KÜMELER VE HARİTALAR 5

Son bölümde, yinelenen değerlere izin verilen şeylerin listelerini takip etmek için kullanılan dizileri inceledik. Örneğin, bir dizide ya da tam sayılar listesinde iki tane altı olabilir. Bu bölümde, yinelenen değerlere izin verilmeyen kümelere bakacağız. Kümeleri inceledikten sonra haritalar hakkında konuşmaya geçeceğiz. Haritalar sözlükler veya karma tablolar olarak da adlandırılabilir. Hash tablosu terimi aslında bir küme ya da harita uygulaması anlamına gelir. Bu bölümün birincil odak noktası hashing'i anlamaktır. Hashing Bilgisayar Bilimlerinde çok önemli bir kavramdır çünkü bir değeri aramak için çok verimli bir yöntemdir. Bölüme başlamak için hashing'e olan ilgimizi motive edeceğiz, ardından bir kümedeki değerleri bulmak için bir hashing algoritması geliştireceğiz. Ayrıca hashing'i kümelerin ve haritaların oluşturulmasına da uygulayacağız. Daha sonra bellekleme adı verilen hashing kullanan önemli bir tekniğe bakacağız ve bu tekniği birkaç probleme uygulayacağız.

5.1 Bölüm Hedefleri

Bu bölümde birkaç soyut veri tipinin nasıl uygulanacağını öğreneceksiniz: kümeler ve eşlemeler. Bu veri türlerinin her ikisi için de hashing'in önemini okuyacaksınız. Ayrıca mutable ve immutable veriler arasındaki farkı anlamanın önemini de öğreneceksiniz. Bölümün sonunda şu sorulara cevap verebiliyor olmalısınız.

• Bir kümede bir değer bulmanın karmaşıklığı nedir?

• Yük faktörü nedir ve bir hash tablosunda aramanın genel verimliliğini nasıl etkiler?

• Değişmez bir kümeyi ne zaman kullanırsınız?

• Ne zaman değiştirilebilir bir set isteyebilirsiniz?

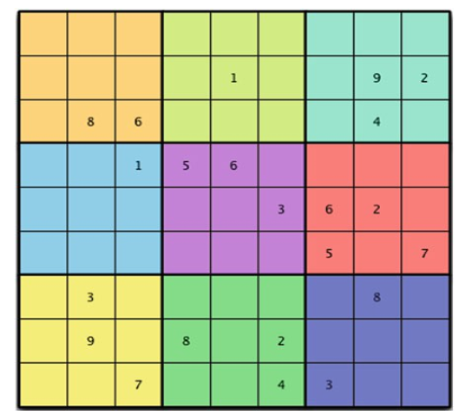
• Bir problemde bellekleme kullanmanın ne zaman bir avantajı vardır?

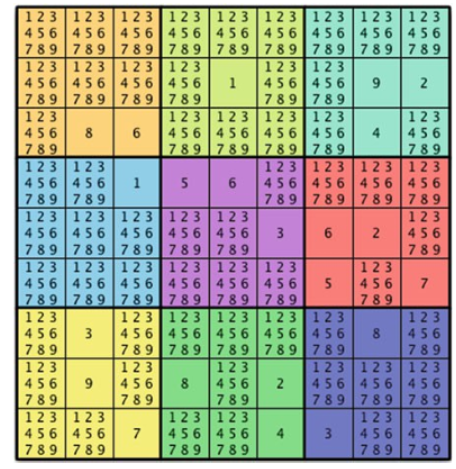
• Hangi durumlarda bir harita veya sözlük kullanmak faydalı olabilir?

Yine bu bölümde, ilk olarak geçen bölümde sunulan tic tac toe oyununun optimizasyonu ve bir Sudoku bulmaca çözücüsü de dahil olmak üzere bazı ilginç programlama problemleri yer alacaktır. Bu ilginç problemleri çözmek için bilmeniz gerekenleri keşfetmek için okumaya devam edin.

5.2 Sudoku Oynamak

Birçok insan Sudoku bulmacalarını çözmekten hoşlanır. Bir Sudoku bulmacasını çözmek için 9 × 9 matrisini dolduracak doğru sayıları bulmalısınız. Tüm sayılar 1- 9 olmalıdır. Her satırda 1-9'dan birer tane bulunmalıdır. Aynı şey her sütun için de geçerlidir. Son olarak, 9 × 9 matris içinde her birinde 1-9 olması gereken dokuz adet 3 3 kare vardır. Başlamak için, size Şekil 5.1'de gösterildiği gibi bazı konumları bilinen bir bulmaca verilir. Göreviniz, bulmacada zaten görünen sayılar göz önüne alındığında geri kalan sayıları bulmaktır. Bu bulmacaları çözmenin yaygın bir yolu eleme sürecidir. Bir bulmaca için olası değerleri yazmak ve ardından olası değerleri teker teker elemek yardımcı olur. Örneğin, yukarıdaki bulmaca Şekil 5.2'de gösterildiği gibi bilinmeyenler için olası değerlerle açıklanabilir. Bulmacayı çözmeye başlamak için, 8, 3 veya 9 içermeyen hücreler için bulmacanın ikinci sütunundan 8, 3 ve 9'u hemen eleyebiliriz. Bu sayıların hiçbiri ikinci sütundaki başka bir hücrede görünemez çünkü zaten bilinmektedirler. Aynı şekilde, 8, 6 ve 4 sayıları da bazı hücrelerden elenebilir

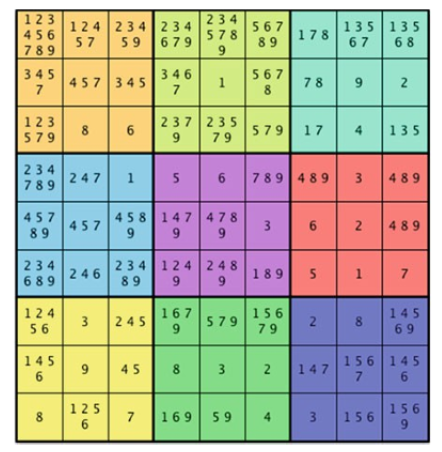




çünkü bu sayılar zaten diğer hücrelerde yer almaktadır. Bu gibi kuralların uygulanması bulmacadaki her bir hücre için olası değer sayısını azaltır. Şekil 5.3 bu kurallardan bazılarını uyguladıktan sonraki bulmacayı göstermektedir. Sudoku ve nasıl çözüleceği hakkında biraz düşünürsek, birçok Sudoku bulmacasını çözmek için kullanılabilecek iki kural türetebiliriz. Bu iki kural bulmaca içindeki herhangi bir gruba uygulanabilir. Bir grup, bulmacada bir satır, sütun veya kare içinde görünen dokuz hücreden oluşan bir koleksiyondur. Her hücrenin içinde, bulmacayı küçültme sürecinin bir noktasında hücre için olası değerleri temsil eden bir dizi sayı bulunur. İşte iki kural:

• KURAL 1: İlk kural, yukarıda hücrelerden bazı değerleri çıkarmak için kullandığımız işlemin bir genellemesidir. Bir grup içinde aynı olası değerler kümesini içeren hücreleri arayın. Kümenin kardinalitesi (yani kümedeki öğe sayısı) bulunan yinelenen kümelerin sayısıyla eşleşiyorsa, yinelenen kümelerin öğeleri gruptaki tüm yinelenmeyen kümelerden güvenli bir şekilde çıkarılabilir. Bu kural, yinelenen kümelerin sayısının 1 ve bu kümenin boyutunun 1 olduğu dejeneratif durumda bile geçerlidir. Dejeneratif durum, yukarıda tek öğeleri diğer kümelerden çıkarmak için kullandığımız durumdur. Bu kural, 2'nin tek başına göründüğü 7. sütun hariç bulmacanın 7. satırındaki tüm hücrelerden 2'yi çıkarmak için Şekil 5.3'e uygulanabilir.

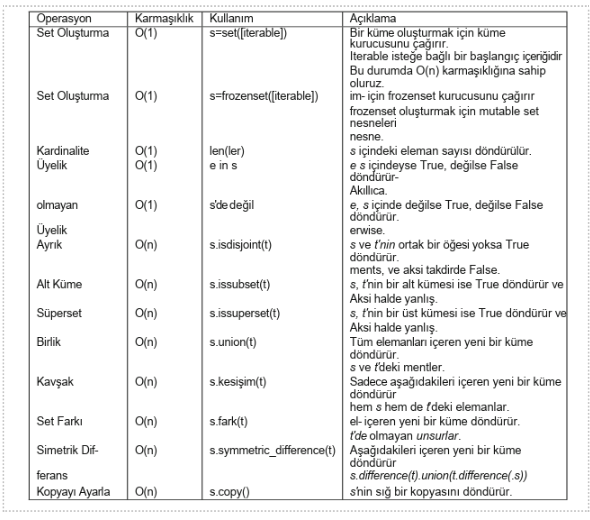
• KURAL 2: İkinci kural, bir grup içindeki her hücreye bakar ve gruptaki diğer hücrelerde görünen tüm öğeleri atar. Seçilen hücrede yalnızca bir değer kaldıysa, bu değer bu hücrede görünmelidir ve hücre aşağıdakiler atılarak güncellenebilir.



seçilen hücrede görünen diğer tüm değerleri uzaklaştırır. Bu kuralın Şekil 5.3'teki beşinci satıra uygulanması, dördüncü sütunun 1 içerecek şekilde indirgenmesiyle sonuçlanır çünkü 1, 5. satırdaki başka hiçbir hücrede görünmez. Bu kural bulmacanın son satırında da geçerlidir. 2 değeri ancak ikinci sütunda 1, 5 ve 6'yı o hücreden çıkardıktan sonra mümkündür çünkü bu değerler o satırdaki diğer hücrelerde yer almaktadır. Şekil 5.3'teki bulmacanın tam olarak indirgenmediğine dikkat edin. İndirgeme işlemi, daha fazla indirgeme mümkün olmayana kadar yinelemeli olarak uygulanabilir. Sudoku çözücü algoritması, bulmacadaki tüm gruplar üzerinde bir azaltma geçişi sırasında daha fazla değişiklik yapılmayana kadar bu azaltma işlemini uygulamaya devam eder. Bu iki kuralı bu şekilde uygulamak birçok Sudoku bulmacasını tamamen indirgeyecektir.

5.3 Setler

Sudoku bulmacaları için indirgeme algoritması sayı kümelerini manipüle eder ve indirgeme ilerledikçe bu kümelerden olası değerleri elimine eder. Küme bir koleksiyon yinelenen değerlere izin vermez. Kümeler herhangi bir değerden oluşabilir. Tamsayılar, çalışan nesneler, karakterler, dizeler, kelimenin tam anlamıyla Python'daki herhangi bir nesne bir kümenin elemanı olabilir. Bir kümenin bir kardinalitesi vardır. Bir kümenin kardinalitesi, kümedeki içindeki öğelerin sayısıdır.

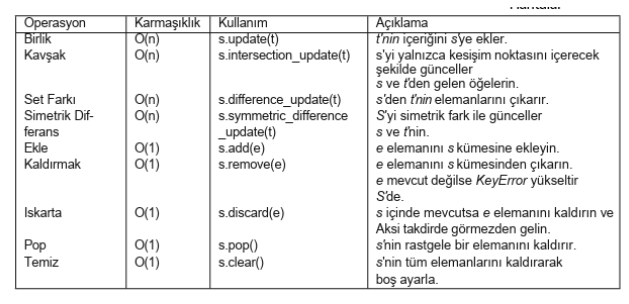


Kümeler, üzerlerinde yaygın olarak tanımlanmış birkaç işlem bulunan nesnelerdir. Bu işlemler bazen birden fazla kümeyi içeren ikili işlemlerdir, bazen de sadece bir küme hakkında bilgi alırlar. Şekil 5.4'teki tablo, kümeler üzerinde yaygın olarak tanımlanan işlemleri ve bunlarla ilişkili hesaplama karmaşıklıklarını açıklamaktadır. Python iki tür küme için yerleşik desteğe sahiptir: set ve frozenset class. Frozenset class değişmezdir. Küme sınıfının nesneleri değiştirilebilir. Şekil 5.4'te s değişkeni bir küme ve t değişkeni de kümeleri içeren yinelenebilir bir dizi olmalıdır.

Infix operatörleri de Şekil 5.4'te tanımlanan bazı işlemler için sözdizimsel şeker olarak tanımlanmıştır. Alt küme içerme için s <= 1 yazabilirsiniz. Uygun alt küme için s < t yazabilirsiniz. Uygun alt küme, üst kümesinden

en az bir daha az elemanı olan bir alt kümedir. Üst küme için s >= t veya uygun üst küme için s > t yazabilirsiniz. Birleştirme işlemi için s❘t yazmak s.union(t) yazmakla eşdeğerdir. Ve kesişim için, s&t, s.intersection(t) yazmaya eşdeğerdir. s t yazmak - s.difference(t) yazmakla aynıdır ve s^t simetrik fark operatörüne eşdeğerdir. Şekil 5.5'teki işlemler, s kümesini değiştirdikleri için frozenset sınıfı üzerinde tanımlanmamıştır.

 Yine, Şekil 5.5'te sunulan bazı yöntemler için operatörler vardır. Mutatör birleştirme yöntemi st olarak yazılabilir. Kesişim güncellemesi s&=t olarak yazılabilir. Son olarak, simetrik fark güncelleme operatörü s^=t olarak yazılır. Bu operatörler kullanışlı olmakla birlikte çok iyi bilinmemektedir.



Yukarıda sunulan hesaplama karmaşıklıkları şaşırtıcıdır! Küme üyeliği O(1) zamanda nasıl test edilebilir? Şimdiye kadar anlatılanlara bakılırsa, küme üyeliğini test etmek O(n) zaman almalıdır. Sonuçta, bir öğenin kümede olup olmadığını anlamak için kümedeki tüm öğelere ya da ortalama olarak en az yarısına bakmamız gerekir. Kümede yinelenen öğe olmadığından emin olmak istiyorsak, iki kümenin birleşimi O(n) zamanda nasıl hesaplanabilir? Küme bir şekilde sıralanmadığı sürece iki kümenin birleşiminin hesaplanması O(n²) zaman alacak gibi görünmektedir. Ancak bir kümenin elemanlarını sıralamak her zaman mümkün değildir çünkü kümelerin tüm elemanları bir sıralamaya sahip değildir.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5.4 HASHING

Bir küme üyelik testini O(1) zamanda uygulamak mümkünse, yukarıdaki diğer işlemleri belirttiğimiz karmaşıklıklarla uygulayabiliriz. O(1) üyelik testi olmadan, iki kümenin birleşimini almak yukarıda belirtildiği gibi çok daha uzun sürecektir. Küme üyeliğini O(1) zamanda test etmek hashing kullanılarak gerçekleştirilir. Hashing, Bilgisayar Biliminde son derece önemli bir kavramdır ve bir bilgisayardaki random erişimi ile ilgilidir. Bölüm 2'de gördüğümüz gibi, bir liste içindeki herhangi bir konuma erişim O(1) zamanda gerçekleştirilebilir. Bu rastgele erişim prensibidir. Rastgele erişilebilir bir liste, liste içindeki herhangi bir konuma O(1) zamanda erişilebileceği anlamına gelir. Listedeki bir konuma erişmek için erişmek istediğimiz konumun indeksine ihtiyacımız vardır. İndeks, listedeki bir öğenin adresi olarak işlev görür. Bir öğeyi listeye kaydettikten sonra, O(1) sürede geri getirmek istiyorsak, indeksini hatırlamamız gerekir. İndeks olmadan listedeki öğeyi aramak zorunda kalırız ki bu da O(1) zaman değil O(n) zaman alır. Dolayısıyla, üyeliği O(1) zamanda test edebileceğimiz bir küme uygulamak istersek, kümenin öğelerini bir listede saklamayı düşünebiliriz. O(1) zamanda tekrar bulmak için her bir öğenin saklandığı indeksi bir şekilde hatırlamamız gerekir. Bu ilk başta mümkün görünmüyor. Ancak, ya öğe adresini bulmak için kullanılabilseydi? Bu, hashing'e yol açan içgörüdür. Bilgisayardaki her nesne bilgisayarlar ikili sistemde çalıştığından sıfırlar ve birlerden oluşan bir dizi olarak saklanır. Bu sıfırlar ve birler, bir listenin indeksi olarak da dahil olmak üzere istediğimiz şekilde yorumlanabilir. Bu kavram o kadar önemlidir ki Python (ve diğer birçok modern dil) bir nesne için bir tamsayı değeri döndürmek üzere herhangi bir nesne üzerinde çağrılabilen hash adlı bir fonksiyon eklemiştir. Bu değere nesnenin hash kodu veya hash değeri diyeceğiz. Hash fonksiyonuna yapılan şu çağrıları düşünün.

Python 3.2 (r32:88452, Feb 20 2011, 10:19:59)

[GCC 4.0.1 (Apple Inc. build 5493)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>> hash("abc")

-1600925533

>>> hash("123")

1911471187

>>> hash(45)

45

>>> hash(45.0)

45

>>> hash(45.3)

1503225491

>>> hash(True)

1

>>> hash(False)

0

>>> hash([1,2,3])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unhashable type: 'list'

>>>

Çoğu nesne hashlenebilir olsa da, her nesne hashlenebilir değildir. Özellikle, listeler gibi değişebilir nesneler hashlenebilir olmayabilir, çünkü bir nesne değiştiğinde hash değeri de değişebilir. Bunun, bu bölümün ilerleyen kısımlarında göreceğimiz gibi, veri yapılarında hash değerleri kullanırken sonuçları vardır. Yerleşik türlere ek olarak, Python programcının bir sınıf üzerinde bir hash yöntemi uygulayarak hash kodları üzerinde biraz kontrol sahibi olmasına izin verir. Bir sınıf için bir hash metodu yazarsanız, o sınıfın örnekleri için istediğiniz hash değeri tamsayısını döndürebilirsiniz. "abc" dizesi üzerinde hash çağrısı negatif bir değer döndürürken, diğer hash çağrılarının son derece büyük tamsayılar döndürdüğüne dikkat edin. Bu hash tamsayısını bir listede kabul edilebilir bir indekse dönüştürmek için bazı çalışmalar yapılması gerektiği açıktır. Hash değerlerinin liste indekslerine nasıl dönüştürüldüğünü keşfetmek için okumaya devam edin.

5.5 HashSet

Sınıfı O(1) öğe arama karmaşıklığı elde etmek için bir listeye indeks hesaplamak üzere bir hash değeri kullanabiliriz. Listenin ayrıntılarını ve öğenin indeksini bulmak için hash fonksiyonunun çağrılmasını gizlemek için bir küme sınıfı yazılabilir. Python'un yerleşik küme sınıfıyla karıştırılmaması için küme sınıfımıza HashSet adını vereceğiz. Yerleşik küme sınıfı da hashing kullanır. Bu bölümde sunulan HashSet sınıfı size kümenin nasıl

sınıfı uygulanmaktadır. Başlangıçta, HashSet nesneleri bir liste ve listedeki öğelerin sayısını içerecektir. Başlangıçta liste bir grup None değeri içerecektir. Liste, içinde bir tür değer olacak şekilde oluşturulmalıdır. None, listede hiçbir değerin saklanmadığı yerler için null değer görevi görür. Liste, olası her hash değeri için bir konuma sahip olacak kadar büyük değildir. Yine de, listenin tüm olası hash değerleri için yeterince büyük olması mümkün değildir. Aslında, son bölümde gördüğümüz gibi, bazı hash değerleri negatiftir ve açıkça bir listedeki indisler negatif değildir. Bir hash değerinin bir liste indeksine dönüştürülmesi Bölüm 5.5.2'de daha ayrıntılı olarak açıklanmıştır. HashSet kurucusu Bölüm 5.5.1'de verilmiştir.

5.5.1 HashSet oluşturucu

class HashSet:

def \_\_init\_\_(self, contents=[]):

self.items = [None] \* 10

self.numItems = 0

for item in contents:

self.add(item)

5.5.2 Bir Öğeyi Saklama

 Bir öğeyi bir hash kümesinde saklamak için önce hash fonksiyonunu kullanarak indeksini hesaplarız. Ele alınması gereken iki sorun vardır. İlk olarak, öğelerin saklandığı liste sonlu uzunlukta olmalıdır ve kesinlikle hash fonksiyonunu çağırarak oluşturacağımız benzersiz hash değerleri kadar uzun olamaz. Listenin maksimum hash değerinden daha kısa olması gerektiğinden, listemiz için bir boyut seçeriz ve ardından hash değerlerini bu listenin uzunluğuna böleriz. Kalan (yani mod operatörü olarak adlandırılan % operatörünün sonucu) listenin indeksi olarak kullanılır. Listenin uzunluğuna bölündükten sonra kalan, hash değeri negatif bir tamsayı olsa bile her zaman 0 ile listenin uzunluğu eksi bir arasında olacaktır. Mod operatörünü kullanmak, seçtiğimiz boyuttaki bir listede bize geçerli indeksler verecektir. Başa çıkmamız gereken başka bir sorun daha var. Hash değerlerinin benzersiz olması gerekmez. Hash değerleri tam sayılardır ve bir bilgisayarda yalnızca sonlu sayıda tam sayı mümkündür. Buna ek olarak, hash değerlerini listenin uzunluğuna böldüğümüz için, kalanlar veya liste indeksleri orijinal hash değerlerinden daha az benzersiz olacaktır. Liste uzunluğu 10 ise, 44 ve -6 hash değerlerinin her ikisi de listede 4. indekste bir değer saklamaya çalışmakla sonuçlanacaktır. Bu elbette mümkün değildir.

 5.5.3 Çarpışma

Çözünürlüğü Uzunluğu 10 olan bir listede hash kullanarak hem "Cow" hem de "Fox "u saklamaya çalıştığınızı düşünün. "Cow "un hash değeri -1432160826 ve "Fox "un hash değeri 1462539404'dur. Her iki değeri de 10 ile modladığımızda, kalan değer her iki hash değeri için de 4'tür ve bu da her ikisinin de listedeki beşinci konumda saklanması gerektiğini gösterir.

Hesaplanan indeksleri aynı olduğu için iki nesnenin karma küme listesinde aynı indekste saklanması gerekiyorsa, buna çakışma diyoruz. Bununla başa çıkmak için bir çarpışma çözümleme şeması tanımlamak gerekir. Mümkün olan birçok farklı şema vardır. Biz Lineer Problama adı verilen bir şemayı inceleyeceğiz. Doğrusal problama kullanırken bir çarpışma meydana geldiğinde, o konumun kullanılabilir olup olmadığını görmek için listedeki bir sonraki konuma ilerleriz. Bir konumun kullanılabilir olup olmadığını, listede o noktada bir None değeri bulursak anlayabiliriz. Listede bulabileceğimiz ve o konumun kullanılabilir olduğu anlamına gelen başka bir değer daha olduğu ortaya çıktı. Placeholder nesnesi adı verilen özel bir nesne türü de listede saklanabilir. Bu sınıfın nedeni bir sonraki bölümde ortaya çıkacaktır. Şimdilik, None ya da Placeholder nesnesi hash set listesi içinde açık bir konumu gösterir. Bölüm 5.5.4'teki kod, HashSet listesine bir öğe ekleme işlemini gerçekleştirir ve asıl add yöntemi için bir yardımcı işlevdir.   
 