8                **for** j **in** range(10):

9                    pair = Point(screen,i,j)

10                   lst.append(pair)

11

12       lst.sort()

13

14       **for** p **in** lst:

15           **print**(p)

Bölüm 4.4.2'deki kod çağrıldığında, noktaları x eksenine olan uzaklıklarına göre aşağıdaki gibi yazdırır.

(0,0)

(1,0)

(2,0)

(3,0)

(4,0)

(5,0)

(6,0)

(7,0)

(8,0)

(9,0)

(0,1)

(1,1)

(2,1)

(3,1)

(4,1)

(5,1)

(6,1)

(7,1)

 ...

Ancak, bu *sıralama (sort)* yöntemi nasıl çalışır ve maliyeti nedir? Python ne tür bir sıralama algoritması kullanır ve bunun hesaplama karmaşıklığı nedir? Bu soruları sonraki bölümlerde inceleyeceğiz.

**4.5**  **Selection Sort**

Son bölümde, listedeki öğeleri artan sırada sıralamak için bir liste üzerinde *sort* adlı bir yöntemi çağırabileceğimizi öğrendik. Artan sıralama, öğeler üzerinde tanımlanan *less than (<)* operatörü tarafından belirlenir. Peki sıralama algoritması nasıl çalışır? İlk sıralama algoritmalarından birinin adı *Selection Sort'tur* ve sıralama algoritmalarını anlamak için iyi bir başlangıç noktasıdır. Ancak bu Python tarafından kullanılan sıralama algoritması değildir. Nedenini yakında öğreneceğiz.

Seçim sıralama algoritmasını tanımlamak oldukça basittir. Algoritma, listedeki ilk konuma yerleştirilecek en küçük değeri bularak başlar. Bunu liste boyunca doğrusal bir arama yaparak ve yol boyunca bulduğu en küçük öğenin indeksini hatırlayarak yapar. Algoritma, önce en küçük öğenin listedeki ilk öğe olduğunu *tahmin ederek ve* ardından yanlış bir tahmin yapıp yapmadığını görmek için sonraki öğeleri kontrol ederek *tahmin et ve kontrol et* modelini kullanır. Algoritmanın bu kısmı *seçim* kısmıdır. Bu seçimi *select* fonksiyonu yapar.

**4.5.1  Selection Sort' un Seçme İşlevi**

1  **def** select(seq, start):

2      minIndex = start

3

4      **for** j **in** range(start+1, len(seq)):

5          **if** seq[minIndex] > seq[j]**:**

6              minIndex = j

7

8      **return** minIndex

   Başlangıç değişkeni select fonksiyonuna en küçük öğeyi aramaya nereden başlayacağını söyler. En küçük öğe için başlangıçtan dizinin sonuna kadar arama yapar.Seçmeli sıralama (selection sort) algoritması, select fonksiyonunu kullanarak en küçük öğeyi bularak çalışır ve bu öğeyi dizinin ilk konumuna yerleştirir. Şimdi, ilk konumdaki değer başka bir yere taşınmalıdır. Bu değer, taşınan değerin bulunduğu konum ile basitçe yer değiştirir.Algoritma, ardından dizideki ikinci en küçük değeri aramaya devam eder. Artık en küçük değer dizinin ilk konumunda olduğundan, seçmeli sıralama algoritması ikinci konumdan itibaren en küçük değeri aramaya başlar. En küçük değer bulunduğunda (bu aslında listenin ikinci en küçük değeridir), ikinci konumdaki değer ile bu değer yer değiştirir. Daha sonra, algoritma dizideki üçüncü konumdan itibaren en küçük öğeyi aramaya devam eder.Bu desen, dizideki tüm öğeler sıralanana kadar tekrarlanır. selSort fonksiyonu, algoritmanın sıralama işlemini gerçekleştiren kısımdır.

**4.5.2** **Selection Sort Kodu**

1           def selSort(seq) :

2                  for i in range (len(seq)-1):

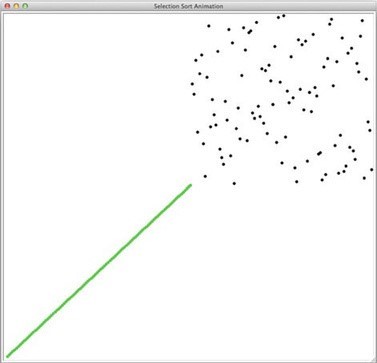
3                         minIndex = select(seq, i)

4                         tmp = seq[i]

5                         seq[i] = seq[minIndex]

6                         seq[minIndex] = tmp

Seçmeli sıralama algoritmasını, sıralama işlemini gerçekleştiren bir animasyonu çalıştırarak görselleştirebiliriz. Şekil 4.3'teki animasyon, bir dizideki değerlerin yarısından fazlası sıralandıktan sonraki durumu göstermektedir. Yeşil noktalar, artık dizide doğru konumlarına yerleştirilmiş olan öğeleri temsil eder. Bir noktanın x ekseninden olan yüksekliği (y değeri), onun değerini gösterir. x ekseni ise listedeki konumu temsil eder. Bu animasyonda, 0 ile 199 arasındaki tüm değerler artan sıraya göre sıralanmaktadır. Üst sağ köşede, henüz sıralanmamış olan değerler bulunmaktadır. Algoritma, bir sonraki en küçük değeri yeşil çapraz çizginin hemen sağında aramaya başlar. Kalan sıralanmamış noktalar arasında x eksenine en yakın olanı (minimum değeri) bulur. Bulunan en küçük ve en kısa noktayı, sol taraftaki nokta ile yer değiştirerek onu doğru konumuna yerleştirir. Bu animasyonun tam kodu Bölüm 20.3'te verilmiştir ve bu metne eşlik eden web sitesinden indirilebilir. Deneyin!

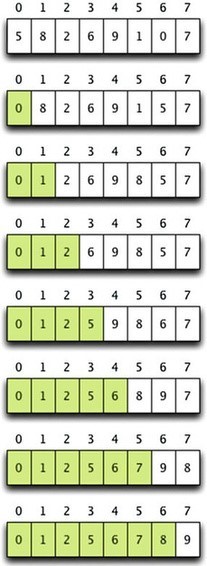


Ş**ekil 4.3** Seçim Sıralaması Anlık Görüntüsü

Şekil 4.4'te gösterildiği gibi [*5 8 2 6 9 1 0 7*] listesini sıraladığımızı düşünelim. *selSort* fonksiyonunun *select* fonksiyonu çağrısından sonra, listenin bir sonraki elemanı son konumuna yerleştirilir. Liste sıralandıkça ara adımlar oluşur. *select* fonksiyonu her çağrıldığında yeni en küçük öğe bulunur ve listenin geri kalanındaki ilk konumla yer değiştirerek, bir sonraki en küçük öğe sıralı liste içindeki yerine taşınır.

Her yeni en küçük öğeyi bulmak için select fonksiyonunu çağırırız. Bu fonksiyon, listenin geri kalanını tarayarak minimum öğeyi aramak zorundadır. Bu algoritmanın zaman karmaşıklığı O(n2 )'dir,  çünkü *selSort* fonksiyonundaki for döngüsü boyunca her seferinde kendi for döngüsü olan *select* fonksiyonunu çağırırız ve **select** fonksiyonunun da kendi içinde bir **for** döngüsü bulunuyor.

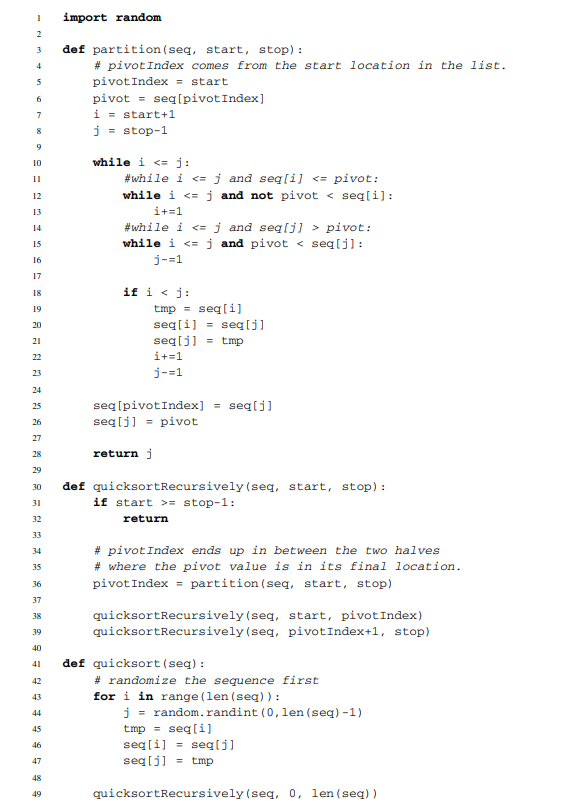
Dıştaki for i döngüsü n kez çalıştırılır ve her çalıştırmada for j döngüsü en küçük değeri bulmak için her seferinde bir öğe daha az olacak şekilde çalışır.İlk çalıştırmada j döngüsü (n - 1) kez çalışır.İkinci çalıştırmada (n - 2) kez çalışır.Üçüncü çalıştırmada (n - 3) kez çalışır ve bu böyle devam eder.Bu daha önce gördüğümüz bir desendir. İlk n tam sayının toplamı formülünde n² terimi bulunduğundan, seçmeli sıralamanın O(n²) olduğu ortaya çıkar. Bu, daha büyük listeleri sıralamaya çalıştıkça algoritmanın gerçekten yavaşlayacağı anlamına gelir. Bu algoritmayı sıralama için asla kullanmamalısınız. Hatta küçük listelerde bile çok daha iyi algoritmalar kullanabiliriz.

  
**Şekil 4.4** Bir Listenin Seçim Sıralaması

**4.6**  **Birleştirme Sıralaması (Merge Sort)**

Eski Romalıların dediği gibi böl ve yönet, etkili bir savaş stratejisidir. Bu kavramın algoritma yazarken de çok önemli olduğu ortaya çıkmıştır. Merge Sortalgoritması, böl ve yönet algoritmasının bir örneğidir. Böl ve yönet algoritmaları genellikle özyinelemeli (recursive) olarak yazılır, ancak öyle olmak zorunda değildir. Temel prensip, bir problemi iki parçaya ayırmaktır. Buiki parçanın her biri, tüm problemi bir kerede çözmeye çalışmaktan daha kolaydır çünkü her biri daha küçük alt problemlerdir.

**4.6.1. Birleştirme Sıralaması Kodu**

  
Birleştirme sıralama (Merge Sort) algoritması, böl ve yönet stratejisini en uç noktaya taşır. Listeyi böler, sonra tekrar böler ve bu işlemi, elimizde boyutu 1 olan listeler kalana kadar devam ettirir. Boyutu 1 olan bir alt liste zaten sıralıdır. İki sıralı alt liste, O(n) zaman karmaşıklığı ile birleştirilerek tek bir sıralı liste haline getirilebilir.Bir liste, sürekli olarak ikiye bölünerek O(log n) zaman içinde boyutu 1 olan listelere ayrılabilir. Daha sonra, bu bölünmüş listeler O(n) zaman içinde birleştirilir. Sonuç olarak, birleştirme sıralama algoritmasının toplam zaman karmaşıklığı O(n log n) olur. Birleştirme sıralama algoritmasının kodu Bölüm 4.6.1'de yer almaktadır.

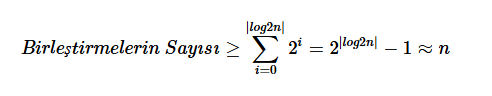
Birleştirme (merge) fonksiyonu, bitişil iki alt listeyi birleştirme işlemini gerçekleştirir. İlk alt liste start indeksinden mid - 1 indeksine kafar uzanır.  İkinci alt liste, mid indeksinden  stop-1 indeksine kadar uzanır. Bu fonksiyon, iki sıralı alt listeyi karşılaştırarak tek bir sıralı listeyi karşılaltırarak tek bir sıralı liste haline gelir. İki sıralı alt listenin elemanları,O(n) zaman karmaşıklığında yeni bir listeye kopyalanır. Daha sonra, sıralı liste tekrar O(n) karmaşıklığında orjinal diziye kopyalanır. Merge fonksiyonunda ilk while döngüsü, iki alt listeyi birleştirerek sıralama işlemini gerçekleştirir. Bu işlem, alt listelerden biri tamamen boşalana kadar devam eder. İkinci ve üçüncü wile döngüleri, artakalan elemanları eklemek için kullanılır. Sadece bir alt listede artakalan elemanlar kalacağı için, ikinci ve üçüncü while döngülerinden yalnızca biri çalışacaktır.

Kodda bulunan üçüncü while döngüsünün yorum satırına alındığını fark edebilirsiniz. Üçüncü while döngüsünde j’den stop’a kadar elemanları kopyalamak gereksizdir, çünkü zaten bu elemanlar, lst’nin içeriği seq dizisine geri kopyalandığında aynı konumlarına yerleştirilecektir. Bu optimizasyon, birleştirme sıralama (merge sort) algoritmasını biraz hızlandırır.Bir diğer optimizasyon ise, değerleri kopyalamak için önceden teşhis edilmiş(pre-allocated) ek bir liste kullanmak ve birleştirme işlemini dönüşümlü olarak orijinal liste ve bu kopya liste arasında gerçekleştirmektedir. Bu yöntem, yeni listeler oluşturma ve onlara eleman ekleme yükünü ortadan kaldırır. Bu iki optimizasyondan herhangi birini uygulamak, algoritmanın asıl hesaplama karmaşıklığını değiştirmez, ancak genel performansını biraz iyileştirebilir.Merge sort algoritmasına yöneltilen eleştirilerden biri, iki alt listenin doğrudan birleştirilememesi ve önce yeni bir listeye kopyalanıp, ardından tekrar orijinal listeye aktarılması gerekliliğidir. Öte yandan, Quicksort gibi diğer sıralama algoritmaları da O(n log n) karmaşıklığına sahiptir, ancak ekstra bir listeye ihtiyaç duymadan çalışabilirler.

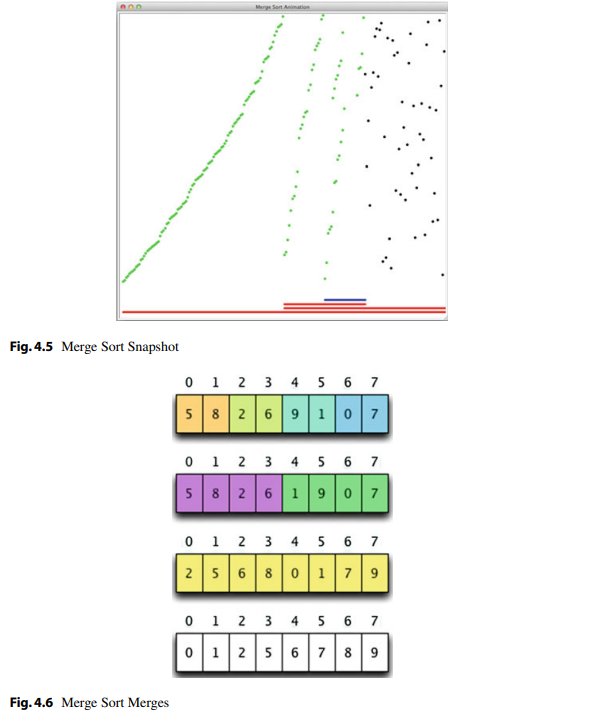
mergeSort fonksiyonu, işlemi başlatmak için bir yardımcı fonksiyon çağırır. mergeSortRecursively fonksiyonunu, sıralanması gereken tüm listeyi temsil eden sequence, start ve stop değerleriyle birlikte çağırır.Start ve stop parametreleri, listeyi bölme işlemi için kullanılır. mergeSortRecursively çağrıldığında liste fiziksel olarak bölünmez. Bunun yerine, start ve stop değerleri kullanılarak bunların arasındaki orta nokta hesaplanır ve ardından iki yarı özyinelemeli (recursive) olarak sıralanır. Her alt liste daha küçük olduğundan, özyinelemeli çağrının görevini yerine getirdiğinden ve iki alt listeyi sıraladığından emin olabiliriz. Daha sonra, merge fonksiyonunu çağırarak bu iki sıralı alt listeyi birleştirmemiz gerekir.Özyinelemeli fonksiyonun temel durumu (base case), alt liste boyutunun 1 olduğu durumdur. Bu noktada, zaten sıralı bir alt listeye sahip oluruz.

Şekil 4.5’te, listenin sol yarısının tamamı sıralanmıştır. Ayrıca, sağ yarıda bulunan üç alt liste sıralanmış ve üçüncü ile dördüncü alt liste birleştirlme aşamasındadır. Yeşil noktalar sıralanmış dizi bölümlerini, siyah noktalar orijinal dizinin sıralanmamış bölümlerini temsil eder.Alttaki kırmızı çizgiler, çalışma zamanında (run-time stack) bulunan özyinelemeli çağrıları(recursive calls) göstreir. Kırmızı çizginin uzunluğu, ilgilii özyinelemeli çağrının sıraladığı dizi bölümünü belirtir. Öaci çizgi, şu anda birleştirlmekte olan iki kat listeyi vurgular.

O (n log n) zaman içinde gelişen ya da devam eden argüman sadece biraz açıklama gerektirir.Listenin tekrarlı olarak bölünmesi O(n logn) bölünmeye neden olur. Sonunda, boyutu 1 olan listeler elde  ederiz. Eğer gerçekleşen tüm birleştirme işlemlerini sayacak olursak en altta n/2 birleştirme, bir üst seviyede n/4 birleştirmeve bu şekilde devam ederek bu toplama ulaşılır.



 Bu analiz, birleştirme sıralama algoritmasının karmaşıklığının O(n^2) olduğunu önermektedir çünkü yaklaşık olarak her biri O(n) olan n birleştirme vardır. Bununla birlikte

  
 algoritma O(n²) değildir. Bunun neden böyle olduğunu görmek için [5, 8, 2, 6, 9, 1, 0, 7] listesini sıralmayı düşünelim. Listeyi tekrar tekrar böldüğümüzde, şekil 4.6’daki ilk listede gösterildiği gibi boyutu 1 olan alt listelere ulaşırız. Bireysel öğeler, ikişerli guruplar halinde birleştirilerek sıralı iki elemanlı listeler oluşturulur. Bu süreç, ikinci listedeki öğelerde gösterilmiştir. En alt seviyede dört birleştirme işlemi gerçekleşir. Her bir birleştirme, iki elemandan(her listeden bir öğe) oluşan listeleri birleştirir. Bu işlemler sonucunda n elemandan oluşan bir liste elde edilir. Dolayısıyla, bu dört dört birleştirme işlemini tek bir grup olarak ele alabiliriz.Sonuç olarak, en derin seviyedeki tüm birleştirme işlemlerinin birleşik süresi O(n)’dir.Burada her bir birleşme O(n) değil, ancak bu seviyedeki tüm birleşmelerin toplamı O(n) zaman alır.Mor renkli kısım bir birleştirme işlemini temsil eder.Yeşil renkli kısım ise ikinci birleştirme işlemine dahil olan öğeleri içerir.Bu iki birleştirme birlikte yine tüm n öğeyi kapsar.   
Dolayısıyla, ikinci en derin seviyede de en fazla n öğe O(n) sürede birleştirilmiş olur.

Son olarak, son birleştirme işlemi, iki sıralı alt listeden gelen sarı renkteki tüm öğeleri kapsar. Bu birleştirme işlemi de O(n) sürede gerçekleşir, çünkü listedeki tüm öğeleri birleştirerek son versiyonda görülen sıralı listeyi oluşturur.

 Bu nedenle, birleştirme işlemi O(n) zaman alırken, birleştirmeler listedeki n öğeden oluşan alt listeler üzerinde yapılır. Bu da demek oluyor ki, her seviyedeki birleştirmeleri O(n) olarak sayabiliriz ve her bir bireysel birleştirme işlemini O(n) olarak saymamıza gerek yoktur.Merge sort algoritmasında log n seviye vardır ve her seviye birleştirme işlemi için O(n) zaman alır. Bu nedenle, algoritma O(n log n) zaman karmaşıklığına sahiptir.

**4.7 HIZLI SIRALAMA (Quicksort)**

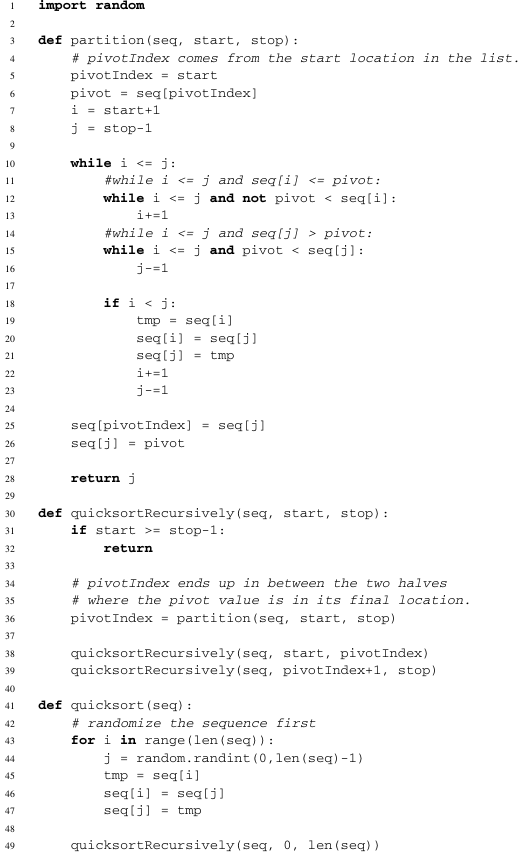
Bir bakıma, quicksort algoritması merge sort algoritmasının tam tersidir. Aynı zamanda en yaygın kullanılan ve bilinen en etkili sıralama algoritmalarından biridir. Quicksort yine böl ve fethet algoritmasıdır ve genellikle rekürsif olarak yazılır. Ancak, birleştirme sıralamasının bir listeyi boyutu 1'e ulaşana kadar böldüğü ve ardından sıralı listeleri birleştirdiği yerde, quicksort algoritması önce birleştirme işlemini yapar ve sonra listeyi böler. Sırasız bir listeyi birleştiremeyiz. Bunun yerine, listeyi iki alt liste olarak bölüyoruz. Hedefimiz, quicksort'un rekürsif olarak çağrılabilmesi için listeyi hazırlamaktır. Ancak, başarılı olmak için iki alt liste orijinal listeden daha kolay sıralanabilir olmalıdır. Bu bölmeye hazırlık, parçalama olarak adlandırılır. Bir listeyi parçalamak için bir pivot elemanı seçeriz. Quicksort'un bir listeyi bir pivot elemanı etrafında, pivottan daha büyük ve daha küçük olan tüm öğeleri içerecek şekilde böldüğünü düşünün. Pivottan daha büyük tüm öğeleri pivotun sağına ve daha küçük tüm öğeleri pivotun soluna yerleştiririz. Bunu yaptıktan sonra, iki şey doğrudur:

• Pivot, listeye son konumunda bulunmaktadır.

• İki alt liste artık daha küçüktür ve dolayısıyla quicksort(hızlı sıralama) uygulanabilir. İki alt liste sıralandığında,  tüm listeyi sıralı hale getir, çünkü solda pivot’a kadar yükselen değerler artan sırada olur, pivot doğru yerde olur ve pivot’tan daha büyük değerler de hepsi doğru konumlarına yerleştirilmiş olur.

Quicksort bir böl ve fethet (divide and conquer)algoritmasıdır. En iyi performansı elde etmek için, diziyi tam ortadan iki eşit alt diziye  bölmek isteriz. Bu, iki listenin ortasını bölmek için kullanılan pivot değerini tam olarak ortaya koymamız gerektiği anlamına gelir çünkü pivot iki listeyi bölmek için kullanılır. Maalesef, bunu verimli bir şekilde yapmak mümkün değildir. Quicksort'un O(n log n) karmaşıklığına sahip olması için, birleştirme sıralaması (merge sort) gibi, listedeki bölme işlemini O(n) zamanında gerçekleştirmesi gerekir. Bu yüzden pivotu hızlı bir şekilde seçmemiz ve soğru bir seçim yapmamız gerekir. Eğer merkeze yakın bir pivot seçmezsek, beklediğimiz O(n log n) karmaşıklığını elde edemeyiz. Ancak rastgele bir pivot seçmenin yeterince iyi olduğu ortaya çıkmıştır. Pivot'un rastgele seçilmesini garanti etmenin bir yolu, quicksort algoritmasının sırayı rastgele hale getirmesiyle başlamasıdır. Quicksort algoritması Bölüm 4.7.1'de verilmiştir.

**4.7.1 The quicksort Code (Hızlı Sıralama Kodu)**

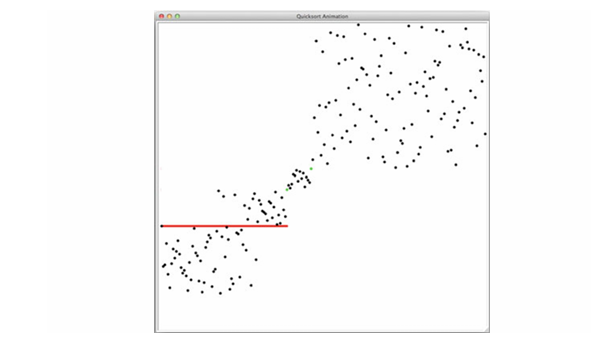


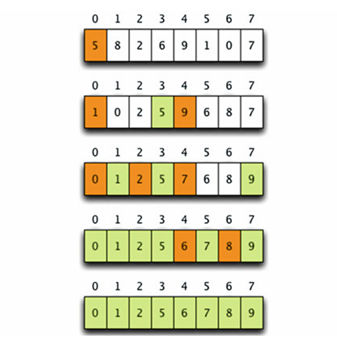
Liste rastgele hale getirildikten sonra, rastgele bir pivot seçmek daha kolay olur. Partiton (bölme) fonksiyonu, dizinin ilk öğesini pivot olarak seçer. Partiton işlemi, listenin sonlarından başlayarak merkeze doğru ilerler. Temel olarak, sol tarafta pivottan büyük bir değer ve sağ tarafta pivottan küçük bir değer bulunduğunda,bu iki değer yer değiştirir.Her iki taraftan da merkeze ulaşıldığında, pivot yerine oturtulur. Dizi bölümlerine ayrıldıktan sonra, hızlı sıralama (quicksort) algoritması iki alt dizi üzerinde özyinelemeli (recusive) oşarak çağrılır. I ve j değişkenleri, bölme işlemi sırasında sol ve sağ değerlerin dizideki indekslerini temsil eder.

Bölme (partition) koduna bakarsanız, yorum satırına alınmış while döngüsü koşullarının, yorum satırına alınmamış koddan daha anlaşılır olduğunu fark edebilirsiniz. Ancak, yorum satırına alınmamış kod sadece küçükten (<) operatörünü kullanır. Hızlı sıralama (quicksort), Python'daki listelerin sort (sıralama) metodunda kullanılan algoritmadır. Bu yöntem yalnızca, dizideki öğeler arasında küçüktür (<) operatörünün tanımlanmasını gerektirir. While döngülerini bu şekilde yazarak, Python’un gerektirdiği gibi yalnızca küçüktür (<) operatörüyle tanımlanan bir sıralama düzenine ihtiyaç duymuş oluruz.

Şekil 4.7’deki anlık görüntü, bir dizide bölmenin (partitioning) etkisini göstermektedir.Bu şekilde, dizi daha önce iki kez bölünmüştür. İlk bölme işlemi, neredeyse tam ortada bir pivot seçmiştir. Ancak, ikinci bölme işlemi o kadar iyi bir pivot seçememiştir. Kırmızı çizgi, şu anda bölme işleminin uygulandığı dizinin bölümünü göstermektedir.Bu alt dizideki en soldaki değerin pivot değeri olduğunu görebilirsiniz. Yeşil noktalar, zaten doğru konumlarına yerleştirilmiş pivot değerleridir.Pivotun sağındaki tüm değerler, pivotun sağındaki bölüme yerleştirilecek; pivotun solundaki tüm değerler ise pivot değerinden küçük olacaktır. İşte bu, hızlı sıralamanın (quicksort) temel çalışma prensibidir. Amortismanlı (amortized) karmaşıklık hesaplaması yapıldığında, quicksort algoritmasının O(n log n) zaman karmaşıklığında çalıştığı görülür.

Quicksort algoritmasını kullanarak [5, 8, 2, 6, 9, 1, 0, 7] listesini sıralamayı düşünelim.Şekil 4.8, her partition (bölme) fonksiyonunun çağrılmasından sonra listenin durumunu göstermektedir. Her çağrıda seçilen pivot, turuncu renkle gösterilmiştir.Partition fonksiyonu, pivotun sağındaki liste bölümünü böler. Bölme işlemi tamamlandıktan sonra, pivot, kendisinden küçük olan son öğeyle yer değiştirilerek nihai konumuna taşınır.Ardından, elde edilen iki alt liste üzerinde tekrar bölme işlemi uygulanır.

  
Şekil 4.7 QuickSort Snapchot (Hızlı Sıralama Anlık Görüntüsü)



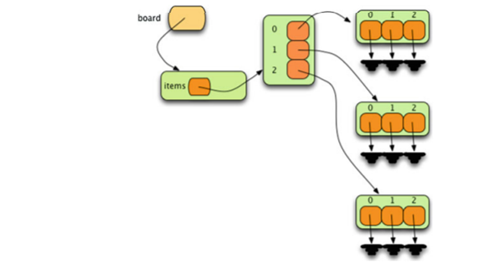
Şekil 4.8: Bir Listeyi Hızlı Sıralama (Quicksorting a List)

Quicksort'un ilk adımında yapılan rastgeleleştirme, daha rastgele bir pivot değeri seçmeye yardımcı olur. Bu, özellikle quicksort algoritmasında gerçek sonuçlara sahiptir, özellikle quicksort'a iletilen dizinin zaten sıralı olma olasılığı varsa. Eğer quicksort'a iletilen dizi sıralı veya neredeyse sıralı ise, artan veya azalan sırada ise, quicksort O(n log n) karmaşıklığına ulaşamaz. Aslında, algoritmanın en kötü durum karmaşıklığı O(n^2) dir. Eğer seçilen pivot, bir sonraki en küçük veya en büyük değer ise, o zaman bölme işlemi, Şekil 4.8'deki alt liste için 9 seçildiğinde olduğu gibi problemi daha küçük alt listelere bölmeyecek. Algoritma sadece bir değeri yerine koyacak ve geri kalan tüm değerlerin büyük bir bölümüyle sonuçlanacak. Eğer her bir pivot seçildiğinde bu olursa, O(n^2) karmaşıklığına yol açar. Listenin quicksort'tan önce rastgeleleştirilmesi, bunun olmamasını sağlamaya yardımcı olacaktır.

Merge sort, bir pivotun seçiminden etkilenmez, çünkü bir seçim yapmak gerekli değildir. Bu nedenle, merge sort'un dikkate alması gereken bir en kötü durumu veya en iyi durumu yoktur. Her zaman O(n log n) karmaşıklığına ulaşacaktır. Bununla birlikte, pratikte quicksort, merge sort'tan daha iyi performans gösterir çünkü quicksort algoritması yeni bir liste kopyalamaya ve sonra tekrar geri kopyalamaya ihtiyaç duymaz. Quicksort algoritması, sıralama algoritmalarının de facto standartıdır.

**4.8 İki Boyutlu Diziler**

Bazen programcılar bir programda iki boyutlu dizileri temsil etmek ihtiyacı duyarlar. Bunun oldukça kolay bir şekilde yapılabilir, bunun için bir liste içinde listeler oluşturulur. Ana liste, matrisin sütunlarını veya satırlarını temsil edebilir. Eğer ana liste,



Şekil 4.9: Bir 2 Boyutlu Matris

Eğer ana liste, matrisin satırlarını içeriyorsa, matris satır sıralı formda denir. Eğer ana liste, matrisin sütunlarını içeriyorsa, o zaman sütun sıralı formda denir. Çoğu zaman, matrisler satır sıralı formda oluşturulur. Şekil 4.9'da bir matris, satır sıralı bir yönelimle çizilmiştir, ancak matris hem satır sıralı hem de sütun sıralı formu temsil edebilir. Verilerin gerçek düzeni her iki durumda da aynıdır. Öğeler ana liste için referans noktalar. Öğeler listesi, matrisin her bir satırına referanslar içerir.

     Örneğin, bir programı düşünelim ki bu program insan bir rakibe karşı üç taş oyunu oynar. Üç taş oyununun oynandığı tahtayı temsil etmemiz gerekecektir. Bunu yapmak için, önceki bölümde olduğu gibi PyList sınıfımızı taklit eden bir Board sınıfı oluşturacağız. Board sınıfımızın organizasyonu Şekil 4.9'da grafiksel olarak gösterilmiştir. Board sınıfının taslağı Bölüm 4.8.1'de verilmiştir.

**4.8.1 Board Sınıfı**

1 class Board:

2          # Bir pano oluşturulduğunda panonun bir kopyasını  çıkarmak isteyebilirsiniz.

3           # Bu , tahtanın yüzeysel bir kopyası olabilir çünkü kaplumbağa nesneleri

4            # bir pano nesnesi açısından değişmez.

5          def \_\_init\_\_(self,board=None):

6                 self.items = []

7                 for i in range(3):

8                       rowlst = []

9                       for j in range(3):

10                           if board==None:

11                                   rowlst.append(Dummy())

12                          else:

13                               rowlst.append(board[i][j])

14

15                      self.items.append(rowlst)

16

17              # Panoya indekslemek için getitem metodu kullanılır. Olması gerekiyor

18             # tahtanın bir sırasını döndür. Bu satırın kendisi indekslenebilir (sadece

19            # bir liste) böylece panodaki bir satıra ve sütuna erişim yazılabilir.

20            # tahta[satır][sütun] bu yöntem nedeniyle.

21           def \_\_getitem\_\_(self,index):

22                  return self.items[index]

24           # Bu yöntem eğer iki kart (self ve other) doğruysa true değerini döndürmelidir.

25          # tam olarak aynı durumu temsil ediyor.

26          def \_\_eq\_\_(self,other):

27                pass

29          # Bu yöntem, bu kartı tüm kuklaları içerecek şekilde değiştirecek

30           # kaplumbağa. Bu şekilde yeni bir oyun başlatıldığında tahta sıfırlanabilir.

31          # seçilir. Başlama dışında KULLANILMAMALIDIR

32           # yeni bir oyun.

33            def reset(self):

34                   screen.tracer(1)

35                   for i in range(3):

36                          for j in range(3):

37                                self.items[i][j].goto(-100,-100)

38                                self.items[i][j] = Dummy()

39

40                 screen.tracer(0)

41

42            # Bu yöntem, değeri temsil eden bir tam sayı döndürmelidir.

43            # kurulun durumu. Bilgisayar kazandıysa 1 değerini döndürün.

44            # Eğer insan kazandıysa -1 değerini döndür. Aksi halde 0 değerini döndür

45            def eval(self):

46                  pass

47

48            # Eğer kartta bu yöntem True değerini döndürmelidir

49            # tamamen dolu (kukla kaplumbağa yok).

50           # Aksi takdirde false değeri döndürmelidir.

51           def full(self):

52                   pass

53

54            # Bu yöntem X ve O'ları çizmelidir

55             # Bu panonun ekranında.

56            def drawXOs(self):

57                     for row in range(3):

58                             for col in range(3):

59                                   if self[row][col].eval() != 0:

60                                     self[row][col].st()

61                                      self[row][col].goto(col\*100+50,row\*100+50)

62

63                     screen.update()