**LAPORAN PRAKTIKUM**

**ALGORITMA DAN STRUKTUR DATA**

**LANJUTAN MODUL 10**

**ANALISIS ALGORITMA**

**Disusun oleh:**

**Bima Triadmaja**

**L200210137**

**E**

**TEKNIK INFORMATIKA**

**FAKULTAS KOMUNIKASI DAN INFORMATIKA**

**UNIVERSITAS MUHAMMADIYAH SURAKARTA**

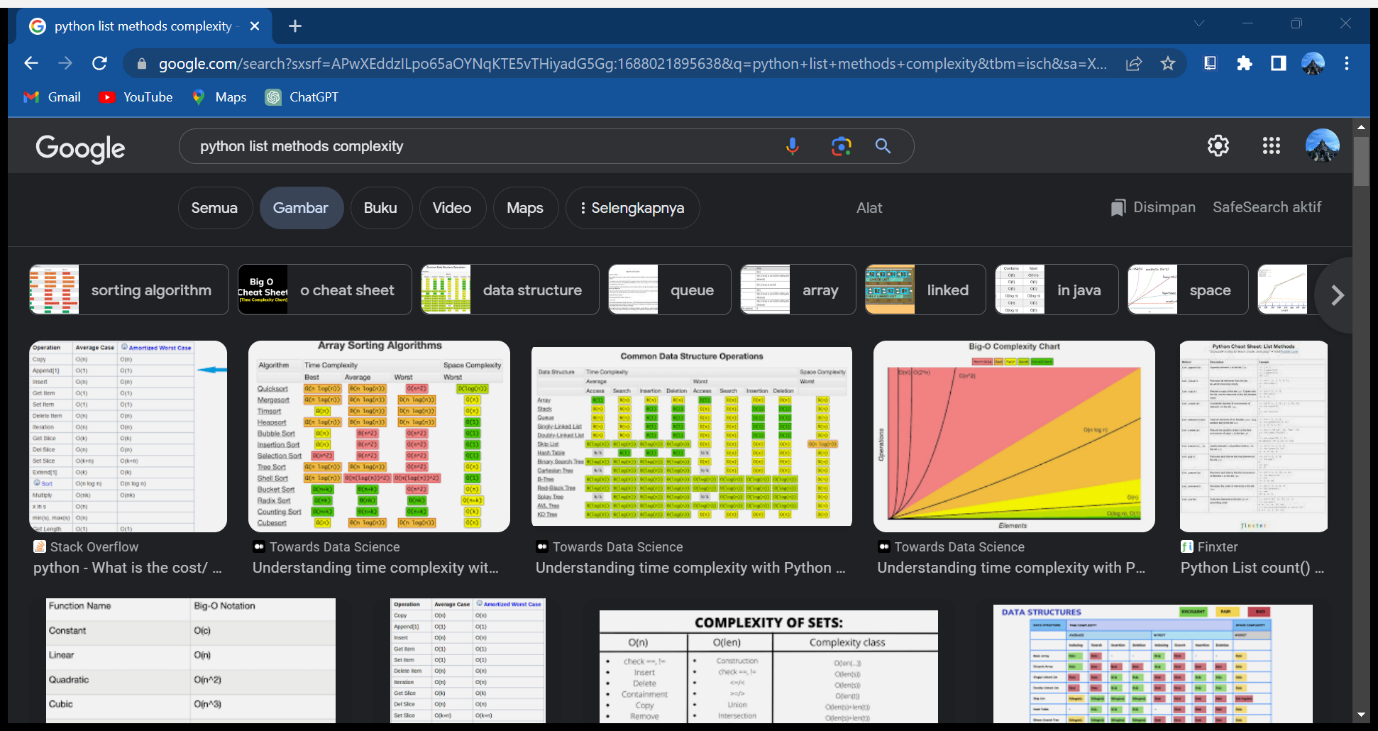
**2022/2023**

**Soal-soal untuk mahasiswa**

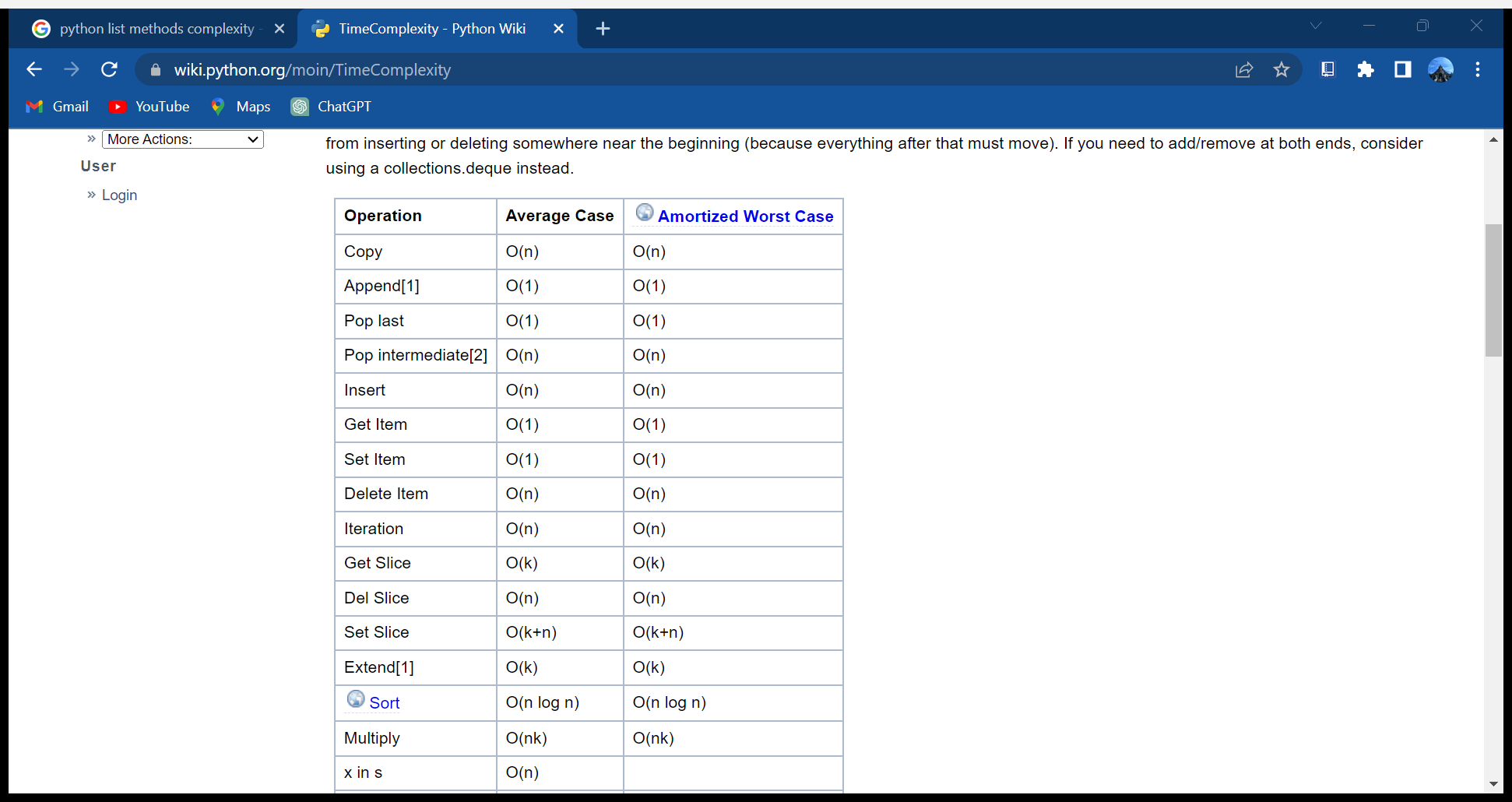
1. Urutkan dari yang pertumbuhan kompleksitasnya lambat ke yang cepat:

log4n ; 10log2n ; n log2n ; 2 log2n ; 5n2 ; n3 ; 12n6 ; 4n

1. Tentukan O(.) dari fungi-fungsi berikut yang mewakili banyaknya langkah yang diperlukan untuk beberapa algoritma.
2. T(n) = n^2 + 32n + 8 = O(n^2)
3. T(n) = 87n + 8n = O(n)
4. T(n) = 4n + 5n log n + 102 = O(n log n)
5. T(n) = log n + 3n^2 + 88 = O(n^2)
6. T(n) = 3(2^n) + n^2 + 647 = O(2^n)
7. T(n, k) = kn + log k = O(kn)
8. T(n, k) = 8n + k log n + 800 = O(n)
9. T(n, k) = 100kn + n = O(kn)
10. Carilah di internet, komplesitas metode-metode pada object list di Python.
11. Google **python list methods complexity** . Lihat juga bagian ”Images”-nya



1. Kunjungi https://wiki.python.org/moin/TimeComplexity



1. Ujicoba untuk mengkonfirmasi bahwa metode append() adalah O(1).

import time

import random

import timeit

import matplotlib.pyplot as plt

def a(n):

L = list(range(30))

L = L[::-1]

for i in range(n):

L.append(n)

def ujia(n):

ls=[]

jangkauan = range(1,n+1)

siap = "from \_\_main\_\_ import a"

for i in jangkauan:

t = timeit.timeit("a(" + str(i) +")",setup=siap, number=1)

ls.append(t)

return ls

n = 10

LS = ujia(n)

plt.plot(LS)

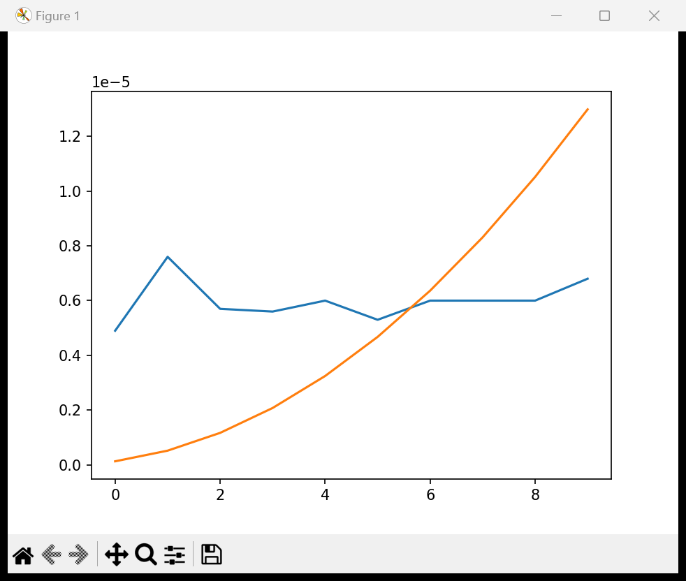
skala = 7700000

plt.plot([x\*x/skala for x in range(1,n+1)])

plt.show()

print('')

print('Sudah selesai.')

print('\n--- Oleh L200210137 ---')

1. Ujicoba untuk mengkonfirmasi bahwa metode insert() adalah O(n).

import time

import random

import timeit

import matplotlib.pyplot as plt

def a(n):

L = list(range(30))

L = L[::-1]

for i in range(n):

for b in range(n):

L.insert(i,b)

def ujia(n):

ls=[]

jangkauan = range(1,n+1)

siap = "from \_\_main\_\_ import a"

for i in jangkauan:

t = timeit.timeit("a(" + str(i) +")",setup=siap, number=1)

ls.append(t)

return ls

n = 100

LS = ujia(n)

plt.plot(LS)

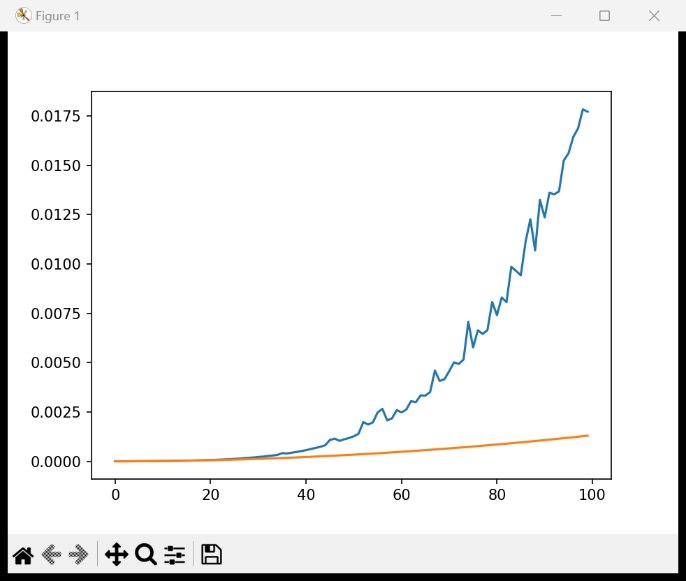
skala = 7700000

plt.plot([x\*x/skala for x in range(1,n+1)])

plt.show()

print('')

print('Sudah selesai.')

print('\n--- Oleh L200210137 ---')

1. Ujicoba untuk mengkonfirmasi bahwa unutk memeriksa apakah-suatu-nilai-berada-di-suatu-list mempunyai kompleksitas O(n).

import time

import random

import timeit

import matplotlib.pyplot as plt

def cari(lis, target):

n = len(lis)

for i in range(n):

if lis[i] == target:

return True

return False

def timee():

n = 200

lis = [1,6,7,8,3,2,11]

awal = time.time()

U = cari(lis, n)

akhir = time.time()

print("Jumlah adalah %d, memerlukan %8.7f detik" % (U, akhir-awal))

timee()

def a(n):

a = [1,6,7,8,3,2,11]

U = cari(a,n)

def ujia(n):

ls=[]

jangkauan = range(1,n+1)

for i in jangkauan:

t = timeit.timeit("a(" + str(i) +")", "from \_\_main\_\_ import a", number=1)

ls.append(t)

return ls

n = 10

LS = ujia(n)

plt.plot(LS)

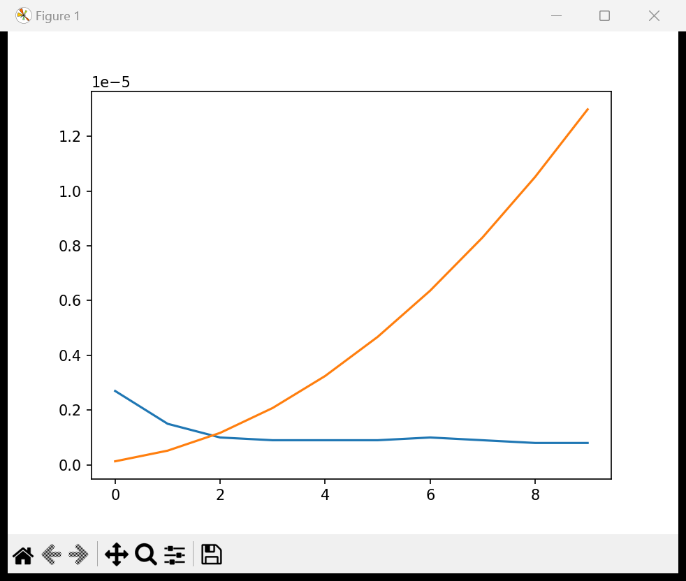
skala = 7700000

plt.plot([x\*x/skala for x in range(1,n+1)])

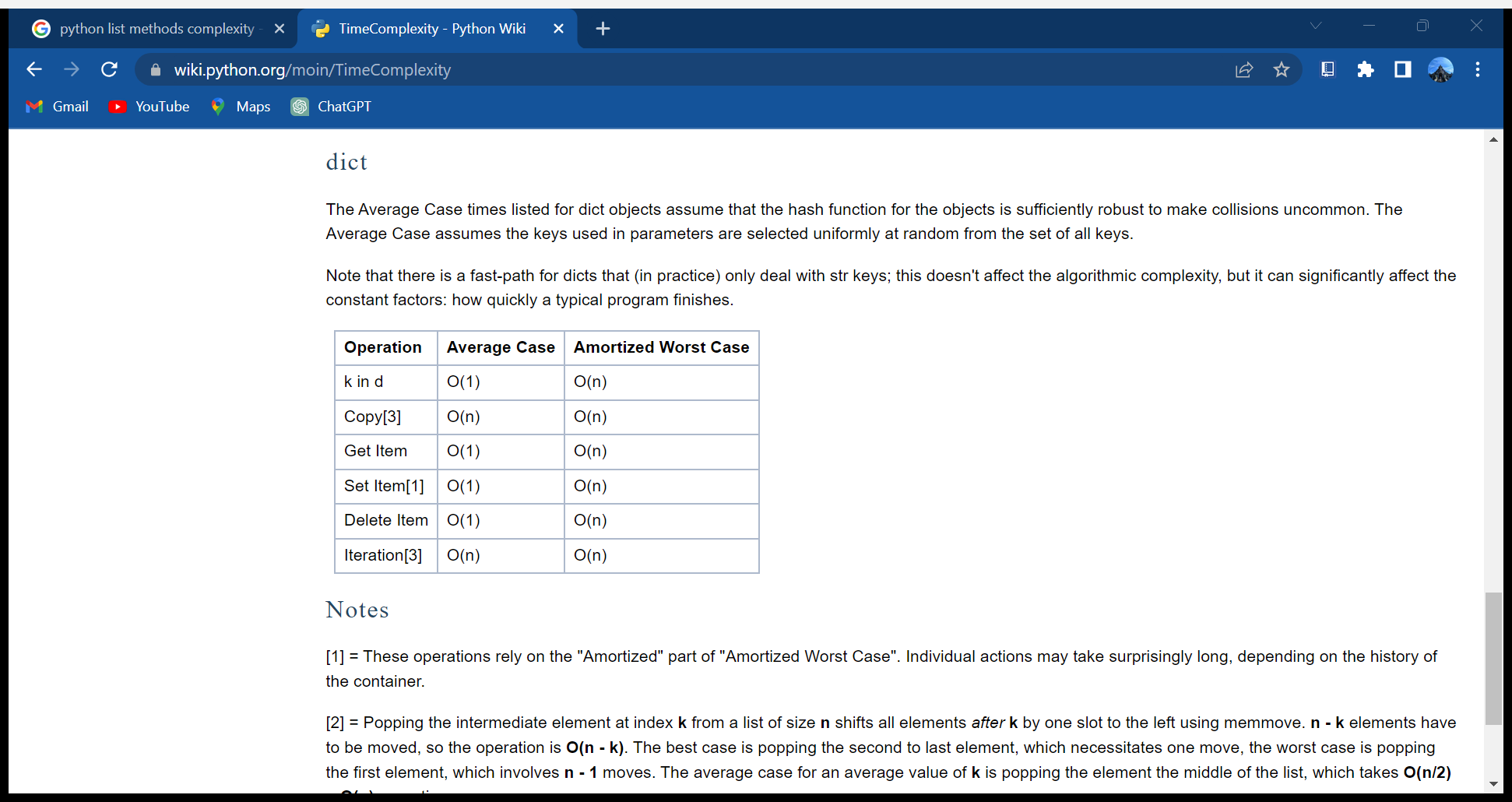
plt.show()

print('')

print('Sudah selesai.')

print('\n--- Oleh L200210137 ---')

1. Carilah di Internet, kompleksitas metode-metode pada object dict di Python.



1. Notasi Big-O (O(·)), notasi Big-Theta (Θ(·)), dan notasi Big-Omega (Ω(·)) adalah ketiga konsep yang berbeda dalam analisis kompleksitas algoritma.
2. **Notasi Big-O (O(·)) :**

Notasi Big-O digunakan untuk memberikan batas atas pada pertumbuhan waktu eksekusi atau kompleksitas algoritma. Notasi Big-O memberikan batas atas pada kinerja algoritma, tetapi tidak memberikan informasi tentang kinerja terbaik atau rata-rata.

1. **Notasi Big-Theta (Θ(·)) :**

Notasi Big-Theta digunakan untuk memberikan batas atas dan batas bawah pada pertumbuhan waktu eksekusi atau kompleksitas algoritma. Algoritma tersebut memiliki kinerja yang stabil dan terikat oleh f(n) baik pada kasus terburuk, terbaik, atau rata-rata.

1. **Notasi Big-Omega (Ω(·)) :**

Notasi Big-Omega digunakan untuk memberikan batas bawah pada pertumbuhan waktu eksekusi atau kompleksitas algoritma. Notasi Big-Omega memberikan batas bawah pada kinerja algoritma, tetapi tidak memberikan informasi tentang kinerja terbaik atau rata-rata.

1. Apa yang dimaksud dengan **amortized analysis** dalam analisis algoritma?

Amortized Analysis adalah metode untuk menganalisis kompleksitas waktu algoritma tertentu, terutama waktu atau memori yang dibutuhkan untuk eksekusi program. Tujuan dari amortized analysis adalah untuk melihat waktu eksekusi rata-rata dari sekumpulan operasi yang dilakukan oleh program.