ÖZYİNELEME
RECURSION

Doç. Dr. Aybars UĞUR

Giriş

Kendini doğrudan veya dolaylı olarak çağıran fonksiyonlara özyineli (recursive) fonksiyonlar adı verilir. Özyineleme (recursion), iterasyonun (döngüler, tekrar) yerine geçebilecek çok güçlü bir programlama tekniğidir. Orijinal problemin küçük parçalarını çözmek için, bir alt programın kendi kendini çağırmasını sağlayarak, tekrarlı işlemlerin çözümüne farklı bir bakış açısı getirir. Yeni başlayan programcılara, bir fonksiyon içinde atama deyiminin sağında fonksiyon isminin kullanılmaması gerektiği söylenmekle birlikte, özyineli programlamada fonksiyon ismi doğrudan veya dolaylı olarak fonksiyon içinde kullanılır.

ÖRNEKLER

Örnek 1 (1) : Faktöryel Fonksiyonu

Faktöryel fonksiyonunun matematik ve istatistik alanında önemi büyüktür. Verilen pozitif bir n tamsayısı için n faktöryel, n! Şeklinde gösterilir ve n ile 1 arasındaki tüm tamsayıların çarpımına eşittir.

Örnekler :

0! = 1

1! = 1

5! = 5 * 4 * 3 * 2 * 1 = 120

Faktöryel fonksiyonunun tanımı (definition of factorial function) :

n! = 1 if n==0

n! = n * (n-1) * (n-2) * ... * 1 if n>0

n tamsayısını alıp n faktöryelin değerini döndüren bir algoritmayı şu şekilde oluşturabiliriz

Örnek 1 (2) : Faktöryel Fonksiyonu

```
prod = 1;
for(x=n; x>0; x--)
  prod *= x;
return prod;
```

Böyle bir algoritma tekrarlı (iterative) bir algoritmadır. Çünkü, belirli bir şart gerçekleşinceye kadar süren belirgin bir döngü veya tekrar vardır.

Faktöryel fonksiyonunun diğer bir tanımı :

n! = 1 if n==0

n! = n * (n-1)! if n>0

Bu, faktöryel fonksiyonunun kendi terimleri cinsinden tanımlanmasıdır. Böyle bir tanımlama, bir nesneyi kendi cinsinden daha basit olarak ifade etmesinden dolayı özyineli tanımlama olarak adlandırılır.

Örnek :

3! = 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 * 0!

= 3 * 2 * 1 * 1

= 3 * 2 * 1

= 3 * 2

= 6

n! değerini hesaplayan c fonksiyonu

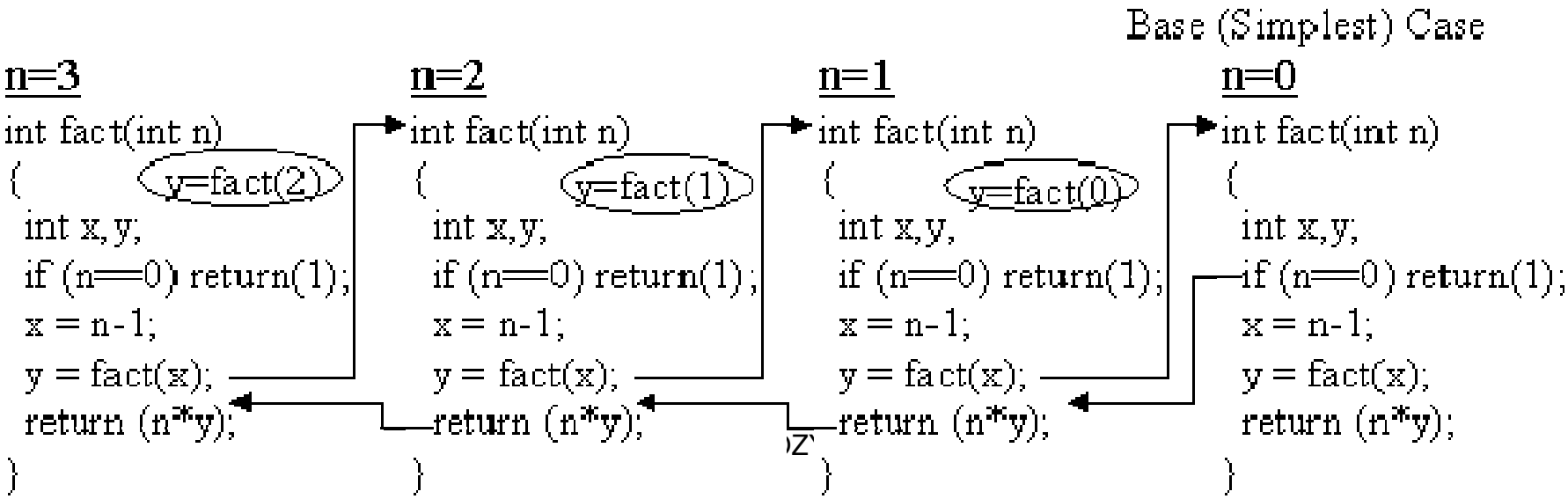
Recursive

```
int fact(int n)
{
  int x,y;
  if (n==0) return(1);
  x = n-1;
  y = fact(x); /*Kendini çağırıyor*/
  return (n*y);
}
```

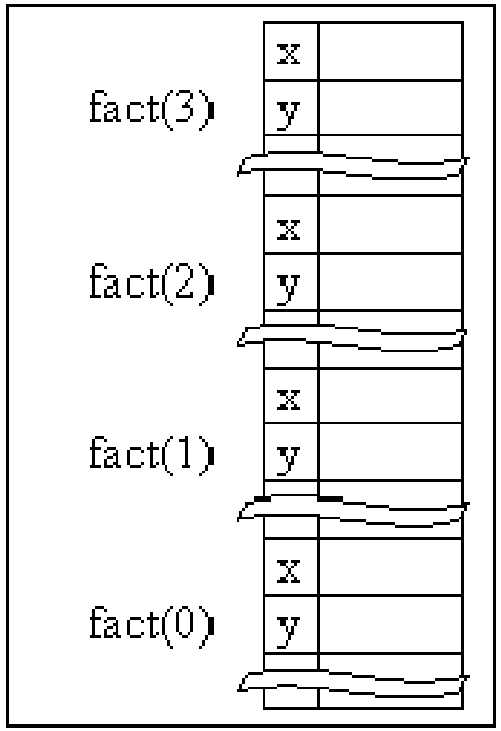
Iterative

```
int fact(int n)
{
  int x, prod;
  prod = 1;
  for(x=n; x>0; x--)
    prod *= x;
  return (prod);
}
```

Bu fonksiyonlar diğer bir fonksiyondan, "printf("%d", fact(3));" şeklinde çağrılabilir.



Bellek Görünümü



Fonksiyon özyineli olarak her çağrılışında yerel değişkenler ve parametreler için bellekten yer ayrılır. Her fonksiyondan çıkışta ise ilgili fonksiyonun (en son çağrılan) değişkenleri için bellekten ayrılan yerler serbest bırakılır ve bir önceki kopya yeniden etkinleştirilir. C bu işlemi yığıt (stack) kullanarak gerçekleştirir. Her fonksiyon çağrılışında fonksiyonun değişkenleri yığıtın en üstüne konulur. Fonksiyondan çıkıldığında ise en son eklenen eleman çıkarılır. Herhangi bir anda fonksiyonların bellekte bulunan kopyalarını parametre değerleri ile birlikte görmek için Debug - Call Stack seçeneği kullanılır (BorlandC 3.0).

n! fonksiyonunun iyileştirilmesi

Özyineli fonksiyonun her çağrılışında bellekte x ve y değişkenleri için yer ayrılması istenmiyorsa fonksiyon kısaltılabilir :

```
int fact(int n)
{
  return ( (n==0) ? 1 : n * fact(n-1) );
}
```

Hata durumları da kontrol edilmelidir. “fact(-1)” gibi bir fonksiyon çağrımında n azaldıkça azalacaktır ve programın sonlandırma koşulu olan “n == 0” durumu oluşmayacaktır. Bir fonksiyon kendisini sonsuza kadar çağırmamalıdır. Bunu engellemek için fonksiyon biraz değiştirilebilir :

```
int fact(int n)
{
  if(n<0)
    { printf(“%s”, “Faktöryel fonksiyonunda negatif parametre!”); exit(1); };
  return ( (n==0) ? 1 : n * fact(n-1) );
}
```

Örnek 2 : Fibonacci Dizisi (Fibonacci Sequence)

Fibonacci dizisi, her elemanı kendinden önceki iki elemanın toplamı şeklinde ifade edilen dizidir :

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
eleman : 0
eleman : 1
eleman : 0+1 = 1
eleman : 1+1 = 2
eleman : 1+2 = 3
eleman : 2+3 = 5
.....

Fibonacci dizisinin tanımı

fib(n) = n if n==0 or n==1

fib(n) = fib(n-2) + fib(n-1) if(n>=2)

Örnek hesaplama :

fib(4) = fib(2) + fib(3)

= fib(0) + fib(1) + fib(1) + fib(2)

= 0 + 1 + 1 + fib(0) + fib(1)

= 2 + 0 + 1

= 3

Fibonacci dizisinin n. elemanının değerini bulan C fonksiyonu

```
int fib(int n)
{
  int x, y;
  if(n<=1) return(n);
  x=fib(n-1);
  y=fib(n-2);
  return(x+y);
}
```

n	x	y	return
3	fib(2)		
2	fib(1)		
1			return(1)->2
2	1	fib(0)	
1			return(0)->2
2	1	0	return(1)->3
3	1	fib(1)	
1			return(1)->3
3	2	1	return(3)->

Örnek 3 : İkili Arama (Binary Search)

Özyineleme sadece matematiksel fonksiyonların tanımlamasında kullanılan başarılı bir araç değildir. Arama gibi hesaplama etkinliklerinde de oldukça kullanışlıdır. Belirli bir sayıdaki eleman içerisinde belli bir elemanı bulmaya çalışma işlemine arama adı verilir. Elemanların sıralanması arama işlemini kolaylaştırır. İkili arama da sıralanmış elemanlar üzerinde gerçekleştirilir. Bir dizinin iki parçaya bölünmesi ve uygun parçada aynı işlemin sürdürülmesi ile yapılan arama işlemidir. Telefon rehberinin ortasını açıp, soyadına göre önce ise ilk yarıda, sonra ise ikinci yarıda aynı işlemi tekrarlama yolu ile arama, telefon rehberindeki baştan itibaren sona doğru sıra ile herkese bakmaktan çok daha etkindir.

İkili Arama Fonksiyonu

```
int binsrch(int a[], int x, int low, int high)
{
  int mid;
  if(low>high)
    return(-1);
  mid=(low+high)/2;
  return(x==a[mid] ? mid : x<a[mid] ?
      binsrch(a,x,low,mid-1) :
      binsrch(a,x,mid+1,high) );
}

int i; int a[8] = { 1,3,4,5,17,18,31,33 };
      // a dizisi : 1  3  4  5  17  18  31  33

i = binsrch(a,17,0,7);
      // 0 ve 7. elemanlar arasında 17
      sayısının aratılmasını sağlar.
```

```
binsrch(a,17,0,7);
mid = (0+7)/2 = 3. eleman
a[mid] = 5 => 17>5
binsrch(a,17,4,7);
mid = (4+7)/2 = 5.eleman
a[mid] = 18 => 17<18
binsrch(a,17,4,4);
mid = (4+4)/2 = 4. eleman
a[mid] = 17 => 17==17
return(4); // Bulundu
```

Global değişkenli binsrch fonksiyonu

a ve x global değişken olarak tanımlanırsa fonksiyon ve çağrımı şu şekilde olacaktır :

```
int binsrch(int low, int high)
{
  int mid;
  if(low>high)
    return(-1);
  mid=(low+high)/2;
  return(x==a[mid] ? mid : x<a[mid]?binsrch(low,mid-1):binsrch(mid+1,high) );
}
... i = binsrch(0,n-1);
```

Örnek 4 : Hanoi Kuleleri Problemi (Towers of Hanoi Problem)

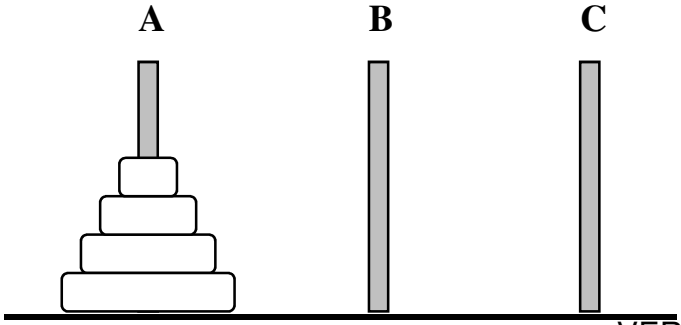
Üç kule (A,B,C) olan bir sistemde yarıçapı birbirinden farklı 4 tane diskin A kulesine yerleştirildiğini düşünün.

Kurallar :

• Bir diskin altında yarıçapı daha küçük bir disk bulunamaz. Bir diskin üzerine yarıçapı daha büyük bir disk yerleştirilemez.

• Bir anda bir kulenin sadece en üstündeki disk diğer bir kuleye yerleştirilebilir. Bir anda bir disk hareket ettirilebilir.

Hanoi Kulesinin ilk yerleşimi :



Problemde istenen : B direğinden de yararlanarak tüm disklerin C'ye yerleştirilmesi.

Çözüm Mantığı

Önce n disk için düşünelim.

- 1 disk olursa, doğrudan A'dan C'ye konulabilir (Doğrudan görülüyor).
- (n-1) disk cinsinden çözüm üretebilirsek özyinelemeden yararlanarak n disk için de çözümü bulabiliriz.
- 4 diskli bir sistemde, kurallara göre en üstteki 3 diski B'ye yerleştirebilirsek çözüm kolaylaşır.

Genelleştirirsek :

1. n==1 => A'dan C'ye diski koy ve bitir.
2. En üstteki n-1 diski A'dan C'den yararlanarak B'ye aktar.
3. Kalan diski A'dan C'ye koy.
4. Kalan n-1 diski A'dan yararlanarak B'den C'ye koy.

Girdi/Çıktı Tasarımı

Problem deyimi : “Hanoi Probleminin Çözümü”
Problem şu an tam olarak tanımlı değil, problem deyimi yeterli değil.
Bir diskin bir kuleden diğerine taşınmasını bilgisayarda nasıl temsil edeceğiz?
Programın Girdileri nelerdir? Çıktıları nelerdir? belli değil.

Girdi/Çıktı tasarımı herhangi bir problemin çözümünün programın algoritmasının oluşturulmasında önemli yer tutar. Oluşturulacak algoritmanın yapısı da büyük ölçüde girdi ve çıktılara bağlı olacaktır. Uygun girdi ve çıktı tasarımı, algoritmaların geliştirilmesini kolaylaştırabilir ve etkinlik sağlar.

Girdi :

disk sayısı : n
kuleler : A, B, C (uygun)

Çıktı :

disk nnn’i, yyy kulesinden alıp zzz kulesine yerleştir.
nnn : disk numarası. En küçük disk 1 numara (en küçük sayı olması doğrudan)
yyy ve zzz de kule adı.

Towers fonksiyonunun çağrılması

Ana programdan towers fonksiyonunun çağrılması :

```
void main()
{
  int n;
  scanf("%d",&n);
  towers(parameters);
}
```

Şimdi parametreleri belirleme aşamasına gelindi :

```
#include <stdio.h>
void towers(int, char, char, char);
void main()
{
  int n;
  scanf("%d",&n);
  towers(n,'A','C','B');
}
```

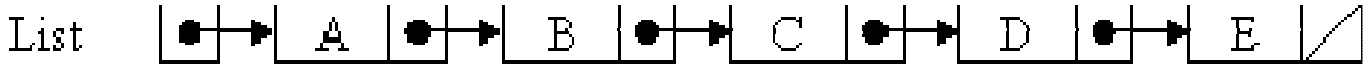
Towers metodu

```
void towers(int n, char frompeg, char topeg, char auxpeg)
{
  if(n==1) {
    printf("\n%d%s%c%s%c%s",n,". diski ",frompeg," kulesinden alıp ",
                                    topeg, " kulesine yerleştir");
    return;
  };
  towers(n-1, frompeg,auxpeg,topeg); // n-1 diskin yardımcı kuleye konulması
  printf("\n%d%s%c%s%c%s",n,". diski ",frompeg," kulesinden alıp ",
                                    topeg, " kulesine yerleştir");
  towers(n-1, auxpeg,topeg,frompeg); // n-1 diskin hedef kuleye konulması
}
```

Programın n=3 disk için çalıştırılması sonucu oluşan ekran çıktısı

1. diski A kulesinden alıp C kulesine yerleştir
2. diski A kulesinden alıp B kulesine yerleştir
1. diski C kulesinden alıp B kulesine yerleştir
3. diski A kulesinden alıp C kulesine yerleştir
1. diski B kulesinden alıp A kulesine yerleştir
2. diski B kulesinden alıp C kulesine yerleştir
1. diski A kulesinden alıp C kulesine yerleştir

Tek Bağlı Listede Elemanların Ters Sırada Listelenmesinde Özyineleme




```
Procedure RevPrint (list:ListType)
begin
  if list<>NIL
    then
      begin
        RevPrint(List^.Next);
        write(list^.info)
      end;
end;
```

Ekran Çıktısı : E D C B A

Elemanları bir yığıta koyup ters sırada listelemek yerine doğrudan özyineleme kullanmak daha basit ve doğal.

Özyineleme Zinciri

Özyineli bir fonksiyonun kendisini doğrudan çağırması gerekmez. Dolaylı olarak da çağırabilir. Örnek bir yapı şu şekildedir :

```
a(parametreler)
{
.....
  b(değerler);
.....
}
```

```
b(parametreler)
{
.....
  a(değerler);
.....
}
```

Özyineleme (Recursion) ve İterasyon (Iteration)

Herhangi bir fonksiyonun iteratif (iterative) yani tekrarlı versiyonu, özyineli (recursive) versiyonundan zaman (time) ve yer (space) bakımından genelde daha etkindir. Bunun nedeni, özyinelemede fonksiyonun her çağrılışında fonksiyona giriş ve çıkışta oluşan yüklerdir.

Bununla birlikte genelde yapısı uygun olan problemlerin çözümünde özyinelemenin kullanılması daha doğal ve mantıklıdır. Tanımlamalardan çözüme doğrudan ulaşılabilir. Yığıt kullanımı gerektiren durumlarda özyineli olmayan çözümlerin geliştirilmesi zordur ve hataları gidermek için daha fazla çaba harcamak gerekir. Örnek olarak, tek bağlı listedeki elemanların ters sırada yazdırılması verilebilir. Özyineli çözümde yığıt otomatik olarak oluşmaktadır ve ayrıca yığıt kullanmaya gerek kalmaz.

Özyineleme (Recursion) ve İterasyon (Iteration)

İterasyonda "Control Structure" olarak döngüler yolu ile tekrar kullanılırken, özyinelemede seçim yapısı kullanılır. İterasyonda tekrar, doğrudan sağlanırken, özyinelemede sürekli fonksiyon çağrıları ile sağlanır. İterasyon, döngü durum şartı geçersizliğinde sonlanırken, özyineleme en basit duruma (simplest case = base case) ulaşıldığında sonlanır. İterasyonda kullanılan sayaç değeri değiştirilerek problem çözülürken, özyinelemede orijinal problemin daha basit sürümleri oluşturularak çözüme ulaşılır.

```
import java.io.*;
import java.util.Scanner;

class FaktoryelOrnek
{
 static int sayi;

 public static void main(String args[]) throws IOException
 {
  System.out.println("Sayi veriniz :");

  Scanner str = new Scanner(System.in);
  int sayi = str.nextInt();

  int sonuc = factorial(sayi);
  System.out.println(sayi+"! ="+sonuc);
 }

 public static int factorial(int n)
 {
  if(n==0)
   return 1;
  else
   return(n*factorial(n-1));
 }
}
```

Verilen bir sayının faktöryelini bulan metodu içeren Java Programı

Ekran Çıktısı

```
Sayi veriniz :
3
3! =6
```