1.2 Soyut Veri Tipleri ve Veri Yapıları

Bir **tip** (type), bir değerler topluluğudur. Örneğin, Boole tipi **true** ve **false** değerlerinden oluşur. Tamsayılar da bir tip oluşturur. Bir tamsayı basit bir tiptir çünkü değerleri alt parça içermez. Bir banka hesap kaydı genellikle ad, adres, hesap numarası ve hesap bakiyesi gibi birkaç bilgi içerir. Böyle bir kayıt, topluluk tipi veya bileşik tip için bir örnektir. Bir **veri öğesi** (data item), değeri bir tipten alınan bir bilgi parçası veya bir kayıttır. Bir veri öğesinin bir tipin **üyesi** olduğu söylenir.

Bir **veri tipi**, tipi işlemek için bir dizi işlemle birlikte bir tiptir. Örneğin, bir tamsayı değişkeni, tamsayı veri tipinin bir üyesidir. Toplama, tamsayı veri tipi üzerinde bir işlem örneğidir.

Bir veri tipinin mantıksal konsepti ile bir bilgisayar programında fiziksel uygulaması arasında bir ayrım yapılmalıdır. Örneğin, liste veri tipi için iki geleneksel uygulama vardır: bağlantılı liste ve dizi tabanlı liste. Liste veri tipi bu nedenle bağlantılı bir liste veya bir dizi kullanılarak uygulanabilir. "Dizi" terimi bile belirsizdir, çünkü ya bir veri tipine ya da bir uygulamaya atıfta bulunabilir. "Dizi", bilgisayar programcılığında yaygın olarak, her bir bellek konumunun bir sabit uzunluklu veri öğesini depoladığı bitişik bir bellek konumları bloğu anlamına gelir. Bu anlamda bir dizi fiziksel bir veri yapısıdır. Bununla birlikte dizi, her bir veri öğesinin bir indeks numarası ile tanımlandığı (tipik olarak homojen) veri öğeleri koleksiyonundan oluşan mantıksal bir veri tipi anlamına da gelebilir. Dizileri birçok farklı şekilde uygulamak mümkündür. Örneğin, büyük bir iki boyutlu dizi olan seyrek bir matrisi uygulamak için kullanılabilir. Bu uygulama, bir dizinin bitişik bellek konumları olarak fiziksel temsilinden oldukça farklıdır.

Bir **Soyut Veri Tipi** (ADT – Abstract Data Type), bir veri tipinin bir yazılım bileşeni olarak gerçekleştirilmesidir. ADT'nin arayüzü, bir tip ve bu tipteki bir dizi işlem açısından tanımlanır. Her işlemin davranışı, girdileri ve çıktıları tarafından belirlenir. ADT, veri tipinin *nasıl* uygulanacağını belirtmez. Bu uygulama ayrıntıları, ADT kullanıcısından gizlenir ve kapsülleme (encapsulation) olarak adlandırılan bir kavram olan dış erişimden korunur.

Bir **veri yapısı** (data structure), bir ADT'nin uygulanmasıdır. Nesne yönelimli bir dilde, bir ADT ve uygulaması birlikte bir **sınıf** (class) oluşturur. ADT ile ilişkili her işlem, bir **üye fonksiyonu** veya **metodu** tarafından uygulanır. Bir veri öğesinin gerektirdiği alanı tanımlayan değişkenlere **veri üyeleri** denir. Bir **nesne** (object), bir sınıfın bir örneğidir, yani bir bilgisayar programının yürütülmesi sırasında oluşturulan ve değer alan bir şeydir.

"Veri yapısı" terimi genellikle bir bilgisayarın ana belleğinde depolanan verileri ifade eder. İlgili bir terim olan **dosya yapısı** ise genellikle bir disk sürücüsü veya CD gibi çevresel depolama birimi üzerindeki verilerin organizasyonu ile ilgilidir.

Örnek 3

Bir tamsayının matematiksel kavramı, tamsayıları manipüle eden işlemlerle birlikte bir veri tipi oluşturur. **int** değişken tipi, soyut tamsayının fiziksel bir temsilidir. **int** değişken tipi, bir **int** değişkeni üzerinde çalışan işlemlerle birlikte bir ADT oluşturur.

Bazı ADT'leri kullanan bir uygulama, o ADT'nin belirli üye fonksiyonlarını ikinci bir uygulamadan daha fazla kullanabilir veya iki uygulamanın çeşitli işlemler için farklı zaman gereksinimleri olabilir. Uygulama gereksinimlerindeki bu farklılıklar, belirli bir ADT'nin birden fazla uygulama tarafından desteklenmesinin nedenidir.

Örnek 4

Büyük disk alanı kullanan veri tabanı uygulamaları için iki popüler uygulama, hashing ve B+ ağacıdır. Her ikisi de kayıtların verimli bir şekilde eklenmesini ve silinmesini destekler ve her ikisi de tam-eşleşme sorgularını destekler. Ancak, tam-eşleme sorguları için hashing, B+-ağacından daha verimlidir. Öte yandan, B+-ağacı, aralık sorgularını verimli bir şekilde gerçekleştirebilirken, hashing verimsizdir. Bu nedenle, veri tabanı uygulaması aramaları tam-eşleşme sorgularıyla sınırlandırırsa, hashing tercih edilir. Öte yandan, uygulama aralık sorguları için destek gerektiriyorsa, B+-ağacı tercih edilir. Bu performans sorunlarına rağmen, her iki uygulama da aynı sorunun sürümlerini çözer.

ADT kavramı, bilgi işlem dışı uygulamalarda bile temel konulara odaklanmamıza yardımcı olabilir.

Örnek 5

Araba kullanırken, temel faaliyetler direksiyon çevirme, hızlanma ve frenlemedir. Hemen hemen tüm binek araçlarda direksiyonu çevirerek yön veriyor, gaz pedalına basarak hızlanıyor, fren pedalına basarak fren yapıyorsunuz. Arabalara yönelik bu tasarım, "yönlendirme", "hızlanma" ve "frenleme" işlemlerine sahip bir ADT olarak görülebilir. İki araba, bu işlemleri farklı şekillerde, örneğin farklı motor tipleriyle veya önden arkadan çekişe karşı uygulayabilir. Yine de çoğu sürücü birçok farklı arabayı çalıştırabilir çünkü ADT, sürücünün herhangi bir motor veya sürüş tasarımının özelliklerini anlamasını gerektirmeyen tek tip bir çalışma yöntemi sunar. Bu farklılıklar kasıtlı olarak gizlenmiştir.

ADT kavramı, herhangi bir başarılı bilgisayar bilimcisi tarafından anlaşılması gereken önemli bir ilkedir. Soyutlama yoluyla karmaşıklık yönetilir. Bilgisayar biliminin önemli bir konusu, karmaşıklık ve onu ele alma teknikleridir. İnsanlar, nesneler veya kavramlardan oluşan bir düzeneğe bir etiket atayarak ve ardından düzeneğin yerine etiketi manipüle ederek karmaşıklığı yönetirler. Belirli bir etiket, diğer bilgi parçaları veya diğer etiketlerle ilgili olabilir. Bu koleksiyona da bir kavram ve etiket hiyerarşisi oluşturan bir etiket verilebilir. Bu etiket hiyerarşisi, gereksiz ayrıntıları görmezden gelirken önemli konulara odaklanmamızı sağlar.

Örnek 6

> "Sabit sürücü" etiketini, belirli bir depolama aygıtı türündeki verileri yöneten bir donanım koleksiyonuna tanımlarız ve bilgisayar talimatlarının yürütülmesini kontrol eden donanıma "CPU" etiketini uygularız. Bunlar ve diğer etiketler “bilgisayar” etiketi altında toplanmıştır. Bugün en küçük ev bilgisayarlarında bile milyonlarca bileşen bulunduğundan, bir bilgisayarın nasıl çalıştığını anlamak için bir tür soyutlama gereklidir.

Veri türlerinin hem mantıksal hem de fiziksel bir biçimi vardır. Veri türünün bir ADT açısından tanımı, mantıksal biçimidir. Veri türünün bir veri yapısı olarak uygulanması, fiziksel biçimidir.

1.3 Tasarım Kalıpları

ADT'lerden daha yüksek bir soyutlama düzeyinde, programların tasarımını (yani nesnelerin ve sınıfların etkileşimlerini) tanımlayan soyutlamalar bulunur. Deneyimli yazılım tasarımcıları, yazılım bileşenlerini birleştirmek için kalıpları öğrenir ve yeniden kullanır. Bunlar tasarım kalıpları (design pattern) olarak anılır.

Bir tasarım deseni, yinelenen bir problem için önemli tasarım kavramlarını içerir ve genelleştirir. Tasarım kalıplarının birincil amacı, uzman tasarımcılar tarafından kazanılan bilgileri daha yeni programcılara hızlı bir şekilde aktarmaktır. Diğer bir amaç, programcılar arasında verimli iletişime izin vermektir. Konuyla ilgili teknik bir kelime dağarcığı paylaştığınız zaman, bir tasarım sorununu tartışmak çok daha kolaydır.

Belirli bir tasarım sorununun birçok bağlamda tekrar tekrar ortaya çıktığının farkına varılmasıyla belirli tasarım kalıpları ortaya çıkar. Gerçek sorunları çözmeyi amaçlarlar. Tasarım kalıpları biraz jenerik gibidir. Bir tasarım çözümünün yapısını, verilen herhangi bir problem için doldurulmuş ayrıntılarla tanımlarlar. Tasarım kalıpları biraz veri yapılarına benzer: Her biri maliyet ve fayda sağlar, bu da

ödünleşimlerin mümkün olduğunu gösterir. Bu nedenle, belirli bir tasarım deseni, belirli bir duruma özgü çeşitli ödünleşimleri eşleştirmek için uygulamasında varyasyonlara sahip olabilir.

Bu bölümün geri kalanında, kitapta daha sonra kullanılacak olan birkaç basit tasarım modeli tanıtılmaktadır.

1.3.1 Flyweight

Flyweight (sinek sıklet) tasarım kalıbı bilgi tekrarını engellemeyi hedefler. Birçok nesne içeren bir uygulamanız var diyelim. Bu nesnelerden bazıları içerdikleri bilgi ve oynadıkları rol bakımından aynıdır. Ancak onlara çeşitli yerlerden ulaşılmalıdır ve kavramsal olarak gerçekten ayrı nesnelerdir. Aynı bilgilerin çok fazla kopyası olduğundan, bu alanı paylaşarak bellek maliyetini azaltma fırsatından yararlanmak istiyoruz.

Flyweight kalıp nesne temelli yazılım mimarilerinde belirlenmiş temel tasarım kalıplarından biridir. Bu kalıbın amacı yapıca aynı nesneleri bellekte çokça oluşturmak yerine her bir nesnenin bir kopyasını oluşturmak ve oluşturulan nesneleri ortak bir noktada tutup paylaştırma işlemini yerine getirmektir. Yani tekrar eden aynı nesneleri gruplayarak hafızada çok fazla yer kaplamaması için, hafıza kullanımını minimuma indirmektir. Mesela bir metin dosyası sıkıştırma olayı buna bir örnektir. Aynı metinlerin referansları tutularak yerden kazanç sağlanır.

1.3.2 Visitor

Visitor (ziyaretçi) tasarım kalıbı; sınıflara, sınıfların içerisinde değişiklik yapmadan fonksiyonellik ekleme imkanı sunar.

Uygulamalarda belirli sınıflara, o sınıfların fiziksel yapılarını değiştirmeksizin yeni fonksiyonlar eklemek gerektiği durumlarda kullanılan bir tasarım desenidir. Bazen belirli nesneler üzerinde var olan özelliklerin dışında yeni davranışlara sahip farklı özelliklere ihtiyacımız olabilmektedir.

Yeni özelliğin ekleneceği sınıfların değiştirilmesi istenmediği ya da kaynak kodu elimizde olmayan bir sınıfa yeni bir fonksiyonun eklenmek istendiği durumlarda visitor kalıbı kullanılır.

1.3.3 Composite

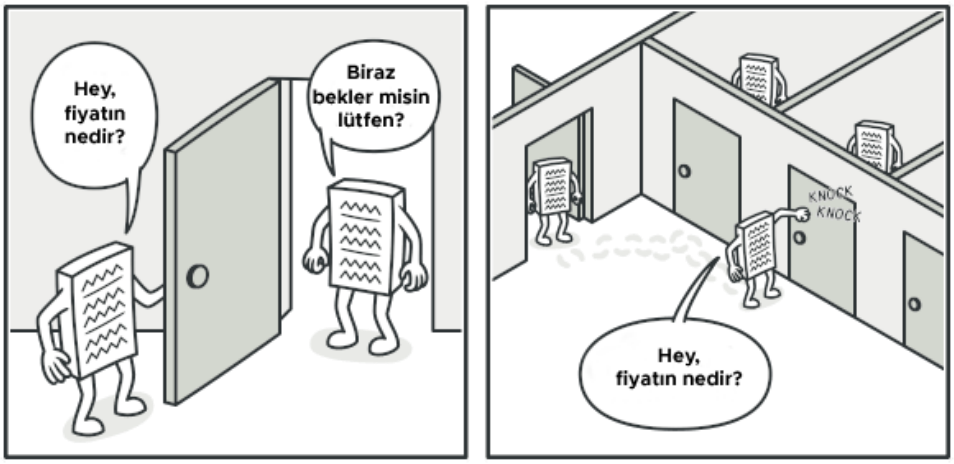
Nesneleri ağaç yapıları halinde oluşturmanıza ve bu ağacın dalları ile tek tek nesnelermiş gibi çalışmanıza olanak veren bir tasarım kalıbıdır.

Örnek olarak Ürünler ve Kutular şeklinde iki tür nesneniz var diyelim. Bir kutu birden fazla ürün içerebilir, hatta daha küçük kutular da içerebilir. Bu kutuların içinde de ürünler ya da kendilerinden daha küçük kutular olabilir. (ve bu böyle gider)

Bu sınıfları kullanan bir sıralama sistemi yapmaya karar verdiğinizi düşünün. Siparişler herhangi bir paketleme yapılmaya gerek olmayan basit ürünler içerdiği gibi, içi birçok ürünle dolu kutulardan da oluşabilir. Böyle bir siparişin toplam fiyatını nasıl belirlersiniz?

Direkt bir yaklaşımla tüm kutuları açabilir, bütün ürünlere bakabilir ve toplamı hesaplayabilirsiniz. Bu gerçek dünyada uygulanabilir bir yöntem fakat bir program için bir döngü çalıştırmak kadar basit değildir. Tarayacağınız kutu ve ürünlerin sınıflarını, kaç seviye aşağıya gidileceğini vb. diğer detayları önceden bilmeniz gerekir. Bu kısıtlar direkt yaklaşımı kullanışsız ve bazen imkânsız hale getirir.

Çözüm: Composite kalıp; ürünler ve kutularla çalışırken, fiyatı hesaplaması için bir metod içeren ortak bir arayüz kullanmanızı önerir. Bir ürün basitçe kendi fiyatını döndürür. Bir kutu kendi içindeki tüm öğelere gidip onlara fiyatlarını sorar (aynı arayüzle) ve toplamı döndürür. Eğer bu öğelerden biri küçük bir kutuysa o da içindekilere fiyatları sorarak toplamı bulacak ve önceki büyük kutuya bu toplamı gönderecektir. Böylece hiç bir kutu daha alt seviyelerde ne olduğunu bilmek zorunda kalmayacaktır.



Bu metodun en avantajlı tarafı ağaçtaki nesnelerin tam sınıflarını bilmeniz gerekmemesidir. Nesnenin tipinden bağımsız olarak composite arayüzdeki metodu içermesi sizin için yeterlidir. Bir nesne basit bir ürün mü, yoksa karmaşık içerikli bir kutu mu önceden bilmeniz gerekmez. Bir metodu çağırdığınızda daha aşağılara inilmesi gerekiyorsa ilgili nesne bunu kendisi yapacaktır.

1.3.4 Strategy

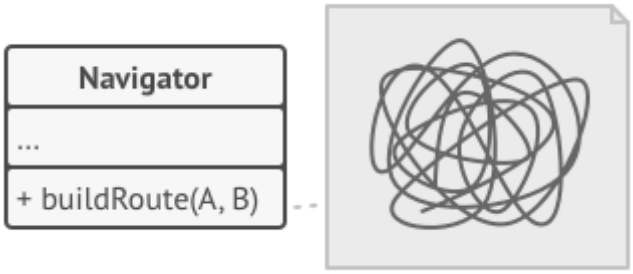
Strategy kalıbı bir algoritma ailesi oluşturup her birini farklı sınıfa yerleştirerek nesnelerini birbiri arasında değişebilir hale getirmeyi sağlayan bir tasarım desenidir.

Sorun: Gezginler için bir navigasyon uygulaması oluşturmaya karar verdiniz. Uygulamayı herhangi bir şehirde, güzel bir harita üzerinde, kullanıcıların yönlerini hızlıca belirlemelerini sağlayacak şekilde inşa ettiniz.

Uygulama için en çok istenen özelliklerden biri de otomatik rota oluşturabilmesi oldu. Bunun için kullanıcı varmak istediği adresi girmeli ve o adrese gitmek için en hızlı rotayı görebilmeli.

Uygulamanın ilk sürümü, yalnızca yollar üzerinde rotalar oluşturabiliyordu. Arabayla seyahat edenler sevinçten havalara uçtu. Ama görünüşe göre, herkes tatilde araba kullanmayı tercih etmiyor. Bir sonraki güncelleme ile yürüyüş rotaları oluşturmak için bir seçenek eklediniz. Hemen ardından insanların güzergâhlarında toplu taşımayı kullanmalarına izin veren bir seçenek daha eklediniz.

Ancak, bu sadece başlangıçtı. Daha sonra bisikletçiler için rota oluşturmayı eklemeyi planladınız. Hatta daha sonra, şehrin tüm turistik mekânları arasında rota oluşturmak için başka bir seçenek eklemeyi düşündünüz.



Uygulama ticari olarak bakıldığında başarılı olsa da teknik kısım birçok baş ağrısına neden oldu. Her yeni rota belirleme algoritması ile ana navigasyon kodunun büyüklüğü iki katına çıktı. Bir yerden sonra bu canavarla baş etmek zor olmaya başladı.

Algoritmalardan birinde yapılacak herhangi bir değişiklik, ister basit bir hata düzeltmesi isterse sokak puanında küçük bir değişiklik olsun, tüm sınıfı etkileyerek hâlihazırda çalışan kodda bir hata oluşturma ihtimalini artırdı.

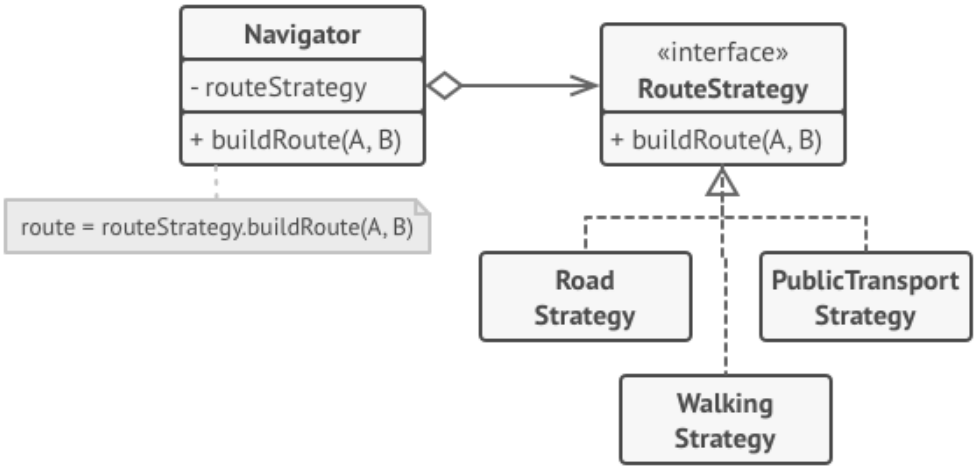
Ayrıca, ekip çalışması verimsiz hale geldi. Başarılı sürümden hemen sonra işe alınan takım arkadaşlarınız, birleştirme çatışmalarını (merge conflict) çözmek için çok fazla zaman harcadıklarından şikâyet ediyorlar. Yeni bir özelliğin eklenmesi, diğerleri tarafından üretilen kodlarla çelişecek şekilde aynı büyük sınıf üzerinde çalışılmasını gerektiriyor.

Çözüm: Strategy kalıbı, spesifik bir işi bir çok farklı yolla yapan bir sınıfı alıp bütün bu algoritmaları strategy adı verilen ayrı sınıflara ayırmanızı öneriyor.

Bu modelde, bağlam (context) adını vereceğimiz orijinal sınıfta, oluşturacağımız bu stratejilere referansları içeren alanlar oluşturulması gerekiyor.

Bağlam adı verilen orijinal sınıf tüm davranışları kendisi uygulamak yerine, her biri kendi durumunu tutan durum nesnelerinin referansını saklar ve durumla alakalı işleri gerektiğinde onlara aktarır.

Yapılacak iş için uygun algoritmayı seçmek bağlamın sorumluluğu değildir. İstemci kod bağlama tercih ettiği stratejiyi gönderir. Aslına bakarsanız bağlam nesnesinin stratejilerin ne yaptığı ile ilgili bir fikri yoktur. Tüm stratejilerle ortak bir arayüz üzerinden tek bir metot ile etkileşime girer ve algoritmanın içeriğini sadece seçilen strateji bilir.



Rota planlama stratejileri

Navigasyon uygulamamızda, her rota algoritması sadece buildRoute (rota oluştur) sınıfı içeren bağımsız bir sınıfa dönüştürülebilir. Metot bir başlangıç ve bitiş konumu alır ve rotayı geri döndürür.

Aynı parametreler ile çağrılsalar bile her rota sınıfı başka bir rota döndürebilir, ana navigasyon uygulaması hangi algoritmanın seçildiği ile ilgilenmez, onun tek görevi geri döndürülen rotayı harita üzerinde çizmektir. Bu sınıfın aktif rota stratejisini değiştirmek için bir metodu vardır, böylece istemciler rota stratejisini bir başkasıyla değiştirebilirler.

2 Matematiksel Giriş

2.2 Kümeler

Matematiksel anlamda küme kavramı, bilgisayar bilimlerinde geniş bir uygulama alanına sahiptir. Küme teorisinin gösterimleri ve teknikleri, algoritmaları tanımlarken ve uygularken yaygın olarak kullanılır, çünkü kümelerle ilişkili soyutlamalar genellikle algoritma tasarımını netleştirmeye ve basitleştirmeye yardımcı olur.

Bir küme, ayırt edilebilir **üyeler** (members) veya **öğeler** (elements) topluluğudur. Üyeler tipik olarak, **temel tip** (base type) olarak bilinen daha büyük bir popülasyondan alınır. Bir kümenin her bir üyesi ya temel tipin bir **ilkel öğesidir** (primitive element) ya da bir kümenin kendisidir. Bir kümede yineleme kavramı yoktur. Temel türden her bir değer ya kümededir ya da kümede değildir. Örneğin, **P** adlı bir küme, 7, 11 ve 42 tamsayılarından oluşabilir. Bu durumda, P'nin üyeleri 7, 11 ve 42'dir ve temel tipi tamsayıdır.

Şekil 2.1, kümeleri ve bunların ilişkilerini ifade etmek için yaygın olarak kullanılan sembolleri gösterir.

{1, 4}	1 ve 4 üyelerinden oluşan bir küme
{x | x is a positive integer}	Bir **küme kalıbı** kullanarak küme tanımı Örnek: tüm pozitif tam sayıların kümesi
$x \in \mathbf{P}$	x, **P** kümesinin bir elemanıdır
$x \notin \mathbf{P}$	x, **P** kümesinin bir elemanı değildir
$\emptyset$	Null yada boş küme
$\lvert\mathbf{P}\rvert$	Nicelik: **P** kümesinin boyutu veya **P** kümesi için üye sayısı
$\mathbf{P} \subseteq \mathbf{Q}$, $\mathbf{Q} \supseteq \mathbf{P}$	**P** kümesi, **Q** kümesine dâhildir, **P** kümesi, **Q** kümesinin bir alt kümesidir, **Q** kümesi, **P** kümesinin bir üst kümesidir
$\mathbf{P} \cup \mathbf{Q}$	Küme birleşimi: **P** VEYA **Q'da** görünen tüm öğeler
$\mathbf{P} \cap \mathbf{Q}$	Küme kesişimi: **P** VE **Q**'da görünen tüm öğeler
$\mathbf{P} - \mathbf{Q}$	Küme farkı: **P**'de olan ve **Q**'da olmayan tüm öğeler

Şekil 2-1. Küme Notasyonu

Aşağıda bu gösterimin kullanımdaki bazı örnekleri görebilirsiniz. İki küme tanımlansın: **P** ve **Q**.

$\mathbf{P} = \{2, 3, 5\}$, $\mathbf{Q} = \{5, 10\}$

|**P**| = 3 (çünkü P'nin üç üyesi vardır) ve |Q| = 2 (çünkü Q'nun iki üyesi vardır). P ∪ Q olarak yazılan küme birleşimi, P veya Q'da olan öğeler {2, 3, 5, 10} kümesidir. P ∩ Q olarak yazılan P ve Q'nun kesişimi, hem P hem de Q'da görünen öğeler {5} kümesidir. P ve Q'nun küme farkı, P − Q olarak yazılır, P'de olan ancak Q'da olmayan, öğeler {2, 3} kümesidir. P ∪ Q = Q ∪ P ve P ∩ Q = Q ∩ P olduğuna dikkat edin, ancak genel olarak P − Q ≠ Q − P. Bu örnekte, Q − P = {10}.

Bir **S** = {a, b, c} kümesinin tüm altkümeleri (S'nin güç kümesi, powerset) şöyledir:

{Ø, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}

Sırasız ögelerden oluşan, yinelenen öğeler içerebilen öğeler koleksiyonuna torba/çanta denir. Çantaları kümelerden ayırt etmek için bir çantanın öğelerinin etrafında köşeli parantez [] kullanılır. Örneğin, [3, 4, 5, 4] torbası, [3, 4, 5] torbasından farklı iken, {3, 4, 5, 4} kümesi {3, 4, 5} kümesinden ayırt edilemez. Ancak [3, 4, 5, 4] torbası, [3, 4, 4, 5] torbasından ayırt edilemez.

Bir **dizi** (sequence), bir sıraya sahip ve yinelenen değerli öğeler içerebilen bir öğeler topluluğudur. Bir diziye bazen **demet**, 2li öge (**tuple**) veya **vektör** de denir. Bir dizide 0. eleman, 1. eleman, 2. eleman vb. vardır. Bir diziyi açılı ayraçlar kullanarak ⟨ ⟩ belirteceğiz. Örneğin ⟨3, 4, 5, 4⟩ bir dizidir. ⟨3, 5, 4, 4⟩ dizisinin ⟨3, 4, 5, 4⟩ dizisinden farklı olduğuna ve her ikisinin de ⟨3, 4, 5⟩ dizisinden farklı olduğuna dikkat edin.

S kümesi üzerinde bir *R* **ilişkisi**, **S**'den alınan sıralı çiftler kümesidir. **S** kümesi {a, b, c} ise:

{⟨a, c⟩, ⟨b, c⟩, ⟨c, b⟩} bir ilişkidir ve {⟨a, a⟩, ⟨a, c⟩, ⟨b, b⟩, ⟨b, c⟩, ⟨c, c⟩} farklı bir ilişkidir.

2.3 Taban ve Tavan

x'in **tabanı** ($\lfloor x \rfloor$ şeklinde yazılır) *x*'in gerçek değerini alır ve ≤*x* olacak şekilde en büyük tamsayıyı döndürür. Örneğin, $\lfloor 3.4 \rfloor = 3$, $\lfloor 3.0 \rfloor = 3$, $\lfloor -3.4 \rfloor = -4$ ve $\lfloor -3.0 \rfloor = -3$ olur. *x*'in tavanı ($\lceil x \rceil$ şeklinde yazılır) *x* gerçek değerini alır ve en küçük ≥ *x* tamsayısını döndürür. Örneğin, $\lceil 3.4 \rceil = 4$, $\lceil 4.0 \rceil$'de olduğu gibi, $\lceil -3.4 \rceil = \lceil -3.0 \rceil = -3$.

2.4 Logaritma

y değeri için *b* tabanının bir **logaritması**, *y*'yi elde etmek için *b*'nin yükseltildiği güçtür. Normalde bu $\log_b y = x$ olarak yazılır. Böylece $\log_b y = x$ ise $b^x = y$ ve $b^{\log_b y} = y$ olur. Logaritmalar programcılar tarafından sıklıkla kullanılır.

Örnek 7

Birçok program, bir nesne grubu için bir kodlama yapar. *n* farklı kod değerini temsil etmek için gereken minimum bit sayısı nedir? Cevap, $\lceil \log_2 n \rceil$ adet bit sayısıdır. Örneğin, depolanacak 1000 kodunuz varsa, 1000 farklı koda sahip olmak için en az $\lceil \log_2 1000 \rceil = 10$ bit gerekir (10 bit, 1024 farklı kod değeri sağlar).

Örnek 8

Küçükten büyüğe sıralanmış bir dizi içinde belirli bir değeri bulmak için ikili arama algoritması kullanılabilir. İkili arama önce ortadaki elemana bakar ve aranan değerin dizinin üst yarısında mı yoksa alt yarısında mı olduğunu belirler. Algoritma daha sonra istenen değer bulunana kadar uygun alt diziyi ikiye bölmeye devam eder. *n* boyutundaki bir dizi, son alt dizide yalnızca bir eleman kalana kadar kaç kez ikiye bölünebilir? Cevap $\lceil \log_2 n \rceil$ kere.

Burada kullanılan hemen hemen tüm logaritmaların tabanı ikidir. Bunun nedeni, veri yapılarının ve algoritmaların çoğu zaman nesneleri yarıya bölmeleri veya kodları ikili bitlerle depolamalarıdır. Burada kullanılan log *n* gösterimleri, ya $\log_2 n$ 'i kastetmektedir ya da terim asimptotik olarak kullanılmaktadır ve bu nedenle gerçek taban önemli değildir.

İkiden farklı bir taban kullanan logaritmalar, tabanı açıkça gösterecektir. Logaritmalar, m, n ve r pozitif değerleri ve pozitif a ve b tamsayıları için aşağıdaki özelliklere sahiptir:

1) $\log(nm) = \log n + \log m$

$\log(n/m) = \log n - \log m$

$\log(n^r) = r \log n$

$\log_a n = \log_b n / \log_b a$

4 nolu özellik için: Teorem:

$$\log_A B = \frac{\log_C B}{\log_C A}; \quad A, B, C > 0,\ A \neq 1$$

İspat:

$$\text{Let } X = \log_C B,\ Y = \log_C A, \text{ and } Z = \log_A B$$

olsun. Logaritma tanımına göre

$$C^X = B,\ C^Y = A, \text{ and } A^Z = B$$
$$C^X = B = (C^Y)^Z.$$

olacağından: $X = YZ$, ve dolayısıyla $Z = X/Y$ olur.

1 nolu özellik için: Teorem:

$$\log AB = \log A + \log B;\ A, B > 0$$

İspat:

$$X = \log A,\ Y = \log B,\quad Z = \log AB.$$

olsun. Varsayılan tabanın 2 olduğu düşünülürse $2^X = A,\ 2^Y = B$ ve $2^Z = AB$ olacaktır. Üç ifade birleştirilirse $2^X 2^Y = AB = 2^Z$ olur ve $X + Y = Z$ elde edilir.

Bazı logaritma değerleri:

$$\log X < X \quad \text{for all } X > 0$$

$$\log 1 = 0, \qquad \log 2 = 1, \qquad \log 1{,}024 = 10, \qquad \log 1{,}048{,}576 = 20$$

2.5 Toplamlar

Çoğu program döngü yapıları içerir. Döngüleri olan programların çalışma süresi maliyetlerini analiz ederken, döngünün her yürütüldüğü zaman için maliyetleri toplamamız gerekir. Bu bir toplama örneğidir. Toplamlar, basitçe, bir dizi parametre değerine uygulanan bazı fonksiyonların maliyetlerinin toplamıdır. Özetler tipik olarak aşağıdaki "Sigma" notasyonu ile yazılır:

$$\sum_{i=1}^{n} f(i)$$

Açık halde $f(1) + f(2) + \cdots + f(n-1) + f(n)$ şeklinde yazılır. Mesela $\sum_{i=1}^{n} i$ ifadesi n tane i toplamı demektir. Bazı sık kullanılan toplam formülleri:

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

Diğer bazı aritmetik seriler de bu formül kullanılarak hesaplanabilir. Mesela $2 + 5 + 8 + \ldots + 3k\text{-}1$ serisini $3(1 + 2 + 3 + \ldots + k) - (1 + 1 + 1 + \ldots + 1)$ şeklinde yazarsak sonuç $3k(k+1)/2\text{-}k$ olacaktır (kısaltırsak: $k\,(3k + 1)/2$ olur). Bu tür problemleri çözmek için diğer bir yol: ilk sıradaki eleman ile son sıradaki eleman eşleştirilir (toplam $3k + 1$), sonra ikinci sıradaki elemanla sondan bir önceki eleman eşleştirilir (toplam $3k + 1$), bu şekilde $k/2$ adet eşleştirme yapılır. Bu şekilde de toplamın $k\,(3k + 1)/2$ bulunacaktır.

Geometrik serilerde sık karşılaşılan bir formül:

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

Mesela *N*=2 kabul edersek $2^0 + 2^1 + 2^2 = 2^3-1$ olacaktır.

2.6 Üslü Sayılar

$$X^A X^B = X^{A+B}$$
$$\frac{X^A}{X^B} = X^{A-B}$$
$$(X^A)^B = X^{AB}$$
$$X^N + X^N = 2X^N \neq X^{2N}$$
$$2^N + 2^N = 2^{N+1}$$