIL) do	c_5	n
min ← tree_get_minimum(tree);	c_6	n - 1
tree ← remove_node(tree, min);	c_7	n - 1
write(min, OUT)	c_8	n - 1
R ← read(min.file);	c_9	n - 1
tree ← tree_insert(tree, R);	c_{10}	n - 1

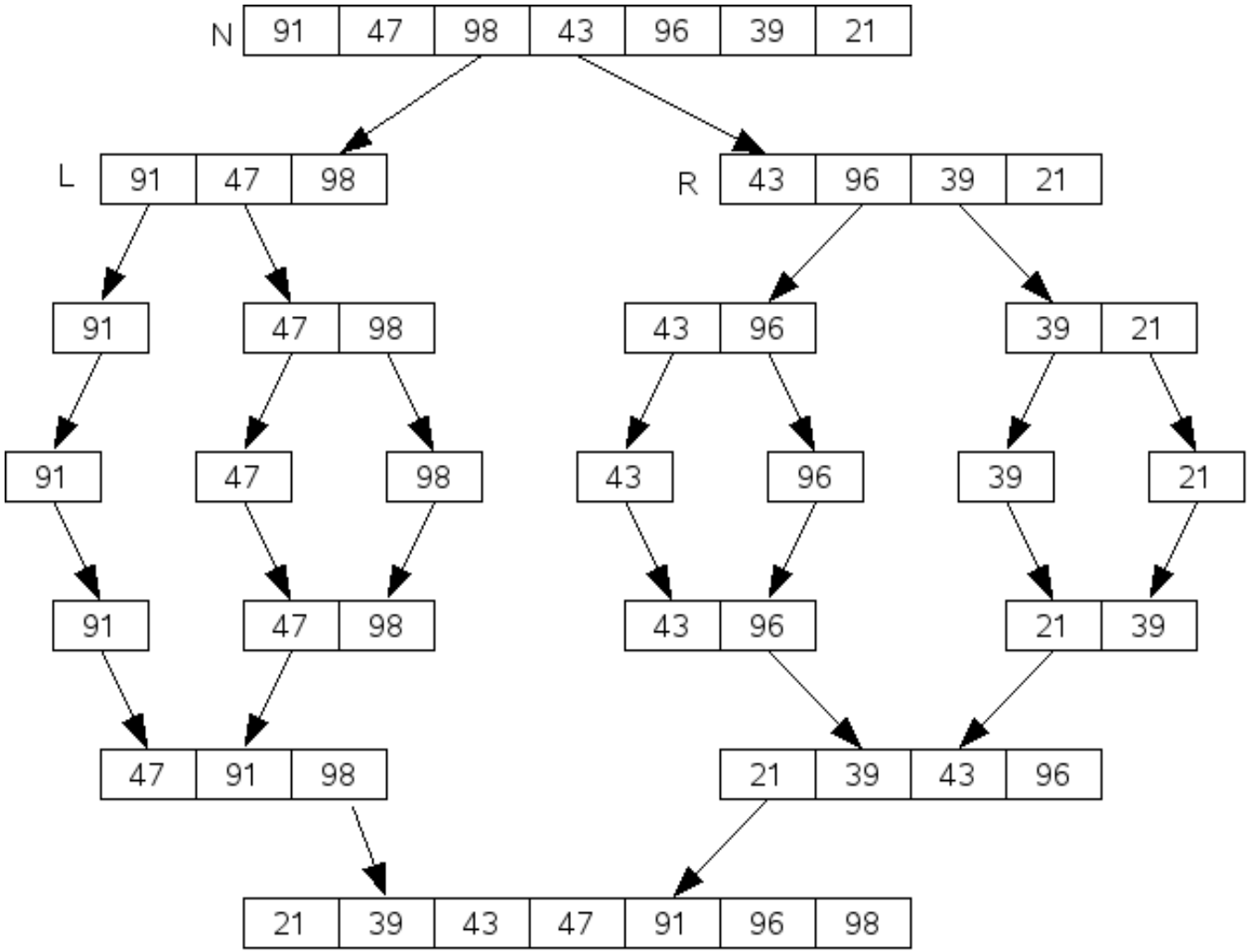
$$T(n) = c_5 n + c_1 s$$
$$= \Theta(n)$$



Merge sort

Merge sort yaitu algoritma sorting yang berdasarkan pembandingan, dengan menggunakan metoda *divide and conqueror*.

Merge Sort



Worst Case running Time : $\Theta(n \lg n)$

Merge Sort

Algoritma:

(1) Input data *N*, dibagi menjadi dua subbagian *L* dan *R*.

(2) Jika *L* > *1*, kembali ke langkah 1 dengan *L* menjadi input data, *N* ← *L*.

(3) Jika *R* > *1*, kembali ke langkah 1 dengan *R* menjadi input data, *N* ← *R*.

(4) Gabungkan *L* dan *R* menjadi satu bagian yang terurut.

Analisis Merge Sort

mergesort(data, n)	Cost	Times
if (n <= 1)	c_1	1
return data;	c_2	1
nleft ← n/2;	c_3	1
nright ← n - nleft;	c_4	1
left ← data;	c_5	1
right ← data;	c_6	1
for i ← 0 to nleft do	c_7	n/2
right ← right.next;	c_8	(n/2)-1
left ← mergesort(left, nleft);	c_9	T(n/2)
right ← mergesort(right, nright);	c_{10}	T(n/2)
while (left <> NIL and right <> NIL) do	c_{11}	n
if (left.key < right.key) then	c_{12}	n - 1
result ← list_add(result, left);	c_{13}	n - 1
else result ← list_add(result, right);	c_{14}	n - 1
add(result, left);	c_{15}	1
add(result, right);	c_{16}	1
return result;	c_{17}	1

$$T(n) = c_9 T(n/2) + c_{10} T(n/2) + c_{11} n$$
$$= 2\,T(n/2) + n$$

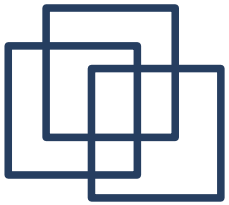
Analisis Merge Sort (2)

Dengan mengaplikasikan master method untuk menyelesaikan T(n) = 2 T(n/2) + n.

diketahui a = 2, b = 2, dan f(n) = n, maka:

$$f(n) = \Theta(n^{\log_b a})$$
$$n = \Theta(n^{\log_{۲} ۲})$$
$$n = \Theta(n)$$

$$sehingga\ didapat\ T(n) = \Theta(n^{\log_{۲} ۲} lg\, n) = \Theta(n\, lg\, n)$$

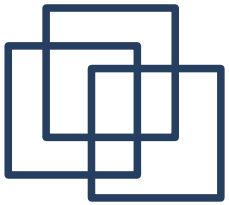


Binary Sort

Binary sort yaitu algoritma sorting yang menggunakan metoda *binary tree* untuk mengurutkan data.

Untuk menjaga keseimbangan tree (kesamaan tinggi dari tree pada kedua bagian kiri dan kanan terhadap *root*) maka digunakan algoritma *red-black tree*.

Red-black tree yaitu *tree* dengan tambahan sebuah attribut: warna, di setiap nodenya. *Red-black tree* menjamin bahwa tidak ada *path* yang panjangnya dua kali dari panjang *path* yang lainnya.

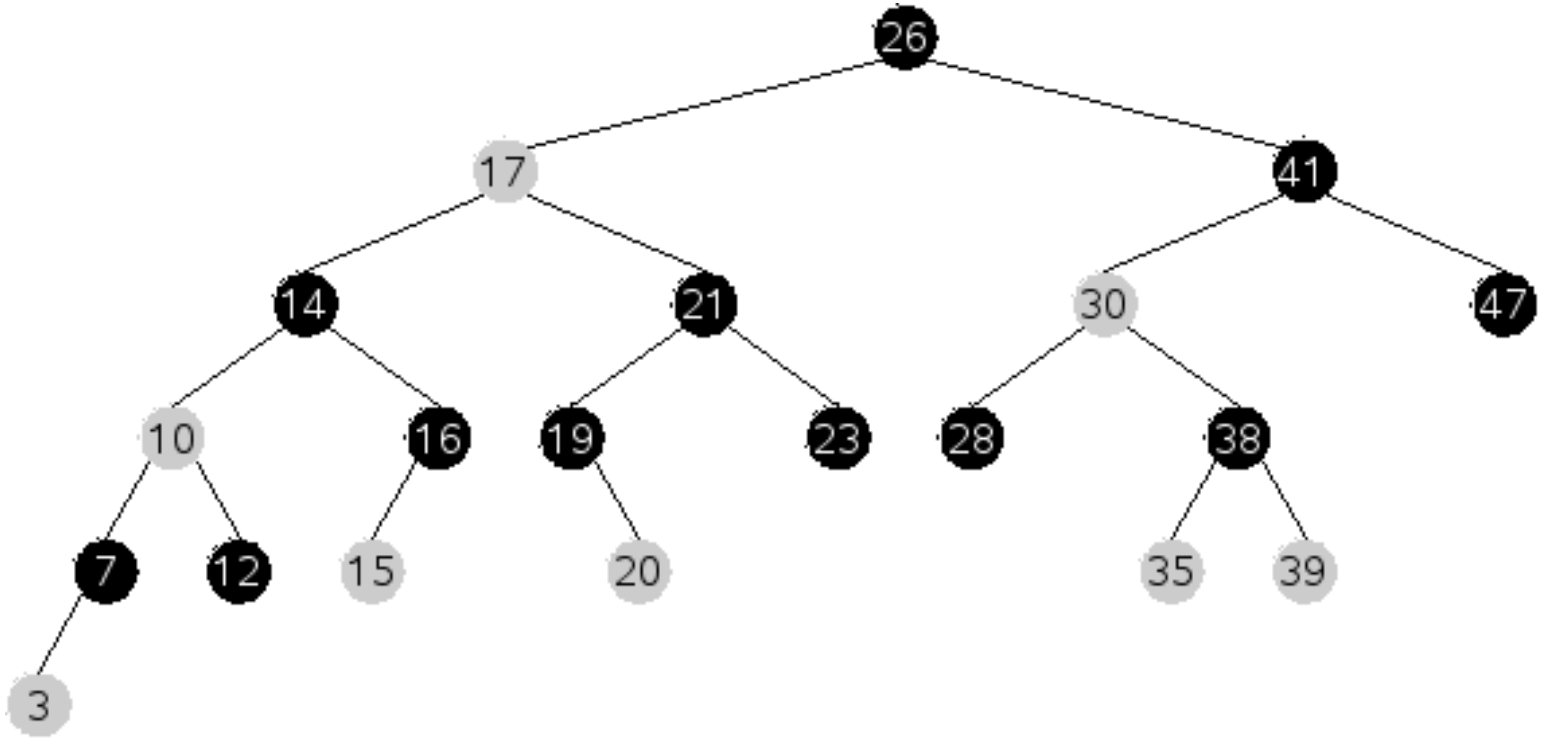


Binary Sort

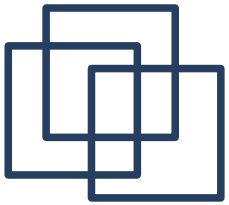
Red-black tree memiliki properti sebagai berikut :

1. Setiap node memiliki warna RED atau BLACK.
2. Root dari tree haruslah berwarna BLACK.
3. Jika sebuah node berwarna RED maka kedua anaknya haruslah berwarna BLACK.
4. Untuk setiap node, semua path dari node tersebut ke turunan dibawahnya, kiri atau kanan, memiliki jumlah node BLACK yang sama.

Binary Sort



Worst Case Running Time : $\Theta(n \lg n)$



Analisis Binary Sort

binarysort(data)	Cost	Times
`while (data <> NIL) do`	c_1	n
`    tree ← redblacktree_insert(tree, data);`	c_2	n - 1
`    data ← data.next;`	c_3	n - 1
`return tree;`	c_4	1

$$T(n) = c1\ n + c2\ (n\text{-}1)$$
$$= \Theta(n)$$

Analisis Binary Sort (2)

redblacktree_insert(tree,node)	Cost	Times
p ← tree;	c_1	1
while (p <> NIL) do	c_2	h
parent = p;	c_3	h - 1
if (p.key < node.key) then	c_4	h - 1
p ← p.left;	c_5	h - 1
else p ← p.right;	c_6	h - 1
if (node.key < parent.key) then	c_7	1
parent.left ← node;	c_8	1
else parent.right ← node;	c_9	1
node.parent ← parent;	c_{10}	1
return tree;	c_{11}	1

h adalah tinggi dari sebuah *tree*. Nilai *h* dapat dihitung dengan,

$$n \geq 2^{h/2} - 1$$

$$n + 1 \geq 2^{h/2}$$

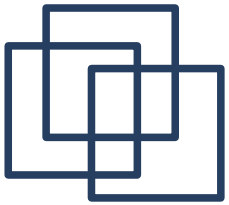
$$\lg (n + 1) \geq h/2$$

$$2 \lg (n + 1) \geq h$$

maka, h = $\Theta(\lg n)$. Sehingga running time dari binary sort,

$$T(n) = n \,.\, \Theta(\lg n)$$

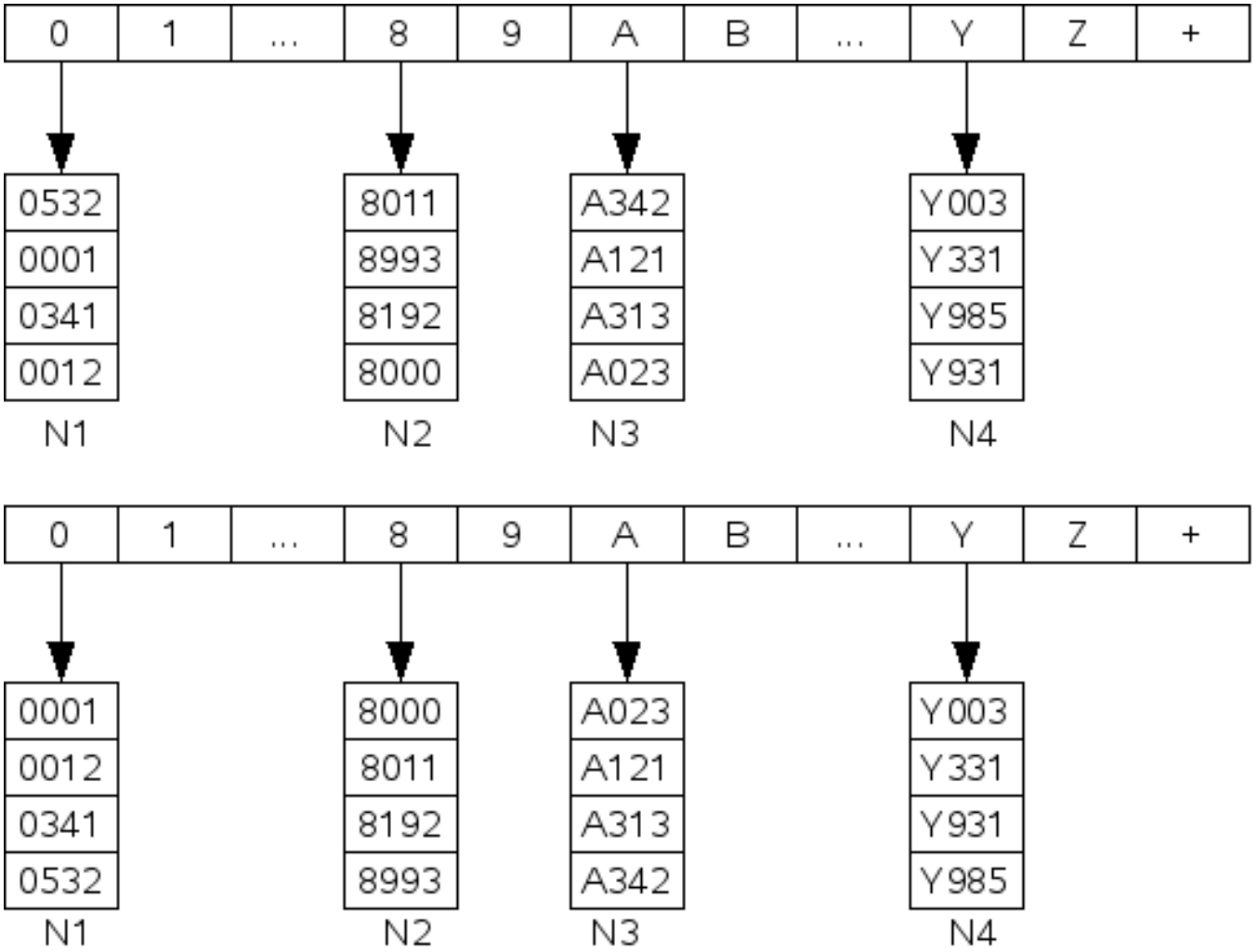
$$= \Theta (n \lg n)$$



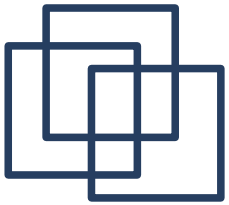
Bucket Sort

Bucket Sort adalah algoritma sorting yang bekerja dengan membagi input data ke dalam sejumlah kantong (*bucket*). Setiap *bucket* memiliki *key* tersendiri yang unik satu dengan yang lainnya. Setiap *bucket* kemudian di-*sort* secara tersendiri, dapat menggunakan algoritma *sorting* yang berbeda, atau dengan menggunakan algoritma *sorting bucket* secara rekursif.

Bucket Sort



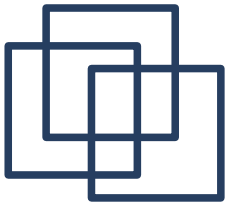
Worst Case Running Time : $\Theta(n \lg n)$



Bucket Sort

Algoritma Bucket sort :

(1) Persiapkan *bucket*, penampung input data.
(2) Untuk setiap input data, masukkan setiap objek data ke *bucket*-nya berdasarkan *key*.
(3) Urutkan tiap *bucket* yang tidak kosong secara tersendiri.
(4) Gabungkan kembali data dari setiap *bucket*, dimulai dari *bucket* dengan *key* terkecil sampai dengan *bucket* dengan *key* terbesar.

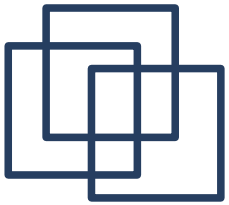


Analisis Bucket Sort

bucketsort(data,bucket,nbucket,OUT)	Cost	Times
while (data <> NIL) do	c_1	n
bucket_insert(bucket[data.key], data);	c_2	n - 1
data ← data.next;	c_3	n - 1
for i ← 1 to nbucket do	c_4	b
bucket[i] ← sort(bucket[i]);	c_5	b - 1
write(bucket[i], OUT);	c_6	b - 1

Kasus terburuk pada bucket sort yaitu bila seluruh data jatuh ke dalam satu bucket. Jika algoritma sorting yang digunakan pada baris ke 5 adalah mergesort, maka running time bucket sort menjadi sama dengan dengan mergesort :

$$T(n) = \Theta(n \lg n) + \Theta(n)$$

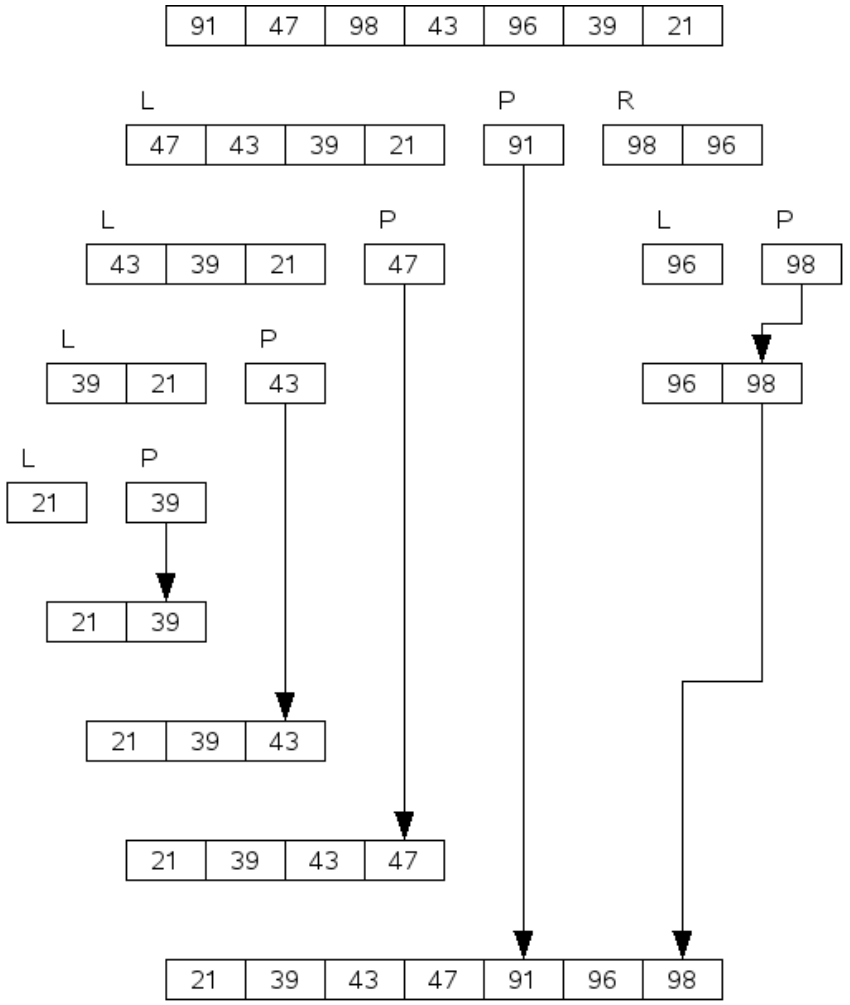


Quicksort

Quicksort adalah algoritma sorting yang berdasarkan pembandingan dengan metoda *divide-and-conqueror* dan, pada implementasi yang sederhana, bukanlah algoritma sorting yang *stable*.

Pada kasus terburuk *running time* dari algoritma adalah $O(n^2)$ untuk input berupa array dengan jumlah n.

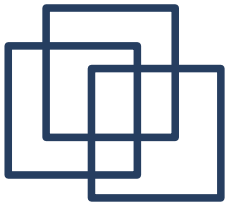
Quicksort



P : pivot
L : *left*, sublist dengan elemen ≤ P
R : *right*, sublist dengan elemen > P

Worst Case Running Time :
$\Theta(n^2)$

* Pivot dipilih secara random



Quicksort

Algoritma :

- ***Divide***, bagi list input N menjadi dua bagian L dan R, berdasarkan sebuah pivot P, dimana elemen-elemen pada sublist L memiliki key yang lebih kecil atau sama dengan P, dan elemen-elemen pada sublist R memiliki key yang lebih besar dari P.

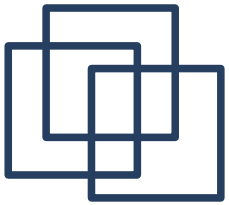
- ***Conqueror***, urutkan kedua sublist L dan R, dengan memanggil quicksort secara rekursif.

- ***Combine***, gabungkan kembali sublist L, pivot P, dan sublist R kembali menjadi sebuah list N.

Analisis Quicksort

quicksort(data, n)	Cost	Times
if (n <= 0)	c_1	1
return data;	c_2	1
pivot ← data;	c_3	1
data ← data.next;	c_4	1
while (data <> NIL) do	c_5	n
if (data.key > pivot.key) then	c_6	n - 1
list_add(right, data);	c_7	n - 1
nright = nright + 1;	c_8	n - 1
else list_add(left, data);	c_9	n - 1
nleft = nleft + 1;	c_{10}	n - 1
data ← data.next;	c_{11}	n - 1
left ← quicksort(left, nleft);	c_{12}	T(0)
right ← quicksort(right, nright);	c_{13}	T(n-1)
result ← add(result, left);	c_{14}	1
result ← add(result, pivot);	c_{15}	1
result ← add(result, right);	c_{16}	1
return result;	c_{17}	1

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$
$$= \Theta(n^2)$$



Analisis Kompleksitas Algoritma (2)

Worst case running time dari setiap algoritma :

- Merge sort + Multiway Merge

$$T(n) = \Theta(n \lg n) + \Theta(n)$$

- Binary sort + Multiway Merge

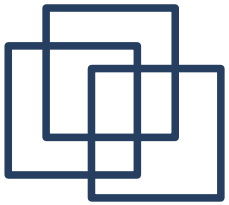
$$T(n) = \Theta(n \lg n) + \Theta(n)$$

- Quicksort + Multiway Merge

$$T(n) = \Theta(n^2) + \Theta(n)$$

- Bucket sort

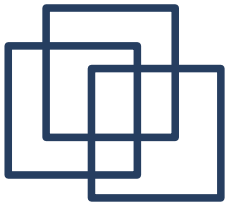
$$T(n) = \Theta(n \lg n) + \Theta(n)$$



Analisis Frekuensi Baca/Tulis

Analisis mengenai jumlah baca dan tulis yang dilakukan pada eksternal memori (*hard disk*).

Penghitungan baca/tulis mengacu pada pembacaan dan penulisan sebuah record dari/ke sebuah file.



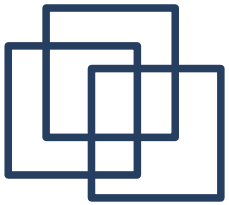
Analisis Frekuensi Baca/Tulis (2)

Contoh kasus :

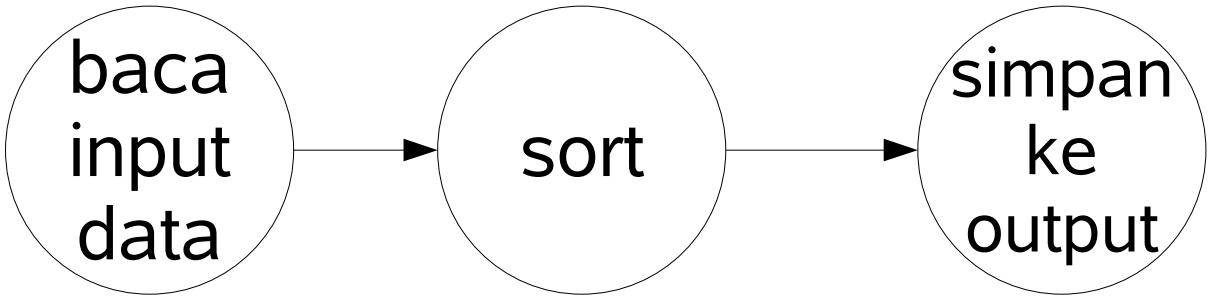
Input data {15, 14, 13, 12, 11}

Asumsi :

- Input berada di dalam file.
- Maksimum lima record yang dapat disimpan dalam memori internal komputer.

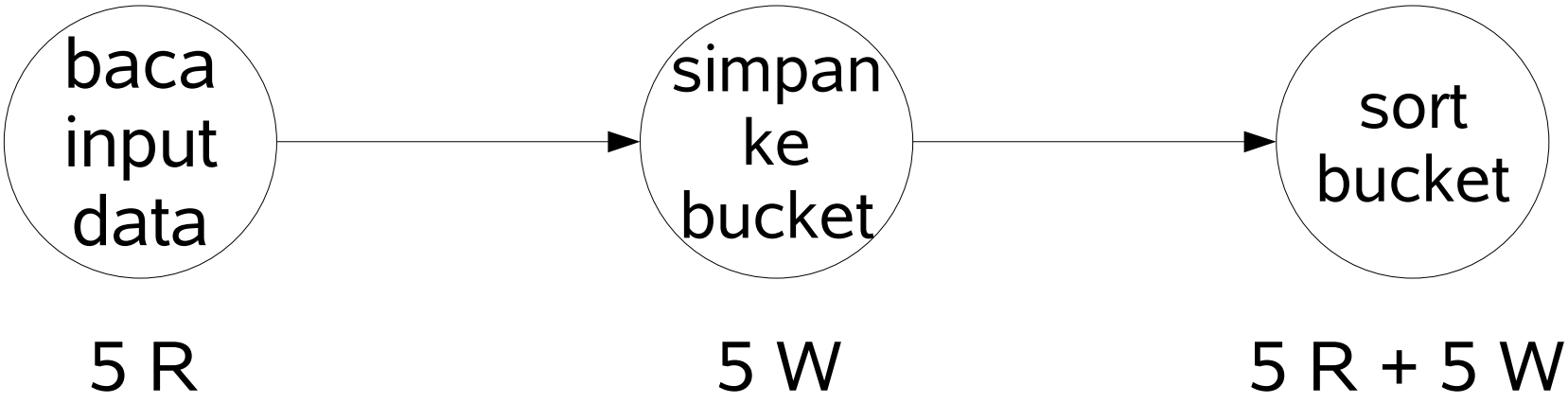


Analisis Frekuensi Baca/Tulis pada Merge sort, Binary sort, dan Quicksort

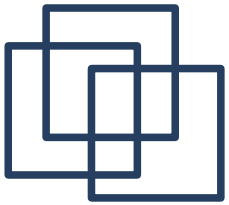


Total : 5R + 5W

Analisis Frekuensi Baca/Tulis pada Bucket sort



Total : 10R + 10W

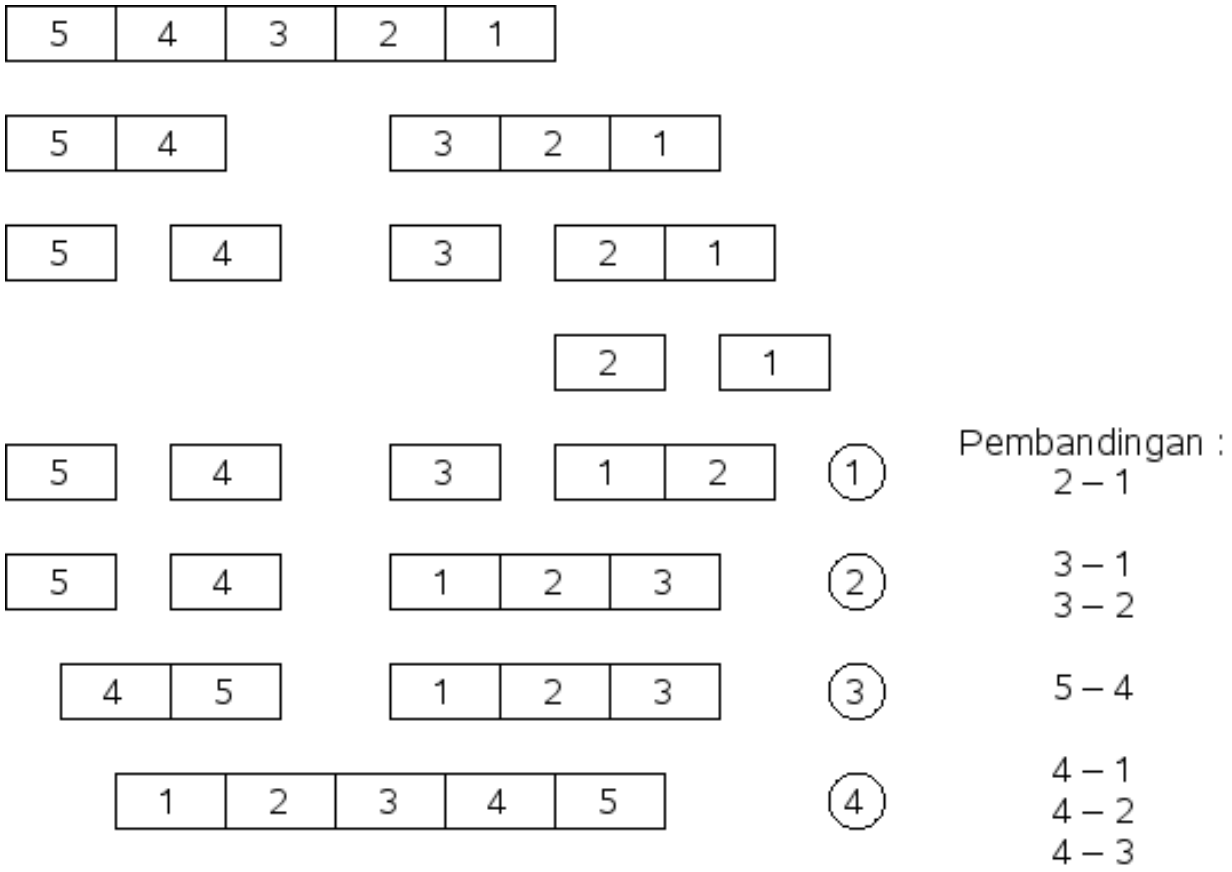


Analisis Jumlah Pembandingan Pada Algoritma

Analisis pembandingan pada algoritma yaitu menghitung jumlah pembandingan antara sebuah *key* dengan *key* yang lainnya yang dilakukan oleh algoritma.

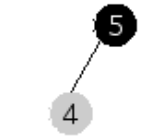
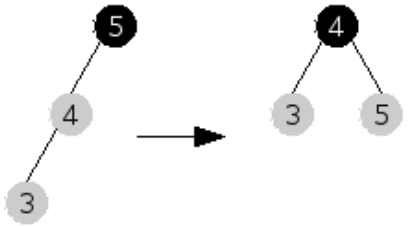
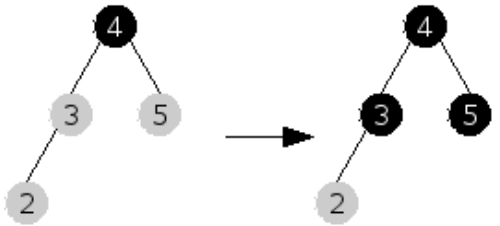
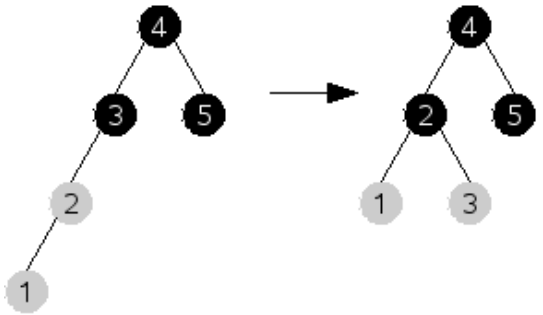
Pembandingan selain antara *key* dengan *key* tidak dihitung disini, misalnya pembandingan variabel dengan konstanta (x = 1), atau variabel dengan variabel (x > y) .

Analisis Jumlah Pembandingan Pada Algoritma Merge sort



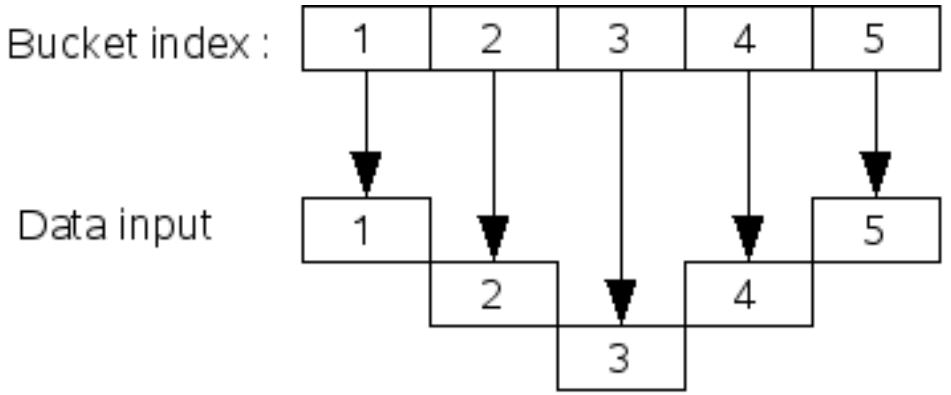
Jumlah pembandingan : 7 kali

Analisis Jumlah Pembandingan Pada Algoritma Binary sort

	Pembandingan
	4 – 5
	3 – 5 3 – 4
	2 – 4 2 – 3
	1 – 4 1 – 3 1 – 2

Jumlah pembandingan : 8 kali

Analisis Jumlah Pembandingan Pada Algoritma Bucket sort

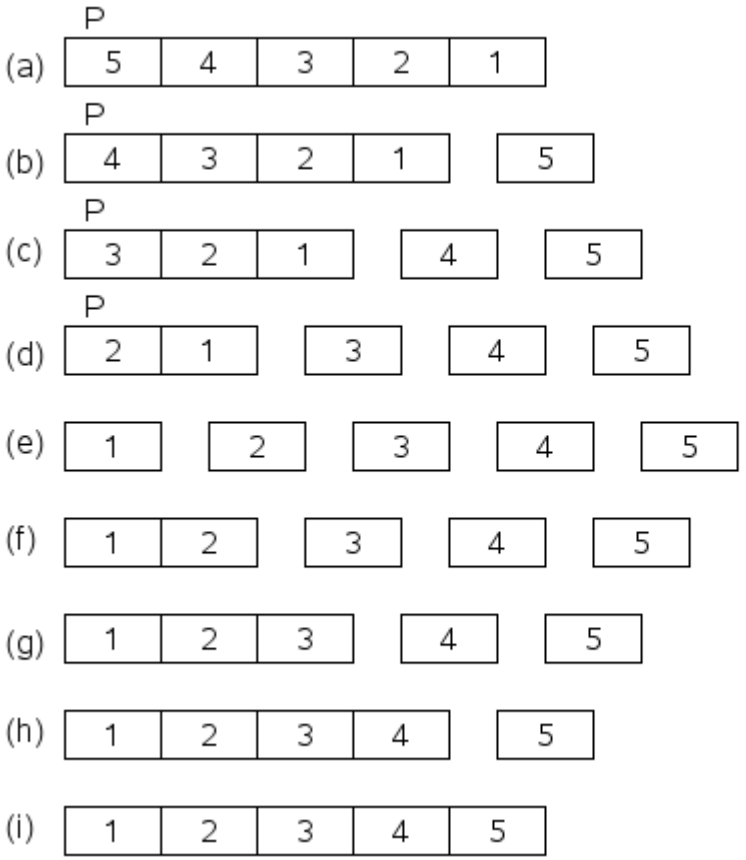


Jumlah pembandingan : 0 kali

Karena sifat algoritma bucket sort yang menyimpan data ke bucket berdasarkan data itu sendiri sebagai *key*.

Contoh dalam pseudo code : bucket[data[i]]

Analisis Jumlah Pembandingan Pada Algoritma Quicksort



Pembandingan:
(a) 5 – 4
5 – 3
5 – 2
5 – 1

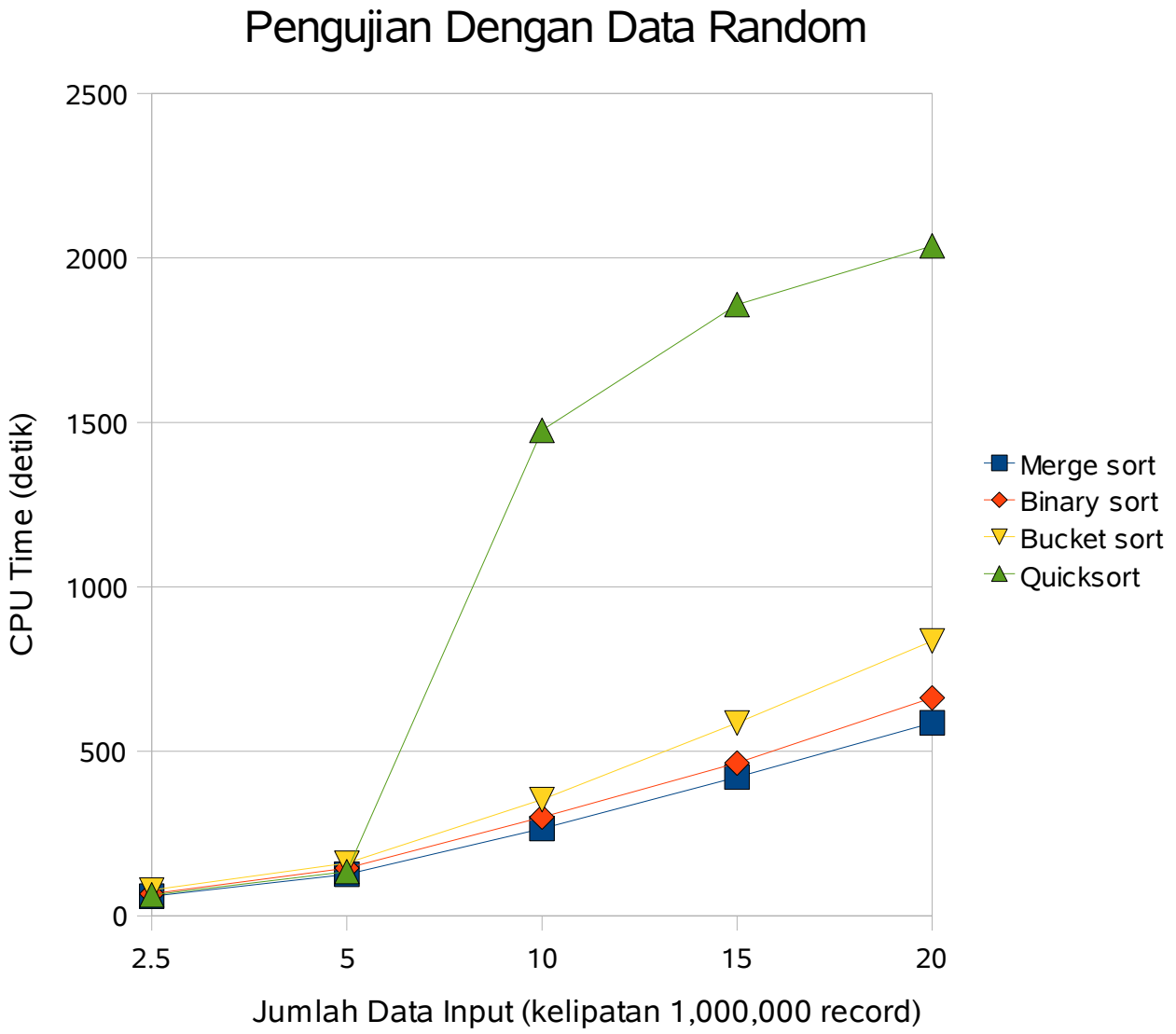
(b) 4 – 3
4 – 2
4 – 1

(c) 3 – 2
3 – 1

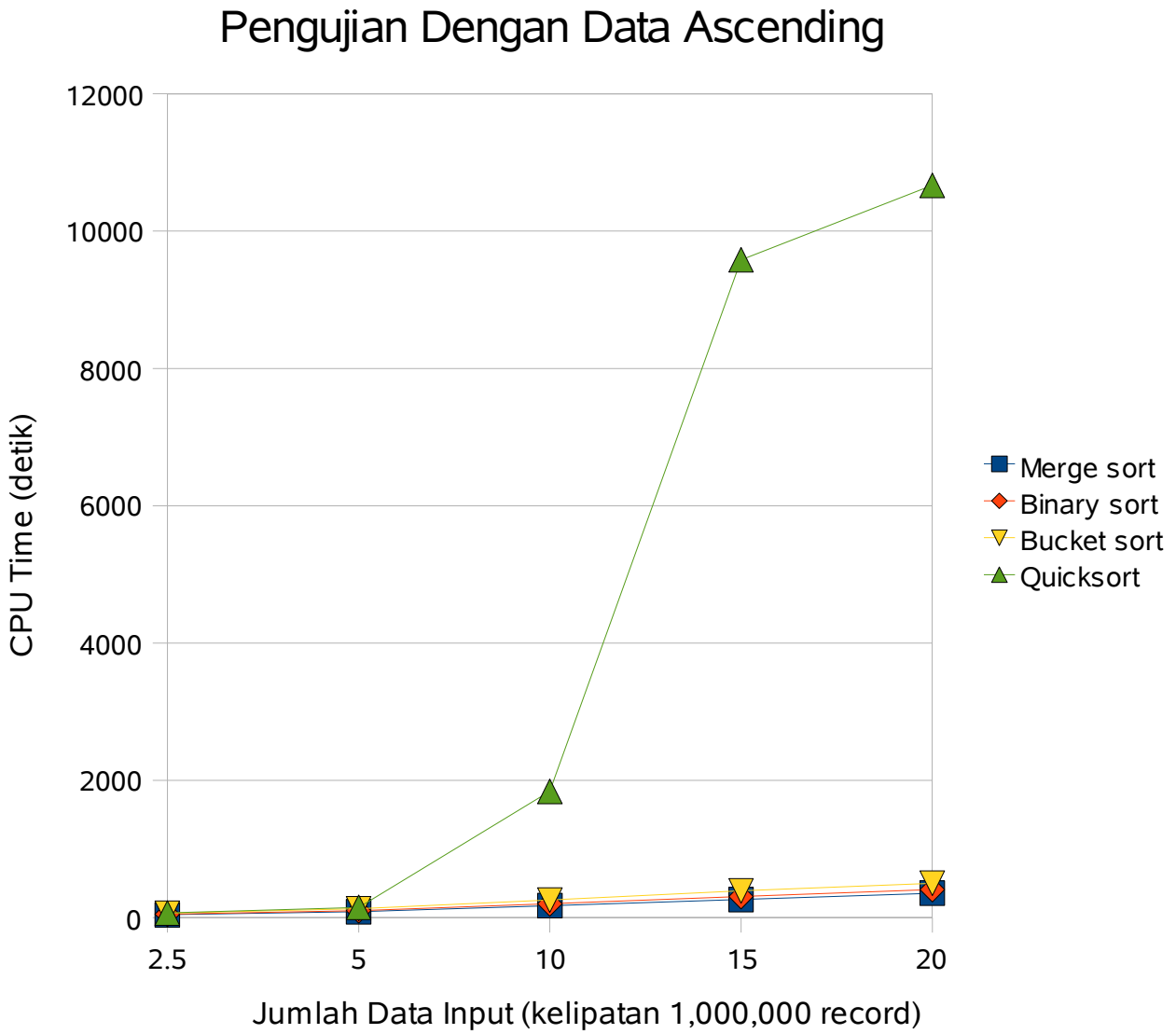
(d) 2 - 1

Jumlah Pembandingan : 10 kali

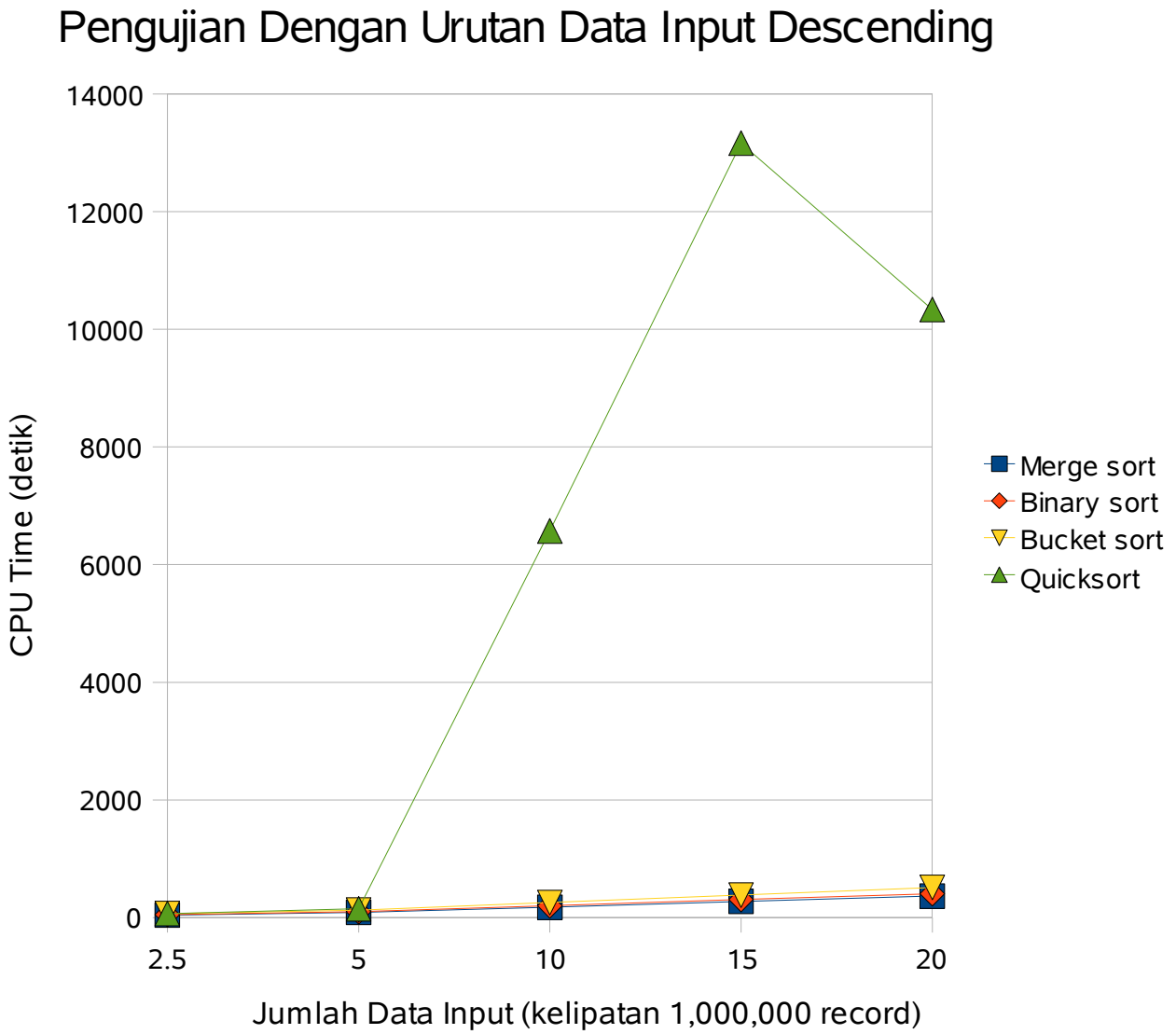
Hasil Uji
Dengan Data Random

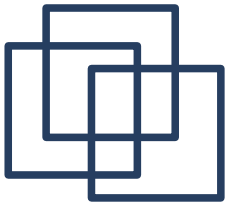


Hasil Uji
Dengan Data Ascending



Hasil Uji
Dengan Data Descending





Kesimpulan

- *Bucket sort* memiliki kelemahan dalam frekuensi baca/tulis yang lebih banyak dari algoritma sorting lainnya.
- *Quicksort* memiliki kelemahan pada frekuensi pembandingan yang dilakukan lebih banyak, pada kasus terburuk.
- Pemilihan pivot yang random pada *Quicksort* tidak selalu menjamin *running time* yang bagus.
- *Merge sort* memiliki kelebihan pada proses *merge* dari dua *sublist*, yang dapat mengurangi jumlah pembandingan yang dilakukan.
- Merge sort lebih efisien, dalam waktu atau kecepatan pemprosesan, untuk mengurutkan data dengan jumlah besar yang melebihi kapasitas memori internal komputer, dibandingkan dengan algoritma *binary sort*, *bucket sort* dan *quicksort*.