5.1 Özyinelemeli Fonksiyonlar

Özyinelemeli (*rekürsif*) fonksiyonlar kendi kendini çağıran fonksiyonlardır. Rekürsif olmayan fonksiyonlar iteratif olarak adlandırılırlar. Bunların içerisinde genellikle döngüler (`for`, `while`… gibi) kullanılır.

Bir fonksiyon ya iteratiftir ya da özyinelemelidir. Rekürsif fonksiyonlar, çok büyük problemleri çözmek için o problemi aynı forma sahip daha alt problemlere bölerek çözme tekniğidir. Fakat her problem rekürsif bir çözüme uygun değildir. Problemin doğası ona el vermeyebilir. Rekürsif bir çözüm elde etmek için gerekli olan iki adet strateji şu şekildedir:

1. Kolayca çözülebilen bir temel durum (base case) tanımlamak, buna çıkış (exitcase) durumu da denir.
2. Yukarıdaki tanımı da içeren problemi aynı formda küçük alt problemlere parçalayan recursive case'dir.

Rekürsif fonksiyonlar kendilerini çağırırlar. Recursive case kısmında problem daha alt parçalara bölünecek ve bölündükten sonra hatta bölünürken kendi kendini çağıracaktır. Temel durumda ise çözüm aşikârdır.

5.1.1 Rekürsif Bir Fonksiyonun Genel Yapısı

Her rekürsif fonksiyon mutlaka if segmenti içermelidir. Eğer içermezse rekürsif durumla base durumu ayırt edilemez Bu segmentin içerisinde de bir base-case şartı olmalıdır. Eğer base-case durumu sağlanıyorsa sonuç rekürsif bir uygulama olmaksızın iteratif (özyinelemesiz) olarak hesaplanır. Şayet base-case şartı sağlanmıyorsa else kısmında problem aynı formda daha küçük problemlere bölünür (bazı durumlarda alt parçalara bölünemeyebilir) ve rekürsif (özyinelemeli) olarak problem çözülür.

Bir fonksiyonun özyinelemeli olması, o fonksiyonun daha az maliyetli olduğu anlamına gelmez. Bazen iteratif fonksiyonlar daha hızlı ve belleği daha az kullanarak çalışabilirler.

```
if(base case şartı)
     özyinelemesiz (iteratif olarak) hesapla
else { /* recursive case
     Aynı forma sahip alt problemlere böl
     Alt problemleri rekürsif olarak çöz
     Küçük çözümleri birleştirerek ana problemi çöz */
```

```
}
```

Örnek olarak Faktöriyel problemi verilebilir.

```
4! = 4.3.2.1 = 24
4! = 4.3! // Görüldüğü gibi aynı forma sahip daha küçük probleme
3! = 3.2! // böldük. 4'ten 3'e düşürdük ve 3! şeklinde yazdık.
2! = 2.1! // Geri kalanları da aynı şekilde alt problemlere
1! = 1.0! // böldük. 0! ya da 1! base-case durumuna yazılabilir.
0! = 1
```

Yukarıdaki problemi sadece nasıl çözeriz şeklinde düşünmeyip, genel yapısını düşünmemiz gerekir. Matematiksel olarak düşünürsek problem $n(n-1)!$ şeklindedir. Yapıyı da düşünürsek,

$$n! = \begin{cases} 1 & eğer\ n = 0 \\ n.(n-1)! & eğer\ n > 0 \end{cases}$$

şeklinde olacaktır. Bu temel bir durum içerir: eğer $n = 0$ ise sonuç 1'dir, değilse $n.(n-1)!$ olacaktır. Şimdi bunu koda dökelim;

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

Fonksiyona 4 rakamını gönderdiğimizi düşünelim. 4 sıfıra eşit olmadığından fonksiyon else kısmına gidecek ve `4*fact(3)` şekliyle kendini 3 rakamıyla tekrar çağıracaktır. Bu adımlar aşağıda gösterilmiştir;

4 * fact(3)
↓
3 * fact(2)
↓
2 * fact(1)
↓
1 * fact(0)
↓
(1)

Şekilde görüldüğü gibi en alttaki fonksiyon çağrısında iteratif durum gerçekleştiği için fonksiyon sonlanacak ve ilk çağrıldığı noktaya geri dönecektir. Bu esnada bellekte fact(0) çağrısı yerine 1 yazılacağı için o satırdaki durum 1*1 şeklini alacak ve 1 sonucu üretilecektir. Bu sefer fact(1) çağrısının yapıldığı yere dönecek ve yine 2*1 şeklini alıp 2 üretecektir. Sonra fact(3) çağrısının olduğu satırda 3*2 şeklini alıp 6 ve fact(4) çağrısının yapıldığı satıra döndükten sonra da 4*6 hesaplanarak 24 sayısı üretilecektir.

return 4 * (return 3 * (return 2 * (return 1 * 1)))
return 4 * (return 3 * (return 2 * 1))
return 4 * (return 3 * 2)
return 4 * 6
return 24

İşi biten fonksiyon bellekten silinir.

Aşağıda vereceğimiz kod örneklerini siz de kendi bilgisayarınızda derleyiniz. Çıktıyı çalıştırmadan önce tahmin etmeye çalışınız. Konuyu daha iyi anlamanız için verilen örneklerdeki kodlarda yapacağınız ufak değişikliklerle farkı gözlemleyiniz.

Örnek 10 Rekürsif bir `hesapla` isimli fonksiyon tanımı yapılıyor. `main()` içerisinde de 1 rakamıyla fonksiyon çağrılıyor.

```
void hesapla(int x) {
     printf("%d", x);
     if(x < 9)
          hesapla(x + 1);
     printf("%d", x);
}
main() {
     hesapla(1);
     return 0;
}
```

Fonksiyonun çıktısına dikkat ediniz.

```
123456789987654321
```

Örnek 11 Klavyeden girilen n değerine kadar olan sayıların toplamını hesaplayan rekürsif fonksiyon:

```
int sum(int n) {
     if(n == 1)
          return 1;
```

```
    else
        return n + sum(n - 1);
}
```

Örnek 12 Fibonacci dizisi, her sayının kendinden öncekiyle toplanması sonucu oluşan bir sayı dizisidir. Aşağıda klavyeden girilecek n değerine kadar olan fibonacci dizisini rekürsif olarak hesaplayan fonksiyon görülüyor. Çıktıya dikkat ediniz.

```
int fibonacci(int n) {
    if(n == 0)
        return 0;
    else if(n == 1)
        return 1;
    else
        return (fibonacci(n - 1) + fibonacci(n - 2));
}
```

Örnek 13 Girilen 2 sayının en büyük ortak bölenini hesaplayan rekürsif fonksiyon alttaki gibi yazılabilir.

```
int ebob(int m, int n) {
    if((m % n) == 0)
        return n;
    else
        return ebob(n, m % n);
}
```

5.2 Yapılar

struct: Birbirleri ile ilgili birçok veriyi tek bir isim altında toplamak için bir yoldur. Örneğin programlama dillerinde reel sayılar için `double`, tamsayılar için `int` yapısı tanımlıyken, karmaşık sayılar için böyle bir ifade yoktur. Bu yapıyı `struct` ile oluşturmak mümkündür.

Örnek 14 Bir *z* karmaşık sayısını ele alalım.

```
z = x + iy
```

x: real `iy`: imaginary

Yapı tanımlamanın birkaç yolu vardır. Şimdi bu sayıyı tanımlayacak bir yapı oluşturalım ve bu yapıdan bir nesne tanımlayalım;

```
struct complex {
     int real;
     int im;
}
struct complex a, b;
```

dediğimiz zaman `a` ve `b` birer *complex* sayı olmuşlardır. Tanımlanan `a` ve `b`'nin elemanlarına ulaşmak için nokta operatörü kullanırız. Eğer `a` veya `b`'nin birisi ya da ikisi de pointer olarak tanımlansaydı ok (->) operatörüyle elemanlarına ulaşacaktık. Bir örnek yapalım.

```
a.real = 4;  b.real = 6;
a.im = 7;          b.im = 9;
```

Şimdi de hem pointer olan hem de bir nesne olan tanımlama yapalım ve elemanlarına erişelim.

```
struct complex obj;
struct complex *p = &obj;
p -> real = 7;     obj.real = 7;
p -> im = 8; obj.im = 8;
```

Bu örnekte `complex` türünde `obj` isimli bir nesne ve `p` isimli bir pointer tanımlanmıştır. `p` pointer'ına ise `obj` nesnesinin adresi atanmıştır. `obj` nesnesinin elemanlarına hem `obj`'nin kendisinden, hem de `p` pointer'ından erişilebilir. `p` pointer'ından erişilirken ok (->) operatörü, `obj` nesnesinden erişilirken ise nokta (.) operatörü kullanıldığına dikkat ediniz. `p` pointer'ından erişmekle `obj` nesnesinde erişmek arasında hiçbir fark yoktur.

Örnek 15 İki karmaşık sayıyı toplayan fonksiyonu yazalım.

```
struct complex {
     int real;
     int im;
}
struct complex add(struct complex a, struct complex b) {
     struct complex result;
     result.real = a.real + b.real;
     result.im = a.im + b.im;
     return result;
}
```

Alternatif struct tanımları

```
typedef struct {
     int real;
     int im;
} complex;
complex a, b;
```

Görüldüğü gibi bir `typedef` anahtar sözcüğüyle tanımlanan `struct` yapısından hemen sonra yapı ismi tanımlanıyor. Artık bu tanımlamadan sonra nesneleri oluştururken başa `struct` yazmak gerekmeyecektir.

```
complex a, b;
```

tanımlamasıyla `complex` türden a ve b isimli nesne meydana getirilmiş

5.3 Veri Yapılarının İmplementasyonu

Veri Yapısı, bilgisayarda verinin saklanması (tutulması) ve organizasyonu için bir yoldur. Veri yapılarını saklarken ya da organize ederken iki şekilde çalışacağız;

1- Matematiksel ve mantıksal modeller (Soyut Veri Tipleri – Abstract Data Types - ADT)

Bir veri yapısına bakarken tepeden (yukarıdan) soyut olarak ele alacağız. Yani hangi işlemlere ve özelliklere sahip olduğuna bakacağız. Örneğin bir TV alıcısına soyut biçimde bakarsak elektriksel bir alet olduğu için onun açma ve kapama tuşu, sinyal almak için bir anteni olduğunu göreceğiz. Aynı zamanda görüntü ve ses vardır. Bu TV'ye matematiksel ve mantıksal modelleme açısından bakarsak içindeki devrelerin ne olduğu veya nasıl tasarlandıkları konusuyla ve hangi firma üretmiş gibi bilgilerle ilgilenmeyeceğiz. Soyut bakışın içerisinde uygulama (implementasyon) yoktur.

Örnek olarak bir listeyi ele alabiliriz. Bu listenin özelliklerinden bazılarını aşağıda belirtirsek,

Liste;

- Herhangi bir tipte belirli sayıda elemanları saklasın,
- Elemanları listedeki koordinatlarından okusun,
- Elemanları belirli koordinattaki diğer elemanlarla değiştirsin (modifiye),
- Elemanlarda güncelleme yapsın,
- Ekleme/silme yapsın.

Verilen örnek, matematiksel ve mantıksal modelleme olarak, soyut veri tipi olarak listenin tanımıdır. Bu soyut veri tipini, yüksek seviyeli bir dilde somut hale nasıl getirebiliriz? İlk olarak Diziler akla gelmektedir.

2- Uygulama (Implementation)

Matematiksel ve mantıksal modellemenin uygulamasıdır. Diziler, programlamada çok kullanışlı veri yapıları olmasına rağmen bazı dezavantajları ve kısıtları vardır. Örneğin;

- Derleme aşamasında dizinin boyutu bilinmelidir,
- Diziler bellekte sürekli olarak yer kaplarlar. Örneğin `int` türden bir dizi tanımlanmışsa, eleman sayısı çarpı `int` türünün kapladığı alan kadar bellekte yer kaplayacaktır,
- Ekleme işleminde dizinin diğer elemanlarını kaydırmak gerekir. Bu işlemi yaparken dizinin boyutunun da aşılmaması gerekir,
- Silme işlemlerinde değeri boş elemanlar oluşacaktır.

Bu ve bunun gibi sorunların üstesinden gelmek ancak Bağlı Listelerle (Linked List) mümkün hale gelir.

Soyut veri tipleri (ADT)'nin resmi tanımını şu şekilde yapabiliriz; Soyut veri tipleri (ADTs) sadece veriyi ve işlemleri (operasyonları) tanımlar, uygulama yoktur.

Bazı veri tipleri;

- Arrays (Diziler)
- Linked List (Bağlı liste)
- Stack (Yığın)
- Queue (Kuyruk)
- Tree (Ağaç)

- Graph (Graf, çizge)

Veri yapılarını çalışırken,

1- Mantıksal görünüşüne bakacağız,
2- İçinde barındırdığı işlemelere bakacağız (operation),

Bir bilgisayar programında uygulamasını yapacağız.