ALGORITMA DAN STRUKTUR DATA
MODUL 10
ANALISI ALGORITMA

Disusun oleh:
Muhammad Ferizal Fadhli
L200210119
D

TEKNIK INFORMATIKA
FAKULTAS KOMUNIKASI DAN INFORMATIKA
UNIVERSITAS MUHAMMADIYAH SURAKARTA
2022/2023

Soal-Soal Untuk Mahasiswa

4. Urutkan dari yang pertumbugan kompleksitasnya lambat ke yang cepat:

   $n\log_2 n$
   4^n
   $10 \log_2 n$
   $5n^2$
   $\log_4 n$
   $12n^6$
   $2^{\log_2 nn}$
   n^3

   Berikut adalah urutan dari pertumbuhan kompleksitas yang lambat ke yang cepat:

   $\log_4 n$
   Pertumbuhan kompleksitasnya adalah logaritmik (O(log n)), karena jumlah operasi tidak tergantung pada ukuran input secara linear, melainkan bergantung pada logaritma basis 4 dari ukuran input.

   $2^{\log_2 nn}$
   Pertumbuhan kompleksitasnya adalah logaritmik (O(log n)), karena jumlah operasi tidak tergantung pada ukuran input secara linear, melainkan bergantung pada logaritma basis 2 dari ukuran input.

   $n\log_2 n$
   Pertumbuhan kompleksitasnya adalah log-linear (O(n log n)), karena jumlah operasi akan meningkat secara proporsional dengan ukuran input yang dikalikan dengan logaritma basis 2 dari ukuran input.

   4^n
   Pertumbuhan kompleksitasnya adalah linear (O(n)), karena jumlah operasi akan meningkat sebanding dengan ukuran input yang diberikan.

   $5n^2$
   Pertumbuhan kompleksitasnya adalah kuadratik (O(n^2)), karena jumlah operasi akan meningkat secara kuadratik dengan ukuran input.

   n^3
   Pertumbuhan kompleksitasnya adalah kubik (O(n^3)), karena jumlah operasi akan meningkat secara kubik dengan ukuran input.

   $10 \log_2 n$
   Pertumbuhan kompleksitasnya adalah log-linear (O(n log n)), karena jumlah operasi akan meningkat secara proporsional dengan ukuran input yang dikalikan dengan logaritma basis 2 dari ukuran input.

   $12n^6$
   Pertumbuhan kompleksitasnya adalah polinomial (O(n^6)), karena jumlah operasi akan meningkat secara polinomial dengan ukuran input.

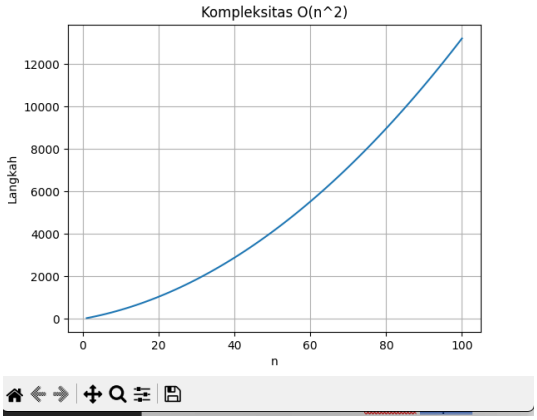
Jadi, urutan yang benar dari pertumbuhan kompleksitas yang lambat ke yang cepat adalah: $\log_4 n < 2^{\log_2 nn} < n\log_2 n < 4^n < 5n^2 < n^3 < 10\log_2 n < 12n^6$

5. Tentukan O(.) dari fungsi-fungsi berikut, yang mewakili banyaknya Langkah yang diperlukan untuk beberapa algoritma.
   a) $T(n) = n^2 + 32n + 8$

      Fungsi $T(n) = n^2 + 32n + 8$ memiliki kompleksitas $O(n^2)$ karena suku dominan pada tingkat pertumbuhannya adalah n^2. Pada tingkat pertumbuhan ini, jumlah langkah yang diperlukan meningkat secara kuadrat dengan ukuran input (n).
      Berikut adalah contoh kode Python untuk menggambarkan fungsi T(n):

```python
import matplotlib.pyplot as plt
import numpy as np


#a) T(n) = n2 + 32n + 8
n = np.arange(1, 101)  # Misalnya, kita mengambil nilai n dari 1 hingga 100
steps = n**2 + 32*n + 8

plt.plot(n, steps)
plt.xlabel('n')
plt.ylabel('Langkah')
plt.title('Kompleksitas O(n^2)')
plt.grid(True)
plt.show()
```

      Berikut output :



   b) $T(n) = 87n + 8n$

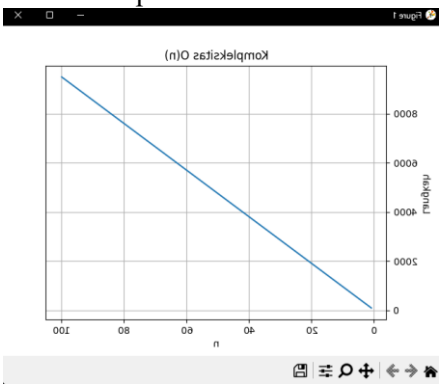
      Fungsi $T(n) = 87n + 8n$ memiliki kompleksitas $O(n)$ karena suku dominan pada tingkat pertumbuhannya adalah n. Pada tingkat pertumbuhan ini, jumlah langkah yang diperlukan meningkat secara linear dengan ukuran input (n).

      Berikut adalah contoh kode Python untuk menggambarkan fungsi T(n):

```python
n = np.arange(1, 101)  # Misalnya, kita mengambil nilai n dari 1 hingga 100
```

```
steps = 87*n + 8*n

plt.plot(n, steps)
plt.xlabel('n')
plt.ylabel('Langkah')
plt.title('Kompleksitas O(n)')
plt.grid(True)
plt.show()
```

Berikut output :



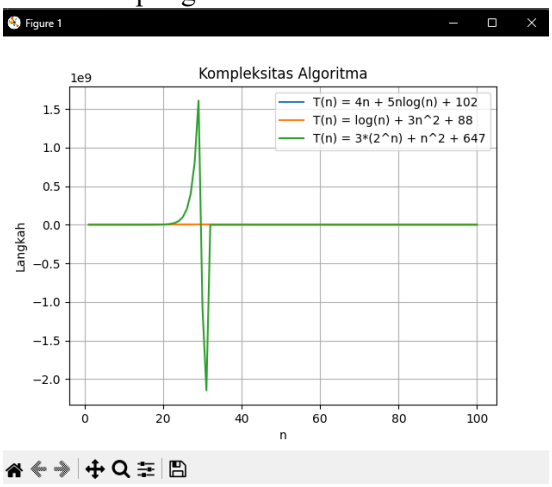
c) $T(n) = 4n + 5n\log n + 102$

O(nlogn) karena suku dominan pada tingkat pertumbuhannya adalah nlogn. Suku-suku lainnya menjadi tidak signifikan saat n meningkat.

Berikut adalah contoh kode Python untuk menggambarkan fungsi T(n):

```
# Plotting T(n) untuk fungsi c
n = np.arange(1, 101)
T_c = 4*n + 5*n*np.log(n) + 102
plt.plot(n, T_c, label='T(n) = 4n + 5nlog(n) + 102')
```

Berikut output grafik:



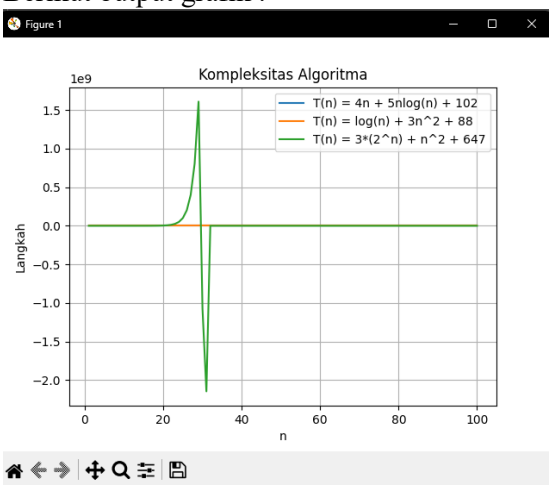
d) $T(n) = \log n + 3n^2 + 88$

O(n^2) karena suku dominan pada tingkat pertumbuhannya adalah n^2. Suku-suku lainnya menjadi tidak signifikan saat n meningkat.

Berikut adalah contoh kode Python untuk menggambarkan fungsi T(n):

```
# Plotting T(n) untuk fungsi d
T_d = np.log(n) + 3*n**2 + 88
plt.plot(n, T_d, label='T(n) = log(n) + 3n^2 + 88')
```

Berikut output grafik :



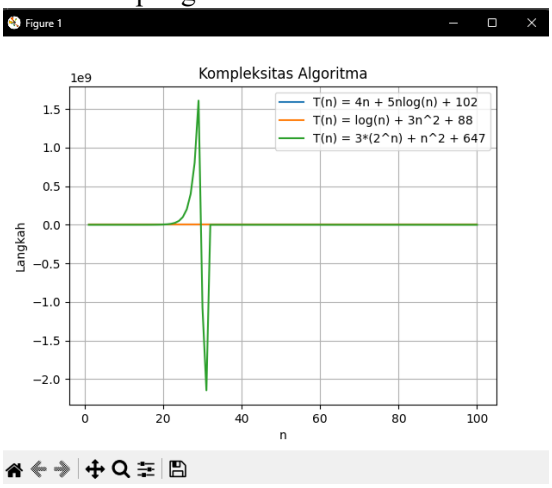
e) $T(n) = 3(2^n) + n^2 + 647$

O(n^2) karena suku dominan pada tingkat pertumbuhannya adalah n^2. Suku konstan 647 dan suku 3(2n) menjadi tidak signifikan saat n meningkat.

Berikut adalah contoh kode Python untuk menggambarkan fungsi T(n):

```
# Plotting T(n) untuk fungsi e
T_e = 3*(2**n) + n**2 + 647
plt.plot(n, T_e, label='T(n) = 3*(2^n) + n^2 + 647')
```

Berikut output grafik :



f) T(n,k) = kn +log k

O(n) karena suku dominan pada tingkat pertumbuhannya adalah n. Suku logk tidak mempengaruhi kompleksitas secara signifikan karena k merupakan konstanta.

g) T(n,k) = 8n + k log n + 800

O(n) karena suku dominan pada tingkat pertumbuhannya adalah n. Suku klogn dan suku konstan 800 menjadi tidak signifikan saat n meningkat.

h) T(n,k) = 100kn +n
O(kn) karena suku dominan pada tingkat pertumbuhannya adalah kn. Suku konstan n menjadi tidak signifikan saat n meningkat.

Note : Untuk fungsi-fungsi f, g, dan h yang melibatkan variabel k, langkah-langkah tersebut tidak dapat digambarkan dalam satu grafik karena melibatkan parameter tambahan.

6. (Literature review) carilah di internet, kompleksitas motode-meetode pada object list di Python.Hint :
   - Google python list meethods complexity. Lihat juga pada bagian “images”-nya
   - Kunjungi https://wiki.python.org/moin/TimeComplexity

   Jawab :

   Pencarian (searching):

   - Memeriksa keanggotaan (membership testing) menggunakan operator in: O(n), di mana n adalah panjang list.
   - Pencarian nilai (searching for a value): O(n), di mana n adalah panjang list.
   - Pencarian indeks (searching for an index): O(n), di mana n adalah panjang list.
   - Penambahan dan penghapusan (addition and deletion):

   Menambahkan elemen di akhir list (appending an element): O(1) pada rata-rata.

   - Menambahkan elemen pada posisi tertentu (inserting an element at a specific position): O(n), di mana n adalah panjang list jika elemen ditambahkan di tengah atau awal list.
   - Menghapus elemen dengan nilai tertentu (removing an element with a specific value): O(n), di mana n adalah panjang list.
   - Menghapus elemen pada indeks tertentu (removing an element at a specific index): O(n), di mana n adalah panjang list jika elemen dihapus dari tengah atau awal list.

   Akses elemen:

   - Mengakses elemen menggunakan indeks: O(1).
   - Slicing (mengambil subset elemen list): O(k), di mana k adalah panjang subset.

   Iterasi:

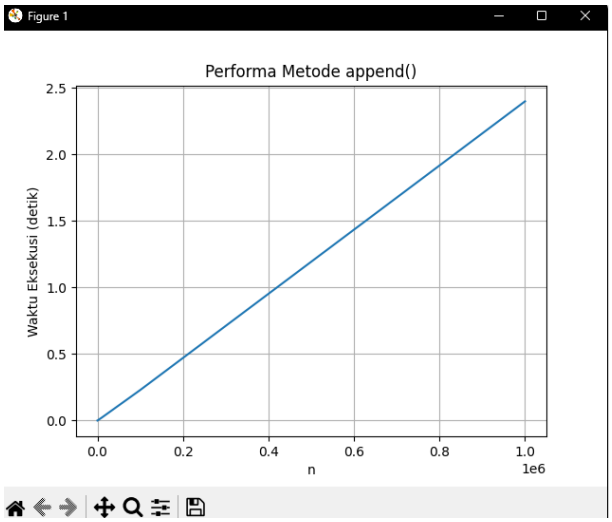
   - Iterasi melalui elemen list menggunakan loop: O(n), di mana n adalah panjang list.
7. Buatlah suatu ujicoba untuk mengkonfirmasi bahwa metode append() adalah O(1). Gunakan timeit dan matplotib.

   Berikut kode:

```
import timeit
import matplotlib.pyplot as plt

def append_test(n):
    my_list = []
    for i in range(n):
        my_list.append(i)

n_values = [10**i for i in range(1, 7)]  # Misalnya, kita menguji n dari 10 hingga 10^6
```

```
execution_times = []

for n in n_values:
    execution_time = timeit.timeit(lambda: append_test(n), number=10)  # Mengukur waktu eksekusi 10 kali
    execution_times.append(execution_time)

plt.plot(n_values, execution_times)
plt.xlabel('n')
plt.ylabel('Waktu Eksekusi (detik)')
plt.title('Performa Metode append()')
plt.grid(True)
plt.show()
```

Berikut output grafik:



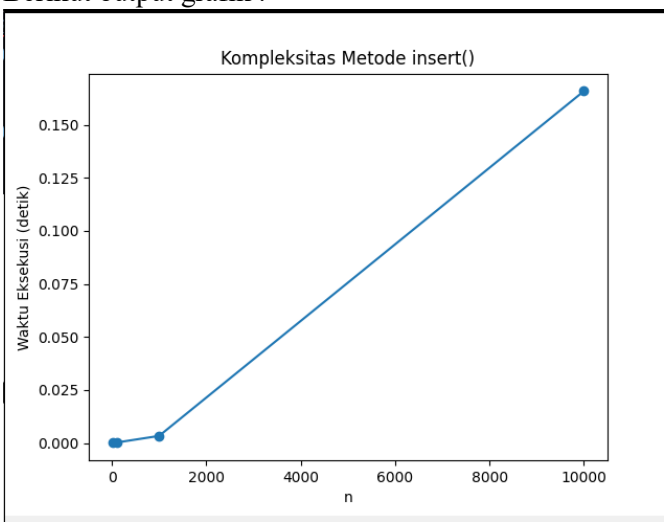
8. Buatlah suatu ujicoba untuk mengkonfirmasi bahwa metode insert() adalah O(n). Gunakan timeit dan matplotib.

Berikut kode:

```
import timeit
import matplotlib.pyplot as plt

# Membuat fungsi untuk menguji kompleksitas metode insert()
def test_insert(n):
    lst = []
    for i in range(n):
        lst.insert(0, i)

# Menguji dan mencatat waktu eksekusi untuk berbagai nilai n
```

```
n_values = [10, 100, 1000, 10000]
execution_times = []

for n in n_values:
    # Menggunakan timeit untuk mengukur waktu eksekusi
    execution_time = timeit.timeit(lambda: test_insert(n), number=10)
    execution_times.append(execution_time)

# Membuat plot waktu eksekusi terhadap nilai n
plt.plot(n_values, execution_times, '-o')
plt.xlabel('n')
plt.ylabel('Waktu Eksekusi (detik)')
plt.title('Kompleksitas Metode insert()')
plt.show()
```

Berikut output grafik :



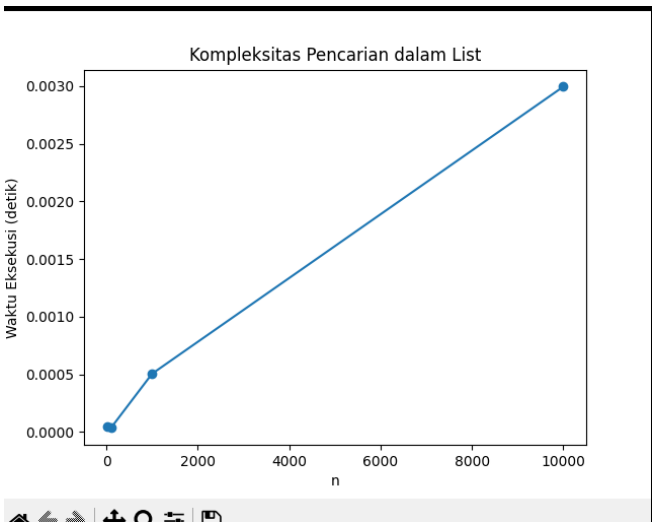
9. Buatlah suatu ujicoba untuk mengkonfirmasi bahwa untuk memeriksa apakah-suatu-nilai-berada-di-suatu-list mempunyai kompleksitas o(n). gunakan timeit dan matplotlib seperti sebelumnya.

Berikut kode:

```
import timeit
import matplotlib.pyplot as plt

# Membuat fungsi untuk menguji kompleksitas pencarian dalam list
def test_search(n):
    lst = list(range(n))
    value = n - 1
    return value in lst

# Menguji dan mencatat waktu eksekusi untuk berbagai nilai n
n_values = [10, 100, 1000, 10000]
execution_times = []
```

```
for n in n_values:
    # Menggunakan timeit untuk mengukur waktu eksekusi
    execution_time = timeit.timeit(lambda: test_search(n), number=10)
    execution_times.append(execution_time)

# Membuat plot waktu eksekusi terhadap nilai n
plt.plot(n_values, execution_times, '-o')
plt.xlabel('n')
plt.ylabel('Waktu Eksekusi (detik)')
plt.title('Kompleksitas Pencarian dalam List')
plt.show()
```

Berikut output grafik:



10. (Literatur review) carilah di internet kompleksitas metode-metode pada object dict di python.

Pencarian (searching):

- Memeriksa keanggotaan (membership testing) menggunakan operator in: O(1) pada rata-rata dan O(n) dalam kasus terburuk, di mana n adalah jumlah elemen dalam dict.
- Mendapatkan nilai berdasarkan kunci (accessing values by key): O(1) pada rata-rata dan O(n) dalam kasus terburuk.

Penambahan dan penghapusan (addition and deletion):

- Menambahkan pasangan kunci-nilai (adding key-value pairs): O(1) pada rata-rata dan O(n) dalam kasus terburuk.
- Menghapus pasangan kunci-nilai (deleting key-value pairs): O(1) pada rata-rata dan O(n) dalam kasus terburuk.

Iterasi:

- Iterasi melalui kunci (iterating through keys): O(n), di mana n adalah jumlah elemen dalam dict.
- Iterasi melalui nilai (iterating through values): O(n), di mana n adalah jumlah elemen dalam dict.
- Iterasi melalui pasangan kunci-nilai (iterating through key-value pairs): O(n), di mana n adalah jumlah elemen dalam dict.

11. (Literatur review) selain notasi bih-O O(.), ada pula notasi Big-θ (.) dan notasi Big-Ω (.). Apakah beda di antara ketiganya.

    Terdapat perbedaan antara notasi Big-O (O(.)), notasi Big-Ω (Ω(.)), dan notasi Big-θ (θ(.)). Berikut adalah penjelasan singkat mengenai perbedaan masing-masing notasi tersebut:

    a. Notasi Big-O (O(.)):

    - Notasi Big-O digunakan untuk memberikan batasan atas terhadap kompleksitas waktu atau ruang suatu algoritma.
    - Notasi Big-O menyatakan kompleksitas terburuk yang mungkin terjadi dalam suatu algoritma.
    - Secara formal, notasi Big-O (O(g(n))) menggambarkan kelas fungsi yang memberikan batasan atas pada pertumbuhan fungsi f(n) ketika n menuju tak terhingga.
    - Contohnya, jika suatu algoritma memiliki kompleksitas O(n^2), ini berarti kompleksitasnya terbatas oleh fungsi kuadratik atau lebih rendah ketika ukuran input (n) meningkat.

    b. Notasi Big-Ω (Ω(.)):

    - Notasi Big-Ω digunakan untuk memberikan batasan bawah terhadap kompleksitas waktu atau ruang suatu algoritma.
    - Notasi Big-Ω menyatakan kompleksitas terbaik yang mungkin terjadi dalam suatu algoritma.
    - Secara formal, notasi Big-Ω (Ω(g(n))) menggambarkan kelas fungsi yang memberikan batasan bawah pada pertumbuhan fungsi f(n) ketika n menuju tak terhingga.
    - Contohnya, jika suatu algoritma memiliki kompleksitas Ω(n), ini berarti kompleksitasnya setidaknya sebanding dengan fungsi linier ketika ukuran input (n) meningkat.

    c. Notasi Big-θ (θ(.)):

    - Notasi Big-θ menggabungkan batasan atas dan batasan bawah terhadap kompleksitas waktu atau ruang suatu algoritma.
    - Notasi Big-θ digunakan untuk memberikan batasan yang tepat terhadap kompleksitas algoritma.
    - Secara formal, notasi Big-θ (θ(g(n))) menggambarkan kelas fungsi yang memberikan batasan atas dan batasan bawah yang sama pada pertumbuhan fungsi f(n) ketika n menuju tak terhingga.
    - Contohnya, jika suatu algoritma memiliki kompleksitas θ(n), ini berarti kompleksitasnya terbatas oleh fungsi linier ketika ukuran input (n) meningkat, dan tidak lebih buruk atau lebih baik daripada itu.

12. (Literatur review) apa yang dimaksud dengan amortized analysis dalam analisis algoritma?

    Jawab :
    Amortized analysis adalah sebuah teknik dalam analisis algoritma yang digunakan untuk mengevaluasi kinerja algoritma yang melakukan sejumlah operasi pada sejumlah elemen data dengan melihat rata-rata kinerja operasi tersebut secara keseluruhan dalam jangka waktu yang

panjang. Teknik ini sering digunakan ketika terdapat variasi yang signifikan dalam kinerja operasi pada elemen data tertentu.

Dalam amortized analysis, kompleksitas waktu rata-rata dari serangkaian operasi dianalisis, meskipun operasi-individu mungkin memiliki kompleksitas yang berbeda-beda. Tujuannya adalah untuk menjamin bahwa dalam jangka waktu yang panjang, kinerja rata-rata setiap operasi tetap terjaga dan tidak ada operasi yang memiliki kinerja yang sangat buruk secara konsisten.

Salah satu contoh penerapan amortized analysis yang terkenal adalah pada struktur data seperti Dynamic Array (Array Dinamis) yang diimplementasikan dengan menggunakan array dan penggandaan kapasitas saat kapasitas penuh. Operasi penambahan elemen pada Dynamic Array memiliki kompleksitas O(1) pada rata-rata, namun terdapat operasi penggandaan kapasitas yang kompleksitasnya O(n). Namun, dengan menggunakan amortized analysis, kita dapat mengatakan bahwa dalam jangka waktu yang panjang, setiap operasi penambahan elemen memiliki kompleksitas rata-rata O(1).

Amortized analysis memberikan pemahaman yang lebih baik tentang kinerja algoritma dalam jangka waktu yang panjang dan membantu menghindari penilaian yang keliru terhadap kompleksitas algoritma berdasarkan kasus terburuk yang jarang terjadi. Dengan demikian, teknik ini memungkinkan kita untuk membuat keputusan yang lebih baik dalam pemilihan algoritma yang efisien untuk suatu masalah.