ÖZYİNELEME (RECURSION)

ÖZYİNELEME(RECURSION)

- Kendi kendisini doğrudan veya dolaylı olarak çağıran fonksiyonlara özyineli (recursive) fonksiyonlar adı verilir.
- Özyineleme bir problemi benzer şekilde olan daha basit alt problemlere bölünerek çözülmesini sağlayan bir tekniktir.
- Alt problemler de kendi içlerinde başka alt problemlere bölünebilir.
- Alt problemler çözülebilecek kadar küçülünce bölme işlemi durur.
- Özyineleme, döngülere (iteration) alternatif olarak kullanılabilir.

ÖZYİNELEME

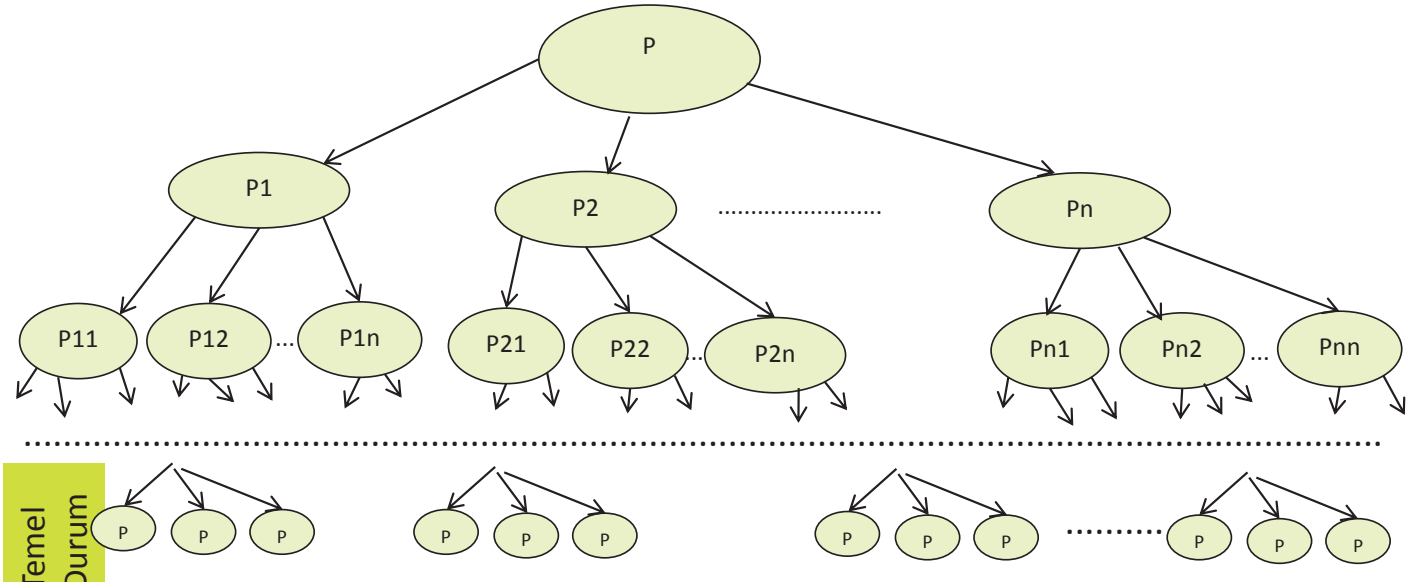
- Bir problem özyineli olmayan basit bir çözüme sahiptir. Problemin diğer durumları özyineleme ile durdurma durumuna (stopping case) indirgenebilir.

- Özyineleme işlemi durdurma durumu sağlanınca sonlandırılır.
- Özyineleme güçlü bir problem çözme mekanizmasıdır.
  - Çoğu algoritma kolayca özyinelemeli şekilde çözülebilir.
  - Fakat sonsuz döngü yapmamaya dikkat edilmeli.

- **Genel yazımı-kaba kod:**
  - *if (durdurma durumu sağlandıysa)*
  - *çözümü yap*
  - *else*
  - *problemi özyineleme kullanarak indirge*

Özyineleme- Böl & Yönet Stratejisi

- Bilgisayar birimlerinde önemli bir yere sahiptir:
  - Problemi küçük parçalara böl
  - Her bir parçayı bağımsız şekilde çöz
  - Parçaları birleştirerek ana problemin çözümüne ulaş



Özyineleme- Böl & Yönet Stratejisi

```
/* P problemini çöz */
Solve(P){
    /* Temel durum(s) */
    if P problemi temel durumda ise
        return çözüm

 /* (n>=2) için P yi P1, P2, ..Pn şeklinde parçalara böl */
 /* Problemleri özyinelemeli şekilde çöz */
    S1 = Solve(P1); /* S1 için P1 problemini çöz */
    S2 = Solve(P2); /* S2 için P2 problemini çöz*/
     …
    Sn = Solve(Pn); /* Sn için Pn problemini çöz */

     /* Çözüm için parçaları birleştir. */
    S = Merge(S1, S2, …, Sn);

     /* Çözümü geri döndür */
     return S;
} //bitti-Solve
```

ÖZYİNELEME

- Faktöryel fonksiyonu: Klasik bir özyineleme örneğidir:
  - **n! = 1 * 2 * 3 * … * (n – 1) * n**
- Faktoriyel işlemini özyineli tanımlamak için küçük sayıların faktöriyeli şeklinde tanımlamak gerekir.
  - **n! = n * (n – 1) !**
  - Durdurma durumu 0 ! = 1 olarak alınır.
- Her çağırmada n değeri bir azaltılarak durdurma durumuna ulaşılır.
- Özyineli tanımlama:

$$f(n) = \begin{cases} n = 1, & \text{if } n = 0 \\ n * f(n-1), & \text{if } n > 0 \end{cases}$$

ÖZYİNELEME

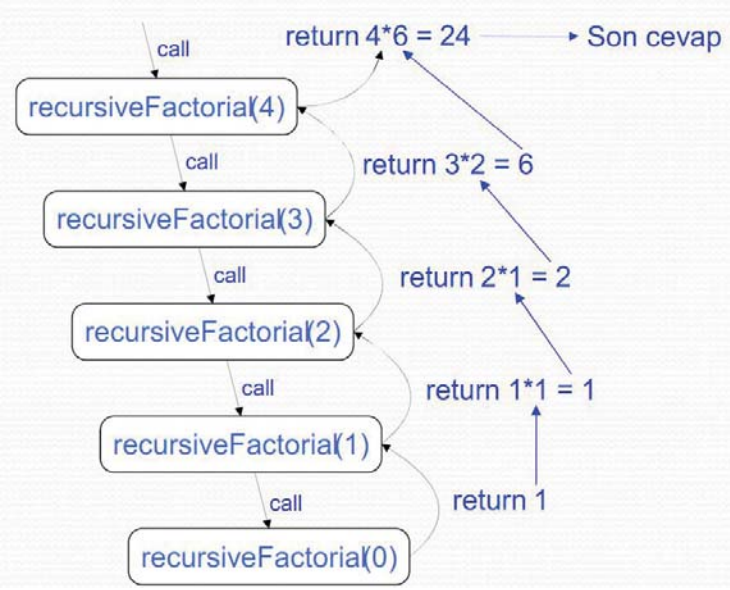
- Aşağıdaki kod çalıştığında n sayısının faktöriyel değerini hesaplar.

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

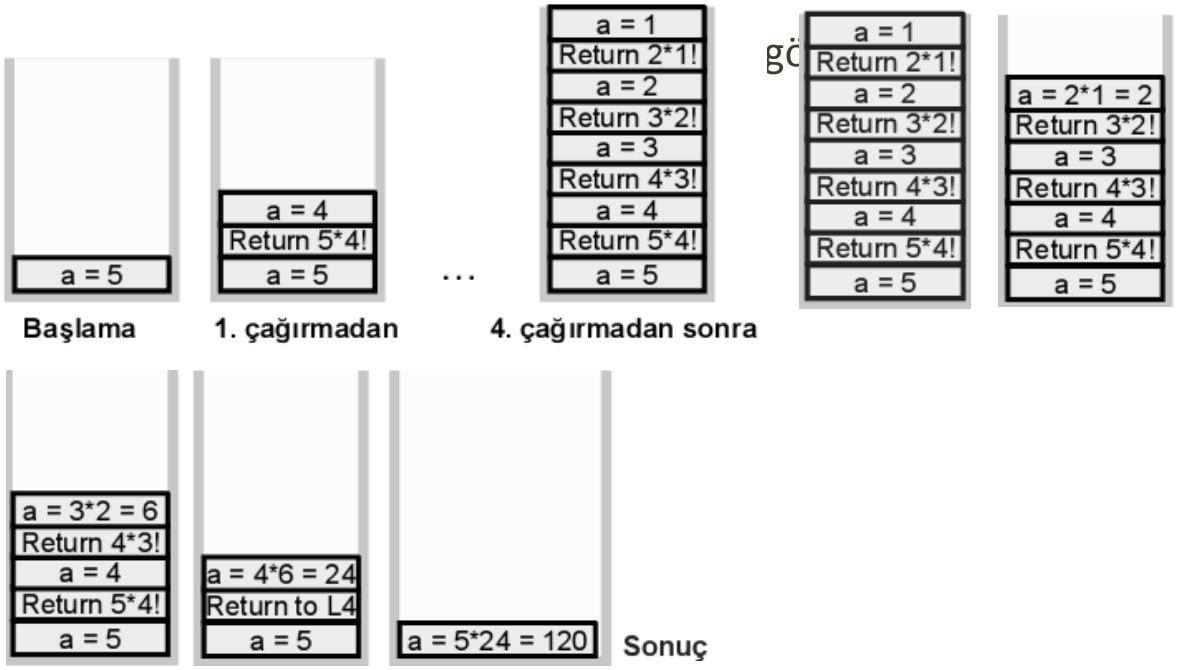
- n! değerini hesaplar ve bulduğu değeri return ile gönderir.

ÖZYİNELEME

- Özyineleme izleme
- Her özyineleme çağrımı için bir kutu
- Her çağırandan çağrılana bir ok
- Her çağrılandan çağırana çizilen ok geri dönüş değerini gösterir.



ÖZYİNELEME


a = 5
a = 4
Return 5*4!
a = 5
...
a = 1
Return 2*1!
a = 2
Return 3*2!
a = 3
Return 4*3!
a = 4
Return 5*4!
a = 5
gö
a = 1
Return 2*1!
a = 2
Return 3*2!
a = 3
Return 4*3!
a = 4
Return 5*4!
a = 5
a = 2*1 = 2
Return 3*2!
a = 3
Return 4*3!
a = 4
Return 5*4!
a = 5
Başlama
1. çağırmadan
4. çağırmadan sonra
a = 3*2 = 6
Return 4*3!
a = 4
Return 5*4!
a = 5
a = 4*6 = 24
Return to L4
a = 5
a = 5*24 = 120
Sonuç

ÖZYİNELEME

- Genellikle iterative fonksiyonlar zaman ve yer bakımından daha etkindirler.
- Iterative algoritma döngü yapısını kullanır.
- Özyineleme algoritması dallanma (branching) algoritmasını kullanır.
- Fonksiyon özyineli olarak her çağrılışında yerel değişkenler ve parametreler için bellekte yer ayrılır.
- Özyineleme problemin çözümünü basitleştirebilir, sonuç genellikle kısadır ve kod kolayca anlaşılabilir.
- Her özyinelemeli olarak tanımlanmış problemin iterative çözümüne geçiş yapılabilir.

ÖZYİNELEME

- **Recursive**
- int recFact(int n)
- {
- if(n ==1) return(1);
- else
- return(n * recFact(n - 1));
- }

- **Iterative**
- int iteFact(int n)
- {
- int araDeger = 1;
- for (int i = n; i > 0; i- -)
- araDeger * = i;
- return araDeger;
- }

FaktoriyelOrnek.java

```
import java.io.*;
 class FaktoryelOrnek
{   static int sayi;
   public static void main(String args[]) throws IOException
  {
    System.out.print("Sayi veriniz :");     System.out.flush();
    sayi=getInt();     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));
  }
```

```
public static String getString() throws IOException
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String s = br.readLine();
    return s;
  }
   public static int getInt() throws IOException
  { String s = getString();
    return Integer.parseInt(s);
  }
 }
```

N'ye Kadar Olan Sayıların Toplamı

- Problemimizin 1'den n'ye kadar sayıların toplamı olduğunu varsayalım.

- Bu problemi özyinelemeli nasıl düşüneceğiz:
  - Topla(n) = 1+2+..+n ifadesini hesaplamak için
    - Topla(n-1) = 1+2+..+n-1 ifadesini hesapla (aynı türden daha küçük bir problem)
    - Topla(n-1) ifadesine n ekleyerek Topla(n) ifadesi hesaplanır.
    - Topla(n) = Topla(n-1) + n;
  - Temel durumu belirlememiz gerekiyor.
    - Temel durum, (alt problem) problemi bölmeye gerek kalmadan kolayca çözülebilen problemdir.
    - n = 1 ise, Topla(1) = 1;

Topla(4) için Özyineleme Ağacı

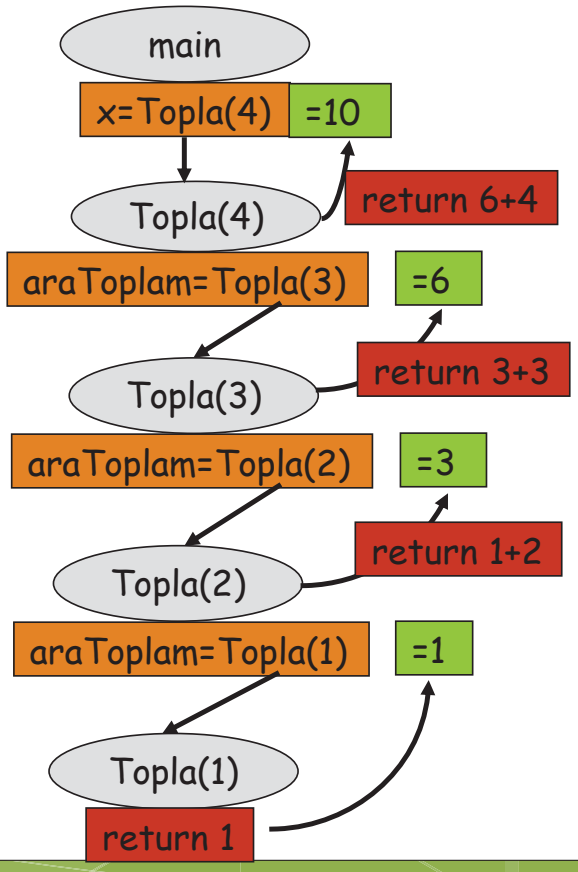
```
/* Topla 1+2+3+…+n */
int Topla(int n){
  int araToplam = 0;

  /* Temel Durum */
  if (n == 1) return 1;

  /* Böl ve Yönet */
  araToplam = Topla(n-1);

  /* Birleştir */
  return araToplam + n;
} /* bitti-Topla */

Public ... main(...){
print("Topla: "+ Topla(4));
} /* bitti-main */
```



Topla(n)'nin çalışma zamanı

```
/* Topla 1+2+3+…+n */
int Topla(int n){
  int araToplam = 0;

  /* Temel durum */
  if (n == 1) return 1;

  /* Böl ve yönet */
  araToplam = Topla(n-1);

  /* Birleştir */
  return araToplam + n;
} /* bitti-araToplam */
```

$$T(n) = \begin{cases} n = 1 \rightarrow 1 \text{ (Temel durum)} \\ n > 1 \rightarrow T(n-1) + 1 \end{cases}$$

a^n İfadesini Hesaplama-Pow(a,n)

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- Böl, yönet & birleştir işlemleri bir ifade ile yapılabilir.

```
/* a^n hesapla */
double Ust(double a, int n){
  double araSonuc;

  /* Temel durum */
  if (n == 0) return 1;
  else if (n == 1) return a;

  /* araSonuc = a^(n-1) */
 araSonuc = Ust(a, n-1);

  /* Birleştir */
  return araSonuc*a;
} /* bitti-Ust */
```

```
/* Hesapla a^n */
double Ust(double a, int n){
  /* Temel durum */
  if (n == 0) return 1;
  else if (n == 1) return a;

  return Ust(a, n-1)*a;
} /* bitti-Ust */
```

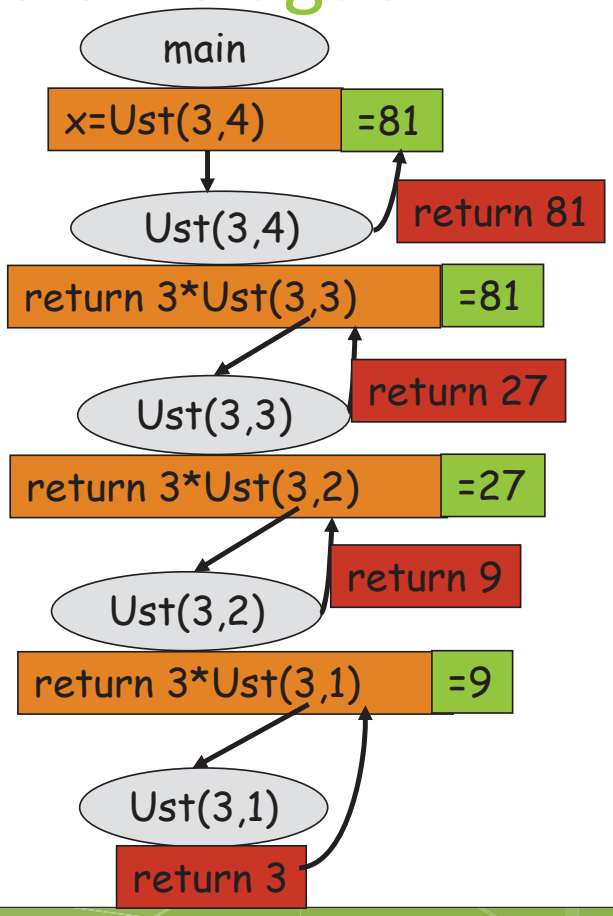
Ust(3, 4) için Özyineleme ağacı

```
/* Hesapla a^n */
double Ust(double a, int n){
  /* Temel durum */
  if (n == 0) return 1;
  else if (n == 1) return a;

  return a * Ust(a, n-1);
} /* bitti-Ust */

Public ... main(...){
  double x;

  x = Ust(3, 4);
} /* bitti-main */
```



Ust(a, n)'nin Çalışma Zamanı

```
/* Hesapla a^n */
double Ust(double a, int n){
  /* temel durum */
  if (n == 0) return 1;
  else if (n == 1) return a;

  return a * Ust(a, n-1);
} /* bitti-Ust */
```

$$T(n) = \begin{cases} n <= 1 \rightarrow 1 \text{ (Temel durum)} \\ N > 1 \rightarrow T(n-1) + 1 \end{cases}$$

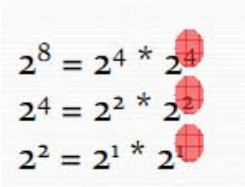
Pow(x, n) 'nin Çalışma Zamanı

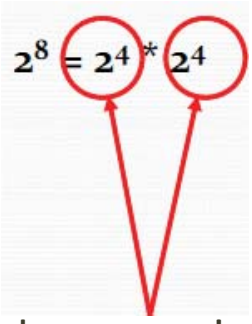
- Bu fonksiyon O(n) zamanında çalışır (n adet özyineli çağrı yapılır)
- Daha iyi bir çözüm var mı?
- Ara sonuçların karesini alarak daha etkin bir doğrusal özyineli algoritma yazabiliriz:

$$p(x,n) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } x > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } x > 0 \text{ is even} \end{cases}$$

Power(2, 8)

- $2^8=2*2*2*2*2*2*2*2$
- Ancak biz bu çözümü iki eşit parçaya bölerek ifade ede biliriz:
- $2^8=2^4*2^4$
- ve $2^4=2^2*2^2$
- ve $2^2=2^1*2^1$





- herhangi bir sayının 1. kuvveti kendisidir.
- Avantaj...
- İkisi de aynı olduğu için, her ikisini de hesaplamıyoruz! ve sadece 3 adet çarpma işlemi yapıyoruz.

Kuvvet değeri tek sayı ise

- Tek olanları şu şekilde yaparız:
- $2^{tek} = 2 * 2^{(tek-1)}$
- Peki öyleyse, 2^{21} 'i hesaplayalım

$2^{21} = 2 * 2^{20}$ (Tek sayı hilesi)
$2^{20} = 2^{10} * 2^{10}$
$2^{10} = 2^5 * 2^5$
$2^5 = 2 * 2^4$
$2^4 = 2^2 * 2^2$
$2^2 = 2^1 * 2^1$

- Görüldüğü gibi 20 defa çarpım yerine sadece 6 kerede sonuca ulaşılıyor

$2^{21} = 2 * 2^{20}$
$2^{20} = 2^{10} * 2^{10}$
$2^{10} = 2^5 * 2^5$
$2^5 = 2 * 2^4$
$2^4 = 2^2 * 2^2$
$2^2 = 2^1 * 2^1$

Özyinelemenin mantığı

- Eğer üst çift ise, iki parçaya ayırıp işleriz.
- Eğer üst tek ise, 1 çıkarır daha sonra kalan kısmı ikiye ayırıp işleriz. Ama üstten bir çıkardığımız için en sonunda 1 adet çarpma işlemi daha yapmayı unutmayın.

- İşlemi formülize etmeye başlayalım:
  - e == 0 , Pow(x, e) = 1
  - e == 1 , Pow(x, e) = x
  - e çift ise , Pow(x, e) = Pow(x, e/2) * Pow(x,e/2)
  - e>1 ve tek ise , Pow(x, e) = x * Pow(x, e-1)

Power(x, n) 'nin Çalışma Zamanı

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$
$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$
$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$
$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$

Algorithm Power(x, n):
Input: x sayısı ve n tamsayısı, n >= 0
Output: xn değeri
if n = 0 **then**
return 1
if n is odd **then**
y = Power(x, (n - 1)/ 2)
return x · y ·y
else
y = Power(x, n/ 2)
return y · y

Her özyineli çağırmada n sayısını 2'ye bölüyoruz; dolayısıyla, log n özyineli çağrı yaparız. Bu metod O(log n) zamanına sahiptir.

Burada ara sonucu y değişkeni ile göstermemiz önemli; şayet metod çağırma yazarsak metod 2 defa çağrılmış olur.

Fibonacci Sayıları

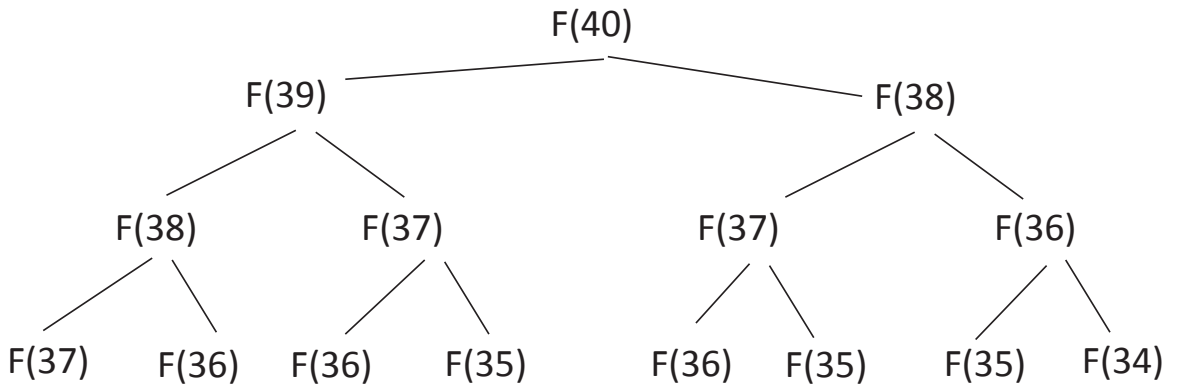
- Fibonacci sayılarını tanımlayacak olursak:
- 1 1 2 3 5 8 13.....
  - F(0) = 0
  - F(1) = 1
  - F(n) = F(n-1) + F(n-2)

```
/* n. Fibonacci sayısını hesaplama*/
int Fibonacci(int n){
  /* Temel durum */
  if (n == 0) return 0;
  if (n == 1) return 1;

  return Fibonacci(n-1) + Fibonacci(n-2);
} /* bitti-Fibonacci */
```

Fibonacci Sayıları

- Fibonacci sayılarının tanımı özyinelemelidir.
- Örneğin 40. fibonacci değerini bulmaya çalışalım.



- F(40) için toplam kaç tane özyinelemeli çağrı yapılır.
- Cevap: 300 000 000 den fazla yordam çağrılır.

Fibonacci Sayıları

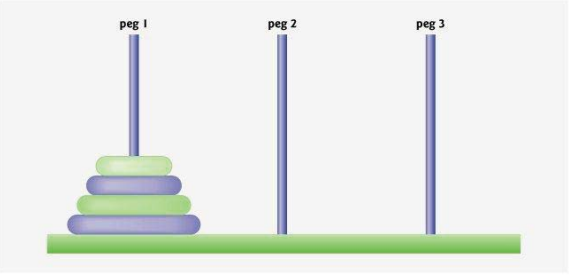
- Basit bir "for" ile çözülebilecek problemler için özyinelemeli algoritmalar kullanılmaz.

```
/* n. Fibonacci sayısını hesaplama*/
public static int fibonacci(int n){
  if(n == 1 || n == 2)  return 1;

  int s1=1,s2=1,sonuc=0;
  for(int i=0; i<n; i++){
    sonuc = s1 + s2;
    s1 = s2;
    s2 = sonuc;
  }
  return sonuc;
}
```

Hanoi Kuleleri

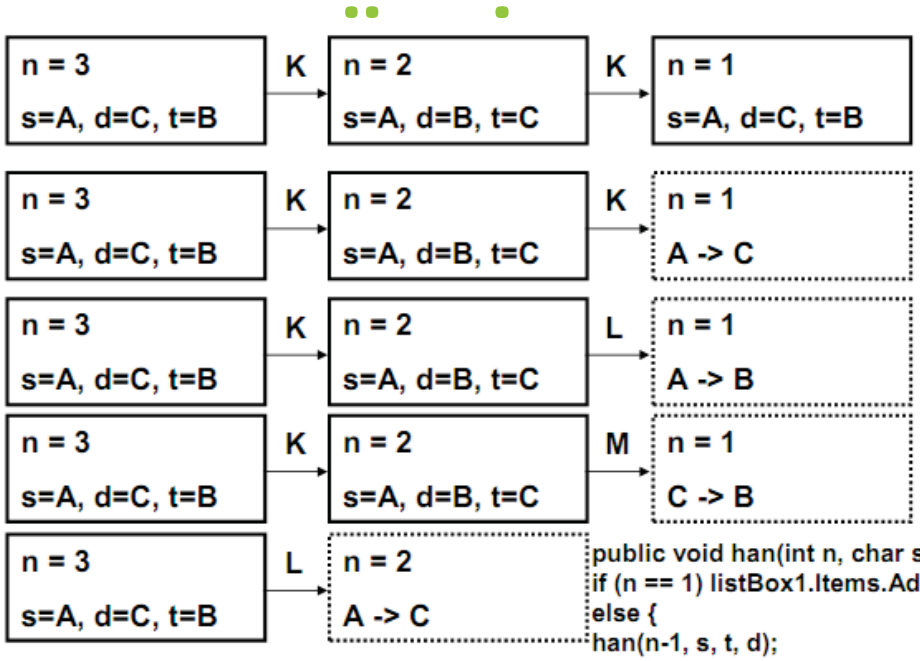
- **Verilen:** üç iğne
  - İlk iğnede en küçük disk en üstte olacak şekilde yerleştirilmiş farklı büyüklükte disk kümesi.
- **Amaç:** diskleri en soldan en sağa taşımak.
- **Şartlar:** aynı anda sadece tek disk taşınabilir.
- Bir disk boş bir iğneye veya daha büyük bir diskin üzerine taşınabilir.



Hanoi Kuleleri– Özyinelemeli Çözüm-Java

```
package hanoikuleleri;
import java.util.*;
public class Hanoikuleleri  {

public static void main(String[] args)
{
 System.out.print("n değerini giriniz : ");
 Scanner klavye = new Scanner(System.in);  int n = klavye.nextInt();
 tasi(n, 'A', 'B', 'C');
}
 public static void tasi(int n, char A, char B, char C)
{if(n==1) System.out.println(A + " --> " + B);
 else
 {
 tasi(n-1, A, C, B);      tasi(1, A, B, C);    tasi(n-1, C, B, A); }
 return;
}
```




```
public void han(int n, char s, char d, char t) {
if (n == 1) listBox1.Items.Add(s + " -> " + d);
else {
han(n-1, s, t, d);                    // K
han(1, s, d, t);                      // L
han(n-1, t, d, s);                    // M
}
```

Kuyruk Özyineleme (Tail Recursion)

- Özyineleme metodunda, özyineli çağrı son adım ise bu durum oluşur. Yineleme çağrısı metodun en sonunda yapılır.
- int recFact(int n)
- {
- if (n<=1) return 1;
- else return n * recFact(n-1);
- }
- void tail() {
- .....
- .....
- tail(); }

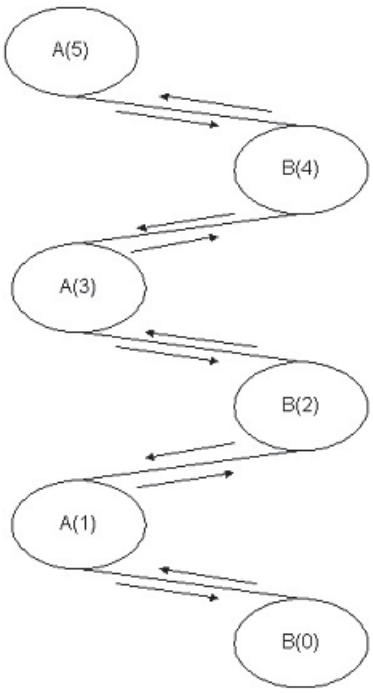
Kuyruk Olmayan Özyineleme (nonTail Recursion)

- Özyineleme metodunda, özyineli çağrı son adım değil ise bu durum oluşur. Yineleme çağrısından sonra başka işlemler yapılır (yazdırma v.b.) veya ikili özyineleme olabilir.
- Taban durumu haricinde iki özyinelemeli çağrı yapılıyorsa, ikili özyineleme oluşur.

```
int nontail(int n)
{
    if (n > 0)  {
      .......
      .......
      nontail(n-1);
      printf(n);
      .......
      .......
      nontail(n-1); }
}
```

Dolaylı Özyineleme (Indirect Recursion)

- Yineleme çağrısı başka bir fonksiyonun içinden yapılır.
- **void A(int n)**
- **{**
  - **if (n <= 0) return 1;**
  - **n- -;**
  - **B(n);**
- **}**
- **void B(int n)**
- **{**
  - **if (n <= 0) return 1;**
  - **n- -;**
  - **A(n);**
- **}**



İç içe Özyineleme (Nested Recursion)

- Yineleme çağrısı içindende yineleme çağrısı yapılır.

```
int A(int n, int m)
{
    if (n <= 0) return 1;
    return A(n-1, A(n-1, m-1));
}
```

Özyineleme mi İterasyon mu?

- Eğer iteratif çözümü tercih ederseniz algoritmanızın karmaşıklığı O(N) olur.
- Gerçek dünyada bu yöntemi kullanırsanız kovulursunuz.
- Eğer özyineleme kullanırsanız, algoritmanızın karmaşıklığı $O(\log_2 n)$ olur.
- Bunun için terfi bile alabilirsiniz.

ÖZET

- Özyineleme bütün doğada mevcuttur.
- Formül ezberlemekten daha kolaydır. Her problemin bir formülü olmayabilir, ancak her problem, küçük, tekrarlayan adımlar serisi olarak ifade edilebilir.
- Özyinelemeli bir işlemde her adım küçük ve hesaplanabilir olmalıdır.
- Kesinlikle sonlandırıcı bir şarta sahip olmalı.
- Özyineleme en çok, özyinelemeli olarak tanımlanmış binary tree dediğimiz veri yapılarında kullanılmak üzere yazılan algoritmalar için faydalıdır. Normal iterasyona göre çok daha kolaydır.
- Özyineleme, böl/yönet (divide & conquer) türündeki algoritmalar için mükemmeldir.

Ödev

- 1- n adet x değeri için standart sapmayı (σ) bulan programı iterasyon ve recursive ile yapınız.

- xm=(1/n) $\sum_{k} x_k$

- (σ)= standart sapma
- V = varyans değeri
- xm = mean değeri

$$\sigma = \sqrt{V}$$

$$V = (1 / (n - 1)) \sum_{k} (x_k - xm)^2$$

Ödev

- 2- Ackerman fonksiyonu aşağıdaki şekilde tanımlanmıştır. Bu fonkisyonu öz yinelemeli olarak gerçekleitiriniz.

$$A(m,n)=\begin{cases} n+1 & ,m=1 \\ A(m-1,1) & ,m>0 \; ve \; n=0 \\ A\big(m-1,A(m,n-1)\big) & ,m>0 \; ve \; n>0 \end{cases}$$