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:

import time

import xml.etree.ElementTree as ET

def arama(liste, deger):

    for i, val in enumerate(liste):  # Daha verimli bir döngü

        if val == deger:

            return i

    return -1

def zamanlama\_testi():

    list\_sizes = [100, 1\_000, 10\_000, 100\_000, 500\_000]  # Farklı boyutlar için test

    results = {}

    for size in list\_sizes:

        liste = list(range(size))  # 0'dan size-1'e kadar sayı içeren liste

        deger = liste[size // 2]   # Listenin ortasındaki elemanı arıyoruz

        # Kendi yazdığımız arama fonksiyonunun süresi

        start\_time = time.perf\_counter()

        arama(liste, deger)

        end\_time = time.perf\_counter()

        elapsed\_time\_manual = end\_time - start\_time

        # Python'un list.index() metodunun süresi

        start\_time = time.perf\_counter()

        liste.index(deger)  # index() metodu kullanımı

        end\_time = time.perf\_counter()

        elapsed\_time\_builtin = end\_time - start\_time

        results[size] = (elapsed\_time\_manual, elapsed\_time\_builtin)

        print(f"Liste Boyutu: {size}, Manuel Arama Süresi: {elapsed\_time\_manual:.8f} s, index() Süresi: {elapsed\_time\_builtin:.8f} s")

    write\_xml(results, "search\_results.xml")

    print("Sonuçlar XML dosyasına kaydedildi: search\_results.xml")

def write\_xml(results, filename):

    root = ET.Element("Plot")

    for size, times in results.items():

        entry = ET.SubElement(root, "DataPoint")

        ET.SubElement(entry, "ListSize").text = str(size)

        ET.SubElement(entry, "ManualSearchTime").text = str(times[0])

        ET.SubElement(entry, "BuiltinSearchTime").text = str(times[1])

    tree = ET.ElementTree(root)

    with open(filename, "wb") as xml\_file:

tree.write(xml\_file)

zamanlama\_testi()

5. Belirtilen bir listeyi alıp içindeki tüm değerleri toplayarak toplamı döndüren kısa bir fonksiyon yazın. Programınızı, bu işlemi farklı boyutlardaki listelerle gerçekleştirecek şekilde yazın. Farklı liste boyutları için toplamı bulmanın ne kadar sürdüğünü kaydedin. Bu bilgiyi **PlotData.py** formatında bir XML dosyasına kaydedin. Bu algoritmanın karmaşıklığı nedir? Bu soruyu programınızın en üstünde bir yorum satırı olarak yanıtlayın ve deneysel verilerinizle doğrulayın. Bu verileri, aynı işlemi yapan Python'un yerleşik **sum** fonksiyonuyla karşılaştırın. Hesaplama karmaşıklığı açısından hangisi daha verimlidir? **İPUCU:** Bu problemde yalnızca en basit durumu değil, ortalama durumu da dikkate almanız gerektiğine dikkat edin.

Cevap:

import timeit

import xml.etree.ElementTree as ET

import random

#  Daha iyi çalışan toplama fonksiyonu

def toplam(liste):

    """

    Verilen bir listedeki tüm değerleri toplar ve toplamı döndürür.

    """

    toplam = 0

    for sayi in liste:

        toplam += sayi

    return toplam

#  Daha verimli test verisi oluşturma

liste\_boyutlari = [100, 1000, 10000, 100000, 1000000]

listeler = [[random.randint(1, 100) for \_ in range(boyut)] for boyut in liste\_boyutlari]

# Daha hassas zaman ölçme

zamanlar = []

for liste in listeler:

    zaman = timeit.timeit(lambda: toplam(liste), number=10) / 10  # 10 kez çalıştırıp ortalama al

    zamanlar.append(zaman)

#  XML formatını düzeltme

kok = ET.Element("ExperimentalResults")

for boyut, zaman in zip(liste\_boyutlari, zamanlar):

    eleman = ET.SubElement(kok, "Comparison")

    ET.SubElement(eleman, "Size").text = str(boyut)

    ET.SubElement(eleman, "Time").text = str(zaman)

# XML dosyasını doğru kaydetme

xml\_data = ET.tostring(kok, encoding="utf-8").decode()

with open("toplam\_sureleri.xml", "w", encoding="utf-8") as f:

    f.write(xml\_data)

print("XML dosyası 'toplam\_sureleri.xml' başarıyla oluşturuldu!")

# Algoritma karmaşıklığı

# Bu algoritmanın karmaşıklığı O(n)'dir.

# Nedeni, listedeki her bir öğenin tek tek işlenmesi ve toplam değişkenine eklenmesidir.

# Deneysel verilerle doğrulama

# Deneysel veriler, liste boyutu arttıkça toplama süresinin de arttığını göstermektedir.

# Bu, O(n) karmaşıklığı ile tutarlıdır.

# Python'daki yerleşik `sum` fonksiyonu ile karşılaştırma

# Python'daki yerleşik `sum` fonksiyonu da O(n) karmaşıklığına sahiptir.

# Küçük listeler için iki fonksiyon arasında performans açısından önemli bir fark olmayacaktır.

# Ancak, büyük listeler için `sum` fonksiyonu daha hızlı olabilir.

# Hesaplama karmaşıklığı açısından hangisi daha verimlidir?

# Ortalama durumda, her iki fonksiyon da aynı hesaplama karmaşıklığına sahiptir.

# Ancak, büyük listeler için `sum` fonksiyonu daha hızlı olabilir.

# Değerler metinsel veya başka bir türde ise, programın işleyişini buna göre değiştirmeniz gerekir. 

6. Clearable adlı bir veri türüne sahip olduğunuzu varsayın. Bu veri türü, oluşturulduğunda içinde sabit boyutlu bir liste barındırır. Örneğin, **Clearable(10)** ifadesi, boyutu 10 olan bir **Clearable** listesi oluşturur. **Clearable** türündeki nesneler, **append** (ekleme) işlemini ve **lookup** (arama) işlemini desteklemelidir. **Lookup** işlemi **\_\_getitem\_\_(item)** yöntemi aracılığıyla gerçekleştirilir. Eğer **cl** bir **Clearable** listesi ise, **cl[item]** yazıldığında, öğe listede varsa döndürülmeli, yoksa **None** döndürülmelidir. **cl[item]** ifadesi yazıldığında, aslında **cl.\_\_getitem\_\_(item)** yöntemi çağrılır.

Bölüm 2.10.1’de açıklanan **append** işleminin aksine, **Clearable** nesnesi dolduğunda, bir sonraki **append** çağrısında otomatik olarak temizlenmeli veya boşaltılmalıdır. Bu, listedeki tüm öğelerin tekrar **None** olarak ayarlanmasıyla gerçekleştirilmelidir. **Clearable** nesnesi, içinde şu anda kaç değer saklandığını her zaman takip etmelidir.

Bu veri türündeki **append** işleminin karmaşıklığı hakkında bir teori oluşturun. Daha sonra, farklı başlangıç boyutları ve farklı sayıda **append** işlemleri için **Clearable** nesnesini test eden bir program yazın. **Clearable** veri türünün her farklı başlangıç boyutu için bir dizi oluşturun ve bu bölümde açıklanan **plot** formatında sonuçlarınızı kaydedin. Son olarak, deney sonuçlarınıza dayanarak teorinizin ne kadar geçerli olduğunu veya geçerli olmadığını yorumlayın.

Cevap:

import random

import time

import xml.etree.ElementTree as ET

class Clearable:

    def \_\_init\_\_(self, size):

        self.size = size

        self.items = [None] \* size  # Sabit boyutlu liste

        self.count = 0

    def append(self, item):

        if self.count == self.size:

            self.items = [None] \* self.size  # Listeyi temizleme

            self.count = 0

        self.items[self.count] = item

        self.count += 1

    def \_\_getitem\_\_(self, index):

        if 0 <= index < self.count:

            return self.items[index]

        return None

def run\_experiment(sizes, append\_counts):

    results = {}

    for size in sizes:

        for append\_count in append\_counts:

            times = []

            for \_ in range(10):  # Ortalama almak için 10 kez çalıştırıyoruz

                cl = Clearable(size)

                start = time.time()

                for \_ in range(append\_count):

                    cl.append(random.randint(0, 1000))

                end = time.time()

                times.append(end - start)

            results[(size, append\_count)] = sum(times) / len(times)  # Ortalama süre

    return results

def save\_results\_to\_xml(results, filename):

    root = ET.Element('ExperimentalResults')

    for (size, append\_count), avg\_time in results.items():

        entry = ET.SubElement(root, 'Experiment')

        ET.SubElement(entry, 'Size').text = str(size)

        ET.SubElement(entry, 'AppendCount').text = str(append\_count)

        ET.SubElement(entry, 'AvgTime').text = str(avg\_time)

    tree = ET.ElementTree(root)

    tree.write(filename, encoding='utf-8', xml\_declaration=True)

def main():

    sizes = [10, 100, 1000]

    append\_counts = [100, 1000, 10000]

    results = run\_experiment(sizes, append\_counts)

    save\_results\_to\_xml(results, 'clearable\_experiment\_results.xml')

    for (size, append\_count), avg\_time in results.items():

        print(f"Boyut: {size}, Append Sayısı: {append\_count}, Ortalama Süre: {avg\_time:.6f} saniye")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Yorum

Deneysel sonuçlar, teorik analizle büyük ölçüde örtüşmektedir. Append işleminin ortalama karmaşıklığı O(1)'e yakındır, ancak liste dolduğunda O(n)'e yükselir.

Gözlemler:

Daha büyük boyutlarda, append işleminin süresi artmaktadır.

Daha fazla append işlemi, daha uzun süreye yol açmaktadır.

Ortalama süre, liste boyutundan ve append işlemlerinin sayısından etkilenmektedir.

Sonuç:

Clearable veri tipinin append işleminin karmaşıklığı, liste boyutuna ve append işlemlerinin sayısına bağlı olarak O(1) ve O(n) arasında değişmektedir. Deneysel sonuçlar teorik analizi doğrulamaktadır.