Algoritmalar ve Karmaşıklık

Ders 11

Algoritma

- Ayrık matematikte karşılaşılan bir çok problem sınıfı mevcuttur. Örneğin,
  - verilen tamsayı grubu içindeki en büyük olanının bulunması,
  - verilen bir kümenin bütün alt kümelerinin listelenmesi,
  - verilen bir tamsayı kümesinin artan sırada sıraya dizilmesi,
  - verilen bir ağ'da iki kenar arasındaki en kısa yol'un bulunması gibi.
- Böyle problemler ile karşılaşıldığında, ilk olarak problemin matematiksel yapısını içeren bir modelinin bulunması gerekir.
- Böyle modellerde, permutasyonlar, bağıntılar, graflar, ağaçlar ve sonlu durumlu makineleri gibi ayrık yapılar sıkça kullanılmaktadır.
- Uygun matematiksel modelin kurulması çözümün ilk adımıdır.
- Modeli kullanarak çözümü gerçekleştiren bir yöntem her zaman gerekli olacaktır.
- Cevabı bulmak için sıralı adımları takip eden bir yordam (procedure) olacaktır.
- Böyle sıralı işlem adımlarına algoritma denir.

Algoritma

- Tanım: *Bir hesaplamayı gerçekleştirmek veya bir problemi çözmek için kesin işlemlerin sonlu bir kümesine **algoritma** denir.*
  - Algoritmadaki her bir adım sonlu zamanda gerçekleştirilebilecek açık ve belirli talimatlar içerir.
  - Her bir adımda çalıştırılacak işlemler açık olarak tanımlanmalıdır.
  - İşlemin, adımların sonlu sayıda çalıştırılması sonucunda sonlandırılması garanti altına alınmalıdır.

- Algoritma kelimesi Türk Matematikçisi olan ve 9. yüzyılda yaşamış olan El-Harezmi'nin isminden türetilmiştir.
- Aslen Türkistan'da Aral gölünün güneyinde bulunan Harzem şehrindedir.
- Algoritma kelimesi Harezmi'nin adından gelen Alkhorizmus=algorisma kelimesinin zamanla farklı dillerdeki kullanımındaki değişiminden türemiştir.

Algoritma

Örnek: Sonlu sayıdaki tamsayılar kümesinin en büyük elemanın bulunması için bir algoritma kurulmaya çalışıldığında:

- Problemin uygulaması çeşitli alanlarda karşımıza çıkabilir:
  - Üniversite öğrencileri arasında derecesi en yüksek olanın belirlenmesi,
  - Bir spor organizasyonunda en yüksek dereceli olan sporcunun belirlenmesi gibi.

- Problemin birçok çözümü olabilir. Bu çözümlerden bir tanesi aşağıda verilmiştir.
  1. Sayıların ilkini geçici en büyük olarak belirle.
  2. Bir sonraki sayı ile geçici en büyük tamsayıyı karşılaştır. Eğer yeni sayı geçici en büyükten büyük ise yeni sayıyı geçici en büyük olarak belirle.
  3. Eğer daha sayı var ise bir önceki adımı tekrarla.
  4. Dizide başka eleman kalmamış ise dur. Bu noktada en büyük sayı geçici en büyük olarak belirlenmiş olan sayı olacaktır.

Pseudocode - Sahtekod

- Algoritma bir bilgisayar dili yardımı ile gösterilebilir. Ancak bunu anlamak çoğu zaman zor olur. Bunun yerine ortak olarak kullanılan ***pseudocode (sahtekod)*** ile ifade edilir.
- Bu kodda anahtar kelimeler İngilizce olarak ifade edilen işlem adımlarını gösterir.

Pseudocode temel bilgileri:

- Bu bölümde sahtekod ile ilgili temel bilgiler(Algoritmaları anlatmak için kullanılan) kısa olarak verilecektir.
- Bir algoritmanın sahtekodu **`procedure (yordam)`** ifadesiyle başlar. Bu algoritmanın ismini, girdi değişkenlerinin listesini ve bunların hangi tip olduklarını tanımlar. Örneğin,

```
Procedure maximum(L: tamsayı listesi, a:integer)
```

ifadesi algoritmanın sahtekod tanımlamasının ilk ifadesidir ve algoritmanın isminin *maximum* ve `L` listesindeki tamsayıların maksimumunu bulur.

Sahtekod

Atama ve Diğer Tipten İfadeler

- Atama için `:=` sembolü kullanılır. Atama ifadesi

```
değişken := ifade
```

formundadır. Örneğin;

```
max := a
x:= L listesindeki en büyük eleman
```

gibi ifadeler atama işlemi için kullanılabilir. Bu ifadeyi gerçek bir programlama diline ifade etmek için birden fazla kod satırı yazılmak zorunda kalınabilir. Burada

```
a ve b değişkenlerinin değerlerini değiştir
```

gibi ifadelerde kullanılabilir.

- Not: Burada bu yazımların kullanılmasının amacı sahtekodu algoritmanın gerçek amacını gösterebilmek için kısa tutmaktır.

Sahtekod

İfade Blokları

- İfadeler (statements) kompleks yordamları bloklar içerisinde gruplamak için kullanılır. Bu bloklar `begin` ve `end` ifadeleri kullanılarak tasarlanır. Bir blok içerisindeki ifadeler aynı miktarda biraz daha içeriden başlatılır.

```
begin
    ifade1
    ifade2
    …
    ifade n
end
```

Blok içerisindeki ifadeler sırasıyla çalıştırılır.

Yorumlar

- Küme parantezleri içerisindeki ifadeler çalıştırılmaz. Böyle ifadeler, yordamın nasıl çalıştığı hakkında bilgi sunan yorum satırları olarak kabul edilir. Örneğin

```
{ x, L listesi içerisindeki en büyük elemandır}
```

ifadesi okuyucuya x değişkeninin L listesinin en büyük elemanı olduğunu hatırlatır.

Sahtekod

Koşullu Yapılar

- En basit koşullu yapı için

```
if koşul then ifade
```

kullanılmaktadır. Blok bir ifadenin çalıştırılması için

```
if koşul then
begin
  ifade bloğu
end
```

- Diğer bir kullanım türü de

```
if koşul then ifade1
else ifade2
```

- Bir diğer türde

```
if koşul_1 then ifade_1
else if koşul_2 then ifade_2
else if koşul_3 then ifade_3
…
else if koşul_n then ifade_n
else ifade_n+1
```

Sahtekod

Döngü Yapıları

- Sahtekod içerisinde kullanılan **for, while do** ve **do until** gibi üç değişik döngü yapısından bahsedilecektir:
- **for** yapısının tek bir ifade ile kullanılan basit formu

```
for değişken:=başlangıç_değeri to son_değeri step artım
  ifade
```

veya blok ifade içeren diğer bir formu

```
for değişken:=başlangıç_değeri to son_değeri step artım
begin
  ifade
end
```

- Örneğin, 1'den n'e kadar tamsayıların toplamını hesaplamak istersek

```
sum:=0
for i:=1 to n
   sum:=sum+i
```


Sahtekod

Döngü Yapıları

- Bir şartın sağlandığı sürece bir veya daha fazla ifadenin çalıştırılabileceği döngüler **while … do** yapısı ile gerçekleştirilir:

```
while koşul [do]
  ifade
```

veya

```
while koşul [do]
begin
    ifade bloğu
end
```

- Örneğin 1'den n'e kadar tamsayıların toplamını hesaplamak istersek

```
sum:=0
k:=1
while k<n [do]
begin
    sum:=sum+k
    k:=k+1
end
```

Sahtekod

Döngü Yapıları

- Bir şart sağlanıncaya kadar bir veya daha fazla ifadenin çalıştırılabileceği döngüler **do … until** yapısı ile gerçekleştirilir.

```
do
begin
    ifade bloğu
end
until koşul
```

- Örneğin 1'den 10'e kadar tamsayıların toplamını hesaplamak istersek

```
sum:=0
n:=1
do
begin
    sum:=sum+n
    n:=n+1
end
until n=10
```


Sahtekod

- Örnek: Liste içerisindeki en büyük tam sayıyı bulma algoritması

```
Procedure enbuyuk(a1, a2, ..., an: integers)
enbuyuk := a1
for i := 2 to n
if enbuyuk < ai then enbuyuk := ai
{işlem sonunda en büyük tamsayı bulunmuş olur}
```

- Başka bir algoritmada bu tamsayıları büyükten küçüğe doğru azalan şekilde sıralayıp ilk elemanı en büyük olarak almak olabilir.

Algoritmaların Özellikleri

- Algoritmalarda aşağıdaki özellikler bulunur ve bu özellikleri akıldan çıkartmamak gereklidir.
  - ***Giriş*** : Belirlenen veri kümesinden algoritma giriş değerleri alır.
  - ***Çıkış*** : Algoritma her bir giriş kümesinde çıkış değerleri üretir. Bu değerler problemin çözümüdür.
  - ***Açıklık*** : Algoritmanın adımları açık olarak tanımlanmalıdır.
  - ***Doğruluk*** : Algoritma her bir giriş kümesi için doğru çıkış üretmelidir.
  - ***Sonluluk*** : Algoritma, her bir giriş kümesi için amaçlanan çıkışı, sonlu işlem adımı(büyük olabilir) sonunda üretmelidir
  - ***Verimlilik*** : Algoritmanın her bir adımı tam ve sonlu bir zaman diliminde gerçeklenmelidir.
  - ***Genellik*** : Yordam formdaki her probleme uygulanabilecek şekilde genel olmalıdır.

- Verilen listedeki en büyük tamsayıyı bulma algoritması bu açıdan değerlendirilirse;
  - Giriş : Sonlu sayıda tamsayı kümesi
  - Çıkış : kümedeki en büyük tamsayı
  - Açık olarak adımlar tanımlanmıştır ve doğru sonuç üretir.
  - Algoritma sonlu işlem adımı kullanır (n adım)
  - Algoritma her bir adımda bir karşılaştırma işlemi yapar (verimlilik).
  - Algoritma bu tür kümelerdeki en büyük tamsayıyı bulacak şekilde geneldir.

Arama Algoritmaları

- Sıralı listedeki bir elemanın yerinin bulunması çok değişik olarak karşılaşılan bir problemdir.
  - Örneğin sözlükten bir kelime aranması gibi problemlere arama problemleri denir.
- Genel arama problemi aşağıdaki şekilde açıklanabilir:
  - Farklı elemanları $a_1, a_2, \ldots, a_n$ olan bir listede bir x elemanın yerinin öğrenilmesi veya listede olup olmadığının öğrenilmesi şeklinde olabilir.
  - Bu arama probleminin çözümü, x elemanına eşit olan a_i elemanın yerinin bulunmasıdır.
  - Eğer $x = a_i$ ise x listenin i. elemandır.

Doğrusal Arama Algoritması

- Problemin çözümü için ilk algoritma doğrusal veya ardışık aramadır.
  - Doğrusal arama algoritması, x ile a_1 karşılaştırarak işleme başlar.
  - Eğer $x = a_1$ ise aranan eleman 1. elemandır.
  - Eğer $x \neq a_1$ ise, x ile a_2 karşılaştırılır.
  - Eğer $x = a_2$ ise çözüm $a_2$nin konumudur.
  - Eğer $x \neq a_2$ ise, x , a_3 ile karşılaştırılır.
  - Bu işlem bir uyuşma bulununcaya kadar devam eder.
  - Uyuşma olmadıkça işlem devam eder.
  - Eğer bir uyuşma bulunamaz ise sonuç sıfır olarak elde edilir.
  - Doğrusal arama algoritmasının sahtekodu aşağıda verilmiştir.

```
Procedure dogrusalarama(x:integer, a1, a2, …, an:farklı tamsayılar)
i:=1
konum:=0
while( i≤n and x≠ai)
   i:=i+1;
if i≤n then konum:=i
else konum:=0
```


İkili Arama Algoritması

- Bu algoritmada verilen veriler artan şekilde sıralanmış olmalıdır.
  - Veriler tamsayı ise en küçükten en büyüğe doğru sıralanmış, eğer kelime iseler alfabetik olarak sıralı şekildedir.
- Böyle bir veri kümesinde bir eleman aranması için algoritma ikili aramadır.
- İkili arama algoritmasının mantığı; sıralı veri kümesi ortadan iki kümeye ayrılarak bulunması istenen veri bu alt kümelerden hangisinin içerisinde olabileceğinin kontrolü prensibine bağlıdır.
- Arama işlemi alt kümelerde tekrarlanarak bulunması gerekli olan veri bulunmaya çalışılır.

Örnek: 1,2,3,5,6,7,8,10,12,13,15,16,18,19,20,22 sıralı dizisi içerisinde 19 tamsayısı aransın.

- Dizide 16 eleman bulunduğundan 1,2,3,5,6,7,8,10 ve 12,13,15,16,18,19,20,22 şeklinde 8'li iki alt kümeye ayrılır.
- 19 tamsayısı birinci alt kümenin en büyük elemanı ile karşılaştırılır. 10 < 19 olduğundan aranan sayı ikinci alt kümededir.
- Bundan sonra ikinci alt küme 12,13,15,16 ve 18,19,20,22 olmak üzere 4 elemanlı iki alt kümeye ayrılır. 16 < 19 olduğundan aranan tamsayı sağ alt kümede olabilecektir.
- Bu nedenle sağ alt küme yine 18,19 ve 20,22 olmak üzere iki elemanlı iki alt kümeye ayrılır.
- Şimdi 19 tamsayısı, son ikili kümenin en büyük elemanından büyük olmadığı için arama ilk kümenin 13. ve 14. elemanını içeren kümeyle sınırlanır.
- Böylece son kümede 18 ve 19 tamsayılı ve birer elemanlı iki alt kümeye ayrılır. 18 < 19 olduğundan arama 19 tamsayısından oluşan son kümeye sınırlanır ve kümenin 14. elemanı olarak bulunur.

İkili Arama Algoritması

- Algoritmanın sahtekodu aşağıda verilmiştir.

```
Procedure ikiliarama(x:integer, a1,a2,...,an: artan tamsayılar)
i:=1; {i, arama aralığının sol bitiş noktasını gösterir}
j:=n; {j, arama aralığının sağ bitiş noktasını gösterir}
while i<j do
begin
   m:=[(i+j)/2];
   if x>am then i:=m+1;
   else j := m;
end
if x=ai then konum:=i
else konum:=0;
{konum, x'e eşit olan terimin indisidir veya eğer x
   bulunamamış ise değeri sıfırdır}
```

Algoritmaların Karmaşıklığı

- Algoritmaların özellikleri içerisinde verimlilik olması gerektiği açıklanmıştı.
- Algoritmanın verimliliği ne demektir? Bunun analizi nasıl yapılır?
- Verimliliğin bir ölçütü, algoritmanın belirli bir giriş verisine karşın, problemin çözümü için bilgisayarın harcadığı zamanın ölçülmesidir.
- Diğer bir ölçü ise belirli giriş verisine karşı bilgisayarın kullandığı bellek miktarıdır.
- Böyle sorular algoritmanın bir hesaplama karmaşıklığının geliştirilmesini gerektirir.
- Problemi çözmek için algoritmanın harcadığı zamanın analizi zaman karmaşıklığını, gerekli belleğin analizi ise yer (space) karmaşıklığının hesabını gerektirir.
- Yer karmaşıklığı probleminin çözümü, algoritmayı gerçeklerken kullanılan veri yapıları ile bağlantılıdır. Ancak bu konular içerisinde yer karmaşıklığından bahsedilmeyecektir.
- Algoritmanın zaman karmaşıklığı ise, belirli miktardaki giriş verisine karşılık, yapılan karşılaştırma, tamsayı toplama, tamsayı çıkartma, tamsayı çarpma ve bölme işlemleri ile diğer basit işlemlerin sayısı olarak hesaplanır.

- Bilgisayarda hesaplamalar genellikle aşağıdaki işlemler yardımıyla gerçekleştirilir:
  - Atama
  - Karşılaştırma (==, ≠, >, <, ≤, ≥)
  - Aritmetik işlemler (+, -, x, /)
  - Mantıksal operasyonlar
- Bu operatörlerin çalıştırılması ne kadar zaman alır?
- Genelde, atama çok hızlı yapılır ve diğer işlemler daha yavaştır. Çarpma ve bölme de toplama ve çıkarmadan daha yavaştır. Kayan noktalarla işlem yapmak, tamsayılarla işlem yapmakta daha yavaştır.
- Çoğu algoritmada bu işlemlerden çok sayıda kullanılmaktadır ve hepsinin ne kadar zaman alacağını teker teker hesaplamak oldukça zor olacaktır. Bunun yerine çoğu algoritmada zaman karmaşıklılığı baskın olan tek bir işlem ile belirlenebilmektedir.

- Algoritma karmaşıklık analizinde dikkat edilmesi gereken yaklaşımlar:
  - Sadece en çok zaman kaybettiren işlemleri hesaba katınız.
  - Hesaplama zamanının girdi boyutuna bağlı olarak değiştiğinde en kötü durumu analiz ediniz.
  - Algoritma girdisinin genelde büyük olduğunu kabul ediniz.
  - Eğer bir algoritmanın zaman karmaşıklılığının artış oranı diğerinin sabit bir katı olduğu durumlardaki iki zaman karmaşıklılığının ayırt etmeye çalışmayınız. Bu durumda her ikisi de aynı kabul edilir.

Örnek (Zaman karmaşıklığı için)

- Bir *A* dizisinin en küçük elemanını bulan algoritmanın zaman karmaşıklığının hesabı:

```
Procedure enkucuk(A real array, enkucuk:real)
enkucuk := A[1];
for i := 2 to n
begin
   if A[i] < enkucuk then enkucuk := A[i]
   {en kötü durumda n-1 defa icra edilir}
end {işlem sonunda en küçük ayı bulunmuş olur}
```

- Bu algoritmanın zaman karmaşıklığı en kötü durumda dizinin büyüklüğü mertebesindedir.
- Bu yordamın işlem sayısını hesaplamaya çalışalım:
- Karşılaştırma işlemlerinin sayısı : *n-1*
- Atama işlemlerinin sayısı: *n-1 ;* Algoritmanın (zaman) karmaşıklığı *O(n)* dir.
- Karmaşıklığı ifade etmek için *O(n)* notasyonu kullanılır. *O* mertebe (order) işareti, *(n)* in sonlu bir çarpanla çarpımından daha küçüktür.
- İşlemlerin sayısı ≤ **k n** { **k** : sınırlı sabit, **n** : problemin büyüklüğü olarak tanımlanır}

Örnekler

- En küçük sayıyı bulma algoritması :

  $n-1$: $O(n)$
- Hem en küçük hem de en büyüğü bulma :

  $n-1 + n-2 = 2n-3$: $O(n)$
- Sıralama yapan algoritma (En küçükten büyüğe):

  $(n-1) + (n-2) + (n-3) + \cdots + 1 = n*(n-1)/2 = n^2/2 - n/2$: $O(n^2)$

Algoritma	Zaman Karmaşıklığı	Çözülebilen En büyük problem			Örnek algoritma
		1 sn.	1 dk.	1 saat	
A1	n	1000	6×10^4	3.6×10^6	En küçüğü bulma
A2	$n \log n$	140	4893	2×10^5	Sıralama(Quicksort)
A3	n^2	31	244	1897	Geleneksel sıralama
A4	n^3	10	39	153	Matris Çarpımı
A5	2^n	9	15	21	Torba doldurma problemi

- Burada dikkat edilmesi gereken ilginç bir nokta, mertebe arttıkça çözebildiğimiz problem sayısı çok çabuk düşmesidir.

Örnek: İkili arama algoritmasının karmaşıklık hesabının yapılması:

Çözüm: Basitlik için $a_1, a_2, .., a_n$ listesinde $n = 2^k$ eleman olduğunu varsayalım ($k>0$).

- Burada $k = \log_2 n$ olacaktır. Eğer kümedeki elemanların sayısı 2'nin katı şeklinde değil ise liste 2^{k+1} elemanlı daha büyük bir liste olacaktır (burada $2^k < n < 2^{k+1}$ gerçeklendiği açıktır).
- Aranan sayı bulununcaya kadar i ve j sayıları birbirine yaklaşır.
- İlk adımda liste 2^{k-1} e sınırlanır. İkinci adımda liste 2^{k-2}'ye sınırlanır. En sonunda liste $2^1 = 2$ elemanlı olarak kalır.
- Listede tek eleman kalınca karşılaştırma başka eleman olmadığını gösterir işlem k+1 adımda biter. İkili arama algoritmasını icra etmek için (her bir adımda 2 karşılaştırma yapıldığından) toplam $2k+2 = 2\log_2 n+2$ karşılaştırma yapılır.
- Buradan ikili arama algoritmasının karmaşıklığının **en kötü durumda** $O(\log_2 n)$ olduğu söylenebilir.

- Diğer bir karmaşıklık analizi ortalama durum analizidir. En kötü durum analizinden daha karmaşık olan bu analiz doğrusal arama algoritmasının karmaşıklık hesabında kullanılmıştır.

Örnek: Aranan elemanın liste içerisinde olduğunu kabul ederek doğrusal arama algoritmasının en ortalama durum analizini yapınız:

Çözüm: *x* in liste içerisinde olduğu n değişik mümkün girdi vardır.

- Eğer *x* listenin birinci terimi ise, üç karşılaştırma gerekir, biri listenin sonuna ulaşıldı mı, diğeri x ile birinci terimi karşılaştır, bir diğeri de döngü dışına çıkılması.
- Eğer listenin ikinci terimi ise ek iki karşılaştırmaya ihtiyaç vardır, dolayısıyla toplam beş karşılaştırmaya ihtiyaç vardır.
- Eğer *x* listenin *i.* terimi ise döngünün her bir i. Adımında iki karşılaştırma yapılmakta ve bir tane de döngü dışında yapılmaktadır. Sonuç olarak 2i+1 karşılaştırmaya ihtiyaç vardır.
- Böylece karşılaştırmaları ortalaması
-

$$\frac{3+5+7+\cdots+(2n+1)}{n} = \frac{2(1+2+3+\cdots+n)+n}{n} = n+2 = O(n)$$

Büyük-O (Big-O) Notasyonu

- *f(n)* ve *g(n)* iki zaman karmaşıklığı olsun. Eğer *n* sayısının yeteri kadar büyük değerleri için $f(n) \leq c\, g(n)$ olacak şekilde bir pozitif *c* sayısı varsa *f(n)* zaman karmaşıklığı *O(g(n))* dir.
- Başka bir deyişle, *n* sayısı yeteri kadar büyük olduğunda, *f(n), g(n)* ile aynı büyüklüktedir.
-