**8. n boyutunda bir veri kümeniz ve bu veri kümesini aynı şekilde işleyen iki algoritmanız olduğunu varsayalım. Algoritma A, veri kümesindeki her öğeyi işlemek için 10 adım attı. Algoritma B, her ögeyi 100 adımda işledi. Bu iki algoritmanın karmaşıklığı nedir?**

Cevap :

Varsayımlar:

Veri kümesi n boyutundadır.

Her iki algoritma da veri kümesini aynı şekilde işlemektedir.

Algoritma A, her öğeyi işlemek için 10 adım atmaktadır.

Algoritma B, her öğeyi işlemek için 100 adım atmaktadır.

Karmaşıklık:

Algoritma A, her öğe için **10 adım** atmaktadır. Veri kümesinin boyutu *n* olduğunda, algoritma A'nın karmaşıklığı:

*O(10n)=O(n)*

Burada, sabit bir katsayı olan 10, büyüme oranını değiştirmez, bu yüzden zaman karmaşıklığı **O(n)** olur.

Algoritma B, her öğe için **100 adım** atmaktadır. Veri kümesinin boyutu *nn*n olduğunda, algoritma B'nin karmaşıklığı:

O(100n)=O(n)

Burada da, sabit bir katsayı olan 100 büyüme oranını değiştirmez, dolayısıyla algoritma B'nin karmaşıklığı da **O(n)** olur.

Sonuç:

Her iki algoritmanın da karmaşıklığı *O(n)O(n)*O(n) şeklindedir. Yani, her iki algoritma da **lineer zaman karmaşıklığına** sahiptir.

Sabit katsayılar (10 ve 100 gibi) büyük n için önemli olsa da, karmaşıklık büyük n için sadece en yüksek büyüme oranını ifade eder, bu nedenle her iki algoritmanın da zaman karmaşıklığı **O(n)** olarak kabul edilir.

**Algoritma A** , **Algoritma B**'den daha verimlidir.

Ancak, burada **Big-O notasyonu** kullanılan bir ölçüttür, ve Big-O, yalnızca bir algoritmanın **zaman karmaşıklığının büyüme oranını** belirler. Big-O, sabit katsayıları (10, 100 gibi) dikkate almaz; sadece *nn*n büyüdükçe algoritmanın nasıl davrandığına bakar.

**9. Bir listenin append işleminin + operatöründen daha verimli olmasının nedenini açıklayın.**

Cevap:

Bir listenin append() işlemi, + operatöründen daha verimli olmasının nedeni, bu iki işlemin arka planda farklı şekillerde çalışmasıdır.

### **append():**

* append() fonksiyonu, mevcut bir listeye eleman eklemek için kullanılan bir yöntemdir.
* Bu işlem, listeyi doğrudan değiştirdiği için, **yeni bir liste oluşturmaz**. Yani mevcut liste üzerinde işlem yapar.
* Bu, **O(1) zaman karmaşıklığı** ile gerçekleşir. Yani listeye her eleman eklemek sabit bir zaman alır.

### **+ Operatörü:**

* + operatörü ile iki liste birleştirildiğinde, aslında yeni bir liste oluşturulur ve her iki listenin elemanları bu yeni listeye kopyalanır.
* Bu işlem, **O(n)** zaman karmaşıklığına sahiptir çünkü her iki listenin elemanları yeni listeye kopyalanır. n, burada ilk listenin eleman sayısıdır.
* Yani + operatörü her defasında yeni bir liste oluşturduğu ve tüm elemanları kopyaladığı için daha maliyetlidir.

### **Özetle:**

* append(), var olan listeyi doğrudan değiştirdiği için **daha hızlı ve daha verimlidir**.
* + operatörü, yeni bir liste oluşturduğundan ve her elemanı kopyaladığından **daha fazla bellek kullanır ve daha yavaş çalışır**.

Bu yüzden, bir listeye tek bir eleman eklemek için append(), daha büyük bir listeyi birleştirmek için ise + operatörü kullanmak daha yaygın ve doğru bir yaklaşımdır.

10. Bir listede belirli bir değeri bulmak için basit bir algoritma tanımlayalım. Bu algoritmanın amacı, bir listenin her elemanını sırasıyla kontrol etmek ve hedef değeri bulmaktır.

Cevap:

Algoritma:

Dizisel arama algoritması kullanabiliriz.

Dizisel arama algoritması, listedeki her öğeyi sırayla kontrol eder.

Aranan değer listedeyse, algoritma o değerin bulunduğu indisi döndürür.

Aranan değer listede değilse, algoritma -1 döndürür.

Varsayımlar:

Liste sıralı değildir.

Aranan değer listede bir veya birden fazla kez bulunabilir.

Hesaplama Karmaşıklığı:

En kötü durumda, algoritma listedeki her öğeyi kontrol etmek zorunda kalacaktır.

Bu nedenle, algoritmanın hesaplama karmaşıklığı O(n) şeklindedir.

Kod Örneği:

def find\_value(my\_list, value):

for i in range(len(my\_list)):

if my\_list[i] == value:

return i

return -1

my\_list = [1, 2, 3, 4, 5]

value = int(input("Sayı giriniz: "))

index = find\_value(my\_list, value)

if index == -1:

print("Değer listede bulunamadı.")

else:

print(f'{value} sayısı {index}. indeksde bulundu')

**2.13 Programlama Problemleri**

1. Python'da string karşılaştırmanın karmaşıklığını keşfetmek için bir deney tasarlayın. String'in boyutu string karşılaştırmasının verimini etkiler mi ve etkiliyorsa, karşılaştırmanın karmaşıklığı nedir?Bu deneyde en iyi durum, en kötü durum ve ortalama durum karmaşıklığını göz önünde bulundurmak isteyebilirsiniz. Sonuçlarınızı bu bölümde belirtilen formatta içeren bir XML dosyası oluşturan bir program yazın. Ardından, bu sonuçları görselleştirmek için PlotData.py programını kullanın.

Cevap:

import time

import random

import string

def string\_karsilastirma(str1, str2):

start\_time = time.time\_ns() # Fonksiyonun başlangıcındaki zamanı al

result = str1 == str2 # İki stringi karşılaştır

end\_time = time.time\_ns() # Fonksiyonun sonundaki zamanı al

return end\_time - start\_time # Geçen süreyi döndür

uzunluk = 100 # Oluşturulacak stringlerin uzunluğunu belirle

# random.choices() fonksiyonu, belirtilen uzunlukta rastgele karakterler seçer. string.#ascii\_lowercase küçük harfleri içeren bir string,

# string.ascii\_uppercase ise büyük harfleri içeren bir string döndürür.

str1 = ''.join(random.choices(string.ascii\_lowercase, k=uzunluk))

str2 = ''.join(random.choices(string.ascii\_uppercase, k=uzunluk))

sure = string\_karsilastirma(str1, str2) # Karşılaştırma işlemi için süreyi hesapla

# Sonucu göster

print("String Uzunluğu:", uzunluk)

print("Karşılaştırma Süresi (nanosaniye):", sure)

1. İki sayının çarpımının, çarpılan iki sayının boyutuna bağlı olmadığını kanıtlamak için bir deney yapın. Farklı boyuttaki sayıları birbirleriyle çarpan sonuçları gösteren bir program yazın. İPUCU: İyi bir okuma elde etmek için bir grup çarpma işlemi yapabilir ve bunları grup olarak zamanlayabilirsiniz, çünkü bir çarpma işlemi bilgisayarda oldukça hızlı gerçekleşir. Gerçekten de bu bir O(1) işlemi midir? Bunu doğrulayın. Herhangi bir anormallik görüyor musunuz? Bu durum, Python'un büyük tam sayıları desteklemesi ile açıklanabilir. Sabit zaman içinde çarpma işlemlerini işlemenin kesme noktası nedir? Neden? Sonuçlarınızı bu bölümde verilen formatta içeren bir XML dosyası üreten bir program yazın. Sonra, bu sonuçları bu bölümde sağlanan PlotData.py programı ile görselleştirin.

Cevap:

import time

def carpma\_suresi(x, y):

start\_time = time.time\_ns()

x \* y

end\_time = time.time\_ns()

return end\_time - start\_time

# İki farklı boyuttaki sayıları belirle

x1 = 5

y1 = 7

x2 = 92342121

y2 = 56782122

# İki farklı boyuttaki sayıları çarpma işlemi için süreyi ölç

sure1 = carpma\_suresi(x1, y1)

sure2 = carpma\_suresi(x2, y2)

# Sonuçları göster

print("1. Boyuttaki Sayılar Çarpma Süresi:", sure1, "mikrosaniye")

print("2. Boyuttaki Sayılar Çarpma Süresi:", sure2, "mikrosaniye")

1. Integer değerlerini karşılaştırmakla ilgili deneysel veriler toplayan bir program yazınız. Farklı boyutlardaki Integerları karşılaştırın ve bu karşılaştırmaları yapmak için geçen süreyi çizerek gösterin. Sonuçlarınızı Ploy.py formatında bir XML dosyası yazarak gösterin. "Karşılaştırma işlemi her zaman O(1) midir? Eğer değilse, nedenini açıklayabilirmisiniz İPUCU: Python'un “large ıntegers” hakkındaki belgeyi okumak isteyebilirsiniz.

Cevap:

Evet, compare\_numbers fonksiyonunun karşılaştırma işlemi **O(1)** karmaşıklığa sahiptir.

**Neden O(1)?**

* Fonksiyon, iki sayıyı (num1 ve num2) karşılaştırır.
* Karşılaştırma, if-elif-else yapısıyla yapılır.
* Bu yapı, her durumda **sabit sayıda** karşılaştırma gerçekleştirir:
  + num1 num2'den küçükse **1 karşılaştırma**.
  + num1 num2'ye eşitse **2 karşılaştırma**.
  + Diğer durumlarda **2 karşılaştırma**.

import time

from lxml import etree as ET

import matplotlib.pyplot as plt

def compare\_numbers(num1, num2):

if num1 < num2:

return -1

elif num1 == num2:

return 0

else:

return 1

def experimental\_data(start, end):

data = []

for i in range(start, end):

start\_time = time.time()

compare\_numbers(i, i+1)

end\_time = time.time()

elapsed\_time = end\_time - start\_time

data.append((i, elapsed\_time))

return data

def write\_xml(data, filename):

root = ET.Element("ExperimentalResults")

for item in data:

element = ET.SubElement(root, "Comparison")

ET.SubElement(element, "Number").text = str(item[0])

ET.SubElement(element, "Time").text = str(item[1])

tree = ET.ElementTree(root)

tree.write(filename, pretty\_print=True)

def plot\_results(data):

numbers = [item[0] for item in data]

times = [item[1] for item in data]

plt.plot(numbers, times)

plt.xlabel('Numbers')

plt.ylabel('Time (s)')

plt.title('Experimental Comparison Times')

plt.show()

# Örnek kullanım

start = 1

end = 10000

experimental\_results = experimental\_data(start, end)

print(experimental\_results)

# XML dosyası oluşturma

write\_xml(experimental\_results, 'experimental\_results.xml')

# Sonuçları çizimleyerek görselleştirme

plot\_results(experimental\_results)

1. Belirli bir değeri bir listede arayan ve bu değerin listenin içindeki konumunu (yani indeksini) döndüren kısa bir fonksiyon yazın. Ardından, farklı boyutlardaki listelerde bir ögeyi aramak için ne kadar zaman aldığını ölçen bir program yazın. Liste boyutu n'nizdir. Bu deneyden elde edilen sonuçları toplayın ve bu sonuçları Ploy.py formatındaki bir XML dosyasına yazın. Bu algoritmanın karmaşıklığı nedir? Bu soruya programınızdaki bir yorumla cevap verin ve deneysel sonuçların tahmininizle eşleşip eşleşmediğini doğrulayın. Ardından, bu durumu bir listenin `index` metoduna karşı karşılaştırın. Hesaplama karmaşıklığı açısından hangisi daha verimlidir? İPUCU: Bu sorun için sadece basit bir durumu değil, ortalama durumu düşünmek önemlidir.

Cevap:

Bu soruyu çözmek için aşağıdaki adımları izleyelim:

1. **Bir değeri liste içinde arama fonksiyonu yazalım:** Bu fonksiyon, belirtilen bir değerin listenin içindeki indeksini döndürecektir.
2. **Farklı boyutlardaki listelerdeki arama süresini ölçmek:** Liste boyutu n olan farklı listelerde bu arama fonksiyonunun çalışma süresini ölçebiliriz.
3. **Sonuçları bir XML dosyasına kaydedelim:** Ploy.py formatında bir XML dosyası oluşturacağız ve deneysel sonuçları bu dosyaya yazacağız.
4. **Algoritmanın karmaşıklığı hakkında yorum yapalım:** Bu algoritmanın karmaşıklığını inceleyeceğiz ve deneysel sonuçlarla tahmin edilen karmaşıklığı karşılaştıracağız.
5. **Python'daki index metodunun karşılaştırması:** Python'da listelerdeki index() metodunun ne kadar verimli olduğunu değerlendireceğiz.

### **1. Değeri Liste İçinde Arama Fonksiyonu**

Öncelikle, bir değeri liste içinde arayan basit bir fonksiyon yazalım:

Python

import time  
  
def find\_index(lst, value):  
 try:  
 return lst.index(value) # Bu Python'un yerleşik index metodunu kullanıyoruz  
 except ValueError:  
 return -1 # Değer bulunamazsa -1 döndür

### **2. Arama Süresi Ölçme Programı**

Farklı boyutlardaki listelerde find\_index fonksiyonunun çalışma süresini ölçmek için, aşağıdaki gibi bir fonksiyon yazabiliriz:

python

import time  
import random  
import xml.etree.ElementTree as ET  
  
def measure\_search\_time():  
 sizes = [100, 1000, 10000, 100000] # Farklı boyutlar  
 results = []  
  
 for size in sizes:  
 lst = random.sample(range(size\*10), size) # 0'dan size\*10'a kadar benzersiz rastgele sayılarla liste oluştur  
 value\_to\_find = lst[size // 2] # Listenin ortasındaki değeri arayalım  
  
 start\_time = time.time()  
 index = find\_index(lst, value\_to\_find)  
 end\_time = time.time()  
  
 elapsed\_time = end\_time - start\_time  
 results.append((size, elapsed\_time))  
   
 return results

### **3. Sonuçları XML Dosyasına Yazma**

Elde ettiğimiz sonuçları bir XML dosyasına yazmak için aşağıdaki kodu kullanabiliriz:

Python

def write\_results\_to\_xml(results, filename="results.xml"):  
 root = ET.Element("Results")  
   
 for size, time\_taken in results:  
 result\_element = ET.SubElement(root, "Result")  
 size\_element = ET.SubElement(result\_element, "Size")  
 time\_element = ET.SubElement(result\_element, "Time")  
  
 size\_element.text = str(size)  
 time\_element.text = str(time\_taken)  
  
 tree = ET.ElementTree(root)  
 tree.write(filename)  
  
# Sonuçları ölçüp XML dosyasına yazalım  
results = measure\_search\_time()  
write\_results\_to\_xml(results)

### **4. Algoritmanın Karmaşıklığı**

find\_index fonksiyonunun karmaşıklığını değerlendirelim:

* Bu algoritma bir listeyi sırayla tarar ve belirli bir öğeyi bulduğunda durur. Bu durumda, karmaşıklığı **O(n)**'dir. Yani, aradığınız öğe listenin ortasında veya sonlarında olduğunda, tüm listeyi taramanız gerekebilir.

Deneysel olarak, farklı liste boyutlarında elde edilen sonuçlar, bu tahminle uyumlu olacaktır. Örneğin, liste boyutuyla doğru orantılı bir zaman artışı gözlemleyebiliriz.

### **5. index() Metodunun Karşılaştırılması**

Python'un yerleşik index() metodu da benzer şekilde listenin her elemanını tek tek kontrol eder. Bu nedenle karmaşıklığı **O(n)**'dir. Bu yöntem, Python'un optimize edilmiş bir versiyonunu kullanıyor olsa da, temel mantık aynı şekilde bir elemanı aramak olduğundan karmaşıklık açısından yine **O(n)** olacaktır.

### **6. Sonuçların Tahmini ile Uyum**

Elde edilen sonuçları gözlemlediğimizde, liste boyutu arttıkça arama süresinin doğrusal olarak arttığını görmeliyiz. Bu, algoritmanın **O(n)** karmaşıklığı ile uyumludur. Büyük liste boyutları için bu sürenin arttığını gözlemlemek, algoritmanın doğruluğunu kanıtlar.

### **7. index ile Kıyaslama**

Her iki yöntemin de (kendi find\_index fonksiyonumuzun ve Python'un index() metodunun) karmaşıklığı **O(n)** olduğu için, uygulamada hangi yöntemin daha verimli olduğu genellikle dilin optimize edilmiş yerleşik fonksiyonları kullanmakla ilgilidir. Python'un index() metodu büyük olasılıkla daha verimli olacaktır, çünkü Python'un yerleşik fonksiyonları genellikle optimize edilmiş ve düşük seviyeli işlemlerle yazılmıştır.

### **Sonuç**

Sonuç olarak, deneysel sonuçlar, teorik karmaşıklık tahminimizle uyumlu olacaktır. Hem kendi yazdığımız fonksiyon hem de Python'un yerleşik index() fonksiyonu aynı karmaşıklığa sahiptir, ancak yerleşik fonksiyon genellikle daha verimli çalışacaktır.