**Blg100/Bölüm 9**

**Sort –Sıralama**

Sıralama(sorting) işlemlerini veritabanında kullanırız.İsim bilgileri harf sırasında,telefon bilgileri alan kodlarına sıralı olarak istenebilir.Programımız bu olanakları sağlamalıdır.Çok gerekli olan bu işlemlerin algoritması karmaşık olabilir.Programınızın sıralama hızıda önemlidir.Örneğin 15 ismin sıralanması dakikalarca sürmemelidir.

**Büyük O Gösterimi**

Bir algoritmanın etkinliğini ölçmek için bilgisayar programcıları “Büyük O Gösterimini”ni tasarlamışlardır.Büyük O,bir algoritmanın yönetmesi gereken bilgi miktarının işlenme hızını ölçer.

Programcılar genellikle aynı miktardaki verinin,farklı algoritmalardaki işlem süresini bilmek isterler.Ortalama(avearge) ve en kötü (worstcase) senaryosu üretirler.”Büyük O” sayesinde program için en uygun algoritma seçilir.

Mesela isimlerin sıralanacağı bir algoritmada,isimlerin sayısı programın hızını etkiler.Bu O(n) ile gösterilir. “O” sıralama büyüklüğü “n” de nesne sayısıdır.”n” boyutundaki bir problemin çözümünde geçen adım sayısı T(n)=4n olarak bulunabilir.

Bilgisayar bilimlerine matematikten miras kalan karmaşıklık teorisi (complexity theory) konularından birisidir. Aslında problem veya algoritmaların çözüme ulaşmadaki karmaşıklığını ölçmek için kullanılır.

Bu tanımın ardında, her problem veya algoritmanın bir fonksiyon gibi düşünülebilmesi, ve matematikteki fonksiyonların karmaşıklığını sınıflandırmak için kullanılan karmaşıklık sınıflarının (complexity classes, asimptotik notasyonlar) , algoritmalar için uygulanması yatmaktadır.

Bilgisayar bilimlerinde kullanılan karmaşıklık sınıfları 5 tanedir.

1. Küçük-o (small-o)
2. Büyük-O (big-o, veya big-oh diye de geçer)
3. Teta (Theta Θ, sadece büyük tetadan bahsedebiliz)
4. Büyük omega (big-Ω )
5. Küçük omega(small-ω )

Bu beş sınıf, aynı fonksiyon için uygulanan ve bir fonksiyonun farklı özelliklerini anlatan sınıftır.

Bu sınıflar, bir algoritma hakkında bilgi edinmemizi sağlar. Bu bilgileri aşağıdaki şekilde sıralayabiliriz:

* Bir algoritmanın ölçeği (Scalability)
* Bir algoritmanın zaman ve hafıza ihtiyacı (hızı ve ne kadar yer kaplayacağı) (time and memory efficiency)

Bu sonuçlara, algoritmanın kullanıldığı giriş bilgisine göre ulaşılır. Bir algoritmanın işlediği veri miktarına göre ne kadar zaman ve ne kadar yer gerektiğini gösterir.

**Örnek**

Örneğin klasik bir arama algoritması olan [doğrusal arama (linear search)](http://www.bilgisayarkavramlari.com/2008/11/09/dogrusal-arama-linear-search/) algoritmasını ele alalım. Bu algoritma verilen bir dizide karışık olarak duran sayılardan bir tanesini bulmayı hedefler.

int a[10] = { 3, 7, 2, 5, 6, 1, 8, -12, 11, 45};

Yukarıdaki dizide bulunan bir sayının kaçıncı sırada olduğunu aramak isteyelim. Örneğin 8 kaçıncı sıradaki sayıdır?

Algoritma, ilk elemandan başlayarak son elemana kadar bütün sayılara bakar ve aranan sayıya rastlarsa sonlanır. Bu durumda aranan 8 sayısını bulmak için sırasıyla 3,7,2,5,6 ve 1 sayılarına bakılacak, nihayet 8 sayısı bulununca 7. Sırada bulunduğu sonucu ile algoritma bitecektir.

Algoritmanın yer ihtiyacı ne kadardır?

Bu algoritmanın üzerinde çalıştığı veri büyüklüğü dizinin büyüklüğüdür ve bu değer 10’dur. Dolayısıyla arama işleminin n sayı içerisinden yapılacağını düşünürsek, n sayıyı hafızada tutmamız gerektiğini görürüz. Demek ki algoritmamızın hafıza karmaşıklığı (memory complexity) n’dir.

Algoritmanın çalışma hızı nedir?

Arama işleminin n adet sayı üzerinde yapıldığını söylersek, aşağıdaki 3 durumu incelememiz gerekir:

1. En kötü ihtimalle kaç adımda bulunur? (worst case)
2. En iyi ihtimalle kaç adımda bulunur? (best case)
3. Ortalama kaç adımda bulunur? (average case)

Bu ihtimal analizini doğrusal arama algoritmamıza uyguladığımızda, karşılaşabileceğimiz en kötü durum aranan sayının dizinin en sonunda bulunması veya dizide hiç bulunmamasıdır. Bu durumda dizideki bütün elemanlara bakılması gerekecektir. Dolayısıyla en kötü durumda karmaşıklığımız n sayı için n olacaktır.

En iyi ihtimal ise, ilk bakılan sayının, aranan sayı olmasıdır. Bu durumda tek bir bakma işlemi yeterlidir. Bu durumdaki karmaşıklığımız ise 1 olacaktır.

Ortalama durum ise bu algoritmanın çok sefer çalışması sonucunda istatistiksel olarak ortalama kaç elemana bakılacağıdır. Bu dizide bulunan sayıların hepsinin aranma oranlarının eşit olduğunu kabul edersek, ortalama durum n/2 olur.

İşte yukarıdaki bu durum analizleri, bizim karmaşıklık sınıflarımızı veren analizlerdir.

O – En kötü durum analizi

Θ – Ortalama durum analizi

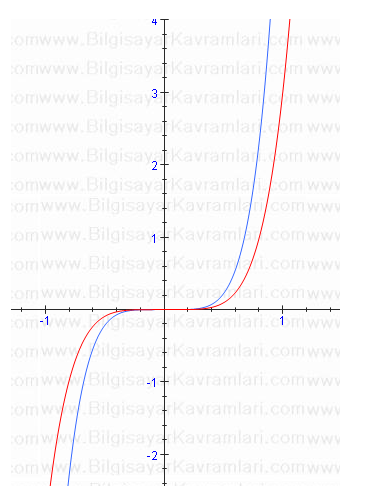
Ω- En iyi durum analizi

Aynı zamanda, bir fonksiyon için de o gösterimi üst sınırı (upper bound), gösterimi ise alt sınırı (lower bound) ifade eder.

Örnek olarak aşağıdaki fonksiyonu ele alacak olursak.

f(x) = 3x5

Bu fonksiyona bir üst sınır çizmek istenirse, örneğin 7x5 fonksiyonu çizilebilir.



Mavi fonksiyon 7x5 ve kırmızı fonksiyon ise f(x) fonksiyonudur.

Yukarıdaki şekilde görüldüğü üzere, x=0 noktasından büyük değerler için yani x>0 için, mavi renkle gösterilen fonksiyon, kırmızı fonksiyonun üst sınırını oluşturmaktadır ve denilebilir ki, kırmızı fonksiyon hiçbir zaman mavi fonksiyonun üzerine geçemez.

Bu durumda mavi fonksiyon, kırmızının üst sınırı (upper bound) olmuş olur. Diğer bir deyişle, mavi fonksiyon, kırmızı fonksiyondan daha kötü bir fonksiyondur ve kırmızı fonksiyon en kötü ihtimalle mavi fonksiyon kadar kötü olabilir.

Bu durumu büyük-o ile gösterirsek:

f(x) ∈ O (x5)

şeklinde yazmamız yeterlidir. Bu gösterimin açılımı aşağıda verilmiştir:

f(x) ∈ O(g(x)) , öyle bir c değeri bulunursa ki, x0>0 için, x>x0 şartıyla, bütün f(x) ≤ c g(x) olsun.

Bu tanım, kısaca, şayet f(x) ∈ O(g(x)) bağlantısı verilirse, herhangi bir x0 noktasından başlanarak sonsuza kadar bütün f(x) ≤ c g(x) koşulunu sağlayan bir c değerinin bulunabileceğini anlatmaktadır.

Yukarıdaki tanımın tam tersi omega için geçerlidir.

f(x) ∈ Ω(g(x)) , öyle bir c değeri bulunursa ki, x0>0 için, x>x0 şartıyla, bütün f(x) ≥ c g(x) olsun.

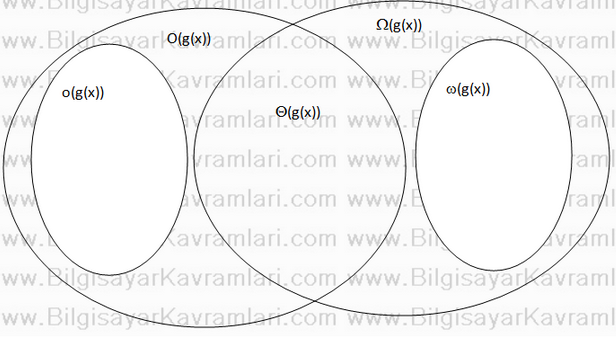
Bu tanım, kısaca, şayet f(x) ∈ Ω(g(x)) bağlantısı verilirse, herhangi bir x0 noktasından başlanarak sonsuza kadar bütün f(x) ≥ c g(x) koşulunu sağlayan bir c değerinin bulunabileceğini anlatmaktadır.

Bu iki tanımda dikkat edilecek bir nokta, f(x) ile c\*g(x) arasındaki bağlantının eşitliği de kapsamasıdır. Yani örneğin big-omega için, c\*g(x) değeri f(x) fonksiyonundan küçük veya eşit olabilir.

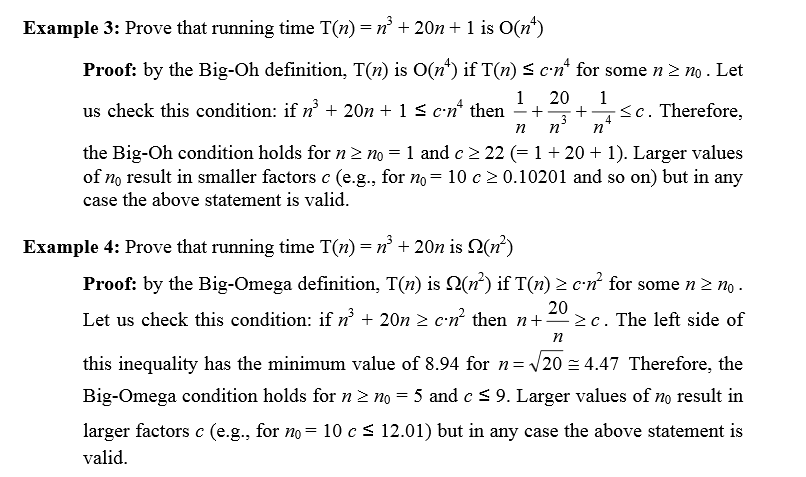
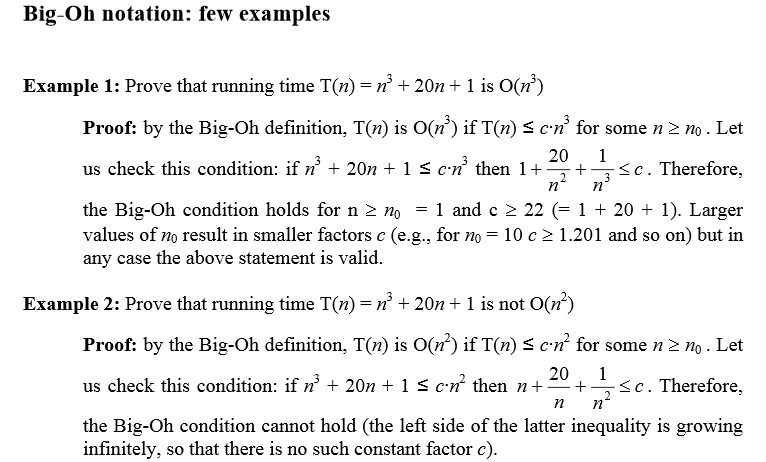
Şayet bu tanımdaki eşitliği kaldırırsak, küçük-o ve küçük-ω tanımını elde ederiz. Yani küçük-o için, c\*g(x) değerinin f(x) değerinden sürekli olarak büyük olması gerekir.

Theta gösterimi ise ortalama durum, yani eşitlik durumu içindir. Burada aranan şart f(x) ∈ Θ(g(x)) için c1\*g(x)≥f(x)≥c2\*g(x)’dir. Yani g(x) fonksiyonunun f(x) fonksiyonu ile aynı şekilde hareket ettiğini göstereniki sabit sayı, c1 ve c2 bulunabiliyorsa f(x) ∈ Θ(g(x)) denilebilir.

Bu durumda yukarıda listelediğimiz 5 sınıf aşağıdaki şekilde ilişkilendirebiliriz:



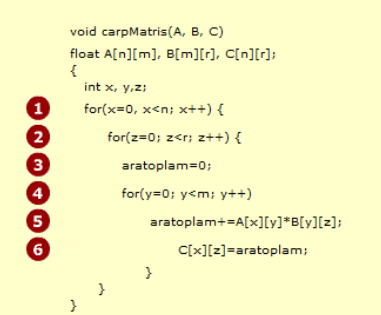
Yukarıdaki şekilde anlatılmak istenen, sınıflar arasında, bazı kesişim durumlarının olduğudur. Görüldüğü üzere, en kötü durum (big-o) ile en iyi durum (big-Ω) kesişmesi sadece eşitlik durumunda olur. Bu durum ise teta ile gösterilmektedir. Benzer şekilde küçük-o ve küçük omega hiçbir zaman kesişemez ve büyük-o ile küçük-o arasında ve büyük omega ile küçük omega arasındaki fark alanında eşitlik durumları bulunur.



Yürütme Zamanı ve Karmaşıklık Hesabı

Örnek:

Aşağıda matris çarpa işlemi yapan bir fonksiyon görülmektedir; A ve B matrisleri giriş C ise sonucun saklandığı matristir. Fonksiyonun n.m boyutlu A matrisiyle n.r boyutlu B matrisini çarpıp n.r boyutlu C matrisini elde etmesi için gerekli T(n)'ni ve büyük O notasyonundaki karmaşıklığı hesaplayınız. Eğer matrisler n.n boyunda kare matrisler olsaydı T(n) ve O(?) ne olurlardı?



Çözüm: Burada üç tane döngü vardır ve sayaç değişkenleri x, y ve z olarak adlandırılmıştır; bağıntılar ise matrisin boyutunu belirleyen n, m ve r'ye bağlı çıkacaktır. Hesaplama yapılırken şöyle bir yol izlenebilir: önce for deyimlerin de yapılan işlem sayısı bulunur ve ardından her for döngüsü içerisinde yapılan işlem sayısı bulunur ve her ikisi toplanır.

1.satırda for deyiminde x=0 işlemi 1 kez; x<n işlemi

(n+1) kez ve x++ işlemi n kez yapılır. Dolayısıyla, en dışdaki for deyimi içerisinde yapılan işlem sayısı 1+(n+1)+n= 2n+2 kez yapılır.

2 .satırdaki for deyiminde ise, işlem sayısı (2r+2)n,

4.satırdaki for deyiminde ise (2m+2)nr çıkarak. Dolayısıyla, yalnızca for deyimleri için yapılan işlem sayısı Tfor(n, m, r)= (2n+2)+(2r+2)n+(2m+2)nr olur. Döngüler içerisindeki işlem sayısı ise,

3.satırdaki işlem sayısı nr kez; benzer şekilde

6.satırda da nr kez yapılır;

5.satırdaki işlem sayısı ise (satırın bütünü 1 işlem kabul edilerek) nmr olur.

Tdöngü içi(n, m, r)= nr+nr+nmr olur. Buna göre toplam işlem sayısı

T(n, m, r)= Tfor(n, m, r)+ Tdöngü içi(n, m, r) bağıntısı uyarınca,

T(n, m, r)= (2n+2)+(2r+2)n+(2m+2)nr+ nr+nr+nmr

= 2n+2+2nr+2n+2mnr+2nr+nr+nr+nmr

= 3nmr+6nr+4n+2

olarak bulunur. Yürütme zamanı bağıntısından O(?) karmaşıklığı,kabaca şöyle bulunur; tüm sabit çarpanlar ve düşük dereceli terimler atılır. Dolayısıyla karmaşıklık O(nmr) bulunur.Eğer matrislerin her yöndeki boyutları n olursa, yürütme zamanı bağıntısındaki m ve r'ler yerine n yerleştirilip sadeleştirme işlemi yapılırsa,

T(n)= 3nnn+6nn+4n+2= 2n3+6n2+4n+2 ve O(n3) bulunur.

**Büyük O notasyonu hesabı**

**Örnek**

Bir algoritmanın yürütme zamanı bağıntısı T(n)= An+Bn+C şeklinde hesaplanmıştır. Burada A, B ve C sabit sayılar olup n ise eleman sayısını göstermektedir. Bu algoritmanın büyük o notasyonuna göre karmaşıklığı ne olur?

**Çözüm:**

Büyük o notasyonunu bulmak için birçok yol vardır. En kolay görüneni aşağıdaki gibi olabilir. Bağıntımız T(n)= An+Bn+C şeklindedir. Sağ taraf n ye bölünürse,



aşağıdaki gibi bir sonuç elde edilir.



Burada 'a giderken birinci terim sabit, ikinci ve üçüncü terimler sıfıra doğru yaklaşır. Dolayısıyla için A sabiti kalır. Dolayısıyla T(n)=An+Bn+C bağıntısı için büyük o notasyonundaki karmaşıklık için O(n) denilebilir. Bu karmaşıklık belirli bir A değeri ve üstü için geçerlidir.

Örneğin ise T(n)= 3n+7n+5, eşitliği sağ tarafı n'ye bölünürse,



elde edilir. Burada A değeri 3'tür.

**En hızlı üç sıralama**

**1.Selecetion Sort(Seçmeli Sıralama)**

**2.Bubble Sort(Elemeli Sıralama)**

**3.Insertion Sort(Eklemeli Sıralama)**

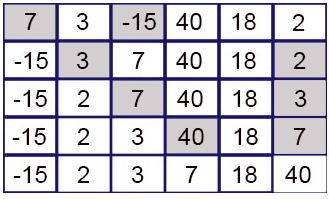
**1.Selecetion Sort(Seçmeli Sıralama)**

Seçmeli sıralama algoritması, küçük boyutlu dizileri sıralarken veya dizinin bir bölümü sıralı ise yer değiştirme işlemi yapılmadığı için tercih edilir. Aksi durumlarda karmaşıklığı O(n2) olduğu için düşük performans gösterir.

Bu algoritma da dizinin ilk elemanı en küçük olarak kabul edilir, sonra dizi içerisindeki en küçük eleman aranır, bulunduğu zaman ilk eleman ile yer değiştirilir; daha sonra kalan elemanlar arasında ikinci en küçük eleman aranır ve ikinci elemanla yer değiştirilir. Bu işlem dizinin son elemanına kadar tekrar edildiğinde dizi küçükten, büyüğe doğru sıralanmış olur.

Selection sort algoritmasının çalışma şekli incelendiğinde dizinin aslında iki alt diziye ayrıldığı birinci bölümün sıralı öğelerden, ikinci bölümün ise sırasız öğelerden oluştuğu görülür. Algoritma çalışmaya devam ettiği sürece sıralı dizinin eleman sayısı artarken, sırasız dizinin eleman sayısı azalır.

Algoritmanın çalışma mantığını A={7, 3, -15, 40, 18, 2} dizisi üzerinde anlatmaya çalışalım. Resimde her adımda yer değiştiren dizi elemanları gri ile boyanarak gösterilmiştir.

****

**Örnek:** 9, 5, 8, 3, 1. rakamlarının azalan şekilde sıralanmasını seçmeli sıralama algoritmasıyla gerçekleştirelim. (Küçük olanı bul)

**Algoritmanın Çalışma Adımları**

Aşağıda seçmeli sıralama algoritmasının python ile yazılmış uygulamasını görebilirsiniz.

Python program

def selectionSort(alist):

for fillslot in range(len(alist)-1,0,-1):

positionOfMax=0

for location in range(1,fillslot+1):

if alist[location]>alist[positionOfMax]:

positionOfMax = location

temp = alist[fillslot]

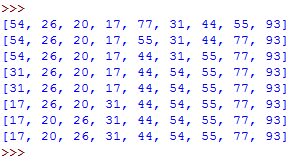
alist[fillslot] = alist[positionOfMax]

alist[positionOfMax] = temp

print(alist)

alist = [54,26,93,17,77,31,44,55,20]

selectionSort(alist)



**2.Bubble Sort(Elemeli Sıralama)**

Sıralama algoritmaları arasında yazılması en kolay olan, ama büyük dizilerde çok yavaş kalan bir sıralama yöntemidir. Zaman karmaşası (time complexity) O(n)dir.

Bu algoritma şöyle çalışır. Verilen dizinin öğelerini, veriliş sırasıyla yatay bir doğru boyunca dizelim.

Verilen dizinin (soldaki) ilk öğesi ikinci öğe ile karşılaştırılır. Bu karşılaştırmada, büyük olan

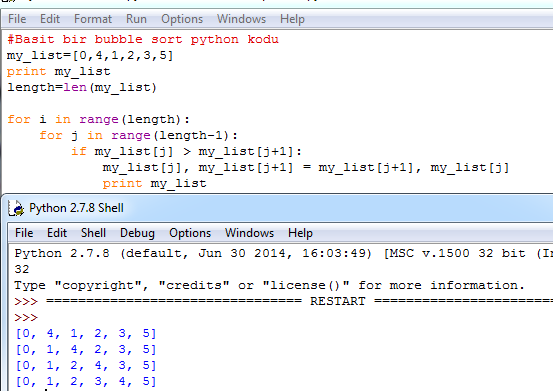
öğe solda kalırsa, sağ yana geçirilmesi için, iki öğenin yerleri değiştirilir (takas, swap). Sonra

büyük olan öğe, üçüncü öğe ile mukayese edilir. Büyük olan sağa geçer ve dördüncü öğe ile

mukayese edilir. Bu işlem verilen dizinin son öğesine gelincene kadar devam eder. Bu geçişin

sonunda, verilen dizinin en büyük terimi, en sağa yerleşmiş olur.

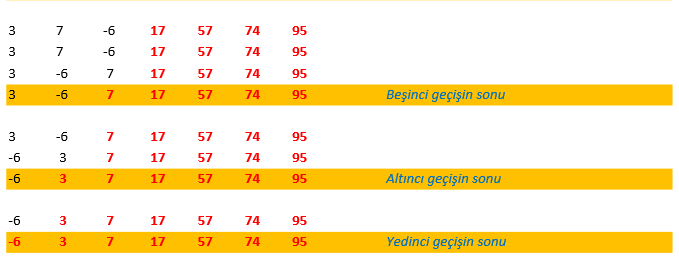
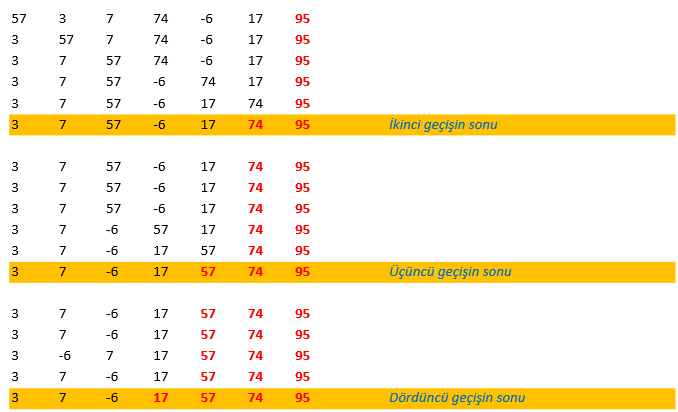
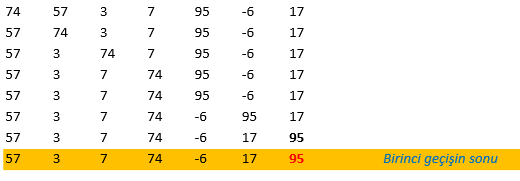
Bubble sort algoritmasını yapan bir python metodu yazalım

****

**Örnek:**

74 57 3 7 95 -6 17

Dizisinin bubble sort algoritması ile sıralayalım. Aşağıdaki tablolarda, dizinin sağında kırmızı ile yazılan alt diziler, sağdan sola doğru azalan sırada sıralanmışlardır.



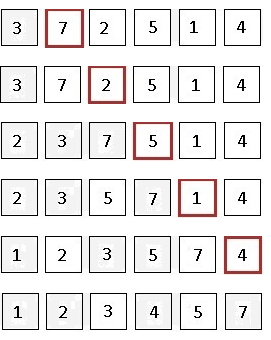
**3.Insertion Sort(Eklemeli Sıralama)**

Bu sıralama Bubble Sort algoritmasının iyileştirilmiş biçimidir. Zaman karmaşası (time complexity) O(n) dir.

Algoritmanın Çalışma Mantığı

Algoritmada, ikinci elemandan başlayarak elemanın kendinden önceki elemanlarla karşılaştırılması suretiyle büyük elemanlar dizide sağa doğru kaydırılır. Böylelikle açılan uygun pozisyona o anda sıralanmakta olan eleman yerleştirilir. Yani, algoritmanın küçükten büyüğe sıralama yaptığı düşünülürse, sayı dizisinin ikinci elemanını kendisine anahtar eleman olarak seçer. Bu anahtar eleman bir önceki elemandan başlayıp, kendinden önceki tüm sayılarla, anahtar olarak seçilen sayıyı kıyaslar. Kendinden büyük olan her sayıyla yerleri değiştirir. Kendinden küçük sayıyla karşılaştığında yer değiştirme işlemi biter. Ardından dizinin son elemanına kadar anahtar sayı seçimi ve devamındaki tüm işlemler devam eder.

Bir sayı dizisinin sıralaması aşağıdaki gibi olur.

****

**Insertion sort algoritmasını yapan bir python metodu yazalım**

def insertion\_sort(items):

""" Implementation of insertion sort """

for i in range(1, len(items)):

j = i

while j > 0 and items[j] < items[j-1]:

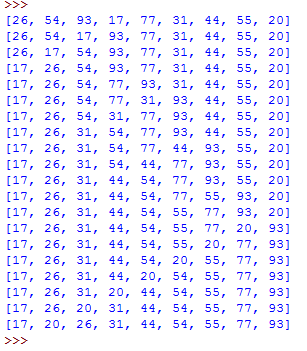
items[j], items[j-1] = items[j-1], items[j]

j -= 1

print(items)

items = [54,26,93,17,77,31,44,55,20]

insertion\_sort(items)



**Referanslar**

[1] <http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html>

[2]<http://ismek.ibb.gov.tr/ismek-el-sanatlari-kurslari/webedition/file/2013_hbo_program_modulleri/temelalgoritmalar.pdf>

[3] <http://www.kodlamamerkezi.com/algoritma/secmeli-siralama-selection-sort-algoritmasi/>

[4]<https://www.cs.auckland.ac.nz/courses/compsci220s1t/lectures/lecturenotes/GG-lectures/BigOhexamples.pdf>

[5]<http://bilgisayarkavramlari.sadievrenseker.com/2010/06/17/karmasiklik-siniflari-complexity-classes/>

[6] <https://0804ce026.files.wordpress.com/2012/12/241012080401026bil34103-r.pdf>

[7]<http://www.baskent.edu.tr/~tkaracay/etudio/ders/prg/dataStructures/sorting/BubbleSort/BubbleSort.pdf>

[8] <https://www.youtube.com/watch?v=HhX_jlBkTD0>

[9]<http://www.baskent.edu.tr/~tkaracay/etudio/ders/prg/dataStructures/sorting/insertionSort/insertionSort.pdf>

[10] http://bidb.itu.edu.tr/seyirdefteri/blog/2013/09/08/insertion-sort-algoritmas%C4%B1

[11]http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html