

# Tujuan Praktikum

1. Mahasiswa memahami dan menerapkan algoritma sorting untuk array
2. Mahasiswa memahami penggunaan jenis-jenis algoritma sorting
3. Mahasiswa menerapkan algoritma sorting di bahasa pemrograman python 3

## Pentingnya Algoritma Pengurutan di Python

Pengurutan (Sorting) adalah fondasi dasar yang menjadi landasan bagi banyak algoritma lainnya. Hal ini berkaitan dengan beberapa konsep menarik yang akan sering Anda temui sepanjang karier pemrograman Anda. Memahami cara kerja algoritma pengurutan di Python secara mendalam adalah langkah penting untuk menerapkan algoritma yang benar dan efisien dalam memecahkan masalah di dunia nyata.

Pengurutan adalah salah satu algoritma yang paling mendalam dipelajari dalam ilmu komputer. Ada puluhan implementasi dan aplikasi pengurutan yang berbeda yang bisa Anda gunakan untuk membuat kode lebih efisien dan efektif.

Anda bisa menggunakan pengurutan untuk memecahkan berbagai macam masalah:

- **Pencarian:** Mencari item dalam daftar bekerja jauh lebih cepat jika daftar tersebut sudah diurutkan.
- **Seleksi:** Memilih item dari daftar berdasarkan hubungannya dengan item lain menjadi lebih mudah dengan data yang sudah diurutkan. Misalnya, menemukan nilai terbesar atau terkecil ke-k, atau menemukan nilai median dari daftar, jauh lebih mudah jika nilai-nilai tersebut diurutkan dalam urutan naik atau turun.
- **Duplikasi:** Menemukan nilai duplikat dalam daftar dapat dilakukan dengan sangat cepat ketika daftar tersebut diurutkan.
- **Distribusi:** Menganalisis distribusi frekuensi item dalam daftar menjadi sangat cepat jika daftar tersebut diurutkan. Sebagai contoh, menemukan elemen yang paling sering atau paling jarang muncul relatif mudah dengan daftar yang diurutkan.

Mulai dari aplikasi komersial hingga penelitian akademis dan di berbagai bidang lainnya, ada banyak cara menggunakan pengurutan untuk menghemat waktu dan usaha Anda.

## Algoritma Pengurutan Bawaan Python

Python, seperti banyak bahasa pemrograman tingkat tinggi lainnya, menyediakan kemampuan untuk mengurutkan data secara langsung menggunakan fungsi `sorted()`. Berikut adalah contoh mengurutkan array bilangan bulat:

```
>>> array = [8, 2, 6, 4, 5]
>>> sorted(array)
[2, 4, 5, 6, 8]
```

Anda dapat menggunakan `sorted()` untuk mengurutkan daftar apa pun selama nilai di dalamnya dapat dibandingkan.

## Pentingnya Kompleksitas Waktu

Pada praktikum ini mencakup dua cara untuk mengukur waktu eksekusi algoritma pengurutan:

- **Secara praktis**, Anda akan mengukur waktu eksekusi implementasi menggunakan modul `timeit`.
- **Secara teoretis**, Anda akan mengukur kompleksitas waktu algoritma menggunakan notasi Big O.

## Mengukur Waktu Eksekusi Kode Anda

Saat membandingkan dua algoritma pengurutan di Python, melihat seberapa lama masing-masing algoritma berjalan sangat berguna. Waktu spesifik yang dibutuhkan setiap algoritma sebagian ditentukan oleh perangkat keras Anda, namun Anda tetap dapat menggunakan perbandingan waktu relatif antara eksekusi untuk membantu menentukan implementasi mana yang lebih efisien.

Pada bagian ini, Anda akan fokus pada cara praktis untuk mengukur waktu eksekusi algoritma pengurutan Anda menggunakan modul `timeit`.

Berikut adalah fungsi yang dapat Anda gunakan untuk mengukur waktu algoritma Anda:

```
In [ ]: from random import randint
from timeit import repeat

def run_sorting_algorithm(algorithm, array):
    # Menyiapkan konteks dan mempersiapkan panggilan
    # ke algoritma yang ditentukan menggunakan array yang diberikan.
    # Hanya mengimpor fungsi algoritma jika bukan `sorted()` bawaan.
    setup_code = f"from __main__ import {algorithm}" \
        if algorithm != "sorted" else ""
    if algorithm != "sorted":
        stmt = f"{algorithm}({array})"
    else:
        stmt = f"sorted({array})"

    # Menjalankan kode sepuluh kali berbeda dan mengembalikan waktu
    # dalam detik untuk setiap eksekusi
    times = repeat(setup=setup_code, stmt=stmt, repeat=3, number=10)

    # Terakhir, menampilkan nama algoritma dan
```

```
# waktu minimum yang dibutuhkan untuk menjalankan
print(f"Algorithm: {algorithm}. Minimum execution time: {min(times)}")
```

Pada contoh ini, `run_sorting_algorithm()` menerima nama algoritma dan array input yang perlu diurutkan. Berikut penjelasan baris demi baris mengenai cara kerjanya:

- **Baris 8** mengimpor nama algoritma menggunakan f-string di Python agar `timeit.repeat()` tahu di mana harus memanggil algoritma tersebut. Ini hanya diperlukan untuk implementasi khusus. Jika algoritma yang ditentukan adalah `sorted()` bawaan, maka tidak ada yang akan diimpor.
- **Baris 11** menyiapkan perintah pemanggilan algoritma dengan array yang diberikan. Ini adalah pernyataan yang akan dieksekusi dan diukur waktunya.
- **Baris 15** memanggil `timeit.repeat()` dengan kode setup dan pernyataan. Ini akan menjalankan algoritma pengurutan yang ditentukan sepuluh kali, mengembalikan waktu eksekusi dalam detik untuk setiap kali dijalankan.
- **Baris 19** memilih waktu terpendek dan mencetaknya bersama dengan nama algoritma.

**Catatan:** Kesalahpahaman umum adalah bahwa Anda harus mencari rata-rata waktu setiap eksekusi daripada memilih waktu terpendek. Pengukuran waktu sering kali "berisik" karena sistem menjalankan proses lain secara bersamaan. Waktu terpendek adalah yang paling tidak "berisik," sehingga memberikan representasi terbaik dari waktu eksekusi sebenarnya dari algoritma.

Berikut adalah contoh penggunaan `run_sorting_algorithm()` untuk menentukan waktu yang dibutuhkan untuk mengurutkan array yang berisi sepuluh ribu nilai integer menggunakan `sorted()`:

```
ARRAY_LENGTH = 10000

if __name__ == "__main__":
    # Membuat array berisi `ARRAY_LENGTH` item
    # dengan nilai integer acak antara 0 dan 999
    array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]

    # Memanggil fungsi dengan nama algoritma pengurutan
    # dan array yang baru saja dibuat
    run_sorting_algorithm(algorithm="sorted", array=array)
```

Jika Anda menyimpan kode di atas dalam file `sorting.py`, maka Anda bisa menjalankannya dari terminal dan melihat hasilnya:

```
$ python sorting.py
Algorithm: sorted. Minimum execution time: 0.010945824000000007
```

Inginlah bahwa waktu dalam detik untuk setiap eksperimen sebagian tergantung pada perangkat keras yang Anda gunakan, jadi Anda mungkin akan melihat hasil yang sedikit berbeda saat menjalankan kode ini.

# Mengukur Efisiensi dengan Notasi Big O

Waktu yang diukur untuk menjalankan suatu algoritma saja tidak cukup untuk mendapatkan gambaran penuh mengenai kompleksitas waktu. Notasi Big O memungkinkan kita untuk membandingkan berbagai implementasi algoritma dan menentukan mana yang paling efisien, mengabaikan detail yang tidak penting dan fokus pada bagian yang paling penting dari waktu eksekusi algoritma.

Karena waktu yang dibutuhkan untuk menjalankan algoritma bisa dipengaruhi oleh faktor-faktor seperti kecepatan prosesor atau memori yang tersedia, notasi Big O menawarkan cara untuk mengekspresikan kompleksitas runtime secara independen dari perangkat keras. Dengan Big O, kita menyatakan kompleksitas dalam istilah seberapa cepat runtime algoritma meningkat relatif terhadap ukuran inputnya, terutama saat input tersebut bertambah besar secara sewenang-wenang.

Jika kita mengasumsikan bahwa  $n$  adalah ukuran input untuk algoritma, notasi Big O mewakili hubungan antara  $n$  dan jumlah langkah yang dibutuhkan algoritma untuk menemukan solusi. Big O menggunakan huruf kapital "O" diikuti dengan hubungan ini dalam tanda kurung. Misalnya,  $O(n)$  mewakili algoritma yang mengeksekusi sejumlah langkah yang sebanding dengan ukuran inputnya.

Berikut adalah lima contoh kompleksitas runtime dari berbagai algoritma:

Big O	Kompleksitas	Deskripsi
$O(1)$	<b>Konstan</b>	Runtime konstan, tidak tergantung pada ukuran input. Contohnya, mencari elemen dalam hash table dapat dilakukan dalam waktu konstan.
$O(n)$	<b>Linear</b>	Runtime bertambah secara linear seiring dengan ukuran input. Fungsi yang memeriksa suatu kondisi pada setiap item dalam daftar adalah contoh algoritma $O(n)$ .
$O(n^2)$	<b>Kuaratik</b>	Runtime berupa fungsi kuadrat dari ukuran input. Implementasi naif untuk mencari nilai duplikat dalam daftar adalah contoh algoritma kuadratik.
$O(2^n)$	<b>Eksponensial</b>	Runtime bertambah secara eksponensial seiring dengan ukuran input. Algoritma eksponensial sangat tidak efisien. Contoh: masalah pewarnaan tiga warna.
$O(\log n)$	<b>Logaritmik</b>	Runtime bertambah secara linear sementara ukuran input bertambah secara eksponensial. Misalnya, jika diperlukan satu detik untuk memproses seribu elemen, maka akan diperlukan dua detik untuk memproses sepuluh ribu elemen, tiga detik untuk seratus ribu, dan seterusnya. Pencarian biner adalah contoh algoritma runtime logaritmik.

## Menentukan Kompleksitas Runtime untuk Algoritma Pengurutan

Pada praktikum ini juga mencakup kompleksitas runtime Big O untuk setiap algoritma pengurutan yang dibahas. Dengan memahami kompleksitas waktu dalam kasus tertentu, Anda akan mendapatkan pemahaman yang lebih baik tentang bagaimana menggunakan Big O untuk mengklasifikasikan algoritma lain.

# Algoritma Bubble Sort dalam Python

Bubble Sort adalah salah satu algoritma pengurutan yang paling sederhana. Nama ini berasal dari cara kerjanya: pada setiap iterasi, elemen terbesar dalam daftar akan "naik" menuju posisi yang benar.

Bubble sort terdiri dari beberapa iterasi melalui sebuah daftar, membandingkan elemen satu per satu, dan menukar item-item yang berdampingan jika urutannya salah.

## Implementasi Bubble Sort dalam Python

Berikut adalah implementasi algoritma bubble sort dalam Python:

```
In [ ]: def bubble_sort(array):
    n = len(array)

    for i in range(n):
        # Membuat flag yang memungkinkan fungsi berhenti lebih awal
        # jika tidak ada yang perlu diurutkan
        already_sorted = True

        # Melakukan pengecekan setiap item dalam daftar satu per satu,
        # membandingkannya dengan nilai yang berdampingan.
        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
                # Jika item lebih besar dari item yang bersebelahan, tukar posisi
                array[j], array[j + 1] = array[j + 1], array[j]

            # Jika ada pertukaran, set `already_sorted` menjadi False
            already_sorted = False

        # Jika tidak ada pertukaran, daftar sudah diurutkan
        if already_sorted:
            break

    return array
```

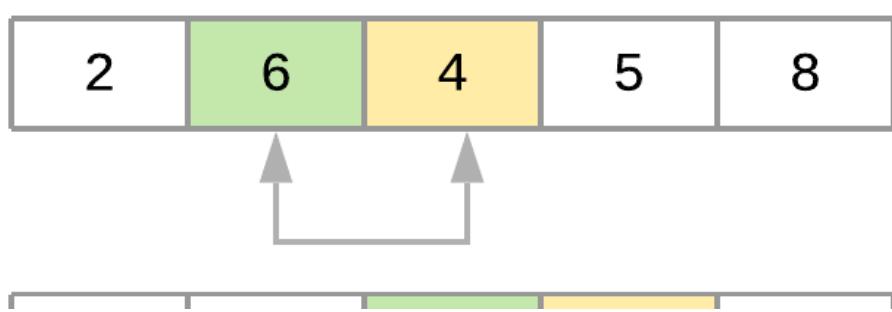
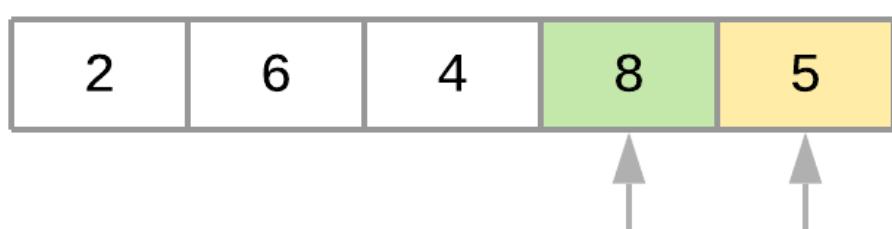
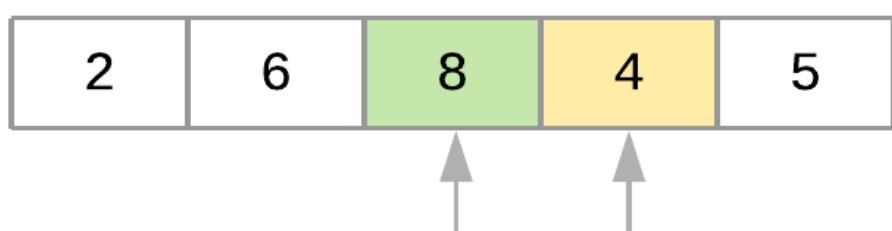
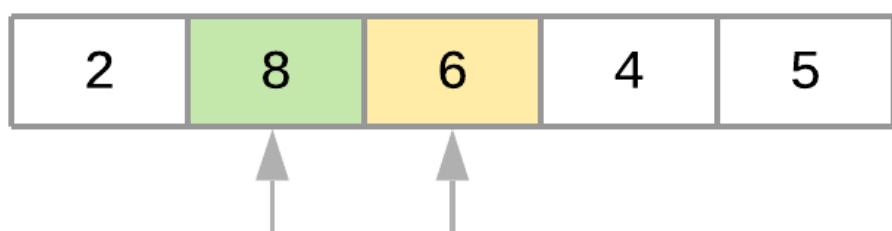
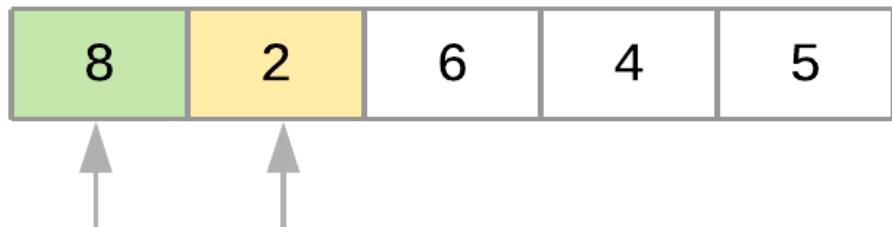
Implementasi ini mengurutkan daftar dalam urutan menaik, di mana setiap langkah "menaikkan" elemen terbesar ke ujung daftar. Artinya, setiap iterasi memerlukan lebih sedikit langkah karena bagian akhir daftar sudah diurutkan.

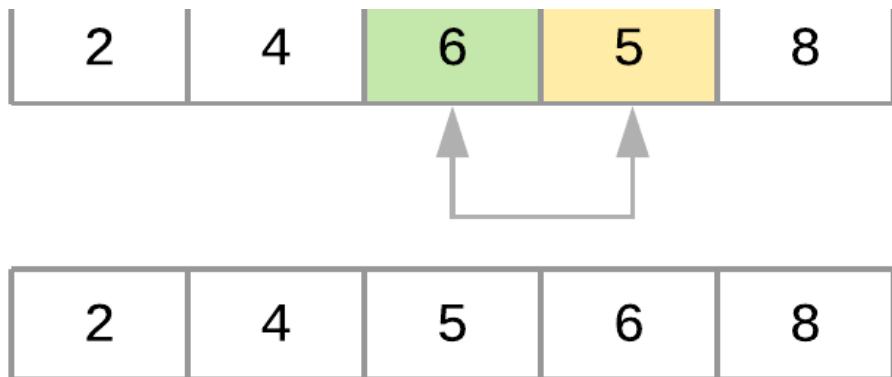
Loop di baris ke-4 dan ke-10 menentukan cara algoritma melalui daftar. Perhatikan bahwa `j` awalnya pergi dari elemen pertama hingga elemen sebelum terakhir, kemudian berkurang pada setiap iterasi karena bagian akhir daftar telah terurut.

Catatan: Flag `already_sorted` di baris 13, 23, dan 27 adalah optimisasi yang memungkinkan fungsi berhenti lebih awal jika daftar sudah terurut.

Sebagai latihan, Anda bisa menghapus flag ini dan membandingkan waktu eksekusi kedua implementasi.

Untuk menganalisis cara kerja algoritma dengan tepat, mari kita perhatikan daftar dengan nilai [8, 2, 6, 4, 5]. Anggaplah kita menggunakan fungsi `bubble_sort()` seperti yang dijelaskan di atas. Berikut adalah ilustrasi tentang bagaimana susunan elemen dalam array pada setiap iterasi algoritma:





Sekarang, mari kita lihat langkah demi langkah apa yang terjadi pada array seiring algoritma berjalan:

1. Algoritma dimulai dengan membandingkan elemen pertama, yaitu 8, dengan elemen yang ada di sebelahnya, yaitu 2. Karena  $8 > 2$ , nilainya ditukar, menghasilkan urutan sebagai berikut: [2, 8, 6, 4, 5].
2. Algoritma kemudian membandingkan elemen kedua, yaitu 8, dengan elemen yang ada di sebelahnya, yaitu 6. Karena  $8 > 6$ , nilainya ditukar, menghasilkan urutan sebagai berikut: [2, 6, 8, 4, 5].
3. Selanjutnya, algoritma membandingkan elemen ketiga, yaitu 8, dengan elemen yang ada di sebelahnya, yaitu 4. Karena  $8 > 4$ , nilainya ditukar, menghasilkan urutan sebagai berikut: [2, 6, 4, 8, 5].
4. Terakhir, algoritma membandingkan elemen keempat, yaitu 8, dengan elemen yang ada di sebelahnya, yaitu 5, dan menukarnya juga, sehingga menghasilkan [2, 6, 4, 5, 8]. Pada titik ini, algoritma telah menyelesaikan satu iterasi pertama pada daftar ( $i = 0$ ). Perhatikan bagaimana nilai 8 naik dari posisinya yang awal hingga ke posisi yang benar di akhir daftar.

Pada iterasi kedua ( $i = 1$ ), algoritma menyadari bahwa elemen terakhir dari daftar sudah berada di posisi yang benar dan fokus pada empat elemen yang tersisa, [2, 6, 4, 5]. Pada akhir iterasi ini, nilai 6 akan menemukan posisinya yang benar. Iterasi ketiga akan menempatkan nilai 5, dan seterusnya hingga seluruh daftar tersusun dengan benar.

## Analisis Kompleksitas Waktu Bubble Sort

Implementasi bubble sort ini terdiri dari dua loop bersarang yang melakukan  $n - 1$  perbandingan, lalu  $n - 2$  perbandingan, dan seterusnya hingga perbandingan terakhir selesai. Ini menghasilkan total  $(n - 1) + (n - 2) + \dots + 1 = n(n-1)/2$  perbandingan, yang dapat disederhanakan menjadi  $\frac{1}{2}n^2 - \frac{1}{2}n$ .

Untuk analisis kompleksitas waktu Big O, kita hanya memperhatikan bagaimana waktu eksekusi tumbuh seiring ukuran input, sehingga persamaan ini disederhanakan menjadi  $O(n^2)$ .

Pada kasus di mana array sudah diurutkan (dengan asumsi menggunakan flag `already_sorted`), kompleksitas waktu terbaik akan menjadi  $O(n)$  karena algoritma tidak perlu meninjau elemen lebih dari sekali.

### Mengukur Waktu Eksekusi Bubble Sort

Menggunakan `run_sorting_algorithm()` dari tutorial sebelumnya, berikut waktu yang diperlukan bubble sort untuk memproses array dengan sepuluh ribu item.

```
if __name__ == "__main__":
    array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]
    run_sorting_algorithm(algorithm="bubble_sort", array=array)
```

### Kelebihan dan Kekurangan Bubble Sort

Kelebihan utama dari algoritma bubble sort adalah kesederhanaannya. Algoritma ini mudah diimplementasikan dan dipahami. Namun, kelemahan bubble sort adalah lambat, dengan kompleksitas waktu  $O(n^2)$ , yang membuatnya kurang cocok untuk mengurutkan daftar yang besar.

## Algoritma Insertion Sort dalam Python

Insertion sort membangun daftar terurut satu elemen dalam satu waktu, membandingkan setiap item dengan item lainnya dan memasukkannya pada posisi yang benar. Algoritma ini mirip dengan cara mengurutkan kartu pada sebuah dek.

### Implementasi Insertion Sort dalam Python

Berikut adalah implementasi insertion sort dalam Python:

```
In [ ]: def insertion_sort(array):
    # Loop dari elemen kedua hingga elemen terakhir
    for i in range(1, len(array)):
        # Elemen yang akan ditempatkan pada posisinya
        key_item = array[i]

        # Inisialisasi variabel untuk menemukan posisi yang benar
        j = i - 1

        # Membandingkan `key_item` dengan nilai di sebelah kiri
        while j >= 0 and array[j] > key_item:
            # Pindahkan nilai satu posisi ke kiri
            array[j + 1] = array[j]
            j -= 1

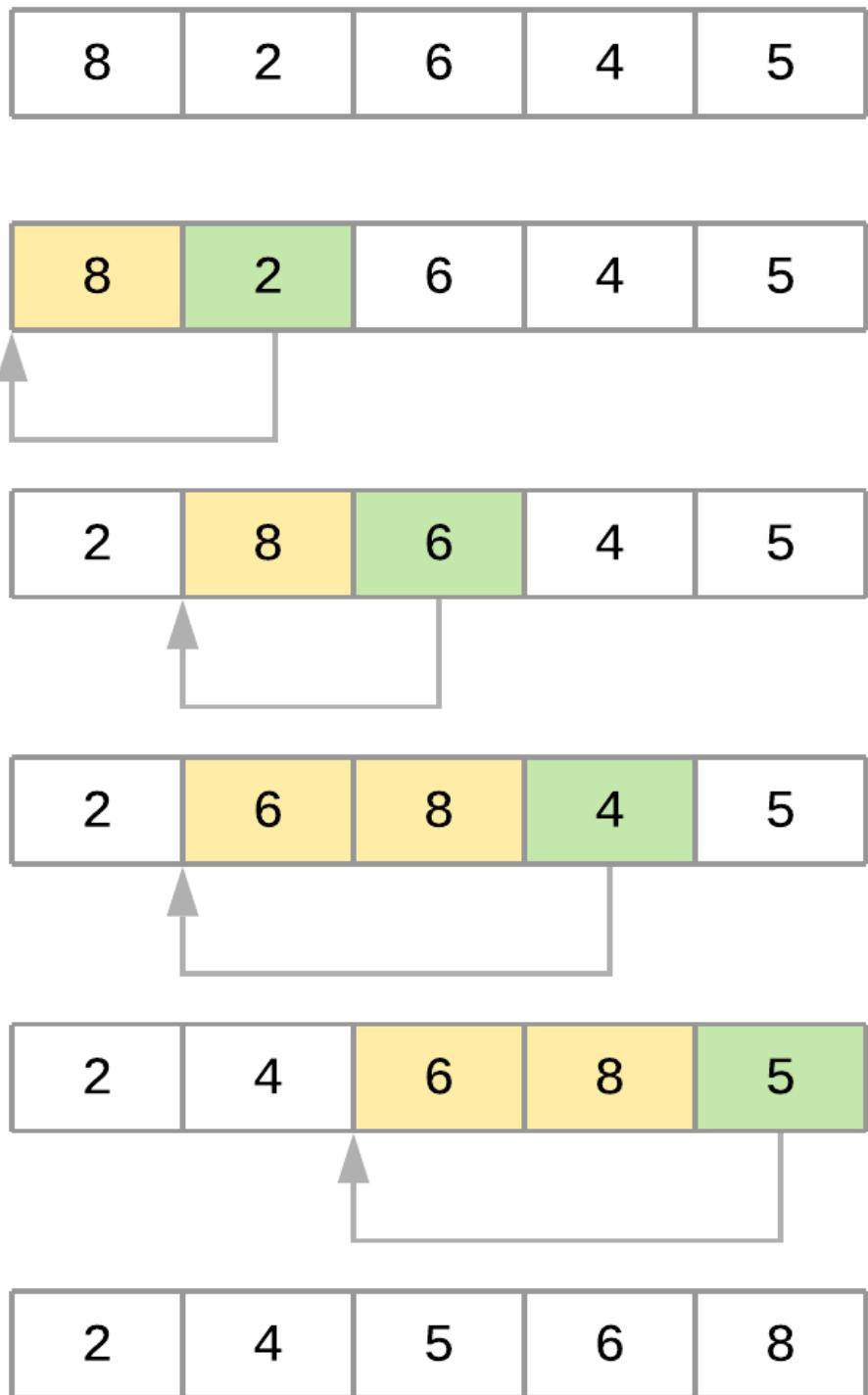
        # Setelah selesai memindahkan elemen, tempatkan `key_item`
        array[j + 1] = key_item

    return array
```

Tidak seperti bubble sort, implementasi insertion sort ini menyusun daftar terurut dengan mendorong elemen yang lebih kecil ke kiri. Mari kita bedah `insertion_sort()` baris demi baris:

- Baris 4 mengatur loop yang menentukan `key_item` yang akan ditempatkan selama setiap iterasi. Perhatikan bahwa loop dimulai dari item kedua dalam daftar hingga item terakhir.
- Baris 7 menginisialisasi `key_item` dengan item yang akan ditempatkan oleh fungsi.
- Baris 12 menginisialisasi variabel yang secara berurutan akan menunjuk ke setiap elemen di sebelah kiri `key_item`. Elemen-elemen ini akan dibandingkan secara berurutan dengan `key_item`.
- Baris 18 membandingkan `key_item` dengan setiap nilai di sebelah kirinya menggunakan while loop, menggeser elemen-elemen untuk memberi ruang bagi `key_item`.
- Baris 27 menempatkan `key_item` di tempat yang benar setelah algoritma menggeser semua nilai yang lebih besar ke kanan.

Berikut adalah ilustrasi berbagai iterasi algoritma saat mengurutkan array [8, 2, 6, 4, 5]:



Sekarang berikut adalah ringkasan langkah-langkah algoritma saat mengurutkan array:

1. Algoritma dimulai dengan `key_item = 2` dan menelusuri subarray di sebelah kirinya untuk menemukan posisi yang tepat untuk item tersebut. Dalam hal ini, subarraynya adalah [8].

Karena  $2 < 8$ , algoritma menggeser elemen 8 satu posisi ke kanan. Array yang dihasilkan pada titik ini adalah [8, 8, 6, 4, 5].

Karena tidak ada lagi elemen di subarray, `key_item` sekarang ditempatkan di posisinya yang baru, dan array akhirnya menjadi [2, 8, 6, 4, 5].

2. Iterasi kedua dimulai dengan `key_item = 6` dan menelusuri subarray di sebelah kirinya, dalam hal ini [2, 8].

Karena  $6 < 8$ , algoritma menggeser 8 ke kanan. Array yang dihasilkan pada titik ini adalah [2, 8, 8, 4, 5].

Karena  $6 > 2$ , algoritma tidak perlu melanjutkan penelusuran di subarray, sehingga `key_item` ditempatkan dan iterasi kedua selesai. Pada saat ini, array yang dihasilkan adalah [2, 6, 8, 4, 5].

3. Iterasi ketiga menempatkan elemen 4 di posisinya yang benar, dan iterasi keempat menempatkan elemen 5 di tempat yang tepat, sehingga array menjadi terurut.

## Mengukur Kompleksitas Waktu Big O untuk Insertion Sort

Mirip dengan implementasi bubble sort, algoritma insertion sort memiliki beberapa loop bersarang yang berjalan melalui daftar. Loop bagian dalam cukup efisien karena hanya akan menelusuri daftar hingga menemukan posisi yang tepat untuk elemen. Namun, algoritma ini tetap memiliki kompleksitas waktu rata-rata  $O(n^2)$ .

Kasus terburuk terjadi saat array yang diberikan diurutkan dalam urutan terbalik. Dalam kasus ini, loop bagian dalam harus melakukan setiap perbandingan untuk menempatkan setiap elemen di posisi yang benar. Hal ini memberikan kompleksitas waktu  $O(n^2)$ .

Kasus terbaik terjadi saat array yang diberikan sudah terurut. Di sini, loop bagian dalam tidak pernah dijalankan, menghasilkan kompleksitas waktu  $O(n)$ , seperti halnya kasus terbaik pada bubble sort.

Meskipun bubble sort dan insertion sort memiliki kompleksitas waktu Big O yang sama, dalam praktiknya, insertion sort jauh lebih efisien daripada bubble sort. Jika melihat implementasi kedua algoritma, dapat terlihat bahwa insertion sort harus melakukan lebih sedikit perbandingan untuk mengurutkan daftar.

## Mengukur Waktu Eksekusi Insertion Sort

Untuk membuktikan bahwa insertion sort lebih efisien daripada bubble sort, Anda dapat menghitung waktu eksekusi algoritma insertion sort dan membandingkannya dengan hasil bubble sort. Untuk melakukannya, cukup ganti pemanggilan

`run_sorting_algorithm()` dengan nama implementasi insertion sort Anda:

```
if __name__ == "__main__":
    # Membuat array dengan panjang `ARRAY_LENGTH`
    # berisi nilai integer acak antara 0 dan 999
    array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]
```

```
# Memanggil fungsi dengan nama algoritma sorting
# dan array yang baru saja dibuat
run_sorting_algorithm(algorithm="insertion_sort", array=array)
```

Anda bisa menjalankan skrip seperti sebelumnya:

```
$ python sorting.py
Algorithm: insertion_sort. Minimum execution time: 56.71029764299999
Perhatikan bahwa implementasi insertion sort memakan waktu sekitar 17 detik lebih
cepat dibandingkan dengan bubble sort untuk mengurutkan array yang sama. Walaupun
keduanya memiliki kompleksitas  $O(n^2)$ , insertion sort lebih efisien.
```

## Menganalisis Kelebihan dan Kekurangan Insertion Sort

Seperti bubble sort, algoritma insertion sort sangat sederhana untuk diimplementasikan. Walaupun insertion sort adalah algoritma  $O(n^2)$ , dalam praktiknya jauh lebih efisien dibandingkan implementasi kuadratik lainnya seperti bubble sort.

Terdapat algoritma yang lebih kuat seperti merge sort dan Quicksort, tetapi implementasi ini bersifat rekursif dan biasanya tidak mengalahkan insertion sort saat bekerja dengan daftar kecil. Beberapa implementasi Quicksort bahkan menggunakan insertion sort secara internal jika daftar cukup kecil agar dapat menghasilkan implementasi yang lebih cepat. Timsort juga menggunakan insertion sort untuk mengurutkan bagian kecil dari array input.

Namun demikian, insertion sort tidak praktis untuk array besar, membuka peluang untuk menggunakan algoritma yang dapat diskalakan dengan cara yang lebih efisien.

## Algoritma Merge Sort dalam Python

Merge sort adalah algoritma pengurutan yang sangat efisien. Algoritma ini didasarkan pada pendekatan divide-and-conquer, teknik algoritmik kuat yang digunakan untuk menyelesaikan masalah kompleks.

Untuk memahami divide and conquer dengan baik, Anda harus terlebih dahulu memahami konsep rekursi. Rekursi melibatkan pemecahan masalah menjadi submasalah yang lebih kecil hingga ukurannya cukup kecil untuk dikelola. Dalam pemrograman, rekursi biasanya diekspresikan melalui fungsi yang memanggil dirinya sendiri.

Algoritma divide-and-conquer umumnya mengikuti struktur yang sama:

1. Input asli dipecah menjadi beberapa bagian, masing-masing mewakili submasalah yang serupa dengan aslinya tetapi lebih sederhana.
2. Setiap submasalah diselesaikan secara rekursif.
3. Solusi untuk semua submasalah digabungkan menjadi satu solusi keseluruhan.

Dalam kasus merge sort, pendekatan divide-and-conquer membagi himpunan nilai input menjadi dua bagian berukuran sama, mengurutkan setiap bagian secara rekursif, dan

akhirnya menggabungkan kedua bagian yang telah diurutkan ini menjadi satu daftar yang terurut.

## Mengimplementasikan Merge Sort dalam Python

Implementasi algoritma merge sort membutuhkan dua bagian yang berbeda:

1. Fungsi yang membagi input secara rekursif menjadi dua.
2. Fungsi yang menggabungkan kedua bagian, menghasilkan array yang terurut.

Berikut adalah kode untuk menggabungkan dua array yang berbeda:

```
In [ ]: def merge(left, right):  
    # Jika array pertama kosong, maka tidak ada yang perlu  
    # digabungkan, dan Anda dapat mengembalikan array kedua sebagai hasilnya  
    if len(left) == 0:  
        return right  
  
    # Jika array kedua kosong, maka tidak ada yang perlu  
    # digabungkan, dan Anda dapat mengembalikan array pertama sebagai hasilnya  
    if len(right) == 0:  
        return left  
  
    result = []  
    index_left = index_right = 0  
  
    # Sekarang telusuri kedua array hingga semua elemen  
    # masuk ke dalam array hasil  
    while len(result) < len(left) + len(right):  
        # Elemen harus diurutkan untuk ditambahkan ke  
        # array hasil, jadi Anda perlu memutuskan apakah akan  
        # mengambil elemen berikutnya dari array pertama atau kedua  
        if left[index_left] <= right[index_right]:  
            result.append(left[index_left])  
            index_left += 1  
        else:  
            result.append(right[index_right])  
            index_right += 1  
  
        # Jika mencapai akhir dari salah satu array, Anda dapat  
        # menambahkan elemen yang tersisa dari array lain ke  
        # hasil dan keluar dari Loop  
        if index_right == len(right):  
            result += left[index_left:]  
            break  
  
        if index_left == len(left):  
            result += right[index_right:]  
            break  
  
    return result
```

Fungsi `merge()` menerima dua array yang sudah diurutkan yang perlu digabungkan. Proses untuk mencapai hal ini cukup sederhana:

- **Baris 4 dan 9** memeriksa apakah salah satu array kosong. Jika salah satunya kosong, tidak ada yang perlu digabungkan, sehingga fungsi mengembalikan array lainnya.
- **Baris 17** memulai loop `while` yang berakhir kapan pun hasilnya sudah mencakup semua elemen dari kedua array yang diberikan. Tujuannya adalah memeriksa kedua array dan menggabungkan item-itemnya untuk menghasilkan daftar yang terurut.
- **Baris 21** membandingkan elemen di awal kedua array, memilih nilai yang lebih kecil, dan menambahkannya ke akhir array hasil.
- **Baris 31 dan 35** menambahkan item yang tersisa ke hasil jika semua elemen dari salah satu array sudah digunakan.

Dengan fungsi di atas, bagian yang hilang hanyalah fungsi yang secara rekursif membagi array input menjadi dua bagian dan menggunakan `merge()` untuk menghasilkan hasil akhir:

```
In [ ]: def merge_sort(array):
    # Jika array input memiliki kurang dari dua elemen,
    # kembalikan array sebagai hasil fungsi
    if len(array) < 2:
        return array

    midpoint = len(array) // 2

    # Urutkan array dengan membagi input menjadi dua bagian yang sama,
    # mengurutkan masing-masing bagian, lalu menggabungkannya
    return merge(
        left=merge_sort(array[:midpoint]),
        right=merge_sort(array[midpoint:]))

# Urutkan array dengan membagi input menjadi dua bagian yang sama,
# mengurutkan masing-masing bagian, lalu menggabungkannya
def merge(left, right):
    if len(left) == 0:
        return right
    if len(right) == 0:
        return left

    result = []
    left_index = right_index = 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            result.append(left[left_index])
            left_index += 1
        else:
            result.append(right[right_index])
            right_index += 1

    result.extend(left[left_index:])
    result.extend(right[right_index:])

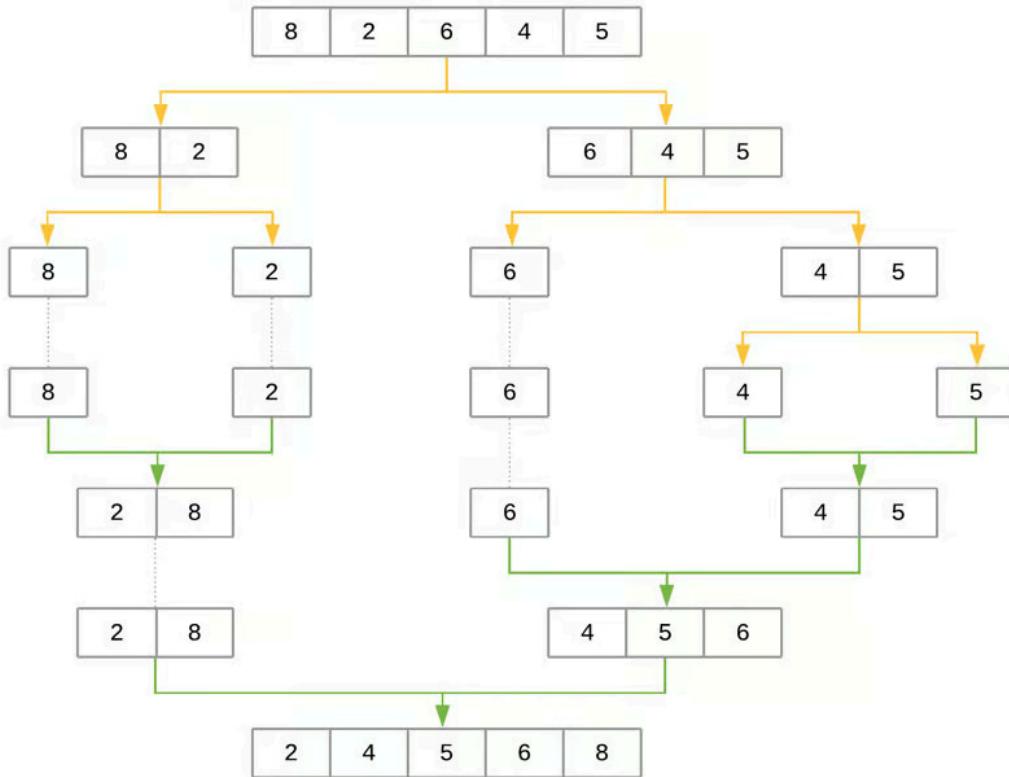
    return result
```

Berikut ringkasan kodennya:

- Baris 44 bertindak sebagai kondisi penghentian rekursi. Jika array input memiliki kurang dari dua elemen, fungsi mengembalikan array tersebut. Kondisi ini bisa terpicu jika menerima satu elemen atau array kosong.
- Baris 47 menghitung titik tengah array.
- Baris 52 memanggil `merge()`, yang menggabungkan kedua bagian yang sudah diurutkan.

Perhatikan bahwa fungsi ini memanggil dirinya sendiri secara rekursif, membagi array setiap kali hingga tersisa kurang dari dua elemen, artinya tidak ada lagi yang perlu diurutkan. Pada titik ini, `merge()` akan menggabungkan kedua bagian menjadi satu daftar terurut.

Mari kita lihat representasi langkah-langkah yang akan diambil merge sort untuk mengurutkan array `[8, 2, 6, 4, 5]`:



Langkah-langkahnya dapat diringkas sebagai berikut:

- Panggilan pertama ke `merge_sort()` dengan `[8, 2, 6, 4, 5]` menghasilkan `midpoint` sebagai 2, membagi array menjadi `[8, 2]` dan `[6, 4, 5]`.
- Panggilan `merge_sort()` dengan `[8, 2]` menghasilkan `[8]` dan `[2]`.
- Panggilan `merge_sort()` dengan `[8]` mengembalikan `[8]`, begitu juga dengan `[2]`.
- Selanjutnya, `merge()` mulai menggabungkan `[8]` dan `[2]` menjadi `[2, 8]`.
- Di sisi lain, `[6, 4, 5]` dipecah dan digabung kembali menjadi `[4, 5, 6]`.
- Pada langkah terakhir, `[2, 8]` dan `[4, 5, 6]` digabungkan dengan `merge()`, menghasilkan hasil akhir: `[2, 4, 5, 6, 8]`.

## Mengukur Kompleksitas Big O Merge Sort

Untuk menganalisis kompleksitas merge sort, kita dapat melihat dua langkahnya secara terpisah:

- `merge()` memiliki runtime linear, karena menerima dua array dengan panjang total maksimum `n`, menggabungkan keduanya dengan melihat setiap elemen maksimal sekali, menghasilkan kompleksitas waktu `O(n)`.
- Langkah kedua membagi array secara rekursif dan memanggil `merge()` untuk setiap bagian. Karena array terus dibagi dua hingga tersisa satu elemen, jumlah total operasi pembagian adalah `log2n`, sehingga kompleksitas runtime total adalah `O(n log2n)`.

# Mengukur Kecepatan Implementasi Merge Sort

Untuk membandingkan kecepatan merge sort, Anda dapat menggunakan mekanisme yang sama dan mengganti nama algoritma pada baris 8:

```
if __name__ == "__main__":
    array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]
    run_sorting_algorithm(algorithm="merge_sort", array=array)
```

Eksekusi skrip untuk melihat waktu eksekusi `merge_sort` :

```
$ python sorting.py
Algorithm: merge_sort. Minimum execution time:
0.6195857160000173
```

Dibandingkan dengan bubble sort dan insertion sort, merge sort sangat cepat, mengurutkan array sepuluh ribu elemen dalam waktu kurang dari satu detik!

## Kelebihan dan Kekurangan Merge Sort

Dengan kompleksitas waktu  $O(n \log_2 n)$ , merge sort adalah algoritma yang efisien dan cocok untuk array besar. Algoritma ini juga mudah diparalelkan karena membagi array input menjadi beberapa bagian yang dapat diproses secara paralel jika diperlukan.

Namun, untuk daftar kecil, waktu eksekusi rekursi membuat algoritma seperti bubble sort dan insertion sort menjadi lebih cepat. Sebagai contoh, eksperimen dengan daftar berisi sepuluh elemen menunjukkan waktu berikut:

```
Algorithm: bubble_sort. Minimum execution time:
0.00001877499999998654
Algorithm: insertion_sort. Minimum execution time:
0.00002978600000000395
Algorithm: merge_sort. Minimum execution time:
0.0001698300000000276
```

Selain itu, merge sort membuat salinan array setiap kali dipanggil secara rekursif, yang membuatnya menggunakan lebih banyak memori daripada bubble sort dan insertion sort, yang dapat mengurutkan daftar di tempat.

## Algoritma Quicksort dalam Python

Seperti halnya merge sort, algoritma Quicksort juga menerapkan prinsip divide-and-conquer untuk membagi array input menjadi dua daftar, yang pertama berisi elemen-elemen kecil dan yang kedua berisi elemen-elemen besar. Algoritma kemudian mengurutkan kedua daftar secara rekursif hingga daftar hasil akhirnya terurut sepenuhnya.

Membagi daftar input disebut dengan *partitioning*. Quicksort pertama-tama memilih elemen pivot dan membagi daftar di sekitar pivot, dengan menempatkan setiap elemen

yang lebih kecil ke dalam daftar rendah (*low*) dan setiap elemen yang lebih besar ke dalam daftar tinggi (*high*).

Dengan menempatkan setiap elemen dari daftar rendah ke kiri pivot dan setiap elemen dari daftar tinggi ke kanan, pivot akan diposisikan dengan tepat di tempatnya dalam daftar yang sudah terurut. Ini berarti fungsi sekarang dapat menerapkan prosedur yang sama secara rekursif pada daftar rendah dan kemudian pada daftar tinggi hingga seluruh daftar terurut.

## Implementasi Quicksort dalam Python

Berikut adalah implementasi Quicksort yang cukup ringkas:

```
In [ ]: from random import randint

def quicksort(array):
    # Jika array input berisi kurang dari dua elemen,
    # kembalikan array tersebut sebagai hasil dari fungsi
    if len(array) < 2:
        return array

    low, same, high = [], [], []
    # Pilih elemen `pivot` secara acak
    pivot = array[randint(0, len(array) - 1)]

    for item in array:
        # Elemen yang Lebih kecil dari `pivot` masuk ke
        # daftar `low`. Elemen yang Lebih besar dari
        # `pivot` masuk ke daftar `high`. Elemen yang
        # sama dengan `pivot` masuk ke daftar `same`.
        if item < pivot:
            low.append(item)
        elif item == pivot:
            same.append(item)
        elif item > pivot:
            high.append(item)

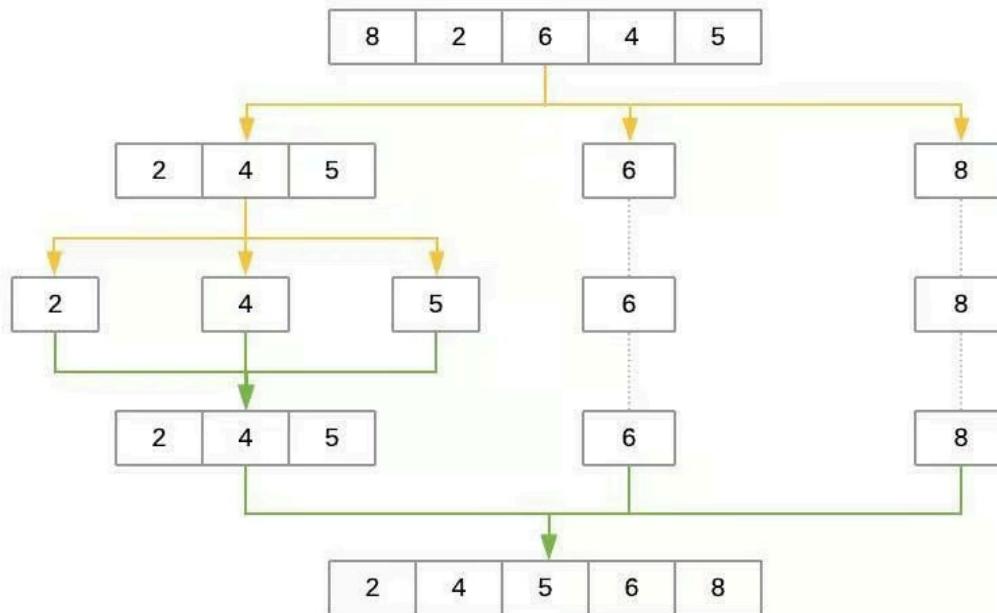
    # Hasil akhir menggabungkan daftar `low` yang terurut
    # dengan daftar `same` dan daftar `high` yang terurut
    return quicksort(low) + same + quicksort(high)
```

### Ringkasan Kode:

- Baris 6 menghentikan fungsi rekursif jika array berisi kurang dari dua elemen.
- Baris 12 memilih elemen pivot secara acak dari daftar dan melanjutkan untuk membagi daftar.
- Baris 19 dan 20 menempatkan setiap elemen yang lebih kecil dari pivot ke dalam daftar *low*.
- Baris 21 dan 22 menempatkan setiap elemen yang sama dengan pivot ke dalam daftar *same*.
- Baris 23 dan 24 menempatkan setiap elemen yang lebih besar dari pivot ke dalam daftar *high*.

- Baris 28 secara rekursif mengurutkan daftar *low* dan *high* dan menggabungkannya dengan daftar *same*.

Ilustrasi Langkah-langkah yang Diambil Quicksort untuk Mengurutkan Array [8, 2, 6, 4, 5]:



Garis kuning mewakili pembagian array menjadi tiga daftar: *low*, *same*, dan *high*. Garis hijau mewakili pengurutan dan penggabungan daftar-daftar ini kembali. Berikut adalah penjelasan singkat tentang langkah-langkahnya:

1. Elemen pivot dipilih secara acak. Dalam hal ini, pivot adalah 6.
2. Pembagian pertama membagi array input sehingga *low* berisi [2, 4, 5], *same* berisi [6], dan *high* berisi [8].
3. Fungsi `quicksort()` kemudian dipanggil secara rekursif dengan *low* sebagai input. Ini memilih pivot acak dan membagi array menjadi [2] sebagai *low*, [4] sebagai *same*, dan [5] sebagai *high*.
4. Proses ini berlanjut, namun pada titik ini, baik *low* maupun *high* masing-masing berisi kurang dari dua elemen. Ini mengakhiri rekursi, dan fungsi menggabungkan array kembali. Menggabungkan *low* dan *high* yang sudah terurut ke sisi masing-masing dari *same* menghasilkan [2, 4, 5].
5. Di sisi lain, daftar *high* yang berisi [8] memiliki kurang dari dua elemen, jadi algoritma mengembalikan array *low* yang sudah terurut, yang sekarang menjadi [2, 4, 5]. Menggabungkannya dengan *same* ([6]) dan *high* ([8]) menghasilkan daftar yang sudah terurut sepenuhnya.

## Memilih Elemen Pivot

Mengapa implementasi di atas memilih elemen pivot secara acak? Bukankah sama saja jika kita selalu memilih elemen pertama atau terakhir dari daftar input?

Karena cara kerja algoritma Quicksort, jumlah level rekursi bergantung pada posisi elemen pivot dalam setiap partisi. Dalam skenario terbaik, algoritma secara konsisten memilih elemen median sebagai pivot. Hal ini akan membuat setiap submasalah yang dihasilkan memiliki ukuran tepat setengah dari masalah sebelumnya, yang mengarah pada maksimal  $\log_2 n$  level.

Di sisi lain, jika algoritma selalu memilih elemen terkecil atau terbesar dari array sebagai pivot, maka partisi yang dihasilkan akan sangat tidak seimbang, yang mengarah pada  $n-1$  level rekursi. Hal ini akan menjadi skenario terburuk untuk Quicksort.

Seperti yang bisa Anda lihat, efisiensi Quicksort sering bergantung pada pemilihan pivot. Jika array input tidak terurut, maka menggunakan elemen pertama atau terakhir sebagai pivot akan berfungsi sama seperti elemen acak. Tetapi jika array input sudah terurut atau hampir terurut, menggunakan elemen pertama atau terakhir sebagai pivot bisa menyebabkan skenario terburuk. Memilih pivot secara acak membuat Quicksort lebih mungkin memilih nilai yang lebih dekat dengan median dan menyelesaikan lebih cepat.

Pilihan lain untuk memilih pivot adalah dengan menemukan nilai median dari array dan memaksa algoritma untuk menggunakananya sebagai pivot. Hal ini dapat dilakukan dalam waktu  $O(n)$ . Meskipun prosesnya sedikit lebih rumit, menggunakan nilai median sebagai pivot untuk Quicksort menjamin Anda akan mendapatkan skenario Big O terbaik.

## Mengukur Kompleksitas Big O Quicksort

Dengan Quicksort, daftar input dipartisi dalam waktu linear,  $O(n)$ , dan proses ini diulang secara rekursif rata-rata  $\log_2 n$  kali. Ini menghasilkan kompleksitas akhir  $O(n \log_2 n)$ .

Namun demikian, ingat diskusi tentang bagaimana pemilihan pivot mempengaruhi waktu eksekusi algoritma. Skenario terbaik  $O(n)$  terjadi ketika pivot yang dipilih dekat dengan median dari array, dan skenario  $O(n^2)$  terjadi ketika pivot adalah nilai terkecil atau terbesar dari array.

Secara teoretis, jika algoritma fokus pertama kali untuk menemukan nilai median dan kemudian menggunakananya sebagai elemen pivot, maka kompleksitas skenario terburuk akan turun menjadi  $O(n \log_2 n)$ . Median dari array dapat ditemukan dalam waktu linear, dan menggunakan median sebagai pivot menjamin bagian Quicksort dari kode akan bekerja dalam  $O(n \log_2 n)$ .

Dengan menggunakan nilai median sebagai pivot, Anda akan mendapatkan waktu eksekusi akhir  $O(n) + O(n \log_2 n)$ . Anda bisa menyederhanakannya menjadi  $O(n \log_2 n)$  karena bagian logaritmik tumbuh jauh lebih cepat daripada bagian linear.

Catatan: Meskipun mencapai  $O(n \log_2 n)$  mungkin dalam skenario terburuk Quicksort, pendekatan ini jarang digunakan dalam praktik. Daftar harus cukup besar untuk implementasinya lebih cepat daripada pemilihan pivot acak sederhana.

Memilih Pivot Secara Acak membuat skenario terburuk sangat tidak mungkin. yang mana membuat pemilihan pivot acak cukup baik untuk sebagian besar implementasi algoritma.

# Mengukur Waktu Eksekusi Implementasi Quicksort

Sekarang Anda sudah familiar dengan proses untuk mengukur waktu eksekusi algoritma. Cukup ganti nama algoritma pada baris 8:

```
if __name__ == "__main__":
    # Menghasilkan array dengan `ARRAY_LENGTH` item yang terdiri
    # dari nilai integer acak antara 0 dan 999
    array = [randint(0, 1000) for i in range(ARRAY_LENGTH)]

    # Panggil fungsi menggunakan nama algoritma pengurutan
    # dan array yang baru saja Anda buat
    run_sorting_algorithm(algorithm="quicksort", array=array)
```

Anda bisa menjalankan skrip seperti sebelumnya:

```
$ python sorting.py
Algoritma: quicksort. Waktu eksekusi minimum: 0.11675417600002902
Tidak hanya Quicksort selesai dalam waktu kurang dari satu detik, tetapi juga jauh lebih cepat daripada merge sort (0,11 detik versus 0,61 detik). Meningkatkan jumlah elemen yang ditentukan oleh ARRAY_LENGTH dari 10.000 menjadi 1.000.000 dan menjalankan skrip lagi, hasilnya adalah merge sort selesai dalam 97 detik, sedangkan Quicksort menyelesaikan pengurutan dalam 10 detik.
```

## Menganalisis Kekuatan dan Kelemahan Quicksort

Sesuai dengan namanya, Quicksort sangat cepat. Meskipun skenario terburuknya secara teoretis adalah  $O(n^2)$ , dalam praktiknya, implementasi Quicksort yang baik mengalahkan kebanyakan implementasi pengurutan lainnya. Selain itu, seperti halnya merge sort, Quicksort cukup mudah untuk diparalelkan.

Salah satu kelemahan utama Quicksort adalah tidak adanya jaminan bahwa algoritma ini akan mencapai kompleksitas waktu rata-rata. Meskipun skenario terburuk jarang terjadi, beberapa aplikasi tidak bisa berisiko dengan kinerja yang buruk, sehingga mereka memilih algoritma yang tetap berada dalam  $O(n \log 2n)$  terlepas dari input.

Seperti halnya merge sort, Quicksort juga mengorbankan ruang memori demi kecepatan. Ini bisa menjadi batasan untuk pengurutan daftar yang lebih besar.

Eksperimen cepat mengurutkan daftar sepuluh elemen menghasilkan hasil berikut:

```
Algoritma: bubble_sort. Waktu eksekusi minimum: 0.000090900000000014
```

```
Algoritma: insertion_sort. Waktu eksekusi minimum:
```

```
0.0000668190000000268
```

```
Algoritma: quicksort. Waktu eksekusi minimum: 0.0001319930000000004
```

Hasil menunjukkan bahwa Quicksort juga membayar harga rekursi ketika daftar cukup kecil, yang menyebabkan waktu eksekusi lebih lama dibandingkan dengan insertion sort dan bubble sort.