ALGORITMA PENGURUTAN

Dasar-Dasar Pemrograman

Semester Gasal 2020/2021

Tujuan

- Memahami konsep dasar pengurutan
- Memahami cara kerja berbagai algoritma pengurutan
- Menganalisis kelebihan dan kekurangan setiap jenis algoritma pengurutan
- Mengimplementasikan algoritma-algoritma pengurutan

Algoritma Pengurutan

Pengurutan (Sorting)

- Data pada umumnya disajikan dalam bentuk terurut (*sorted*)
  - Contoh: Kata-kata dalam kamus, nama di buku telepon, indeks sebuah buku, dll.

- **Pengurutan** ➔ Proses penyusunan data yang awalnya acak menjadi teratur menurut aturan tertentu
  - Data acak: `5 6 8 1 3 25 10`
  - *Ascending*: `1 3 5 6 8 10 25`
  - *Descending*: `25 10 8 6 5 3 1`

Algoritma Pengurutan

- Tidak ada algoritma pengurutan terbaik → disesuaikan dengan kondisi data
- Contoh jenis algoritma pengurutan
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Quick Sort
  - Shell Sort
  - Counting Sort
  - Radix Sort
  - Heap Sort

Karakteristik Algoritma Pengurutan

- Modifikasi **in-place** → melakukan **manipulasi langsung pada kontainer** yang isinya akan diurutkan; tidak membutuhkan kontainer tambahan yang ukurannya bergantung pada jumlah data
- **Stable** → dua objek dengan **kunci yang sama** muncul dengan **urutan yang sama** baik sebelum maupun sesudah diurutkan
- **Kompleksitas algoritma (Big-Oh)** → ukuran performansi algoritma pengurutan; mengukur berapa banyak operasi pada algoritma dilakukan jika dibandingkan terhadap tingkat pertumbuhan data
  - $O(1)$
  - $O(\log N)$
  - $O(N)$
  - $O(N^2)$

Latihan

Suatu algoritma memiliki kompleksitas O(N). Saat memroses 100 data, algoritma tersebut memerlukan waktu sebesar 5 detik. Berapa waktu yang diperlukan oleh algoritma tersebut untuk memroses 1000 data?

Bagaimana jika algoritma tersebut memiliki kompleksitas O(log N)?

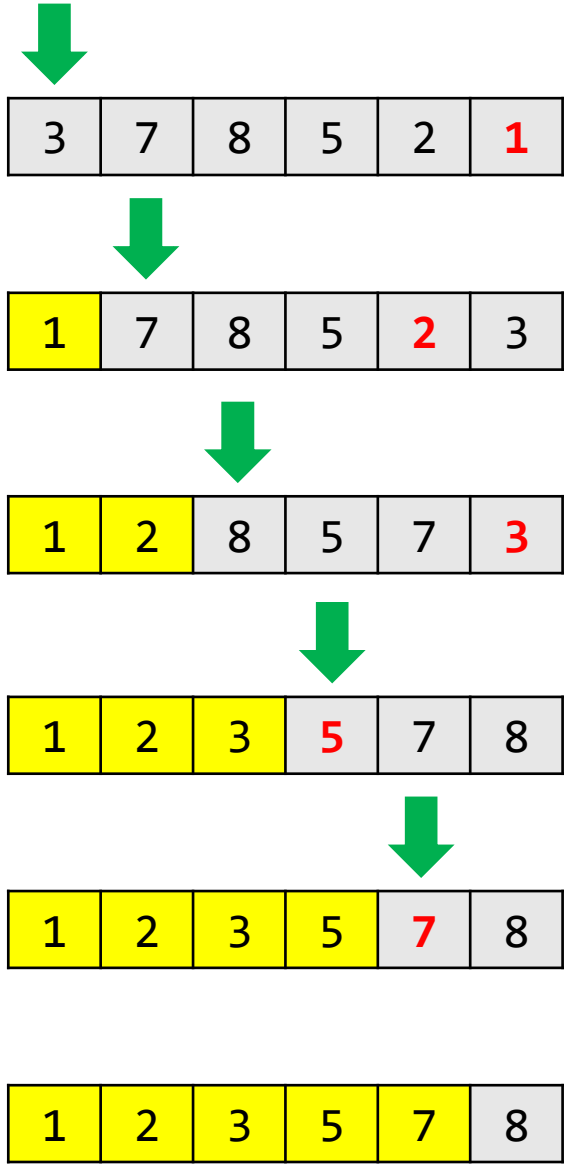
Bagaimana jika algoritma tersebut memiliki kompleksitas $O(N^2)$?

Selection Sort

Selection Sort

- Kombinasi antara *searching* dan *sorting*.
  1. Mencari elemen dengan **nilai terbaik (select)**.
  2. Menempatkan elemen terbaik ke posisi yang seharusnya (tukar dengan elemen yang menempati posisi tersebut saat ini).

```
def selection_sort(arr):
    for i in range(len(arr)-1):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```



Analisis Algoritma Selection Sort

- Dalam kondisi array seperti apapun, algoritma *selection sort* pasti akan tetap melakukan pencarian nilai terbaik di setiap iterasi → tidak ada *best case* maupun *worst case*
- Implementasi sederhana
- Algoritma *selection sort* bersifat **not stable** dan implementasinya **in-place**
- Kompleksitas algoritma selection sort: **$O(N^2)$**
  Ditandai dengan adanya dua blok perulangan yang masing-masing melakukan paling banyak N operasi

Bubble Sort

Bubble Sort

- Diilustrasikan seperti pergerakan busa (*bubble*) dalam air. Busa akan bergerak ke atas karena massa jenisnya lebih ringan dibandingkan air. Membandingkan busa dengan air: yang busa naik, yang air turun.
- Dalam setiap iterasi, elemen dibandingkan dengan elemen sebelahnya: jika posisi elemen yang lebih kecil ada di kanan (di indeks yang lebih besar), maka lakukan pertukaran.
- Di akhir setiap iterasi, posisi elemen terbesar pasti akan ada di sebelah kanan (di indeks yang seharusnya)

Bubble Sort

Iterasi 1

3	7	8	5	2	1
3	7	8	5	2	1
3	7	8	5	2	1
3	7	5	8	2	1
3	7	5	2	8	1
3	7	5	2	1	8

Iterasi 2

3	7	5	2	1	8
3	7	5	2	1	8
3	5	7	2	1	8
3	5	2	7	1	8
3	5	2	1	7	8

Iterasi 3

3	5	2	1	7	8
3	5	2	1	7	8
3	2	5	1	7	8
3	2	1	5	7	8

Iterasi 4

3	2	1	5	7	8
2	3	1	5	7	8
2	1	3	5	7	8

Iterasi 5

2	1	3	5	7	8
1	2	3	5	7	8

```
def bubble_sort(arr):
    for i in range(len(arr)-1):
        for j in range(len(arr)-1-i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Analisis Algoritma Bubble Sort

- **Best case**: ketika kondisi array sudah terurut
- **Worst case**: ketika kondisi array terurut terbalik
- Algoritma *bubble sort* bersifat **stable** dan implementasinya **in-place**
- Kompleksitas algoritma *bubble sort*:
  - Best case: **O(N)**
    Tergantung pada implementasinya; perulangan dalam dapat tidak dilakukan jika kondisi array sudah terurut
  - Worst case: **$O(N^2)$**
    Perulangan dalam (pertukaran elemen yang bersebelahan) pasti dilakukan karena kondisi array terurut terbalik

Insertion Sort

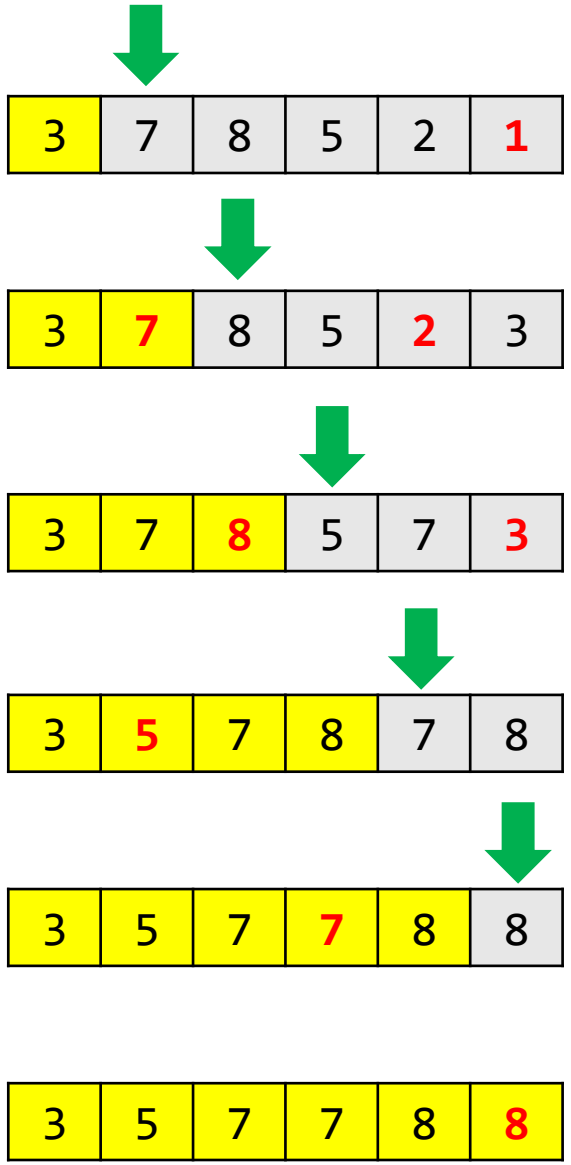
Insertion Sort

- Seperti cara mengurutkan kartu, selembar demi selembar diambil dan **disisipkan (*insert*)** ke tempat yang seharusnya.

1. Pengurutan dilakukan dari elemen ke-2 sampai elemen terakhir, dibandingkan dengan elemen-elemen sebelumnya.
2. Jika ditemukan data yang lebih kecil, maka data yang dibandingkan akan ditempatkan di posisi yang seharusnya.
3. Pada penyisipan elemen, elemen-elemen lain bergeser ke belakang.

Insertion Sort

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j = j - 1
        arr[j+1] = key
```



Analisis Algoritma Insertion Sort

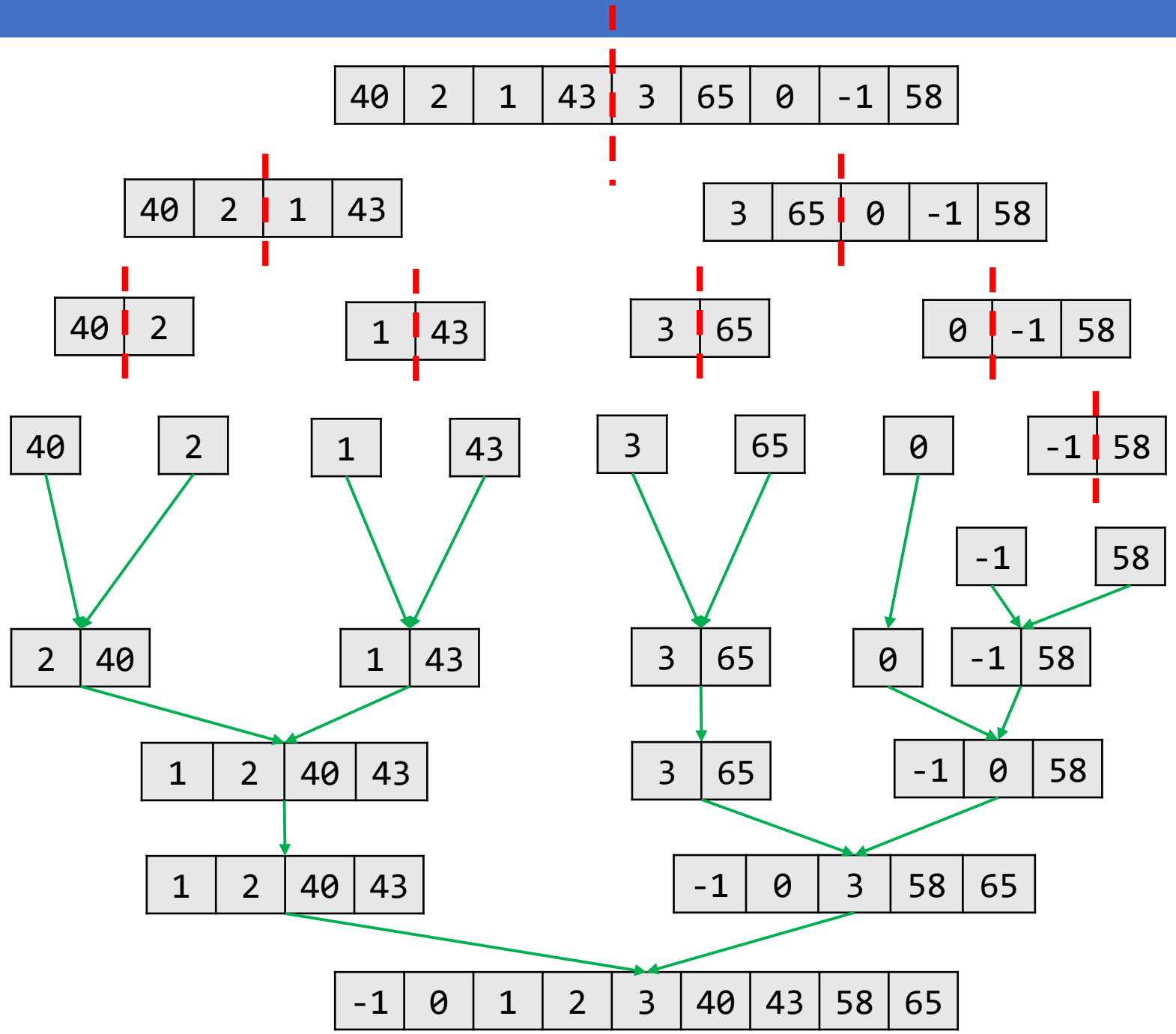
- Efisien untuk mengurutkan array dengan elemen sedikit (**n kecil**)
- **Best case**: ketika kondisi array sudah terurut
- **Worst case**: ketika kondisi array terurut terbalik
- Algoritma *insertion sort* bersifat **stable** dan implementasinya **in-place**
- Kompleksitas algoritma *insertion sort*:
  - Best case: **O(N)**
    Perulangan dalam (`while`) tidak dilakukan jika kondisi array sudah terurut
  - Worst case: **$O(N^2)$**
    Perulangan dalam (`while`) pasti dilakukan sampai posisi `j = 0`

Merge Sort

Merge Sort

- Menggunakan pendekatan iteratif **divide-and-conquer** (membagi dan menyelesaikan)
- **Divide**: membagi masalah menjadi dua submasalah yang lebih kecil
- **Conquer**: menyelesaikan setiap masalah secara rekursif
- **Combine**: menggabungkan dua submasalah yang telah diselesaikan

Merge Sort


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

Merge Sort

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        leftarr = arr[:mid]
        rightarr = arr[mid:]
        merge_sort(leftarr)
        merge_sort(rightarr)
        merge(leftarr, rightarr, arr)
```

```
def merge(left, right, arr):
    i, j, k = 0, 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            arr[k] = left[i]
            i = i + 1
        else:
            arr[k] = right[j]
            j = j + 1
        k = k + 1

    while i < len(left):
        arr[k] = left[i]
        k = k + 1
        i = i + 1

    while j < len(right):
        arr[k] = right[j]
        k = k + 1
        j = j + 1
```

Analisis Algoritma Merge Sort

- Dalam kondisi array seperti apapun, algoritma *merge sort* pasti akan tetap melakukan pembagian (*divide*) array menjadi dua bagian, lalu mengurutkan setiap bagian tersebut → tidak ada *best case* maupun *worst case*
- Running time untuk kasus rata-rata tidak berbeda jauh dengan kasus terburuk
- Algoritma *merge sort* bersifat **stable** dan implementasinya **not-in-place**
- Kompleksitas algoritma *merge sort*: **`O(N log N)`**
  Fungsi `log N` merepresentasikan operasi pembagian array menjadi dua bagian yang sama panjang
  Fungsi `N` merepresentasikan operasi penggabungan (*merge*) array `left` dan array `right` menjadi menjadi satu array `arr` yang terurut

Quick Sort

Quick Sort

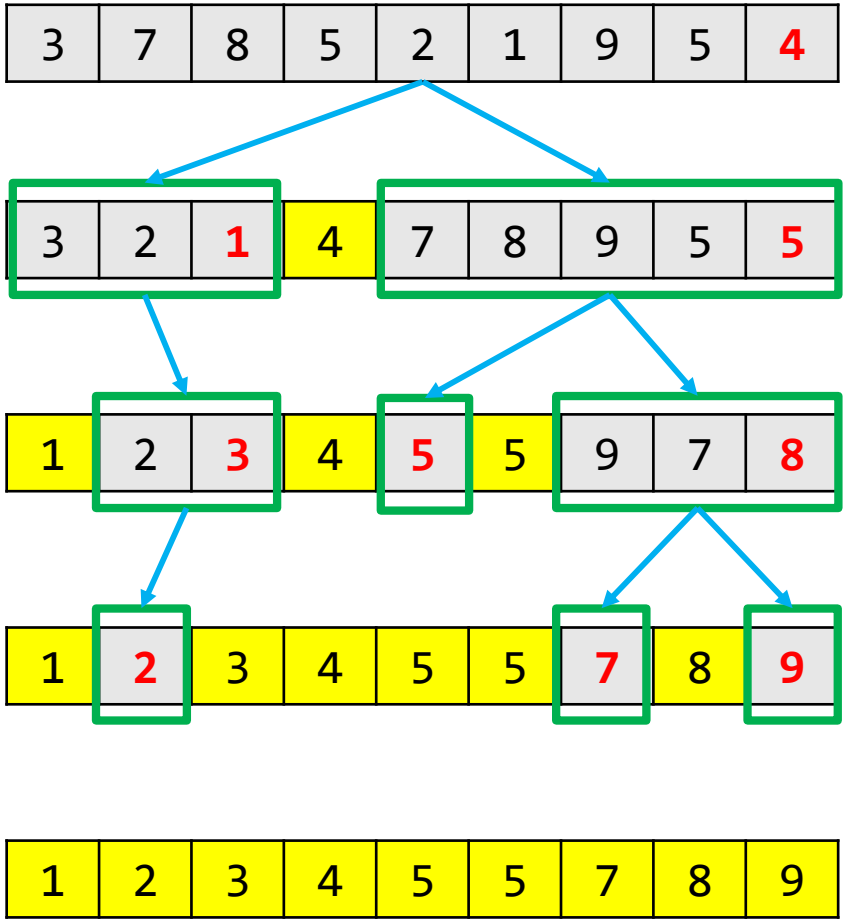
- Seperti Merge Sort, menggunakan pendekatan divide-and-conquer
- Membagi kontainer menjadi dua partisi berdasarkan suatu **pivot** (nilai acuan) yang dipilih dari salah satu elemen pada kontainer yang akan diurutkan
- Cara kerja *partitioning*:
  Elemen yang lebih kecil dari nilai pivot akan ditempatkan di sebelah kiri pivot, sedangkan elemen yang lebih besar dari nilai pivot akan ditempatkan di sebelah kanan pivot.
- Di setiap akhir *partitioning*, elemen pivot pasti berada di indeks yang seharusnya
- **Pemilihan pivot**: elemen pertama, elemen terakhir, elemen tengah, random

Quick Sort

```
def quick_sort(arr):
    quick_sort_rec(arr, 0, len(arr)-1)

def quick_sort_rec(arr, start, end):
    if start < end:
        pivot_idx = partition(arr, start, end)
        quick_sort_rec(arr, start, pivot_idx-1)
        quick_sort_rec(arr, pivot_idx+1, end)

def partition(arr, start, end):
    pivot = arr[end]
    i = start
    for j in range(start, end):
        if arr[j] <= pivot:
            arr[i], arr[j] = arr[j], arr[i]
            i = i + 1
    arr[i], arr[end] = arr[end], arr[i]
    return i
```



Analisis Algoritma Quick Sort

- *Running time* bergantung pada **pemilihan pivot**. Jika pivot menyebabkan *partitioning* relatif seimbang (*balanced*), maka *running time* semakin kecil.
- **Best case**: Pivot membagi array menjadi dua partisi yang seimbang
- **Worst case**: Pivot membagi suatu array dengan panjang `N` menjadi dua partisi, yaitu partisi kosong dan partisi yang panjangnya `N-1`; terjadi ketika pivot yang dipilih merupakan elemen terkecil atau terbesar, sehingga prosesnya sama seperti *selection sort*
- Algoritma quick sort bersifat **not stable** dan implementasinya **in-place**
- Kompleksitas algoritma *quick sort*:
  - Best case: **`O(N log N)`**
    Ada pembagian array menjadi dua partisi yang seimbang (seperti merge sort)
  - Worst case: **$O(N^2)$**
    Tidak terjadi pembagian partisi, seperti selection sort

Selamat Belajar ...!!