**Hesaplamalı karmaşıklık**

**2**

Geçen bölümde bir çizim programı geliştirdik. Çizim komutlarını tutmak için yerleşik Python liste sınıfına benzeyen *PyList* sınıfını oluşturduk. Bu sınıf, ilk veri yapımızı göstermeyi kolaylaştırıyor. Bir çizim komutunu eklediğimizde, append yöntemini çağırıyoruz. Ancak bu yöntemin özellikle çok sayıda komut ekleyeceğimiz durumlarda fazla kullanıldığını fark ettik. Örneğin, birinci bölümdeki çiçek resmi yaklaşık 700 komut gerektiriyordu. Serbest çizim içeren karmaşık resimlerde ise bu sayı binlerce komuta ulaşabilir. Serbest çizim oluştururken bir sonraki çizim komutunu hızlı bir şekilde eklemek önemlidir. Peki, bir çizim komutunu listeye eklemek ne kadar sürer? Tahminde bulunabilir miyiz? Bu zaman miktarını önemsemeli miyiz?

Bu bölümde, aşağıdaki soruları nasıl cevaplayacağınızı ve bir bilgisayar programcısı olarak sizin için hangi soruların önemli olduğunu öğreneceksiniz. Öncelikle, bir bilgisayarın bazı basit işlemleri gerçekleştirmesinin ne kadar sürdüğünü anlamak için bilgisayar mimarisinin bazı ilkelerini öğreneceksiniz. Bu bilgiyle, yazdığınız kodun yürütülmesinin ne kadar sürebileceğine dair bilinçli kararlar vermek için ihtiyaç duyacağınız araçlara sahip olacaksınız.

**2.1 Bölüm Hedefleri**

Bu bölümün sonunda aşağıdaki soruları cevaplayabilmelisiniz.

* Bilgisayarın gerçekleştirebileceği bazı temel işlemler nelerdir?
* Bu temel işlemleri gerçekleştirmek ne kadar sürer?
* Hesaplamalı karmaşıklık terimi ne anlama gelir?
* Neden hesaplamalı karmaşıklık ile ilgileniyoruz?
* Bir kod parçasının karmaşıklığı hakkında ne zaman endişelenmeliyiz?
* Bir kod parçasının verimliliğini nasıl artırabiliriz?
* Big-Oh gösteriminin tanımı nedir?
* Theta notasyonunun tanımı nedir?
* Amorti edilmiş karmaşıklık nedir ve önemi nedir?
* PyList konteyner sınıfını daha iyi hale getirmek için öğrendiklerimizi nasıl uygulayabiliriz?

**2.2 Bilgisayar Mimarisi**

Bir bilgisayar, bir Merkezi İşlem Birimi (CPU) ile klavye, fare, ekran ve ağ arabirimi gibi Giriş/Çıkış (I/O) cihazlarının etkileşiminden oluşur. Bir program çalıştırdığınızda, önce sabit disk gibi bir depolama cihazından bilgisayarın Rastgele Erişimli Belleği' ne (RAM) okunur. RAM, güç kesildiğinde içeriğini kaybeder, bu nedenle programların kopyaları yalnızca çalışırken RAM' de saklanır. Programın kalıcı kopyası sabit diskte veya başka bir kalıcı depolama cihazında saklanır.

Bir bilgisayarın RAM' i, program çalışırken hem programın kendisini hem de programın kullandığı verileri tutar. Program çalışırken CPU, giriş cihazlarından veri okur ve bu verileri RAM' e kaydeder. Ayrıca CPU, genellikle register olarak adlandırılan çok sınırlı miktarda bir belleğe sahiptir. CPU tarafından iki sayıyı toplamak gibi bir işlem gerçekleştirildiğinde, işlem elemanları (operandlar) CPU'daki registerlarda bulunmalıdır. CPU tarafından gerçekleştirilen tipik işlemler şunlardır: toplama, çıkarma, çarpma, bölme, RAM' den değerleri depolama ve geri alma.

**2.2.1 Programın Çalıştırılması**

Bir kullanıcı bilgisayarda bir program çalıştırıldığında şu adımlar gerçekleşir:

1. Program diskte veya başka bir depolama cihazında bulunur. İlk olarak, program bu depolama cihazından RAM' e okunur
2. İşletim sistemi (örneğin Mac OS X, Microsoft Windows veya Linux) programın kullanımı için RAM' de iki ek alan oluşturur: Çalışma Zamanı Yığını (run-time stack) ve Yığın (heap).
3. İşletim sistemi, CPU'ya programın ilk talimatını yürütmeye başlaması talimatını göndererek programı başlatır.
4. Program, klavye, fare, disk ve diğer giriş kaynaklarından veri okur.
5. Programın her bir talimatı, RAM' den küçük veri parçaları alır, bunları işler ve yeni verileri RAM' e geri yazar.
6. Veriler işlendikten sonra, sonuç ekran veya başka bir çıkış cihazında görüntülenir.

CPU'nun çok az belleği olması nedeniyle, normal çalışma modu, bir değer bir CPU işlemi için gerekli olana kadar RAM' de tutmaktır. RAM, CPU'dan çok daha büyük bir depolama alanıdır. Ancak daha büyük olduğu için CPU'dan da yavaştır. Bir değeri RAM' e kaydetmek veya RAM' den bir değer almak, birkaç CPU işlemi kadar zaman alabilir. Gerektiğinde, değerler RAM' den CPU'nun kayıtlarına kopyalanır ve işlem tamamlandıktan sonra geri kopyalanır. Bu kopyalama işlemi, programın performansını etkileyebilir.

metin, ekran görüntüsü, diyagram, Post-it notu içeren bir resim

Açıklama otomatik olarak oluşturuldu

RAM CPU'ya aktarılır, işlem gerçekleştirilir ve sonuç genellikle tekrar RAM' e yazılır. Bir bilgisayarın RAM' i bir program çalıştığında sıkça erişilir, bu yüzden erişildiğinde neler olduğunu anlamak önemlidir (Şekil 2.1).

Sıkça kullanılan bir benzetme, postane kutuları gibidir. Bir bilgisayarın RAM' i, bir dizi posta kutusu koleksiyonuna benzer. Her kutu bir adres içerir ve bir değeri tutabilir. RAM' e koyabileceğiniz değerler "byte" olarak adlandırılır (yani sekiz bit gruplandırılmıştır). Sekiz bit ile 256 farklı değer depolanabilir. Genellikle byte'lar sayılar olarak yorumlanır, bu yüzden bir byte 0 ile 255 arasındaki değerleri tutabilir. Daha büyük değerler depolamak istiyorsak, byte'ları kelimeler (Word) oluşturmak için bir araya getirebiliriz. Bir bilgisayarın kelime boyutu, bilgisayarın donanım mimarisine bağlı olarak 32 bit (dört byte) veya 64 bittir. Tüm modern bilgisayar donanımları, tek seferde bir kelimeyi alıp veya yazabilir.

Posta kutusu benzetmesi, bir bilgisayarın RAM' inin nasıl organize edildiğini görselleştirmemize yardımcı olur, ancak RAM' in nasıl davrandığını göstermede iyi bir benzetme değildir. Bir postane kutusundan bir şey almak veya koymak için önce kutuyu bulmanız gerekir. Ardından, mektubu kutuya koyabilir veya çıkarabilirsiniz. Postane kutularındaki sayı arttıkça, arama işlemi o kadar uzun sürer. Bu, bu metinde incelediğimiz temel sorunu anlamamıza yardımcı olur: Problem alanı büyüdükçe, bir program veya algoritma nasıl davranır? Bu benzetme bağlamında, posta kutusu sayısı arttıkça bir değeri saklamak veya almanın ne kadar daha uzun sürdüğünü düşünebiliriz.

Bir bilgisayarın RAM' i bir postane gibi davranmaz. Bilgisayar, bir değeri almadan veya saklamadan önce doğru RAM konumunu bulmak zorunda değildir. Daha iyi bir benzetme, her biri bilgisayarın RAM' inde bir bellek konumunu temsil eden bir grup insan gibidir. Her kişiye bir adres veya isim atanmıştır. Bir değeri bir konumda saklamak için, kişinin adını seslenir ve sonra onlara ne hatırlamaları gerektiğini söylersiniz. Doğru kişiyi bulmak için hiçbir zaman harcanmaz çünkü tüm insanlar dinliyorlar, belki de adları çağrılır diye. Bir değeri almak için, kişinin adını çağırırsınız ve size hatırlamaları gereken değeri söylerler. Bu şekilde, herhangi bir bellek konumundan herhangi bir değeri almak tam olarak aynı miktarda zaman alır. İşte bir bilgisayarın RAM' i böyle çalışır. Bir değeri RAM'ın herhangi bir konumunda saklamak tam olarak aynı miktarda zaman alır. Benzer şekilde, bir değeri almak, ilk RAM konumunda olsun veya en sonuncusunda olsun, aynı miktarda zaman alır.

**2.3 Python Listesinde Ögelere Erişme**

Python Listelerinde Öğelere Erişim Deneyimlerimizle bir bilgisayarın RAM'ındaki tüm konumlara aynı miktarda zamanda erişilebileceğini doğrulayabiliriz. Bir Python listesi, ardışık bellek konumlarının bir koleksiyonudur. Ardışık kelimesi, bir listenin bellek konumlarının RAM' de art arda bir araya gruplandığı anlamına gelir. Bir bilgisayarın RAM'ının tüm adlarını ve değerlerini hatırlayan bir grup insan gibi davrandığını doğrulamak için, farklı boyutlardaki Python listeleriyle birkaç test çalıştırarak rastgele bir listenin elemanından bir değeri almak veya bir değer saklamak için ortalama zamanı bulabiliriz.

Python listelerinin davranışını test etmek için, rastgele bir liste içinde değerleri saklayıp alacak bir program yazabiliriz. Bu programda iki farklı teoriyi test edebiliriz.

1. Bir listenin boyutu, listenin ortalama erişim zamanını etkilemez.
2. Listenin herhangi bir konumundaki ortalama erişim zamanı, listenin içindeki konumundan bağımsız olarak aynıdır.

Bu iki teoriyi test etmek için bir listenin içinde değerlerin alınması ve saklanması sürelerini ölçmemiz gerekecek. Neyse ki, Python'un mevcut zamanı kaydetmek için kullanılabilecek bir datetime modülü bulunmaktadır. İki datetime nesnesini çıkararak, bir program içindeki herhangi bir işlem için mikro saniye cinsinden (yani saniyenin milyonda biri) süreyi hesaplayabiliriz. Bölüm 2.3.1'deki program, liste erişimini test etmek ve Python listesinde değerleri almak ve saklamak için erişim süresini kaydetmek amacıyla yazılmıştır.

* + 1. **Liste Erişim Zamanlaması**

**import datetime**

**import random**

**import time**

**def** main():

# Sonuçları içeren bir XML dosyası yazın

dosya = open("ListAccessTiming.xml","w")

dosya.write('<?xml version="1.0" encoding="UTF-8" standalone="no" ?>\n')

dosya.write('<Plot title="Ortalama Liste Eleman Erişim Zamanı">\n')

# Boyutları 1000 ila 200000 olan listeleri test edin.

xmin = 1000

xmax = 200000

# xList'te liste boyutlarını ve yList'te o boyuttaki ortalama erişim süresini kaydedin,

#1000 alıntı için.

xList = []

yList = []

**for** x **in** range(xmin, xmax+1, 1000):

xList.append(x)

prod = 0

# Boyutu x olan tüm 0'lardan oluşan bir liste oluşturun

lst = [0] \* x

# Çöp toplama/bellek tahsisi tamamlansın

# veya en azından durulsun

time.sleep(1)

# 1000 test alıntısından önceki zaman

starttime = datetime.datetime.now()

**for** v **in** range(1000):

# Liste içinde rastgele bir konum bulun ve bir değeri alın.

# Gerçekten alındığından emin olmak için o değerle sahte bir işlem yapın.

index = random.randint(0,x-1)

val = lst[index]

prod = prod \* val

# 1000 test alıntısından sonraki zaman

endtime = datetime.datetime.now()

# Başlangıç ve bitiş arasındaki fark.

deltaT = endtime - starttime

# Ortalama erişim zamanı için 1000'e bölün, ancak mikrosaniye cinsinden

çarpmak için 1000000 ile çarpın.

accessTime = deltaT.total\_seconds() \* 1000

yList.append(accessTime)

dosya.write(' <Axes>**\n'**)

dosya.write(' <XAxis min="'+str(xmin)+'" max="'+str(xmax)+'">Liste Boyutu</XAxis>**\n'**)

dosya.write(' <YAxis min="'+str(min(yList))+'" max="'+str(60)+'">Mikrosaniye</YAxis>**\n'**)

dosya.write(' </Axes>**\n'**)

dosya.write(' <Sequence title="Liste Boyutuna Göre Ortalama Erişim Zamanı" color="red">**\n'**)

**for** i **in** range(len(xList)):

dosya.write(' <DataPoint x="'+str(xList[i])+'" y="'+str(yList[i])+'"/>**\n'**)

dosya.write(' </Sequence>**\n'**)

# Bu bölüm, bir listenin içindeki 100 rastgele konuma erişimi test eder

# 200.000 öğe ile, tüm konumların yaklaşık olarak aynı miktarda zamanda erişilebileceğini görmek için.

xList = lst

yList = [0] \* 200000

time.sleep(2)

**for** i **in** range(100):

starttime = datetime.datetime.now()

index = random.randint(0,200000-1)

xList[index] = xList[index] + 1

endtime = datetime.datetime.now()

deltaT = endtime - starttime

yList[index] = yList[index] + deltaT.total\_seconds() \* 1000000

dosya.write(' <Sequence title="Erişim Zamanı Dağılımı" color="blue">\n')

**for** i **in** range(len(xList)):

**if** xList[i] > 0:

dosya.write(' <DataPoint x="'+str(i)+'" y="'+str(yList[i]/xList[i])+'"/>\n')

dosya.write('</Sequence>**\n'**)

dosya.write('</Plot>**\n'**)

dosya.close()

**if** \_\_name\_\_ == "\_\_mai n\_\_":

main()

Bu tür bir programı çalıştırırken elde ettiğiniz süreler, gerçekleştirilen işlemlerin yanı sıra, testin yapıldığı bilgisayarda hangi diğer etkinliklerin gerçekleştiğine de bağlı olacaktır. Mac OS X, Linux veya Microsoft Windows gibi tüm modern işletim sistemleri çoklu görevli olduğu için, örneğin bir bilgisayar programı yazarken e-posta alabiliriz. Bir şeyi zamanladığımızda, sadece kendi programımızın çalışmasının etkilerini görmeyeceğiz, aynı zamanda bilgisayarda şu anda çalışan tüm programları da göreceğiz. Çoklu görevli bir sistemde bir programı tamamen izole etmek neredeyse imkansızdır. Ancak, çoğu zaman kısa bir program çok fazla kesinti olmadan çalışacaktır.

Bölüm 2.3.1'deki program sonuçlarını bir XML dosyasına yazar. XML dosya biçimi, bir veya daha fazla veri dizisinin iki boyutlu bir çizim için deneysel olarak toplanan verilerin açıklamasını destekler. Bu programın oluşturduğu verilerden bir örneğin formatı Bölüm 2.3.2'de gösterildiği gibidir. Veri kısaltılmış olsa da, format Bölüm 2.3.2'de gösterildiği gibidir.

**2.3.2 Bir XML Örneği**

**<?xml version="1.0" encoding="UTF-8" standalone="no" ?>**

**<Plot title="Average List Element Access Time">**

**<Axes>**

**<XAxis min="1000" max="200000">List Size</XAxis>**

**<YAxis min="20.244" max="60">Microseconds</YAxis>**

**</Axes>**

**<Sequence title="Average Access Time vs List Size" color="red">**

**<DataPoint x="1000" y="33.069"/>**

**<DataPoint x="2000" y="27.842"/>**

**<DataPoint x="3000" y="23.908"/>**

**<DataPoint x="4000" y="26.349"/>**

**<DataPoint x="5000" y="23.212"/>**

**<DataPoint x="6000" y="23.765"/>**

**<DataPoint x="7000" y="21.251"/>**

**<DataPoint x="8000" y="21.321"/>**

**<DataPoint x="9000" y="23.197"/>**

**<DataPoint x="10000" y="21.527"/>**

**<DataPoint x="11000" y="35.799"/>**

**<DataPoint x="12000" y="22.173"/>**

**...**

**<DataPoint x="197000" y="26.245"/>**

**<DataPoint x="198000" y="30.013"/>**

**<DataPoint x="199000" y="25.888"/>**

**<DataPoint x="200000" y="23.578"/>**

**</Sequence>**

**<Sequence title="Access Time Distribution" color="blue">**

**<DataPoint x="219" y="41.0"/>**

**<DataPoint x="2839" y="38.0"/>**

**<DataPoint x="5902" y="38.0"/>**

**<DataPoint x="8531" y="58.0"/>**

**<DataPoint x="11491" y="38.0"/>**

**<DataPoint x="15415" y="38.0"/>**

**<DataPoint x="17645" y="31.0"/>**

**<DataPoint x="18658" y="38.0"/>**

**<DataPoint x="20266" y="40.0"/>**

**<DataPoint x="21854" y="31.0"/>**

**...**

**<DataPoint x="197159" y="37.0"/>**

**<DataPoint x="199601" y="40.0"/>**

**</Sequence>**

**</Plot>**

Bu yazımızda oldukça fazla deneysel veriye göz atacağımız için, Bölüm 1'de verilen formatta bir XML dosyasını okuyacak bir Tkinter programı yazdık. (Tkinter, Python kurulumu ile birlikte gelen ve pencereli-menülü modern programlar yazmamızı sağlayan grafik arayüz geliştirme takımlarından biridir.)

Liste erişim deneyi tarafından toplanan verileri çizmek için programı kullanarak şekildeki gibi bir grafik görürsünüz. Bu grafik, Python'daki listeler hakkında daha önce yaptığımız iki ifadeyi destekleyen deneysel verileri sağlar. Kırmızı çizgi, verilen boyuttaki bir listedeki 1.000 öğe’ye erişiminin ortalama öğe erişim süresini gösterir. Bu ortalama erişim süresi (1.000 rastgele liste erişiminden oluşan bir örneklemden hesaplanır) 10.000 kişilik bir listede 160.000 kişilik bir listede olduğundan daha uzun. Önbellek, bazı durumlarda belleğe erişimi hızlandırmanın bir yoludur ve muhtemelen gerçekten bilgisayarın düşük erişim sürelerinin RAM için bir önbelleğin varlığından faydalandığı söz konusudur. Deneysel veriler, bir listenin boyutunun listedeki ortalama metin, ekran görüntüsü, çizgi, öykü gelişim çizgisi; kumpas; grafiğini çıkarma içeren bir resim

Açıklama otomatik olarak oluşturulduerişim süresini etkilemez.

Grafikteki mavi çizgi 200.000 elemanlı bir liste üzerinde 100 liste alma ve saklama işleminin sonucudur . Mavi çizginin kırmızı çizgiden daha yüksek olmasının nedeni büyük olasılıkla hem öğeden alma hem de öğeye depolama işlemi yapmanın sonucudur. Buna ek olarak, bellekteki değerler birbirinden ne kadar uzak olursa, önbellek erişim süresini azaltmaya yardımcı olacaktır. Mavi çizginin daha yüksek olmasının nedeni ne olursa olsun dikkat edilmesi gereken önemli nokta, 0 indeksindeki elemana erişimin daha fazla zaman almamasıdır. Kesin değerler olmamakla birlikte grafikte basılıdır, kesin değerler önemli değildir. Görmek isteyeceğimiz şey, ortalama erişim sürelerinin daha uzun veya daha kısa olmasına yönelik herhangi bir eğilimdir. Açıkça görülüyor ki eğilimi, listenin boyutunun ortalama erişim süresini etkilemediği yönündedir. Deneysel verilerde bazı iniş çıkışlar mevcut ancak bu durum sistemin çok görevli bir sistem olmasından kaynaklanıyor. Başka bir faktör muhtemelen bellek konumlarının önbelleğe alınmasıdır. Önbellek, bazı durumlarda belleğe erişimi hızlandırmanın bir yoludur ve gerçekten düşük erişim sürelerinin, bilgisayarın RAM‘ i için bir önbelleğin varlığından faydalanması muhtemeldir. Liste içindeki tüm konumlar eşit muamele görmektedir. Bu durum, herhangi bir noktadaki ortalama erişim süresinin liste içindeki konumu, liste içindeki konumundan bağımsız olarak aynıdır.

**2.4 Big-oh Notasyonu**

Deneysel verilerde hangi satıra bakarsak bakalım, bilgisayarın yaptığı diğer işlemlerde dahi, bellek erişimlerinin hiçbirinde erişim süresi asla 100μs'yi Belleğe erişimin 100 μs'den daha az sürdüğü sonucuna varabiliriz. Aslında 100μs, bir değere erişmek veya bir bellek konumunda saklamak için gerekenden çok daha fazla bir süredir. Deneysel verilerimiz daha önce öne sürdüğümüz iki iddiayı desteklemektedir. Ancak teknik olarak hafızaya ulaşmanın sabit bir zaman aldığı iddiamızı kanıtlamaz. Bir bilgisayardaki RAM' in mimarisi, herhangi bir bellek konumuna erişmenin sabit bir zaman aldığını kanıtlamak için incelenebilir. Belleğe erişmek, bir grup insanda bir ismi seslenmek ve o kişinin kendisine verilen değerle yanıt vermesini sağlamak gibidir. Hangi kişinin adının anıldığı önemli değil. Tepki süresi aynı veya hemen hemen aynı olacaktır. Bir bilgisayarın RAM' ine erişim süresi, eğer önbellek mevcutsa biraz farklılık gösterebilir ancak en azından bir bellek konumuna erişimin ne kadar zaman alacağına dair bir üst sınır olduğunu söyleyebiliriz.

Bu üst sınır fikri daha resmi olarak ifade edilebilir. Bir üst sınırın resmi ifadesine **Big-Oh notasyonu** denir. Big-Oh, genellikle üst sınırlardan bahsederken kullanılan Yunanca Omicron harfini ifade eder. Bilgisayar programcıları olarak bir numaralı endişemiz, büyük miktarda veriye sahip olduğumuzda programlarımızın nasıl performans göstereceğidir. Bir bilgisayarın hafızası açısından, çok geniş bir öğe listemiz varsa programımızın nasıl performans göstereceğini bilmek istedik. Bir listenin tüm öğelerine, listenin büyüklüğünden bağımsız olarak aynı sürede erişildiğini gördük. Listenin boyutunu n adında bir değişkenle temsil edelim. Boyutu n olan bir listedeki bir öğeye erişim için ortalama erişim süresi f(n) ile verilsin. Artık şunu söyleyebiliriz:

*O(g(n))* = { *f* | ∃*d >* 0*, n*0 ∈ *Z*+ 0 ≤ *f (n)* ≤ *d g(n),* ∀*n* ≥ *n*0}

Matematikte bu şu şekilde okunur: O( g(n)) ile gösterilen işlevler sınıfı, 0'dan büyük bir d'nin ve 0'ın ona eşit veya ondan küçük olduğu bir n0'ın (pozitif bir tamsayı) bulunduğu tüm f işlevlerinden oluşur. f(n), n0'dan büyük veya ona eşit tüm n'ler için d çarpı g(n)'den küçük veya eşittir.

Eğer f, O(g(n))‘nin bir elemanı ise, f(n)‘nin O(g(n)) olduğunu söyleriz. Bu durumda g fonksiyonuna f için asimptotik üst sınır adı verilir. Yukarıdaki matematiksel açıklamadan memnun olmayabilirsiniz. İngilizce olarak ifade edilirse O(g(n)) adlı küme, n sonsuza yaklaşırken üst sınırı d\* g(n) olan tüm f(n) fonksiyonlarının kümesinden oluşur. Asimptotik kelimesinin anlamı budur. Asimptotik sınır fikri, n'nin bazı küçük değerleri için f(n) değerinin d\* g(n) değerinden daha büyük olabileceği anlamına gelir; ancak n yeterince büyüdüğünde (yani n0'dan büyük olduğunda), o zaman n'den büyük olanların tümü f(n)'nin d\* g(n)'den küçük olduğu her zaman doğru olacaktır. Bu asimptotik üst sınır fikri Şekil 2.3'te resmedilmiştir. Bazı küçük değerler için fonksiyonun yeşil renkle gösterilen performansı mavi üst sınır çizgisinden daha kötü olabilir, ancak sonuçta üst sınır n'nin tüm büyük değerleri için daha çizgi, öykü gelişim çizgisi; kumpas; grafiğini çıkarma, diyagram, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldubüyüktür.

Bir elemana erişim süresi n'ye bağlı olmadığından, g(n) = 1 olarak seçebiliriz. Dolayısıyla, n boyutlu bir listedeki bir elemana erişmek için gereken ortalama sürenin O(1) olduğunu söyleyebiliriz. Eğer Python'da bir listenin bir elemanına erişmenin asla 100μs'den uzun sürmediğini varsayarsak, o zaman d için iyi bir seçim 100 olacaktır. Yukarıdaki tanıma göre o zaman n yeterince büyüdüğünde f(n) değerinin 100'den küçük veya 100'e eşit olması gerekir.

Bir listenin bir öğesine erişmenin karmaşıklığının hesaplanmasında g(n) = 1 seçimi keyfidir. g(n) = 2 seçebilirdik. Eğer g(n) = 2 seçilirse d, 100 yerine 50 olarak seçilebilir. Ancak, biz sadece g fonksiyonundaki genel büyüme ile ilgilendiğimiz için, 1 veya 2 seçimi önemsizdir ve en basit fonksiyon seçilir, bu durumda O(1). İngilizcede bir işlem veya program O(1) olduğunda buna sabit zamanlı işlem veya program deriz. Bu, işlemin n'nin boyutuna bağlı olmadığı anlamına gelir.

Bir bilgisayarın gerçekleştirebileceği çoğu işlemin O(1) olduğu ortaya çıktı. Örneğin iki sayının toplanması bir O(1) işlemidir. İki sayının çarpımı da öyle. Her iki işlem de bilgisayarda birkaç döngü gerektirse de, toplam döngü sayısı, eklenen veya çarpılan tam sayıların veya kayan nokta sayılarının boyutuna bağlı değildir. Bir döngü, bilgisayardaki basit bir zaman birimidir. İki değeri karşılaştırmak da sabit zamanlı bir işlemdir. Karmaşıklığı hesaplarken, herhangi bir aritmetik hesaplama veya karşılaştırma, sabit zamanlı bir işlem olarak düşünülebilir.

Bu hesaplama karmaşıklığı fikri, bir kod parçasının karmaşıklığı n'ye bağlı olduğunda özellikle önemlidir. Bir sonraki bölümde, üzerinde çalıştığı listenin boyutuna bağlı olan bazı kodları ve küçük bir kod parçasını bile nasıl yazdığımızın sonuçlarını anlamanın ne kadar önemli olduğunu göreceğiz.

**2.5 Pylist Append (ekleme) İşlemi**

Bir bellek konumuna erişmenin veya bir değeri bellek konumunda saklamanın bir O(1) veya sabit zamanlı işlem olduğunu tespit ettik. Aynı şey bir listenin bir öğesine erişmek veya listede bir değer saklamak için de geçerlidir. Listenin boyutu, bir öğeye erişmek veya bir öğeyi saklamak için gereken süreyi değiştirmez ve bir değere erişmek veya bir değeri bellekte veya listede saklamak için gereken süre miktarı için sabit bir üst sınır vardır.

Bu bilgiyle çizim programına tekrar bakalım ve özellikle PyList'e grafik komutlarını ekleyen kod parçasına bakalım. Bu kod programda çokça kullanılmaktadır. Her yeni grafik komutu oluşturulduğunda diziye eklenir. Kullanıcı serbest çizim yaparken her dakika yüzlerce çok sayıda grafik komutu eklenmektedir. Serbest el çizimi biraz yoğun işlem gerektirdiğinden, bu kodun mümkün olduğunca verimli olmasını istiyoruz.

metin, makbuz, cebir, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturuldu**2.5.1 Verimsiz Ekleme**

Bölüm 2.5.1'deki kod listeye aşağıdaki gibi yeni bir öğe ekler:

1. Öğe etrafına [ve] getirilerek liste haline getirilir. Bunu nasıl söylediğimize dikkat etmeliyiz. Öğenin kendisi değiştirilmez. Öğeden yeni bir liste oluşturulur.

2. İki liste + operatörü kullanılarak birleştirilir. + operatörü, her iki orijinal listeyi de değiştirmeyen bir erişimci yöntemidir. Birleştirme, iki listedeki öğelerden yeni bir liste oluşturur.

3. self.items öğesinin bu yeni listeye atanması PyList nesnesini günceller, böylece artık yeni listeye başvurur.

Sormak istediğimiz soru şu: PyList ‘in boyutu büyüdükçe bu ekleme yöntemi nasıl performans gösteriyor? Append yönteminin ilk kez çağrıldığı zamanı düşünelim. Self.items tarafından başvurulan listede kaç öğe var? Sıfır, değil mi? Ve [öğe]'de her zaman bir öğe vardır. Bu nedenle, ekleme yönteminin, içinde bir öğe bulunan yeni listeyi oluşturmak için listenin bir öğesine erişmesi gerekir.

Append yöntemi ikinci kez çağrıldığında ne olur? Bu sefer, self.items tarafından başvurulan listede bir öğe var ve yine [item] içinde bir öğe var. Artık yeni listeyi oluşturmak için iki öğeye erişilmesi gerekiyor. Bir dahaki sefere ekleme çağrıldığında, yeni listeyi oluşturmak için üç öğeye erişilmesi gerekir. Elbette bu kalıp PyList'e eklenen her yeni öğe için devam eder. N' inci eleman diziye eklendiğinde, yeni listeyi oluşturmak için n tane elemanın kopyalanması gerekecektir. Genel olarak, n öğeyi eklemek için kaç öğeye erişilmesi gerekir?

**2.6 Tümevarım Yoluyla Bir İspat**

**yazı tipi, beyaz, metin, diyagram içeren bir resim

Açıklama otomatik olarak oluşturuldu**Bir listenin her bir öğesine erişmenin sabit bir zaman aldığını zaten tespit etmiştik. Yani, n'yi eklemek için gereken süreyi hesaplamak istersek öğeleri PyList‘ e eklersek, tüm liste erişimlerini toplamamız ve bir liste öğesine erişim için geçen süre artı bir liste öğesini depolamak için gereken süre ile çarpmamız gerekir. Erişim ve depolama işlemlerinin toplam sayısını saymak için, bir öğe ilk kez eklendiğinde listenin kopyalanması için erişim ve depolama işlemlerinin sayısıyla başlamalıyız. Bu kopyalanan bir öğedir. İkinci ekleme iki kopyalama işlemi gerektirir. Üçüncü ekleme üç kopyalama işlemi gerektirir. Yani, aşağıdaki sayıda liste öğesinin kopyalanması var.

Matematikte bu toplamı bir toplama sembolüyle (örn.  ) ifade edebiliriz. Bu, ilk n tamsayının toplamını ifade etmenin matematiksel yoludur. Peki ama bu neye eşit? Biraz çalışmayla ortaya çıkıyor, aşağıdakilerin doğru olduğunu bulabiliriz.

Bunun doğru olduğunu Matematiğin matematiksel tümevarım adı verilen bir ispat tekniğini kullanarak kanıtlayabiliriz. Matematiksel tümevarımın birkaç çeşidi vardır. Bunu kanıtlamak için zayıf tümevarım denilen şeyi kullanacağız. Tümevarım kullanarak bir şeyi ispatlarken aslında bir meta-kanıt inşa ediyorsunuz. Meta-kanıt, istediğiniz sonucu bulmak için tekrar tekrar tekrarlayabileceğiniz bir dizi adımdır. Tümevarımın gücü, meta-kanıtı oluşturduğumuzda, sonucun n'nin tüm olası değerleri için doğru olduğunu kanıtlamış olmamızdır.

 Yukarıda verilen formülün tüm n'ler için geçerli olduğunu kanıtlamak istiyoruz. Bunu yapmak için öncelikle bunun basit bir n değeri için doğru olduğunu gösteriyoruz. Bizim durumumuzda n değeri olarak 1'i seçeceğiz. Bu durumda elimizde aşağıdakiler var.

Bu kesinlikle doğrudur. Bu adıma tümevarımsal kanıtın temel durumu denir. Tümevarım yoluyla her kanıtın bir temel durumu olmalıdır ve bu genellikle önemsizdir.

 Bir sonraki adım meta-kanıtı oluşturmaktır. Bu meta-kanıta tümevarımsal durum denir. Tümevarımsal durumu oluştururken formülün, n'den küçük olan tüm m değerleri için geçerli olduğunu varsayarız. Buna güçlü indüksiyon denir. Bu kesinlikle doğrudur. Bu adıma tümevarımsal kanıtın temel durumu denir. Tümevarım yoluyla her kanıtın bir temel durumu olmalıdır ve bu genellikle önemsizdir. Bir sonraki adım meta-kanıtı oluşturmaktır. Bu meta-kanıta tümevarımsal durum denir. Tümevarımsal durumu oluştururken formülün, n'den küçük olan tüm m değerleri için geçerli olduğunu varsayarız. Buna güçlü indüksiyon denir. Zayıf tümevarımda formülün n−1 için geçerli olduğunu varsayıyoruz ve bunun n için geçerli olduğunu göstermek istiyoruz. Bu problemde ispatımızı bitirmek için zayıf tümevarım kullanacağız. Yine bu adım, temel durumumuzdan bulmamız gereken n değerine ulaşmak için tekrar tekrar uygulayabileceğimiz bir dizi adım oluşturmamıza yardımcı olur. Başlamak için aşağıdakileri not edeceğiz.

Toplamanın tanımı gereği bu doğrudur. Ama şimdi n−1'e giden bir toplamımız var ve zayıf tümevarım, denklemin n−1 için geçerli olduğunu bildiğimizi söylüyor. Buna tümevarımsal hipotez denir. n - 1 için geçerli olduğundan aşağıdakinin doğru olduğunu biliyoruz. Bunu, orijinal formülde n gördüğümüz her yere n -1 koyarak elde ederiz.

Artık bu gerçeği orijinal formülümüzün eşitliğini kanıtlamak için kullanabiliriz. İşte başlıyoruz!



Bu formülün sol tarafına ve sağ tarafına bakarsanız, eşit olduğunu kanıtlamaya çalıştığımız iki şeyin aslında eşit olduğunu görebilirsiniz. Bu, tümevarım yoluyla kanıtımızı tamamlıyor.

Meta-kanıt yukarıdaki formüldedir. Bu, eşitliğin n = 2 için geçerli olduğunu kanıtlamak için kullanabileceğimiz bir şablondur. Eşitliğin n = 2 için geçerli olduğunu kanıtlamak için eşitliğin n = 1 için geçerli olduğu gerçeğini kullanmamız gerekiyordu.

Bu bizim temel durumumuzdu. Bunun n = 2 için geçerli olduğunu kanıtladıktan sonra eşitliğin n = 3 için de geçerli olduğunu kanıtlamak için aynı formülü kullanabiliriz. Matematiksel tümevarım tüm adımları uygulamamızı gerektirmez. Bu meta-kanıtı yarattığımız sürece eşitliğin tüm n'ler için geçerli olduğunu kanıtlamış oluyoruz. Bu, tümevarımın gücüdür.

**metin, ekran görüntüsü, öykü gelişim çizgisi; kumpas; grafiğini çıkarma, çizgi içeren bir resim

Açıklama otomatik olarak oluşturuldu**ilerleyen kısımlarında göreceğimiz gibi, bir listeye öğe eklemek O(1) işlemidir. Bu, eklemek anlamına gelir n öğeyi bir listeye ekleyerek O()'den O(n)'ye geçtik. Bu bölümün ilerleyen kısımlarında Python'un ekleme işlemi için O(1) karmaşıklığı elde etmeyi öğreneceğiz. Şekil 2.4'teki mavi çizgi, PyList append yönteminin aşağıdaki durumlarda nasıl çalıştığını göstermektedir + operatörünün yerine liste ekleme yöntemi çağrılır. PyList'teki 100.000 öğede, başka bir öğe eklemek 27 saniyede yapılır. Bu, programımızda güzel bir hızlanma demek. Bu değişikliği yaptıktan sonra PyList ekleme yöntemi bölüm 2.7.1'de verilmiştir.

metin, ekran görüntüsü, yazı tipi, makbuz içeren bir resim

Açıklama otomatik olarak oluşturuldu**2.7.1 Verimli Ekleme**

**2.8 Yaygın Olarak Ortaya Çıkan Hesaplama Karmaşıklıkları**

Bu metinde inceleyeceğimiz algoritmalar O(1) karmaşıklığında olacaktır, O(log n),O(n log n),O() veya O().Bu fonksiyonların şekillerinin bir grafiği şu şekilde görünür.(Şekil 2.5'te gösterilmiştir).Çoğu algoritma,algoritmalara karşılık gelen bu karmaşıklıklardan birine sahip n faktörü içerir.Bir hesaplamayı tamamlamak için gereken süreyi ölçen bir formüldeki terimlere eklenen veya çarpılan sabit değerler bu işlemin genel karmaşıklığı etkilemez.Hesaplama karmaşıklığı sadece en yüksek güçten etkilenir. Şekil 2.5'te grafiği çizilen karmaşıklıklar n veya O()'nin gerçekten korkunç üstel karmaşıklığı dışında n'nin logaritma ve buradaki c gibi bazı sabit değerler içerir.

Metni okurken farklı karmaşıklıklara sahip algoritmalarla karşılaşacaksınız, Şekil 2.5'te gösterilen karmaşıklıklardan biri olacaktır. Her zaman olduğu gibi, değişken n algoritmaya girdi olarak sağlanan verilerin boyutunu temsil eder. Alınan zaman, bu veriyi işlemek için gereken grafikteki dikey eksendir. Biz dikey ekseni önemsemezken bu fonksiyonların genel şeklini önemsiyoruz. Çizgi ne kadar düz, eğim ne kadar düşük olursa algoritma o kadar iyi performans gösterir. Açıkça bir üstel karmaşıklığa (yani O()) veya karmaşıklığa (yani O()) karmaşıklığı n'nin çok küçük değerleri dışında çok iyi performans göstermeyecektir.Algoritmanızın n'nin büyük değerleri için asla çağrılmayacağını bilmiyorsanız bu verimsiz bir algoritma kabul edilebilir, bunu bildiğinizden gerçekten emin olmanız gerekir.Veri boyutunuz her zaman küçük olacaktır.Tipik olarak mümkün olduğunca verimli olan algoritmalar tasarlamak isteriz.

İlerleyen bölümlerde O() olan sıralama algoritmaları ile karşılaşacaksınız.Daha iyisini yapabileceğimizi ve O(n log n) karmaşıklığına ulaşabileceğimizi öğreneceksiniz.O(n) olan arama algoritmalarını görecek ve ardından O(log n) karmaşıklığına nasıl ulaşacağınızı öğreneceksiniz.Ayrıca hashing adı verilen ve O(1) zamanda arama yapabilen bir teknik de öğreneceksiniz. Öğreneceğiniz bu teknikler, büyük miktarda veriyle mümkün olduğunca verimli bir şekilde başa çıkmanıza yardımcı olacaktır. Bu tekniklerin her biri keşfedildikçe, bazı eğlenceli programlar yazma fırsatınız olacak ve nesne yönelimli yazılımlar hakkında çok şey öğreneceksiniz.

**metin, çizgi, ekran görüntüsü, öykü gelişim çizgisi; kumpas; grafiğini çıkarma içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**Şekil 2.5** Yaygın Big-Oh Karmaşıklıkları

mümkün olduğunca. Bu tekniklerin her biri keşfedilirken, bazı eğlenceli programlar yazma fırsatınız olacak ve nesne yönelimli programlama hakkında çok şey öğreneceksiniz.

**2.9 Daha Fazla Asimptotik Gösterim**

Bu bölümün başlarında, bir algoritmanın karmaşıklığın üst sınırını tanımlamak için Big-Oh notasyonunu geliştirdik. Burada, verimlilik fikrinin sezgisel bir anlayışıyla başladık ve bir fonksiyonun, n'nin algoritmaya verilen verinin boyutunu temsil ettiği n'nin bir fonksiyonu ile yukarıda sınırlandırılmışsa bir karmaşıklık sergilediğini söyledik. Bu bölümde, bir algoritmanın verimliliğini hem yukarıdan hem de aşağıdan sınırlamak için bu kavramları daha da geliştiriyoruz.

Bilgisayar Bilimlerinde verimliliğin ve ölçümünün derinlemesine bir tartışması ile başlıyoruz. Algoritma verimliliği ile ilgilenirken göz önünde bulundurulması gereken iki konu vardır.

2.9 Daha Fazla Asimptotik Notasyon 57

• Bir algoritmanın çalışması için gereken süre

• ve bununla bağlantılı olarak, bir algoritmanın çalışırken kullandığı alan miktarı.

Genellikle bilgisayar bilimcileri algoritmalarda bir alan/zaman ödünleşiminden bahsederler. Bazen daha fazla bellek kullanarak daha hızlı bir çalışma süresi elde edebiliriz. Ancak, eğer çok fazla bellek kullanırsak bilgisayarı ve çalışan diğer programları yavaşlatabiliriz. Bahsedilen alan, bir sorunu çözmek için gereken RAM miktarıdır. İlgilendiğimiz zaman ise veri boyutu büyüdükçe işlem sayısının nasıl arttığının bir ölçüsüdür.

Bir algoritmanın çalışma süresini tanımlayan bir T(𝑛) fonksiyonu düşünün, burada n algoritmaya verilen verinin boyutudur. Bilgisayar bilimcileri olarak bu fonksiyonun asimptotik davranışını incelemek istiyoruz. Başka bir deyişle, T(𝑛)'nin 𝑛→ ∞ olarak nasıl arttığını incelemek isteriz. 𝑛 değeri, girdi verilerinin olası boyutlarını temsil eden bir Doğal sayıdır. Doğal sayılar negatif olmayan tam sayılar kümesidir. Bölüm 2.9.1'deki tanım, bu bölümde daha önce sunulan Big-Oh notasyon tanımının yeniden ifadesidir.

**2.9.1 Big-Oh Asimptotik Üst Sınır**

Şöyle yazıyoruz:



ve 𝑓nin n'nin big-oh 𝑔 'si olduğunu söyleriz. Big-Oh tanımı, bir algoritmanın çalışması için gereken süre için bir üst sınır bulabileceğimizi söyler. Şekil 2.3'te verilen veri boyutuna karşı zaman grafiğini düşünün. Veri boyutu veya 𝑛 x ekseni, zaman ise y eksenidir.

Yeşil çizginin bazı algoritmaların gözlemlenen davranışını temsil ettiğini düşünün.

Mavi çizgi, yaklaşık 𝑛 = 4'ten sonra yeşil çizginin açıkça bir üst sınırıdır. BüyükOh tanımının anlamı budur. Bir süre için üst sınırlayıcı fonksiyon bir üst sınır olmayabilir, ancak sonunda bir üst sınır haline gelir ve 𝑛 sonsuza yaklaştıkça limite kadar bu şekilde kalır.

Ancak, mavi çizgi, çalışma süresi yeşil çizgi ile gösterilen algoritmanın karmaşıklığı üzerinde sıkı bir sınırı temsil ediyor mu? Bir algoritmanın karmaşıklığın tanımladığımı tanımladığımızda, bunun gerçek çalışma süresini gerçekten temsil ettiğini bilmek isteriz. Algoritmanın O(𝑛^2)'de çalıştığını söylemek, algoritma 𝑛 ile orantılı bir zamanda çalışsa bile doğrudur çünkü Big-Oh notasyonu yalnızca bir üst sınırı tanımlar. Eğer algoritmanın çalışma süresinin gerçekten 𝑛 ile orantılı olduğunu söylemek istiyorsak, o zaman biraz daha fazla güce ihtiyacımız var. Bu da bizi Bölüm 2.9.2'deki bir sonraki tanımımıza götürür

58 2 Hesaplamalı Karmaşıklık

**2.9.2 Asimptotik Alt Sınır**

Omega notasyonu, bir fonksiyonun alt sınırını tanımlamanın bir yolu olarak hizmet eder. Bu durumda alt sınır tanımı bir süre için daha büyük olabileceğini söyler, ancak sonunda n'nin tüm büyük değerleri için T(𝑛)'nin g(𝑛)'ye baskın olduğu bir n0 vardır. Bu durumda algoritmanın (g(𝑛)) olduğunu yazabiliriz. Grafiğimize bir kez daha baktığımızda, mor çizginin n = 75'ten sonra gözlemlenen davranışa hakim olduğunu görüyoruz. Üst sınırda olduğu gibi, bir süre için alt sınır gözlemlenen davranıştan daha büyük olabilir, ancak bir süre sonra alt sınır n'nin tüm büyük değerleri için gözlemlenen davranışın altında kalır.

Hem alt sınır hem de üst sınır tanımıyla, artık asimptotik olarak sıkı bir sınır tanımlamak için notasyona sahibiz. Buna Theta notasyonu denir.

**2.9.3 Teta Asimptotik Sıkı Sınır**



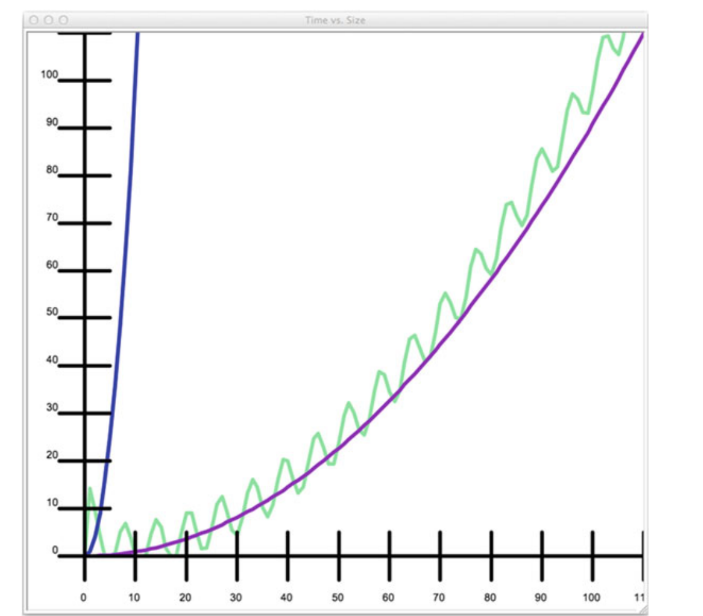
Eğer böyle bir g fonksiyonu bulabilirsek, o zaman (g(n))nin bir algoritmanın gözlemlenen davranışı olan T(n) için asimptotik olarak sıkı bir sınır olduğunu beyan edebiliriz. Şekil 2.6'da üst sınır mavi çizgi g(n) = n^2 ve alt sınır mor çizgi g(n)/110'un bir grafiğidir. Eğer c = 1 ve d = 1/110 olursa, T(n)'nin asimptotik olarak sıkı sınırı (n^2) olur.

Şimdi, n-kare değerinin algoritmanın davranışı üzerinde bir üst sınır olduğunu söylemek yerine, algoritmanın gerçekten n-kare ile orantılı bir zamanda çalıştığını ilan edebiliriz. Davranış, algoritmanın bir n-kare algoritması olduğu iddiasını kanıtlayan n-kare fonksiyonları tarafından yukarıda ve aşağıda sınırlandırılmıştır.

**2.10 Amorti Edilmiş Karmaşıklık**

Bazen bir algoritma için sıkı bir üst sınır bulmak mümkün değildir. Örneğin, çoğu işlem c\*g(n) fonksiyonu ile sınırlandırılabilir, ancak arada bir daha uzun süren bir işlem olabilir. Bu gibi durumlarda Amortize Karmaşıklık adı verilen bir yöntemin kullanılması faydalı olabilir. Amortisman, muhasebeciler tarafından bazı ticari işlemlerin maliyetinin tamamını bir mali yılda defterlere uygulamak yerine birkaç yıla yayarken kullanılan bir terimdir. Aynı fikir Bilgisayar Biliminde de bir işlemin maliyetinin ortalaması alındığında kullanılır. Tüm amortisman yöntemlerinin arkasındaki temel fikir, bir veri yapısı (genellikle boş başlar) üzerindeki herhangi bir n işlem dizisinin en kötü durum çalışma süresi için olabildiğince sıkı bir üst sınır elde etmektir. N'ye bölerek dizideki her bir işlemin ortalama veya amorti edilmiş çalışma süresini elde ederiz.

2.10 Amorti Edilmiş Karmaşıklık 59

 **Şekil 2.6** Bir Alt ve Üst Sınır

Bu bölümün başlarında tartışılan PyList ekleme işlemini düşünün. PyList append metodunun son versiyonu basitçe listeler üzerinde Python append işlemini çağırır. Python listeler için bir ekleme işlemini desteklerken, listeler C'de diziler olarak uygulanır ve C'de bir diziye eklemek mümkün değildir. Bir dizi sabit bir boyutla tahsis edilebilir, ancak oluşturulduktan sonra boyutu artırılamaz

Bir an için Python listelerinin, C dizileri gibi, listeler üzerinde append yöntemini desteklemediğini ve bir liste oluşturmanın tek yolunun n'nin sabit bir değer olduğu *[None]\*n* gibi bir şey yazmak olduğunu varsayın. *[None]\*n* yazmak, her biri *None* değerine referans veren n elemandan oluşan sabit boyutlu bir liste oluşturur. C ve C++ dizileri bu şekilde tahsis edilir. Örneğimizde, Python'un append'i desteklemediğini varsaydığımız için PyList append metodumuzu farklı bir şekilde uygulamalıyız. Append metodunu kullanamayız ve bu bölümün başlarında + operatörü ile bir seferde bir öğe eklemenin kötü bir fikir olduğunu gördük. Biz biraz daha farklı bir şey yapacağız. PyList append işlemimiz, sabit boyutlu listede yer kalmadığında, Bölüm 2.10.1'deki kodda gösterildiği gibi eski listedeki tüm öğeleri yeni listeye kopyalayarak listenin boyutunu iki katına çıkaracaktır.

60 2 Hesaplamalı Karmaşıklık

**2.10.1 Bir PyList Sınıfı**

class PyList:

# The size below is an initial number of locations for the list object. The 3

# numItems instance variable keeps track of how many elements are currently stored 4

# in the list since self.items may have empty locations at the end.

def \_\_init\_\_(self,size=1):

self.items = [None] \* size

self.numItems = 0

def append(self,item):

if self.numItems == len(self.items):

# We must make the list bigger by allocating a new list and copying

# all the elements over to the new list.

newlst = [None] \* self.numItems \* 2

for k in range(len(self.items)):

newlst[k] = self.items[k]

self.items = newlst

self.items[self.numItems] = item 20

self.numItems += 1

def main():

p = PyList()

for k in range(100):

p.append(k)

print(p.items)

print(p.numItems)

print(len(p.items) )

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

main()

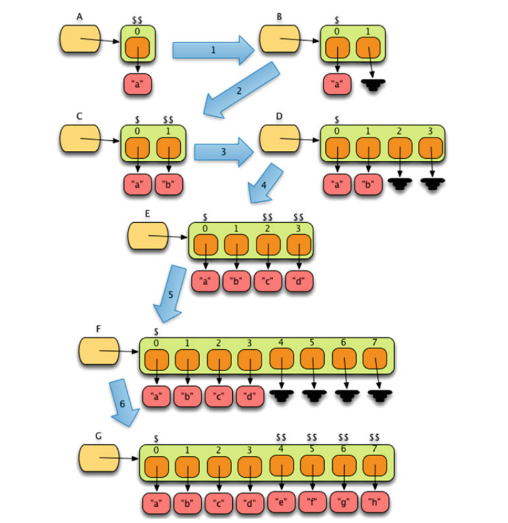
İddia, bu yeni PyList ekleme yöntemini kullanarak, boş bir listeden başlayarak bir PyList nesnesi üzerinde n ekleme işlemi dizisinin O(n) zaman aldığı, yani tek tek işlemlerin O(1) zamandan daha uzun sürmemesi gerektiğidir. Bu nasıl doğru olabilir? Listede yer kalmadığında yeni bir liste ayrılır ve tüm eski öğeler yeni listeye kopyalanır. Açıkçası, n elemanı bir listeden diğerine kopyalamak O(1) zamandan daha uzun sürer. Append'in nasıl O(1) karmaşıklık sergileyebileceğini anlamak, append işleminin amorti edilmiş karmaşıklığını hesaplamaya dayanır. Teknik olarak, liste boyutu iki katına çıktığında append işleminin karmaşıklığı O(n) olur. Ancak bu ne sıklıkla gerçekleşir? Cevap o kadar da sık değil.

**2.10.2 Ekleme Karmaşıklığının Kanıtı**

Append yönteminin O(1) karmaşıklığa sahip olduğunun kanıtı, append'in amorti edilmiş karmaşıklığını bulmak için muhasebe yöntemi olarak adlandırılan yöntemi kullanır. Muhasebe yöntemi, daha sonra pahalı operasyonlar için ödeme yapmak üzere siber dolarları depolar. Buradaki fikir, istenen karmaşıklıktan daha pahalı olan herhangi bir işlem için ödeme yapmaya yetecek kadar siber dolar olması gerektiğidir

61 2 Hesaplamalı Karmaşıklık

Başlangıçta boş olan bir liste üzerinde n adet ekleme işlemi dizisi düşünün. Listeye ilk öğenin eklenmesi O(1) zamanda yapılır çünkü listeye eklenen ilk öğe için yer vardır çünkü listede başlangıçta bir yuva ayrılmıştır. Önceden ayrılmış bir yuvaya bir değer depolamak O(1) zaman alır. Bununla birlikte, hesaplama yöntemine göre, ekleme işlemini yapmanın maliyetinin ek iki siber dolar gerektirdiğini iddia edeceğiz. Bu hala O(1) karmaşıklıktır. Alanımız her tükendiğinde, sabit boyutlu listedeki yuva sayısını iki katına çıkaracağız. Sabit boyutlu bir listeyi ayırmak, liste boyutundan bağımsız olarak bir O(1) işlemidir. Ekstra iş, elemanları eski listeden yeni listeye kopyalarken ortaya çıkar.



**Şekil 2.7** Siber Dolar Ekleme

62 2 Hesaplamalı Karmaşıklık

Boyutu ilk kez ikiye katlamamız gerektiğinde ikinci ekleme çağrılır. Bu noktada depolanan iki siber dolar vardır. Bunlardan biri, eski listede saklanan bir elemanı iki eleman tutabilen yeni sabit boyutlu listeye kopyalarken gereklidir. Şekil 2.7'deki birinci geçiş, depolanan iki siber doları ve A adımından B adımına geçerken yeni listeye kopyalandıktan sonraki sonucu göstermektedir.

Listenin B versiyonuna append çağrıldığında sonuç C versiyonu olur. Bu noktada, liste boyutunu dörde katlarken kullanılmak üzere üç siber dolar saklanır. İlk ikisi listenin eski içeriği ile doldurulur. Saklanan üç siber dolardan ikisi bu değerler yeni listeye kopyalanırken kullanılır. Dört boyutlu liste dolduğunda, beş siber dolar depolayan iki ekleme işlemi daha gerçekleşmiştir. Bu siber dolarlardan dördü adım E'den adım F'ye kopyalama işleminde kullanılır. Yine, adım G'de sekiz boyutlu liste dolduğunda, liste boyutunu ikiye katlamak ve öğeleri kopyalamak için kullanılacak dokuz saklı siber dolar vardır.

Ama ya listenin boyutunu her seferinde iki katına çıkarmazsak? Liste boyutunu her seferinde bir önceki boyutunun yarısı kadar artırırsak, her ekleme işlemi için dört siber dolar depolarsak bu argümanı yine de çalıştırabiliriz. Aslında, listenin boyutu her genişletildiğinde mevcut boyutuyla orantılı olarak büyüdüğü sürece bu argüman, listelerin sabit bir boyutla tahsis edilmesi gerektiğinde bir listeye ekleme yapmanın O(1) işlemi olduğunu kanıtlamak için hala çalışır.

Daha önce de belirtildiği gibi, Python liste nesnesi C dilinde uygulanmıştır. Python bir ekleme işlemi sağlarken, C dili yalnızca C'de diziler olarak adlandırılan sabit boyutlu listeler tahsis edebilir. Yine de, Python liste nesneleri, deneylerle veya Python liste nesnelerini uygulayan C kodunu analiz ederek gözlemlenebileceği gibi O(1) sürede nesneleri ekleyebilir. Python liste ekleme uygulaması bunu, sabit boyutlu dizide yer kalmadığında bu bölümde açıklandığı gibi liste boyutunu artırarak O(1)'lik bir amortize karmaşıklık elde ederek başarır.

**2.11 Bölüm Özeti**

Bu bölümde algoritmaların verimliliği ile ilgili bazı önemli konular ele alınmıştır. Verimlilik önemli bir konudur çünkü en hızlı bilgisayarlar bile kendileri için yazılan programlar verimsizse problemleri makul bir sürede çözemeyeceklerdir. Aslında, program nasıl yazılmış olursa olsun bazı problemler makul bir sürede çözülemez. Yine de, bu verimlilik konularını anlamamız önemlidir. Bir kod parçasının karmaşıklığını bulmak, ne kadar çok pratik yaparsanız o kadar iyi olacağınız önemli bir beceridir. İşte bu bölümde öğrenmiş olmanız gereken bazı şeyler. Öğrenmelisiniz:

* Bir listeden veya bilgisayarın belleğinden bir değerin saklanması veya alınmasının karmaşıklığını bilir.
* Hafızanın nasıl bir postane gibi olduğunu
* Hafızanın postane gibi olmadığını bilir

2.11 Bölüm Özeti 63

* bir programda bir işlemin tamamlanması için geçen süre hakkında bilgi almak için datetime modülünün nasıl kullanılacağını bilmek
* Bir algoritmanın veya kod parçasının performansı hakkında bilgi çizmek için çizim programı tarafından kullanılabilecek bir XML dosyasının nasıl yazılacağını bilmek.
* big-Oh gösteriminin tanımını ve bir kod parçasının performansı üzerinde nasıl bir üst sınır oluşturduğunu anlamak..
* liste + işleminin neden ekleme işlemi kadar verimli olmadığını anlamak.
* O(n), O(n²) ve diğer hesaplama karmaşıklıkları arasındaki farkı ve bu farklılıkların bilgisayar programcıları olarak bizim için neden önemli olduğunu anlamak.
* Theta notasyonunu ve asimptotik olarak sıkı bir sınırın bir algoritma hakkında ne söylediğini anlayın.
* Amortismana tabi karmaşıklığı ve bazı basit durumlarda nasıl uygulanacağını anlamak.

**2.12 Soruları İnceleyin**

Bu kısa cevaplı, çoktan seçmeli ve doğru/yanlış soruları yanıtlayarak bu bölümdeki hakimiyetinizi test edin.

1. **Bir liste nasıl bir grup posta kutusu benzer?**

Cevap: Bir liste her biri bir değere sahip bir dizi bellek konumudur. Bu, bir posta kutusunun bir dizi posta kutusu gibi organize edilmesi ve her bir posta kutusunun bir mektup içerebilmesi gibi bir listedeki her bellek konumunun bir değer içerebilmesine benzer.

1. **Bir listenin bir öğesine erişmek nasıl olur da bir posta kutusunun içeriğini almak gibi DEĞİLDİR?**

Cevap: Bir listenin bir öğesine erişmek, doğrudan belirli bir elemanı indeks numarasıyla çağırmakla yapılır. Yani listenin bir öğesini almak, belirli bir yerden veri çekmeyi içerir ve genellikle hızlıdır. Buna karşılık, bir posta kutusunun içeriğini almak, o kutudaki tüm mesajları incelemeyi veya sıralamayı gerektirir ve genellikle bir arama ya da filtreleme işlemi yapılması gerekir. Listenin öğesine erişim belirli ve doğrudan iken, posta kutusunun içeriğini almak daha kapsamlı ve dolaylı bir işlem olduğu için farklı bir yaklaşım gerektirir. Bu fark, listenin öğelerine hızlı erişim sağlarken, posta kutusunda tüm mesajların sıralanıp aranması gerektiği için zaman alıcı olabilir.

1. **Python kullanarak bir bilgisayarda bir işlemi tamamlamak için gereken süreyi nasıl hesaplayabilirsiniz?**

Cevap:

Python kullanarak bir işlemin tamamlanma süresini hesaplamak için, time modülünü kullanabilirsiniz. Bu modül, işlemin başlangıcını ve bitişini kaydederek, aradaki farkı hesaplamanızı sağlar. İşlemi başlatmadan önce zamanı alırsınız, işlem tamamlandığında ise bitiş zamanını kaydedersiniz ve bu iki zamanı çıkararak geçen süreyi öğrenirsiniz. Bu işlem, genellikle performans testi yapmak veya işlem sürelerini ölçmek için kullanılır. Örnek olarak:

import time

# İşlemi başlatma

start\_time = time.time()

# Yapılacak işlem (örneğin bir döngü veya hesaplama)

for i in range(1000000):

pass

# İşlem tamamlandıktan sonra bitiş zamanını al

end\_time = time.time()

# Geçen süreyi hesapla

elapsed\_time = end\_time - start\_time

print(f"İşlem tamamlanma süresi: {elapsed\_time} saniye")

1. **Hesaplama karmaşıklığı açısından hangisi daha iyidir, O(n²) olan bir algoritma mı yoksa O(2^n) olan bir algoritma mı?**

Cevap:

O(n^2) algoritması, n değeri büyüdükçe **polinomik** bir şekilde artar.

O(2^n) algoritması, n değeri büyüdükçe **üstel** bir şekilde artar.

Üstel artış, polinomik artıştan **çok daha hızlıdır**.

O(n²), özellikle n büyüdükçe, işlemlerin sayısının hızlıca arttığı ancak yine de polinomsal bir büyüme gösterdiği için genellikle daha verimlidir. Ancak **O(2ⁿ)** algoritması, girişin boyutu arttıkça çok hızlı şekilde büyür. Örneğin, n=10 için O(2ⁿ) algoritması 1024 işlem yaparken, n=20 için 1,048,576 işlem yapar ki bu çok büyük bir sayı olabilir.

1. **Bir algoritmanın O(n^2) olmasının ne anlama geldiğini İngilizce olarak açıklayın**.

Cevap:

An algorithm with a time complexity of O(n²) means that as the input size (n) increases, the time required for the algorithm to run increases quadratically. In other words, if the input size doubles, the execution time will increase by a factor of four. This usually happens when the algorithm contains nested loops where each loop iterates over the input dimension. For example, in an algorithm with two loops each running n times, the total number of operations will be proportional to n × n

1. **Tümevarım yoluyla ispat yaparken, ispatın hangi iki kısmı vardır?**

de:

Tümevarım yoluyla ispat yaparken, ispatın iki ana kısmı vardır: **temel adım** ve **indüksiyon adımı**:

1. **Temel Adım (Base Case):** Bu adımda, genellikle en küçük değer için doğruluğu gösterirsiniz. Yani, ispat edilmek istenen ifadenin ilk durumu (örneğin, n=1) doğru kabul edilir ve doğruluğu gösterilir. Bu adım, ispatın başlangıç noktasını oluşturur.
2. **İndüksiyon Adımı (Inductive Step):** Bu adımda, ispat edilen ifadenin daha büyük bir durumda doğru olduğunu varsayarsınız (indüksiyon hipotezi). Ardından, bu varsayımı kullanarak bir sonraki değerin doğru olduğunu gösterirsiniz. Yani, eğer n=k için doğruysa, n=k+1 için de doğru olduğunu kanıtlamak gerekir.

Bu iki adım, tümevarımın temelini oluşturur ve genellikle tüm doğal sayılar için doğru olduğunu ispatlamak amacıyla kullanılır.

1. **İlk seferde n adım, ikinci seferde n - 2 adım, sonraki seferde n - 4 adım yürüten ve son seferde 2 adım yürütene kadar tekrar eden bir döngüye sahip bir algoritmanız olsaydı, bu döngünün karmaşıklık ölçüsü ne olurdu? Cevabınızı bu bölümde öğrendiklerinizle gerekçelendirin.**

Cevap:

.Bu algoritmanın karmaşıklığını analiz ederken, her seferde yapılan adımlar şu şekilde azalır: ilk seferde n, ikinci seferde n-2, üçüncü seferde n-4 adım vb. Son seferde ise 2 adım yapılır. Toplam adım sayısı, bir aritmetik seri oluşturur ve bu serinin toplamı **O(n²)** olur. Çünkü, her seferde yapılan adımlar azalırken, toplamda n² büyüklüğüne yakın bir işlem yapılır. Sonuç olarak, bu algoritmanın karmaşıklığı **O(n²)** olur.

1. **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:

Açıklama:

Algoritma A, her öğeyi sabit bir sürede (10 adım) işlediği için karmaşıklığı n ile orantılıdır.

Algoritma B ise her öğeyi 100 adımda işlediği için karmaşıklığı 100n ile orantılıdır.

Sonuç:

Algoritma A, Algoritma B'den daha verimlidir.

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

Cevap:

Append İşlemi:

Bir listeye yeni bir öğe eklemek için kullanılır.

Listenin sonuna yeni bir öğe ekler.

Sabit bir sürede tamamlanır.

+ Operatörü:

İki listeyi birleştirmek için kullanılır.

Yeni bir liste oluşturur.

Her iki listenin de her öğesini kopyalar.

Karmaşıklığı O(n) şeklindedir.

Verimlilik:

Append işlemi, + operatöründen daha verimlidir.

Append işlemi sabit bir sürede tamamlanırken, + operatörünün karmaşıklığı O(n) şeklindedir.

Nedeni:

Append işlemi, listenin sonuna yeni bir öğe ekler. Bunu yapmak için bellekte sadece bir konum ayarlamak gerekir.

+ operatörü ise iki listeyi birleştirmek için yeni bir liste oluşturur. Bunu yapmak için her iki listenin de her öğesini kopyalamak gerekir.

1. **Bir listede belirli bir değeri bulmak için bir algoritma tanımlayın. Ardından bu algoritmanın hesaplama karmaşıklığını verin. İstediğiniz varsayımları yapabilirsiniz, ancak varsayımlarınızı algoritmanızla birlikte belirtmelisiniz.**

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:

```python

def find\_value(list, value):

for i in range(len(list)):

if list[i] == value:

return i

return -1

list = [1, 2, 3, 4, 5]

value = int(input(“sayi giriniz: “))

index = find\_value(list, value)

if index == -1:

print("Değer listede bulunamadı.")

else:

print(f’{value} sayisi {index}. İndexde 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.

Cavap:

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.

Cavap:

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:

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.