**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.