

Struktur Data dan Algoritma

Sorting

Suryana Setiawan, Ruli Manurung & Ade Azurat
(acknowledgments: Denny)

Fasilkom UI



Outline

- Beberapa algoritma untuk melakukan sorting:
 - Bubble sort
 - Selection sort
 - Insertion sort
 - Shell sort
 - Merge sort
 - Quick sort
- Untuk masing–masing algoritma:
 - Ide dasar
 - Contoh eksekusi
 - Algoritma
 - Analisa *running time*/kompleksitas



Sort

- *Sorting* = pengurutan
- *Sorted* = terurut menurut kaidah/aturan tertentu
- Data pada umumnya disajikan dalam bentuk *sorted*.
- Contoh:
 - Nama di buku telpon
 - Kata-kata dalam kamus
 - File-file di dalam sebuah directory
 - Indeks sebuah buku
 - Data mutasi rekening tabungan
 - CD di toko musik
- Bayangkan jika data di atas tidak terurut!

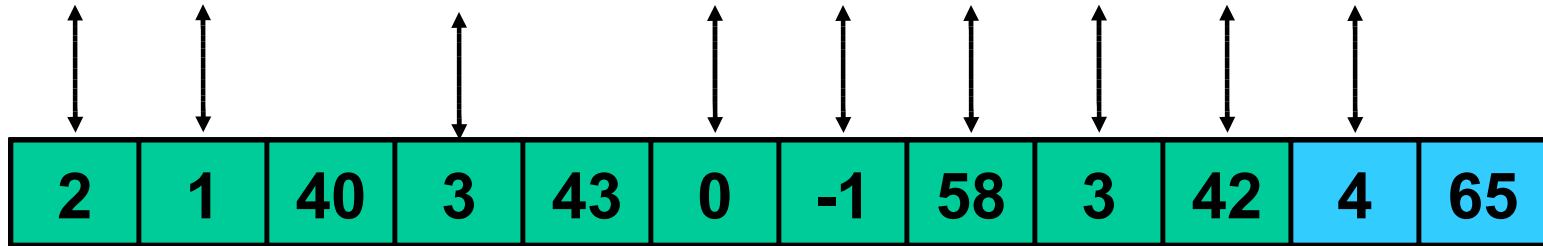


Bubble sort: Ide dasar

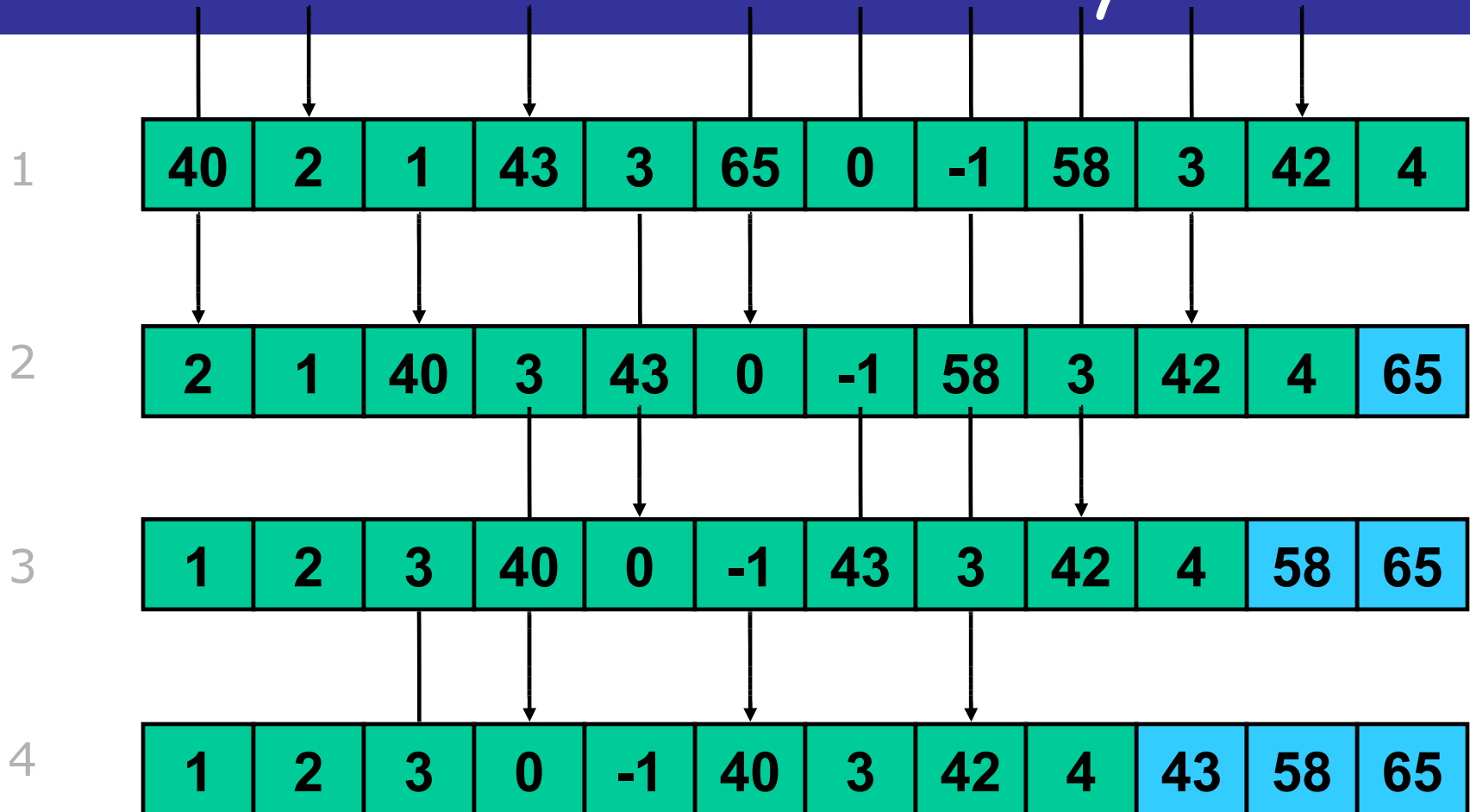
- Bubble = busa/udara dalam air – apa yang terjadi?
 - Busa dalam air akan naik ke atas. Mengapa?
 - Ketika busa naik ke atas, maka air yang di atasnya akan turun memenuhi tempat bekas busa tersebut.
- Pada setiap iterasi, bandingkan elemen dengan sebelahnya: **yang busa naik, yang air turun!**



Bubble sort: Sebuah iterasi



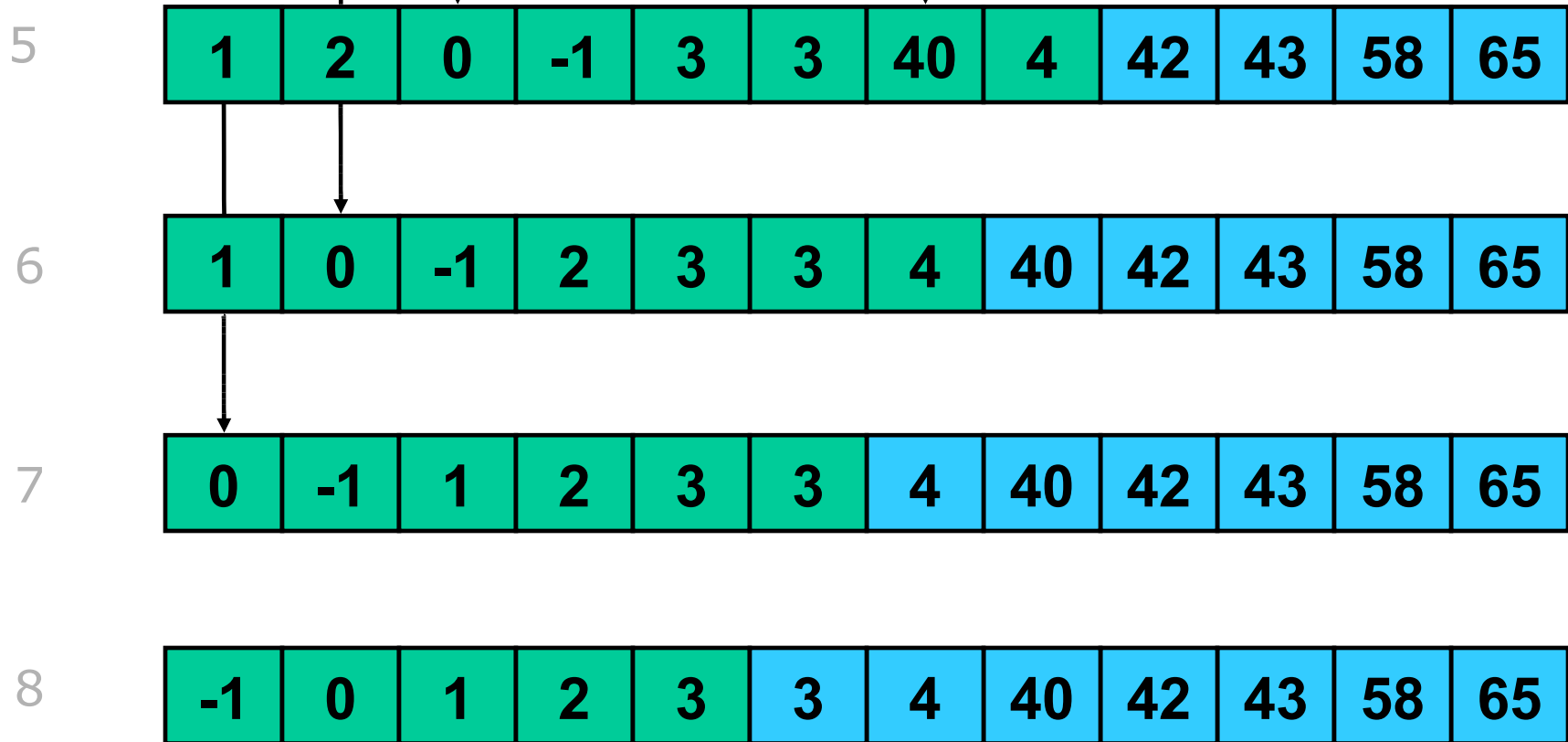
Bubble sort: Iterasi berikutnya



- Perhatikan bahwa pada setiap iterasi, dapat dipastikan satu elemen akan menempati tempat yang benar



Bubble sort: Terminasi algoritma



Berhenti di sini! Mengapa?



Bubble sort: Algoritma

```
void sort(int a[]) throws Exception
{
    for (int i = a.length; i>=0; i--) {
        boolean swapped = false;
        for (int j = 0; j<i; j++) {
            ...
            if (a[j] > a[j+1]) {
                int T = a[j];
                a[j] = a[j+1];
                a[j+1] = T;
                swapped = true;
            }
            ...
        }
        if (!swapped)
            return;
    }
}
```



Bubble sort

- Running time:
 - Worst case: $O(n^2)$
 - Best case: $O(n)$ – kapan? Mengapa?
- Variasi:
 - bi-directional bubble sort
 - original bubble sort: hanya bergerak ke satu arah
 - bi-directional bubble sort bergerak dua arah (bolak balik)

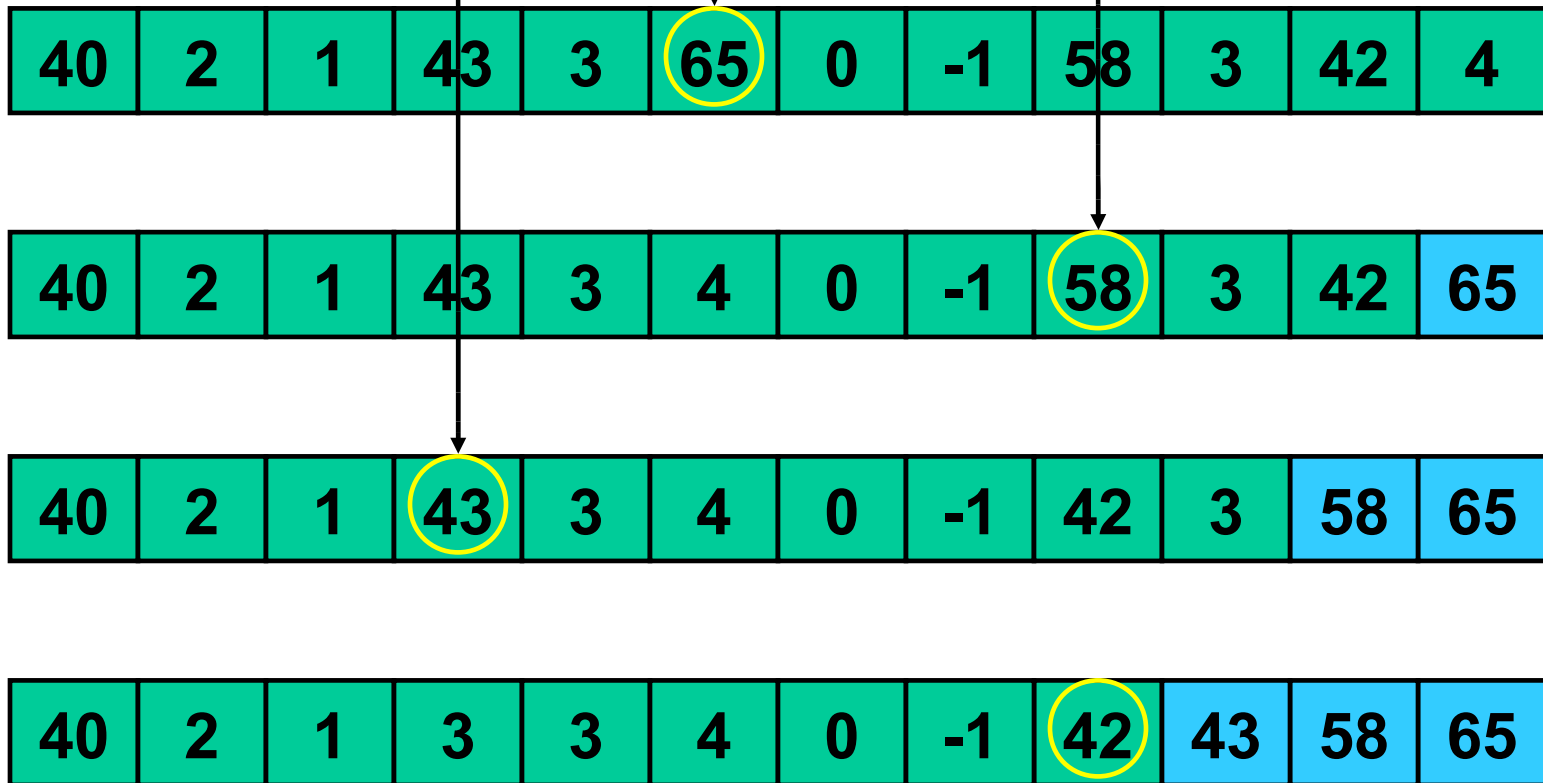


Selection sort: Ide dasar

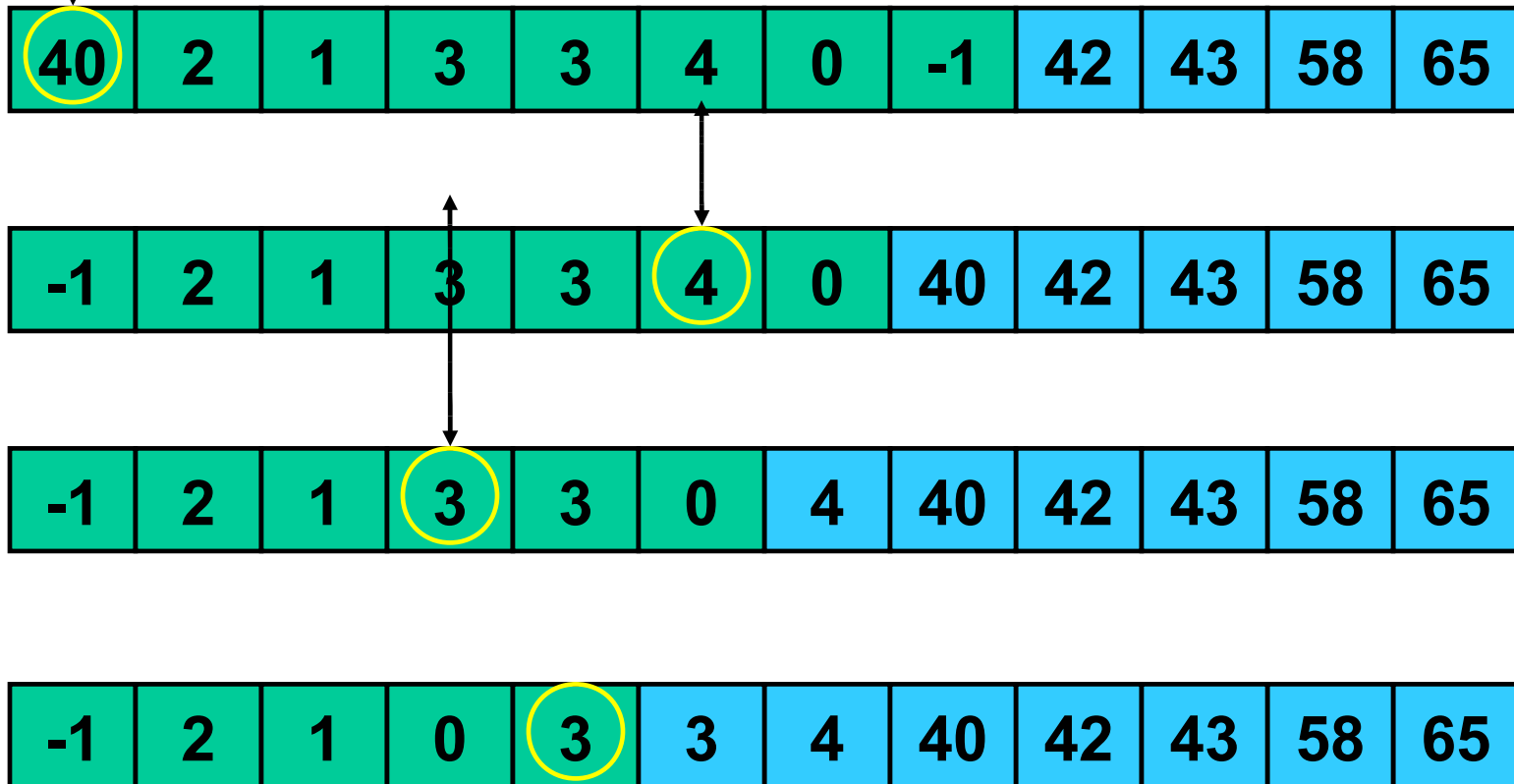
- Kondisi awal:
 - Unsorted list = data
 - Sorted list = kosong
- Ambil yang terbaik (**select**) dari unsorted list, **tambahkan** di belakang sorted list.
- Lakukan terus sampai unsorted list habis.



Selection sort: Contoh



Selection sort: Contoh (lanj.)



Selection sort: Contoh (lanj.)

↓

-1	2	1	0	3	3	4	40	42	43	58	65
----	---	---	---	---	---	---	----	----	----	----	----

-1	0	1	2	3	3	4	40	42	43	58	65
----	---	---	---	---	---	---	----	----	----	----	----

-1	0	1	2	3	3	4	40	42	43	58	65
----	---	---	---	---	---	---	----	----	----	----	----

-1	0	1	2	3	3	4	40	42	43	58	65
----	---	---	---	---	---	---	----	----	----	----	----

-1	0	1	2	3	3	4	40	42	43	58	65
----	---	---	---	---	---	---	----	----	----	----	----



Selection sort: Algoritma

```
void sort(int a[]) throws Exception
{
    for (int i = 0; i < a.length; i++)
        int min = i;

        for (int j = i + 1; j < a.length; j++)
            if (a[j] < a[min])
                min = j;

        int T = a[min];
        a[min] = a[i];
        a[i] = T;
    }
}
```

Cari elemen terkecil dari unsorted list.

Pindahkan ke akhir sorted list.



Selection sort: Analisis

- Running time:
 - Worst case: $O(n^2)$
 - Best case: $O(n^2)$
- Berdasarkan analisis *big-oh*, apakah selection sort lebih baik dari bubble sort?
- Apakah running time yang *sebenarnya* merefleksikan analisis tersebut?

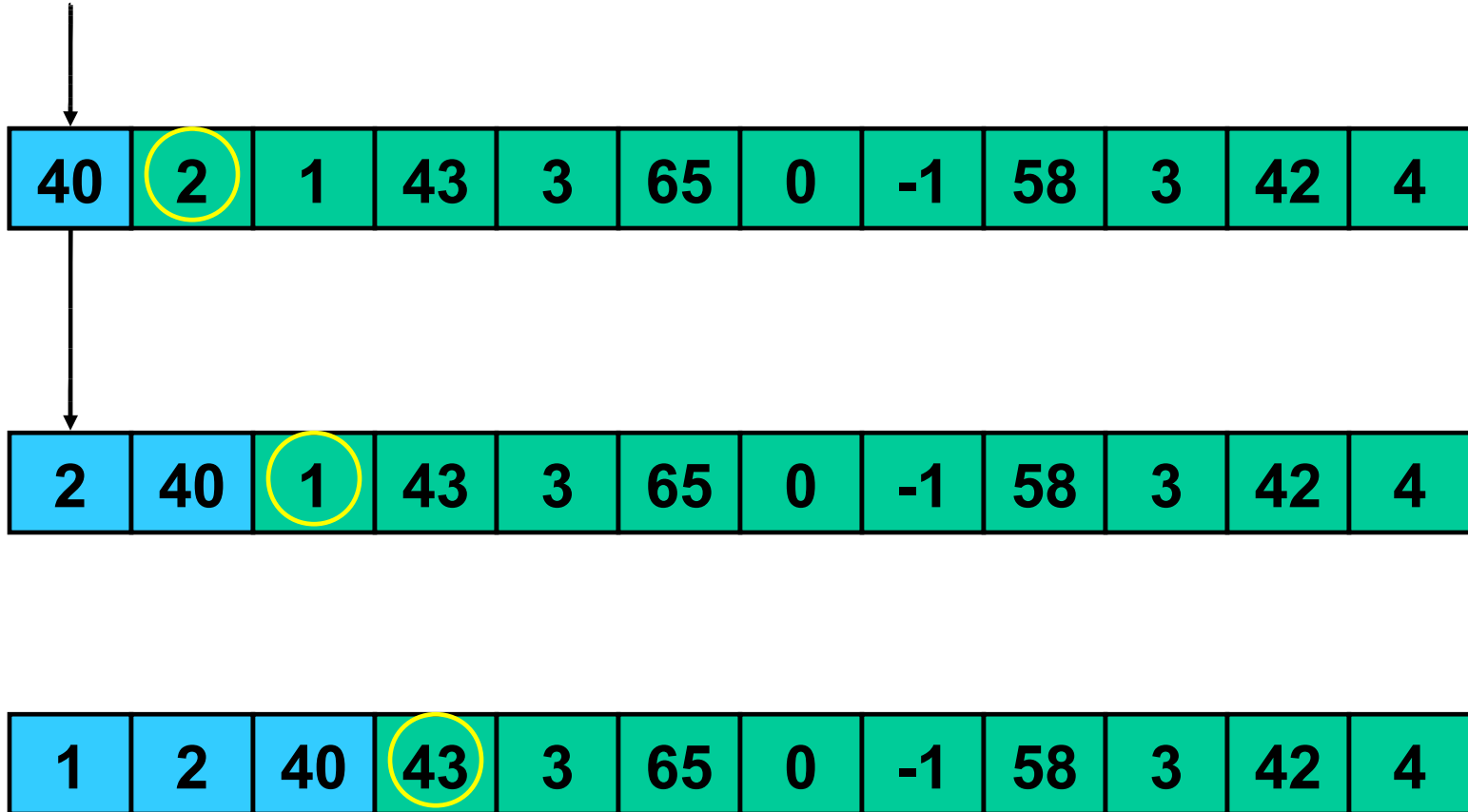


Insertion sort: Ide dasar

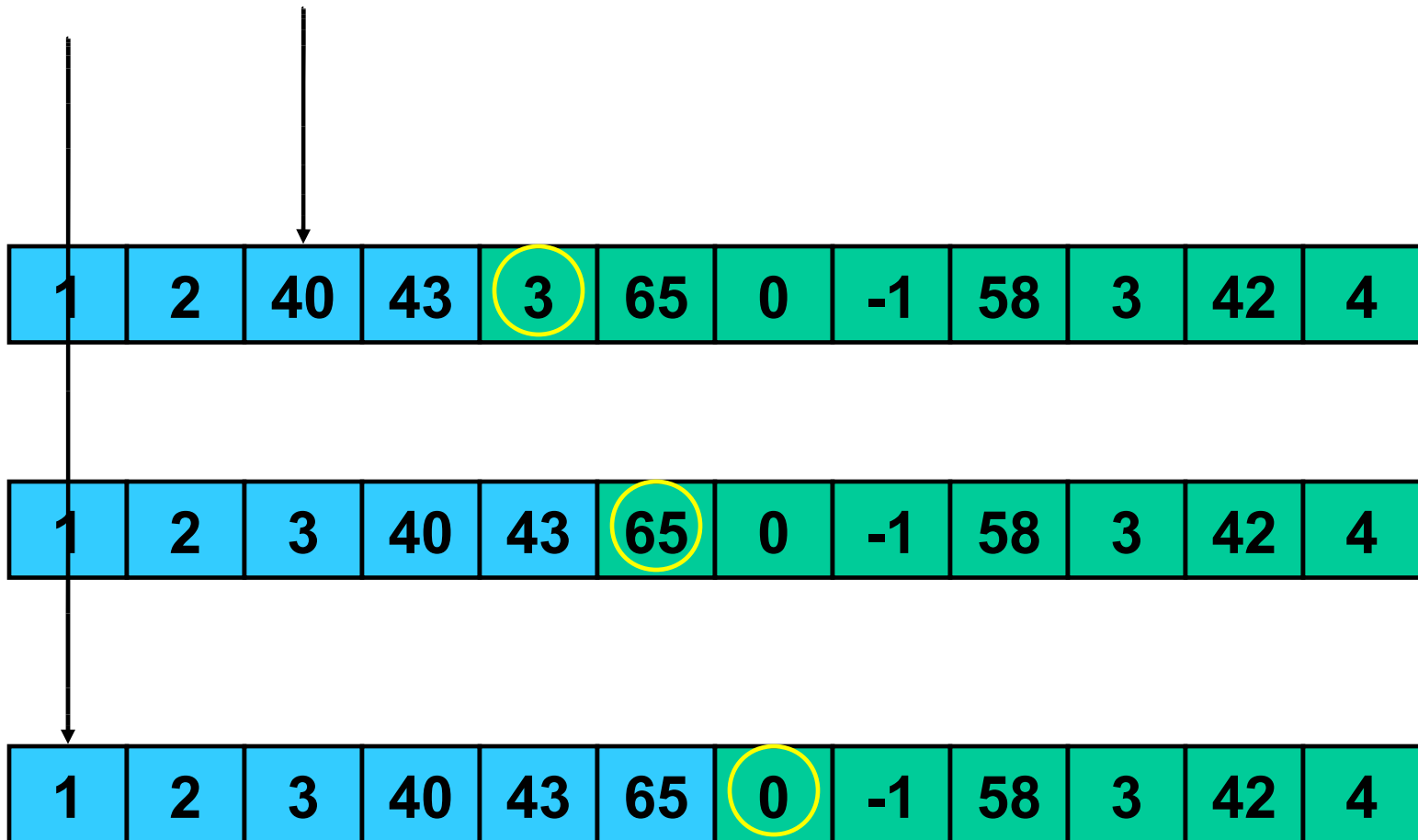
- Kondisi awal:
 - Unsorted list = data
 - Sorted list = kosong
- Ambil **sembarang** elemen dari unsorted list, sisipkan (**insert**) pada posisi yang benar dalam sorted list.
- Lakukan terus sampai unsorted list habis.
- Bayangkan anda mengurutkan kartu.



Insertion sort: Contoh



Insertion sort: Contoh (lanj.)



Insertion sort: Contoh (lanj.)

1	2	3	40	43	65	0	-1	58	3	42	4
---	---	---	----	----	----	---	----	----	---	----	---

0	1	2	3	40	43	65	-1	58	3	42	4
---	---	---	---	----	----	----	----	----	---	----	---

-1	0	1	2	3	40	43	65	58	3	42	4
----	---	---	---	---	----	----	----	----	---	----	---



Insertion sort: Contoh (lanj.)

-1	0	1	2	3	40	43	58	65	3	42	4
----	---	---	---	---	----	----	----	----	---	----	---

-1	0	1	2	3	3	40	43	58	65	42	4
----	---	---	---	---	---	----	----	----	----	----	---

-1	0	1	2	3	3	40	42	43	58	65	4
----	---	---	---	---	---	----	----	----	----	----	---

-1	0	1	2	3	3	4	40	42	43	58	65
----	---	---	---	---	---	---	----	----	----	----	----



Insertion sort: Algoritma

- Insertion sort untuk mengurutkan array integer

```
public static void insertionSort (int[] a)
{
    for (int ii = 1; ii < a.length; ii++) {
        int jj = ii;
        while (( jj > 0) && (a[jj] < a[jj - 1]
            int temp = a[jj];
            a[jj] = a[jj - 1];
            a[jj - 1] = temp;
            jj--;
        }
    }
}
```

Ambil
elemen
pertama
dalam
unsorted

list.

Sisipkan ke
dalam sorted
list.

- Perhatikan: ternyata nilai di `a[jj]` selalu sama \Rightarrow kita dapat melakukan efisiensi di

sini!



Insertion sort: Algoritma (modif.)

- Insertion sort yang lebih efisien:

```
public static void insertionSort2 (int[] a)
{
    for (int ii = 1; ii < a.length; ii++) {
        int temp = a[ii];
        int jj = ii;
        while (( jj > 0) && (temp < a[jj - 1])) {
            a[jj] = a[jj - 1];
            jj--;
        }
        a[jj] = temp;
    }
}
```



Insertion sort: Analisis

- Running time analysis:
 - Worst case: $O(n^2)$
 - Best case: $O(n)$
- Apakah *insertion sort* lebih cepat dari *selection sort*?
- Perhatikan persamaan dan perbedaan antara *insertion sort* dan *selection sort*.



Terhantam tembok kompleksitas...

- Bubble sort, Selection sort, dan Insertion sort semua memiliki *worst case* sama: $O(N^2)$.
- Ternyata, untuk algoritma manapun yang pada dasarnya menukar elemen bersebelahan (**adjacent items**), ini adalah “best worst case”: $\Omega(N^2)$
- Dengan kata lain, disebut *lower bound*
- Bukti: Section 8.3 buku Weiss



Shell sort

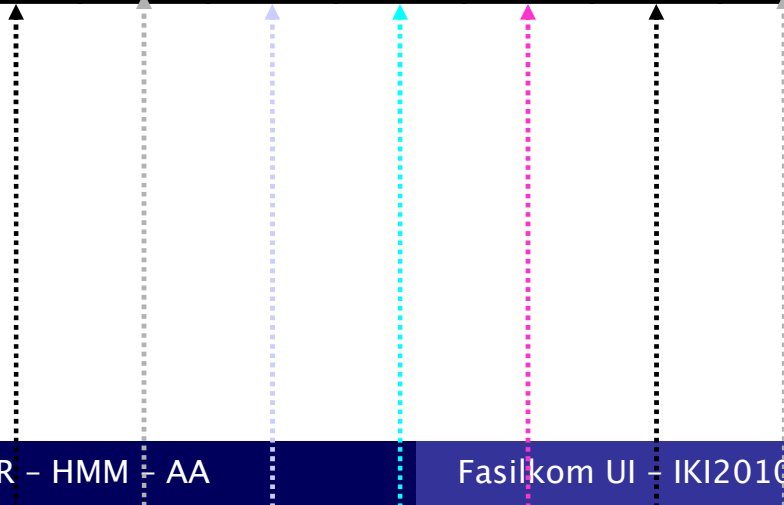
Ide **Donald Shell (1959)**: Tukarlah elemen yang berjarak jauh!

Original:

40	2	1	43	3	65	0	-1	58	3	42	4
----	---	---	----	---	----	---	----	----	---	----	---

5-sort: Sort setiap item yang berjarak 5:

40	2	1	43	3	65	0	-1	58	3	42	4
----	---	---	----	---	----	---	----	----	---	----	---



Shell sort

Original:

40	2	1	43	3	65	0	-1	58	3	42	4
----	---	---	----	---	----	---	----	----	---	----	---

After 5-sort:

40	0	-1	43	3	42	2	1	58	3	65	4
----	---	----	----	---	----	---	---	----	---	----	---

After 3-sort:

2	0	-1	3	1	4	40	3	42	43	65	58
---	---	----	---	---	---	----	---	----	----	----	----

After 1-sort:

-1	0	1	2	3	3	4	40	42	43	58	65
----	---	---	---	---	---	---	----	----	----	----	----



Shell sort: Gap values

- **Gap**: jarak antara elemen yang di-sort.
- Seiring berjalannya waktu, gap diperkecil. Shell sort juga dikenal sebagai **Diminishing Gap Sort**.
- Shell mengusulkan mulai dengan ukuran awal gap = $N/2$, dan dibagi 2 setiap langkah.
- Ada banyak variasi pemilihan gap.



Kinerja Shell sort

N	Insertion sort	Shell sort		
		Shell's	Nilai gap ganjil	Dibagi 2.2
1000	122	11	11	9
2000	483	26	21	23
4000	1936	61	59	54
8000	7950	153	141	114
16000	32560	358	322	269
32000	131911	869	752	575
64000	520000	2091	1705	1249

$O(N^{32})$ $O(N^{54})$

$O(N^{76})$

Ada 3 "nested loop", tetapi Shell sort masih lebih baik dari Insertion sort. Mengapa?



Merge sort: Ide dasar

- Divide and conquer approach
- Idanya:
 - Menggabungkan (**merge**) 2 **sorted array** butuh waktu $O(n)$
 - Membagi sebuah array jadi 2 hanya butuh waktu $O(1)$



merge

- Implementasikan method yang me-*merge* 2 *sorted array* ke dalam 1 *sorted array*!
- Asumsi: a dan b sudah ter-sort, $|c| = |a| + |b|$

```
public static void merge (int[] a, int[] b, int[] c)
{

}
}
```

- Bisakah anda implementasikan tanpa perlu *temporary space*?

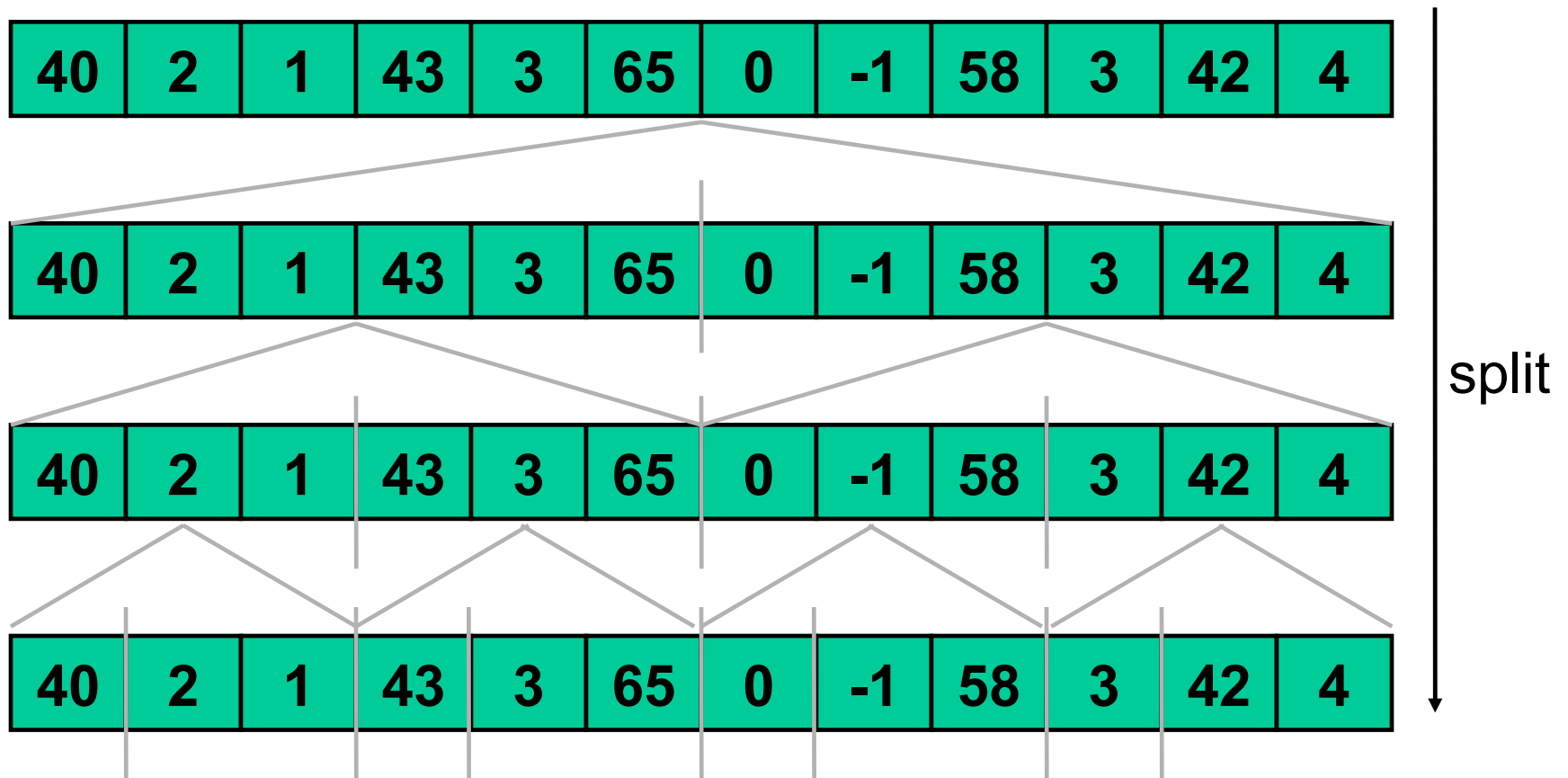


Merge sort: Algoritma rekursif

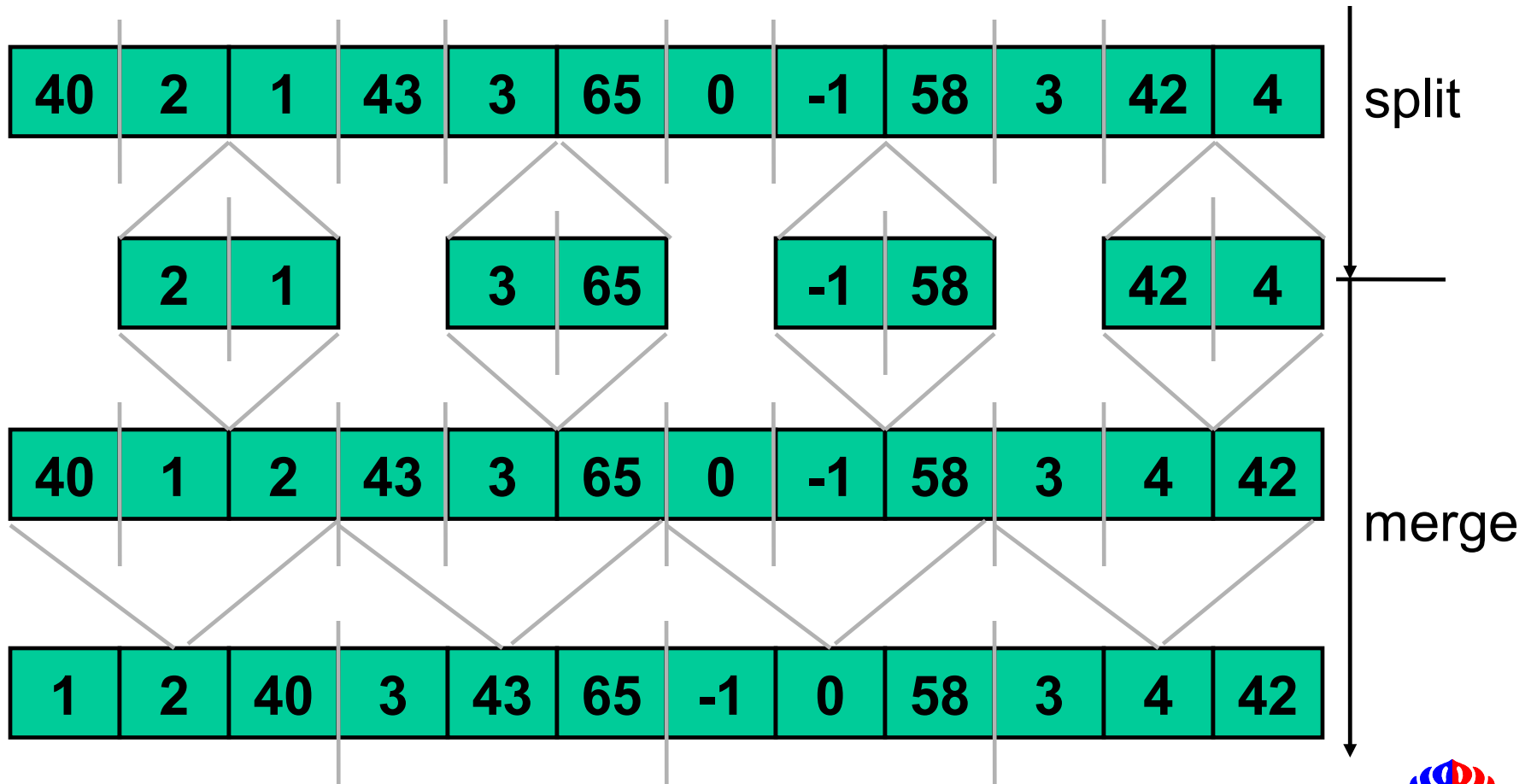
- **Base case:** jika jumlah elemen dalam array yang perlu di-sort adalah 0 atau 1.
- **Recursive case:** secara rekursif, sort bagian pertama dan kedua secara terpisah.
- **Penggabungan:** Merge 2 bagian yang sudah di-sort menjadi satu.



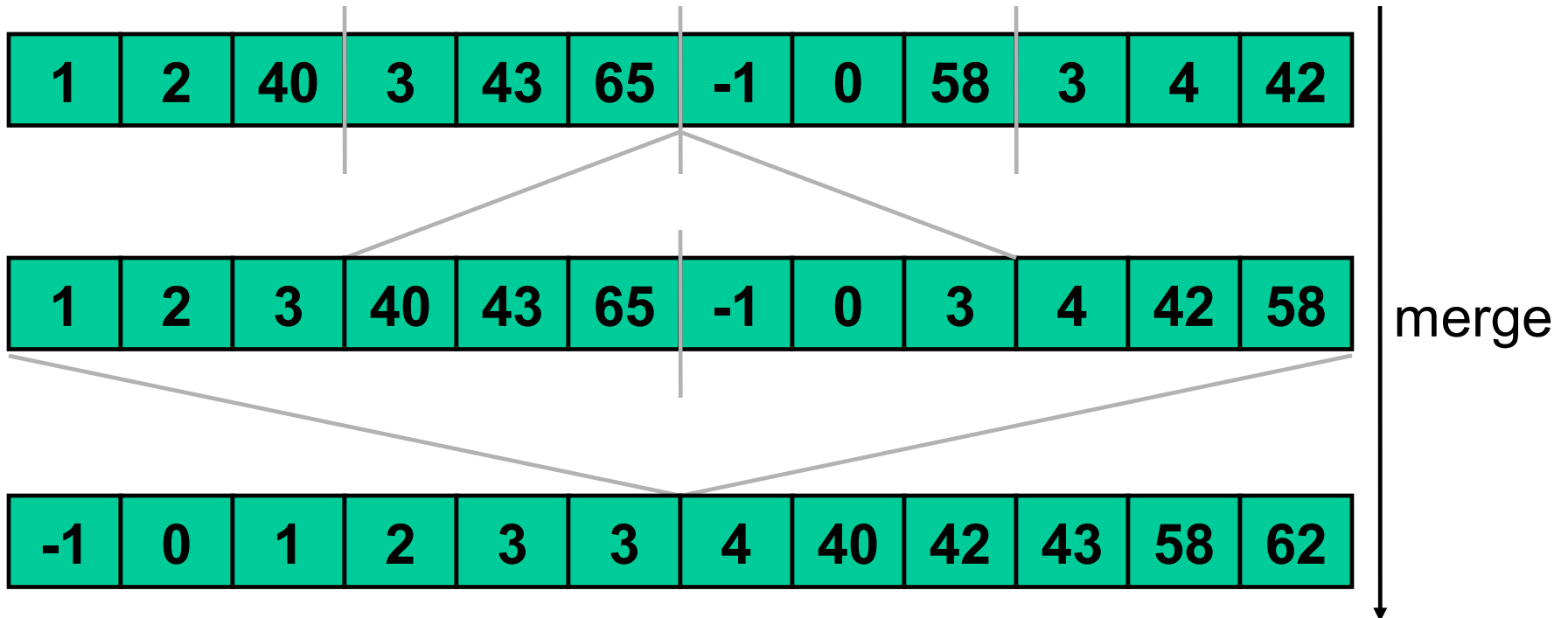
Merge sort: Contoh



Merge sort: Contoh (lanj.)



Merge sort: Contoh (lanj.)



Merge sort: Analisis

- Running Time: $O(n \log n)$
- Mengapa?

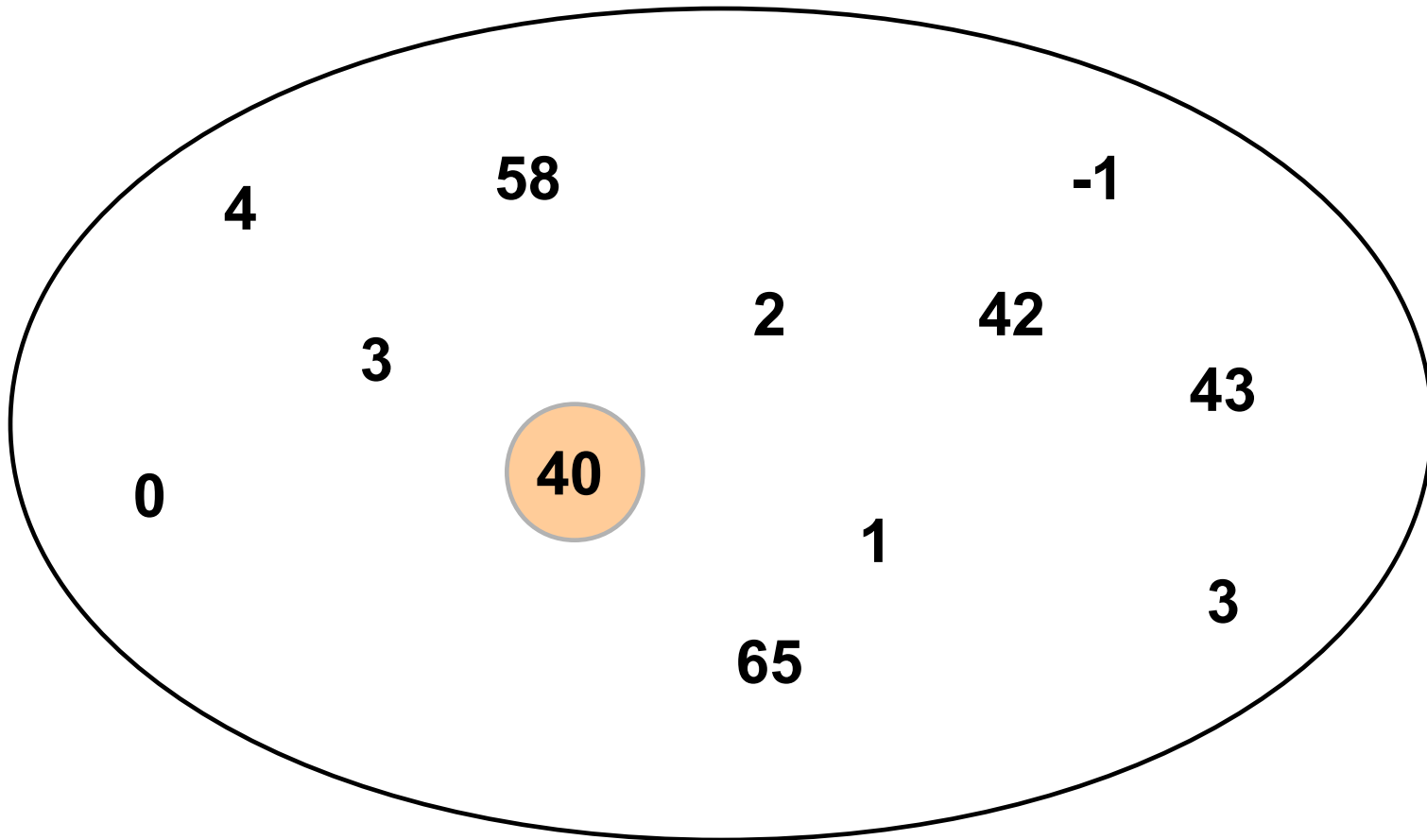


Quick sort: Ide dasar

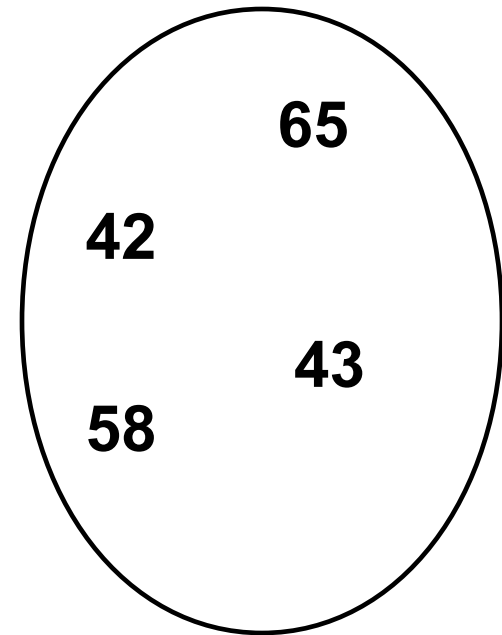
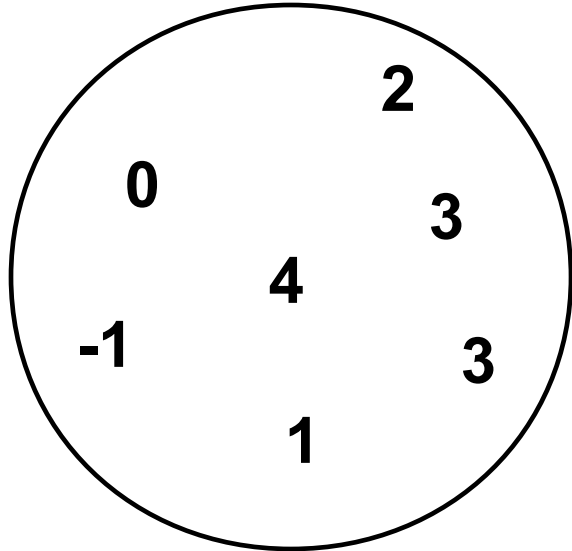
- Divide and conquer approach
- Algoritma *quickSort(S)*:
 - Jika jumlah elemen dalam $S = 0$ atau 1 , return.
 - Pilih sembarang elemen $v \in S$ – sebutlah *pivot*.
 - Partisi $S - \{v\}$ ke dalam 2 bagian:
 - $L = \{x \in S - \{v\} \mid x \leq v\}$
 - $R = \{x \in S - \{v\} \mid x \geq v\}$
 - Kembalikan nilai *quickSort(S)*, diikuti v , diikuti *quickSort(S)*.



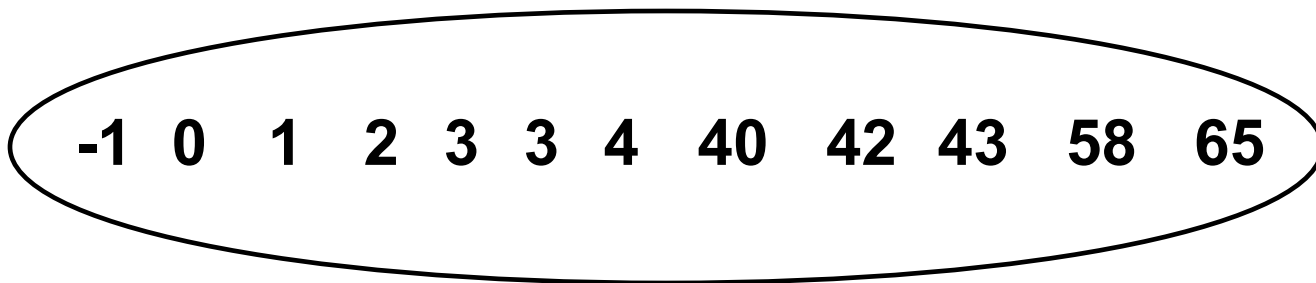
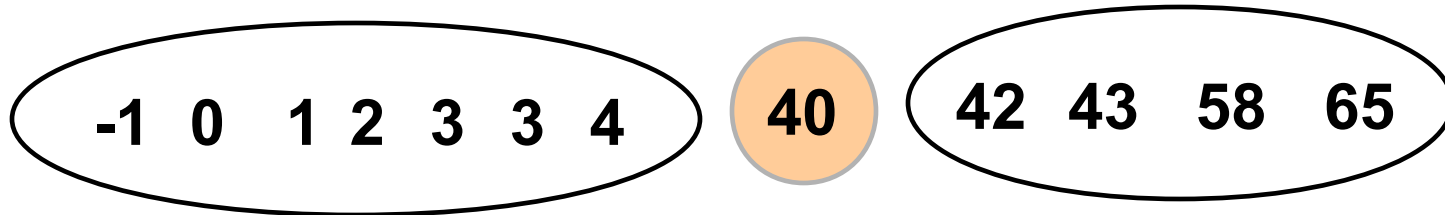
Quick sort: Pilih elemen *pivot*



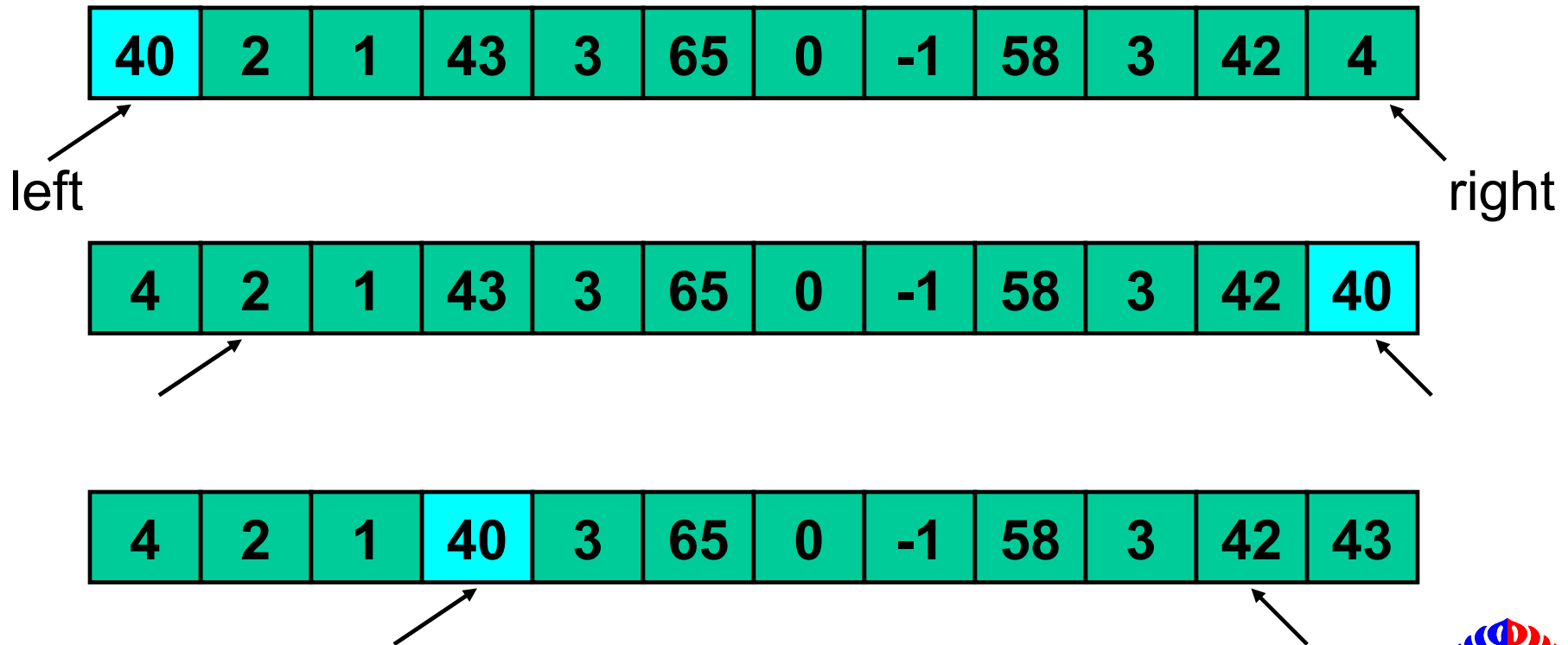
Quick sort: Partition



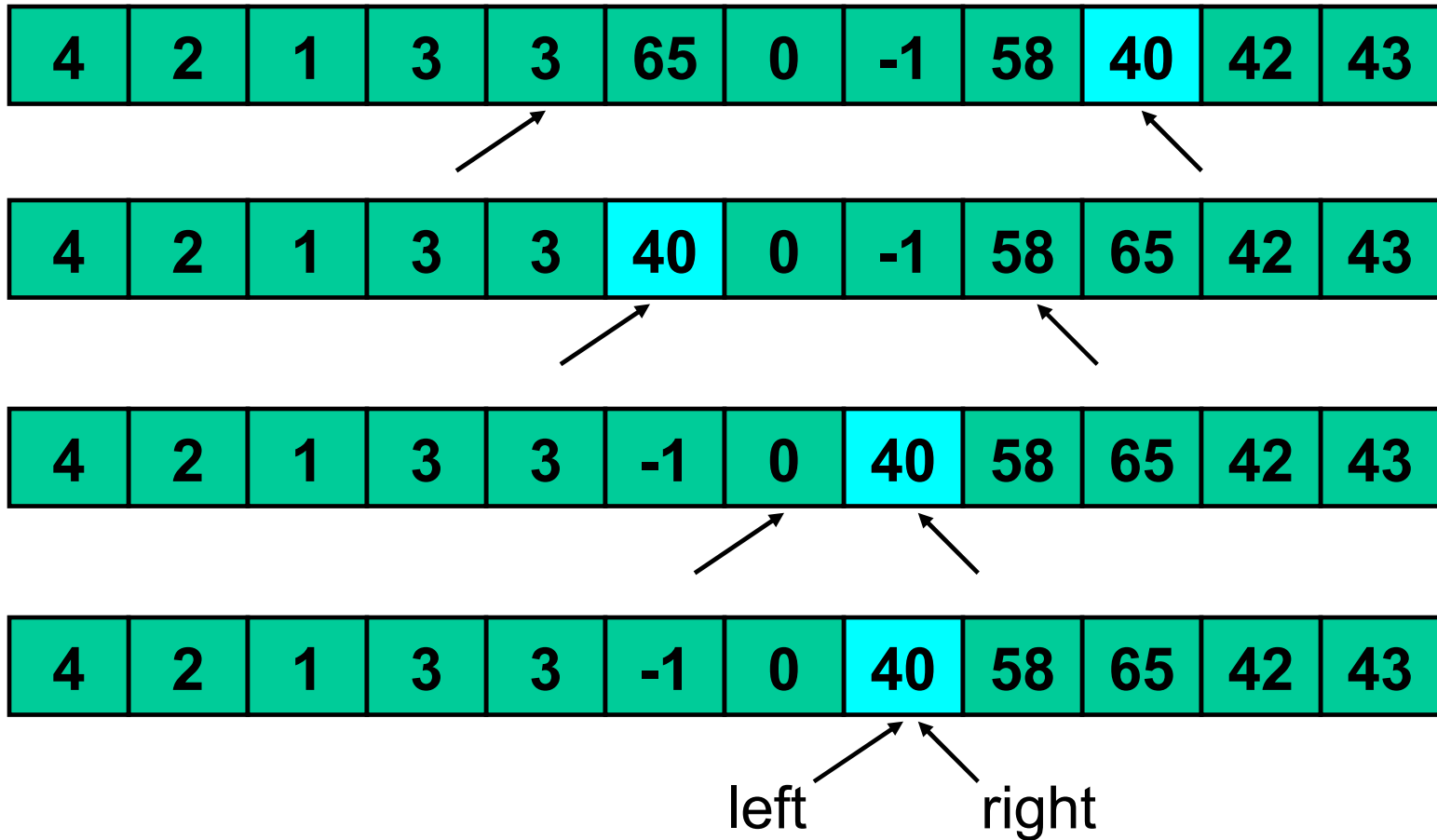
gabungkan



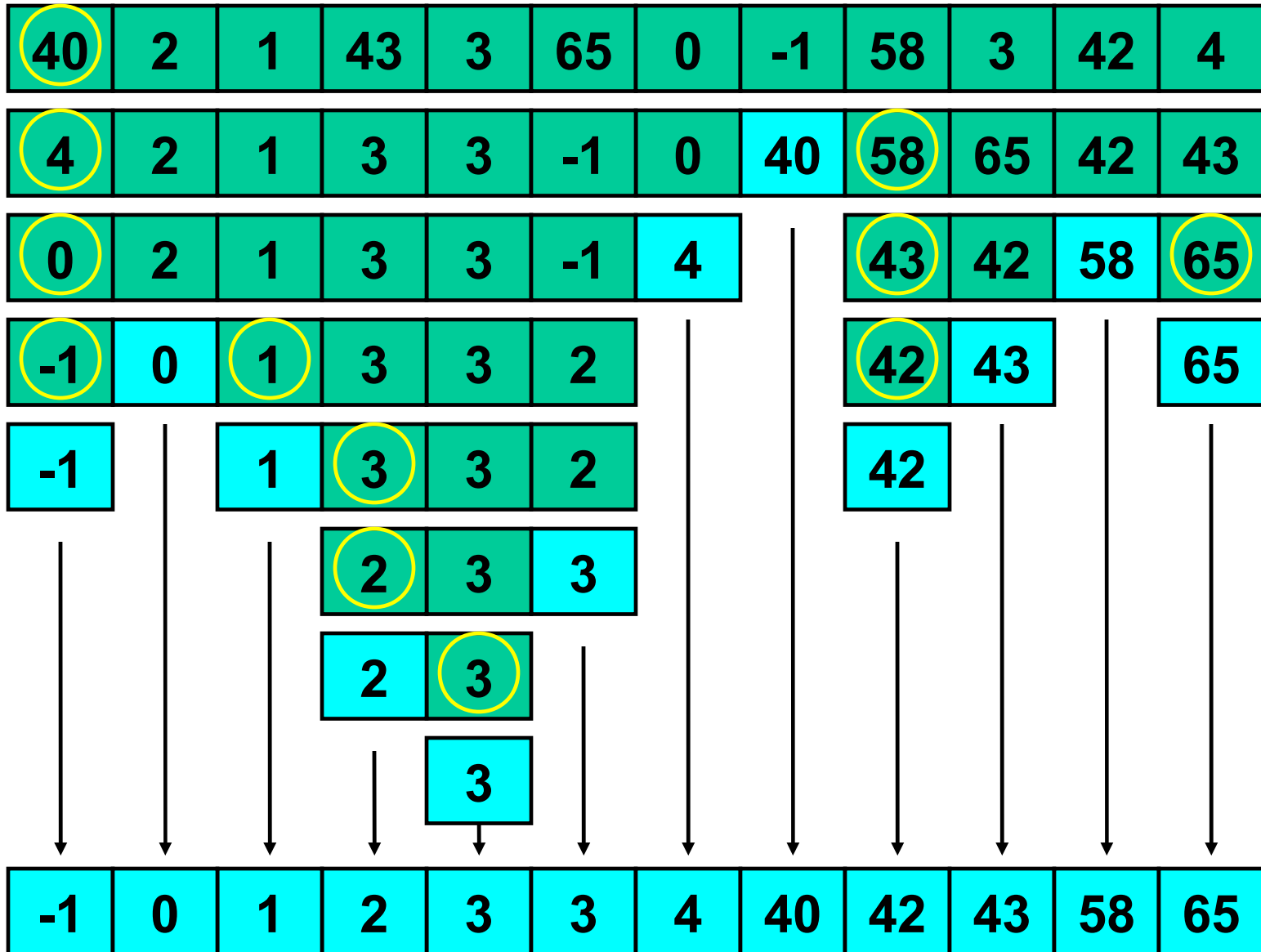
Quick sort: Partition algorithm 1



Quick sort: Partition algorithm 1



Quick sort: Partition algorithm 1



Quick sort: Partition algorithm 2

Original:

40	2	1	43	3	65	0	-1	58	3	42	4
----	---	---	----	---	----	---	----	----	---	----	---

pivot =
40

“buang” pivot sementara

4	2	1	43	3	65	0	-1	58	3	42	40
---	---	---	----	---	----	---	----	----	---	----	----

while < pivot left++

while >= pivot right++

4	2	1	43	3	65	0	-1	58	3	42	40
---	---	---	----	---	----	---	----	----	---	----	----

left

right

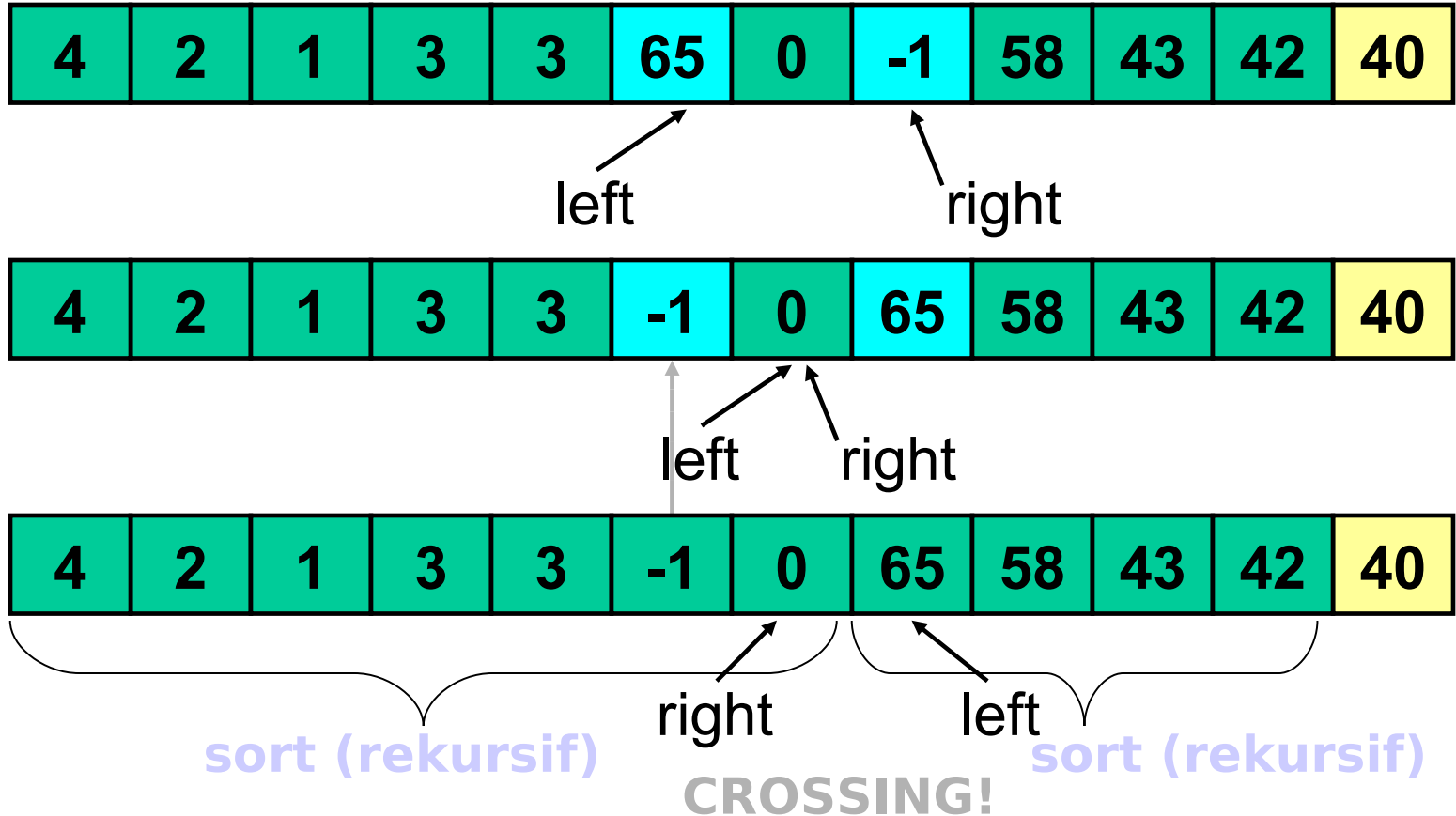
4	2	1	3	3	65	0	-1	58	43	42	40
---	---	---	---	---	----	---	----	----	----	----	----

left

right



Quick sort: Partition algorithm 2



Quick sort: Implementasi

```
static void quickSort(int a[], int low, int high)
{
    if(high <= low) return; // base case
    pivot = choosePivot(a); // select "best" pivot

    int i=low, j=high-1;
    swap(a,pivot,a[j]); // move pivot out of the way
    while(i <= j)
    {
        // find large element starting from left
        while(i<high && a[i]<pivot) i++;

        // find small element starting from right
        while(j>low && a[j]>=pivot) j--;

        // if the indexes have not crossed, swap
        if(i>j) swap(a, i, j);
    }
    swap(a,i,high-1); // restore pivot
    quickSort(a,low,i-1); // sort small elements
    quickSort(a,i+1,high); // sort large elements
}
```



Quick sort: Analisis

- Proses **partitioning** butuh waktu: $O(n)$
- Proses **merging** butuh waktu: $O(1)$
- Untuk setiap **recursive call**, algoritma quicksort butuh waktu: $O(n)$
- Pertanyaannya: berapa **recursive call** dibutuhkan untuk men-sort sebuah array?



Quick sort: Memilih pivot

- Pivot ideal:
 - Elemen media
- Yang biasa dijadikan calon pivot:
 - Elemen pertama
 - Elemen terakhir
 - Elemen di tengah-tengah
 - Median dari ketiga elemen tsb.



Generic sort

- Semua algoritma yang telah kita lihat men-
sort int.
- Bagaimana jika kita ingin sort *String*?
DataMhs? *CD*?
- Apa perlu dibuat method untuk setiap class?
Tidak!
- Agar object bisa di-sort, mereka harus bisa
dibandingkan dengan object lainnya
(*comparable*).
- Solusinya:
 - Gunakan *interface* yang mengandung method
yang dapat membandingkan dua buah object.



Interface `java.lang.comparable`

- Dalam Java, sifat generik “*comparable*” didefinisikan dalam interface `java.lang.comparable`:

```
public interface Comparable
{
    public int compareTo (Object ob);
}
```

- Method `compareTo` returns:
 - <0 : object (**this**) “lebih kecil” dari parameter ‘**ob**’
 - 0 : object (**this**) “sama dengan” parameter ‘**ob**’
 - >0 : object (**this**) “lebih besar” dari

