

Bölüm 3: Özyineleme

Java ile Nesne Yönelimli Programlama

Temel durum

Çağrı yığını

Faktöriyel ve toplam

Fibonacci, Lucas, Catalan, Bell

Palindrom ve EBOB

Hanoi kuleleri

Verimlilik ve döngüler

Kuyruk özyinelemesi

Hafızalama

Sercan KÜLCÜ · Giresun Üniversitesi · Elektrik-Elektronik Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Kavram

Özyinelemeyi (recursion), temel durumu ve özyinelemeli adımı tanımlamak; bir problemi kendisinin küçük bir kopyasına indirgemek.

Yazma

Faktöriyel, toplam, seri, rakamlar toplamı, palindrom, EBOB ve Hanoi kuleleri için özyinelemeli Java metotları yazmak.

Verimlilik

Özyinelemeli ve ilerlemeli (iterative) çözümleri karşılaştırmak; gerektiğinde döngüye çevirmek.

Çalışma biçimi

Çağrı yığnında çerçevelerin nasıl biriktiğini ve geri döndüğünü adım adım izlemek; çağrı ağacı çizmek.

Hatalar

Sonsuz özyinelemeyi,

`StackOverflowError` 'ı ve eksik `return`

hatalarını tanımak ve önlemek.

İleri teknikler

Kuyruk özyinelemesini ve hafızalamayı (memoization) açıklamak; tekrarlanan hesaplamaları ortadan kaldırmak.

Özyineleme Nedir?

- **Özyineleme (Recursion)**: bir metodun, görevini küçük parçalara bölerek **kendini çağırmasıdır**.
- Her çağrı, problemin **daha küçük ama aynı tipten** bir kopyasını çözer; en küçük kopya doğrudan çözülür.
- Tekrarlama süreci kendini benzer biçimde tekrar eder: bir yapının içinde **aynı tipten** bir yapı bulunur.
- Bazı problemleri (ağaçlar, seriler, böl ve yönet) döngülerden **daha anlaşılır ve temiz** çözer.

Java'da özyineleme için özel bir anahtar kelime yoktur: metod, gövdesinde **kendi adını** çağırır.

Matruşka bebekleri

Her bebeğin içinde daha küçük bir bebek vardır; en küçüğü açılmaz (temel durum).

Sinemada kaçınıcı sıra?

Öndekine sorarsınız, o da öndekine... İlk sıradaki "1" der; cevaplar geri gelirken her kişi 1 ekler.

Karşılıklı aynalar

Görüntünün içinde görüntü: temel durumu olmayan özyineleme sonsuza gider.

Temel Durum ve Özyinelemeli Adım

1. Temel durum (base case)

Cevabı **doğrudan** bilinen en küçük problem.
Burada metod kendini çağırılmaz ve özyineleme durur. Örn. $1! = 1$.

2. Özyinelemeli adım (recursive step)

Problemi **bir adım küçültüp** kendine sorar, gelen cevabı kullanarak kendi cevabını kurar.
Örn. $4! = 4 \times 3!$.

Matematikteki tümevarımsal tanımın birebir karşılığıdır:

$n! = 1$ ($n \leq 1$ ise), $n! = n \times (n - 1)!$ ($n > 1$ ise)

$\text{faktoriyel}(4) = 4 \times \text{faktoriyel}(3)$

$\text{faktoriyel}(3) = 3 \times \text{faktoriyel}(2)$

$\text{faktoriyel}(2) = 2 \times \text{faktoriyel}(1)$

$\text{faktoriyel}(1) = 1$

temel durum: kendini çağırılmaz

ÖZYİNELEME KAVRAMI

Çağrı Yığını

- Her metod çağrısı için bellekte bir **çerçeve** (stack frame) açılır: parametreler, yerel değişkenler ve dönülecek yer burada tutulur.
- Çerçeveler **çağrı yığının**a (call stack) üst üste konur; yalnızca **tepedeki** çerçeve çalışır, alttakiler bekler.
- Özyinelemede aynı metodun **birden çok çerçevesi** aynı anda yaşar; her birinin kendi `n` 'si vardır.
- Metot `return` ettiğinde çerçevesi silinir ve değer, bir alttaki çerçeveye iletilir (son giren ilk çıkar: LIFO).
- Yığının boyutu sınırlıdır; çok derin özyineleme `StackOverflowError` ile biter.



Faktöriyel Alma

```
int faktoriyel(int n) {  
    if (n == 0 || n == 1) {  
        return 1;    // temel durum  
    } else {  
        return n * faktoriyel(n - 1); // özyineleme  
    }  
}
```

Matematik

$$n! = 1 \times 2 \times \dots \times n$$

$$n! = n \times (n - 1)!, 0! = 1! = 1$$

Satır 2-3: temel durum

$0! = 1! = 1$; burada özyineleme durur.

Satır 5: özyinelemeli adım

$n!$, $(n - 1)!$ 'in n ile çarpımıdır.

İlerleme

Her çağrıda n 1 azalır $\rightarrow n \geq 0$ için temel duruma mutlaka ulaşılır.

Taşma

`int` 13!'i taşıyamaz: `faktoriyel(13)` = 1932053504 (yanlış!). `long` 20!'e kadar yeter.

Çağrı: `faktoriyel(5)` $\rightarrow$ 120 $\cdot$ `faktoriyel(0)` $\rightarrow$ 1

faktoriyel(4) Açılımı

```
faktoriyel(4)
= 4 * faktoriyel(3)
= 4 * (3 * faktoriyel(2))
= 4 * (3 * (2 * faktoriyel(1)))
= 4 * (3 * (2 * 1))
= 4 * (3 * 2)
= 4 * 6
= 24
```

İniş (çağrılar)

İlk dört satırda her çağrı yeni bir çağrı açar; çarpımlar **bekletilir**, çünkü sağ taraftaki değer henüz bilinmiyor.

Dip (temel durum)

`faktoriyel(1)` doğrudan 1 döndürür. Kod `n == 1`'de durduğu için `faktoriyel(0)` hiç çağrılmaz.

Çıkış (dönüşler)

Bekleyen çarpımlar **içten dışa** doğru tamamlanır: 2×1 , 3×2 , 4×6 .

faktoriyel(4): Kodu Adım Adım İzle

faktoriyel(4)

faktoriyel(3)

faktoriyel(2)

faktoriyel(1)

main , faktoriyel(4) çağrısını yapar; sonucu yazdırmak için beklemeye başlar.

Adım 1 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

main()

DEĞİŞKENLER

n = -

ÇIKTI

faktoriyel(4): Kodu Adım Adım İzle

faktoriyel(4)
n = 4

faktoriyel(3)

faktoriyel(2)

faktoriyel(1)

Yeni çerçeve: `n = 4` . Koşul `n == 0 || n == 1` yanlış → temel durum değil.

Adım 2 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

faktoriyel(4)

main()

DEĞİŞKENLER

n = 4

ÇIKTI

faktoriyel(4): Kodu Adım Adım İzle

faktoriyel(4)

4 * ?

faktoriyel(3)

faktoriyel(2)

faktoriyel(1)

4 * faktoriyel(3) hesaplanmalı. Çarpım bekler; önce faktoriyel(3) çağrılır.

Adım 3 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         ▶ return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

faktoriyel(4)

main()

DEĞİŞKENLER

n = 4

ÇIKTI

faktoriyel(4): Kodu Adım Adım İzle

faktoriyel(4)
4 * ?

faktoriyel(3)
n = 3

faktoriyel(2)

faktoriyel(1)

Yeni çerçeve: `n = 3` . Koşul `n == 0 || n == 1` yanlış → temel durum değil.

Adım 4 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

faktoriyel(3)

faktoriyel(4)

main()

DEĞİŞKENLER

n = 3

ÇIKTI

faktoriyel(4): Kodu Adım Adım İzle

faktoriyel(4)
4 * ?

faktoriyel(3)
3 * ?

faktoriyel(2)

faktoriyel(1)

3 * faktoriyel(2) hesaplanmalı. Çarpım bekler; önce faktoriyel(2) çağrılır.

Adım 5 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

faktoriyel(3)

faktoriyel(4)

main()

DEĞİŞKENLER

n = 3

ÇIKTI

faktoriyel(4): Kodu Adım Adım İzle

faktoriyel(4)
4 * ?

faktoriyel(3)
3 * ?

faktoriyel(2)
n = 2

faktoriyel(1)

Yeni çerçeve: `n = 2` . Koşul `n == 0 || n == 1` yanlış → temel durum değil.

Adım 6 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

faktoriyel(2)

faktoriyel(3)

faktoriyel(4)

main()

DEĞİŞKENLER

n = 2

ÇIKTI

faktoriyel(4): Kodu Adım Adım İzle

faktoriyel(4)
4 * ?

faktoriyel(3)
3 * ?

faktoriyel(2)
2 * ?

faktoriyel(1)

2 * faktoriyel(1) hesaplanmalı. Çarpım bekler; önce faktoriyel(1) çağrılır.

Adım 7 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

faktoriyel(2)

faktoriyel(3)

faktoriyel(4)

main()

DEĞİŞKENLER

n = 2

ÇIKTI

faktoriyel(4): Kodu Adım Adım İzle

faktoriyel(4)
4 * ?

faktoriyel(3)
3 * ?

faktoriyel(2)
2 * ?

faktoriyel(1)
return 1

n = 1 : temel durum. Yeni çağrı yapılmaz, **1** döndürülür. Artık çerçeveler geri dönmeye başlar.

Adım 8 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

faktoriyel(1)

faktoriyel(2)

faktoriyel(3)

faktoriyel(4)

main()

DEĞİŞKENLER

n = 1

dönüş = 1

ÇIKTI

faktoriyel(4): Kodu Adım Adım İzle

faktoriyel(4)
4 * ?

faktoriyel(3)
3 * ?

faktoriyel(2)
2 * 1 = 2

faktoriyel(1)
= 1

faktoriyel(1) çerçevesi yığından çıktı, 1 getirdi: 2 * 1 = 2 döner.

Adım 9 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

faktoriyel(2)

faktoriyel(3)

faktoriyel(4)

main()

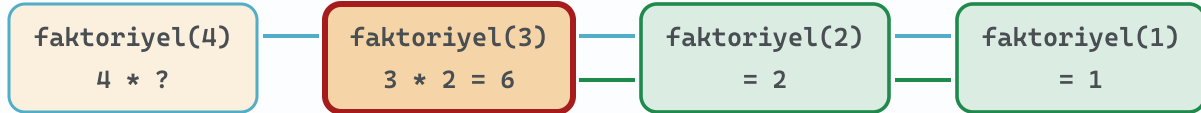
DEĞİŞKENLER

n = 2

dönüş = 2

ÇIKTI

faktoriyel(4): Kodu Adım Adım İzle



faktoriyel(2) çerçevesi yığından çıktı, 2 getirdi: $3 * 2 = 6$ döner.

Adım 10 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

faktoriyel(3)

faktoriyel(4)

main()

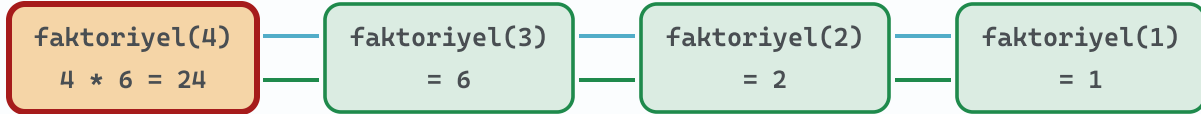
DEĞİŞKENLER

n = 3

dönüş = 6

ÇIKTI

faktoriyel(4): Kodu Adım Adım İzle



faktoriyel(3) çerçevesi yığından çıktı, 6 getirdi: $4 * 6 = 24$ döner.

Adım 11 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

faktoriyel(4)

main()

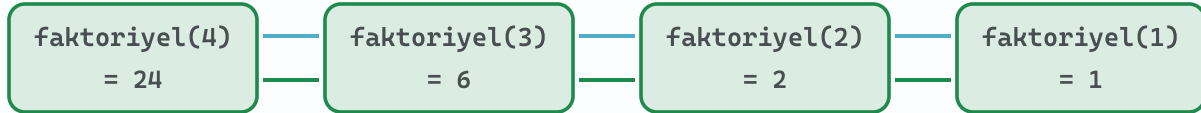
DEĞİŞKENLER

n = 4

dönüş = 24

ÇIKTI

faktoriyel(4): Kodu Adım Adım İzle



24 `main` 'e döndü ve yazdırıldı. 4 çağrı yapıldı; en derin anda yığında 5 çerçeve vardı.

Adım 12 / 12

```
1 static int faktoriyel(int n) {  
2     if (n == 0 || n == 1) {  
3         return 1;  
4     } else {  
5         return n * faktoriyel(n - 1);  
6     }  
7 }  
8  
9 public static void main(String[] args) {  
10     System.out.println(faktoriyel(4));  
11 }
```

ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

`n = -`

ÇIKTI

24

FAKTÖRİYEL VE TOPLAM

Toplam Bulma

```
int toplam(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return n + toplam(n - 1);  
    }  
}
```

1'den n'ye kadar sayıların toplamı:

`toplam(n) = n + toplam(n - 1)` ,

`toplam(1) = 1` .

```
toplam(5) = 5 + toplam(4)  
          = 5 + (4 + (3 + (2 + toplam(1))))  
          = 5 + (4 + (3 + (2 + 1)))  
          = 15
```

Faktöriyelle aynı kalıp; yalnızca $\times$ yerine $+$ ve temel durum farklı. `toplam(100)` = **5050**.

Dikkat: temel durum yalnızca `n == 1` .

`toplam(0)` ya da negatif bir sayı çağrılırsa `n` hiç 1 olmaz: 0, -1, -2, ... sonsuz özyineleme → `StackOverflowError` .

Güvenli sürüm: temel durumu `if (n <= 0) return 0;` yapın; tüm geçersiz girdiler de yakalanır.

Fibonacci Serisi

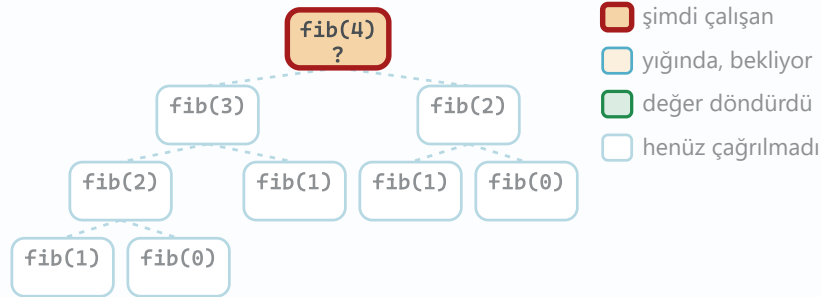
- 0 ile başlar, ardından 1 gelir: **$F(0) = 0$** , **$F(1) = 1$** .
- Genel formül: **$F(n) = F(n - 1) + F(n - 2)$** : her terim önceki iki terimin toplamı.
- Metot kendini **iki kez** çağırır: çağrılar bir **ağaç** oluşturur (ağaç özyinelemesi).

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

n	0	1	2	3	4	5	6	7	8	9	10	11	12
F(n)	0	1	1	2	3	5	8	13	21	34	55	89	144

Tavşan çoğalması problemiyle tanınır (Leonardo Fibonacci, 1202). Ardışık terimlerin oranı altın orana ($\approx 1,618$) yaklaşır: $144 / 89 \approx 1,618$.

fibonacci(4): Çağrı Ağacı ve Yiğın



`fibonacci(4)` : $n > 1$. Toplamın önce **sol** terimi hesaplanır: `fibonacci(3)` çağrılır.

Adım 1 / 17

```
1 static int fibonacci(int n) {  
2     if (n <= 1) {  
3         return n;  
4     } else {  
5         return fibonacci(n - 1) + fibonacci(n - 2);  
6     }  
7 }
```

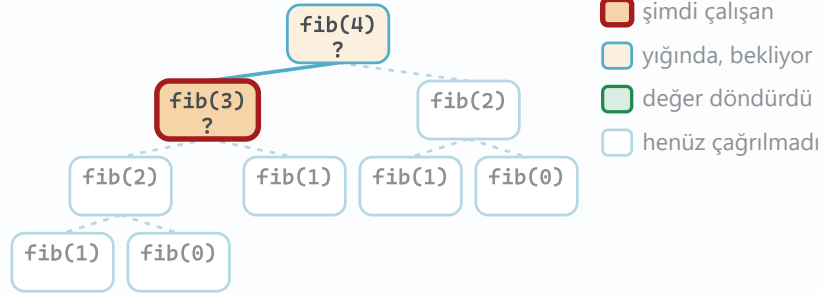
ÇAĞRI YIĞINI

`fibonacci(4)`

DEĞİŞKENLER

`n = 4`

fibonacci(4): Çağrı Ağacı ve Yiğın



`fibonacci(3)` : $n > 1$. Toplamın önce **sol** terimi hesaplanır: `fibonacci(2)` çağrılır.

Adım 2 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

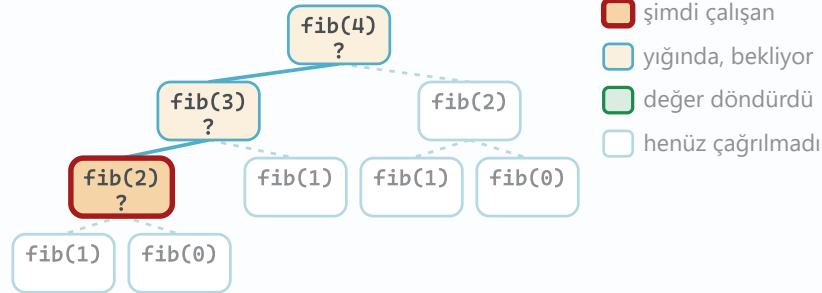
`fibonacci(3)`

`fibonacci(4)`

DEĞİŞKENLER

`n = 3`

fibonacci(4): Çağrı Ağacı ve Yiğın



`fibonacci(2)` : $n > 1$. Toplamın önce **sol** terimi hesaplanır: `fibonacci(1)` çağrılır.

Adım 3 / 17

```

1 static int fibonacci(int n) {
2   if (n <= 1) {
3     return n;
4   } else {
5     return fibonacci(n - 1) + fibonacci(n - 2);
6   }
7 }

```

ÇAĞRI YIĞINI

fibonacci(2)

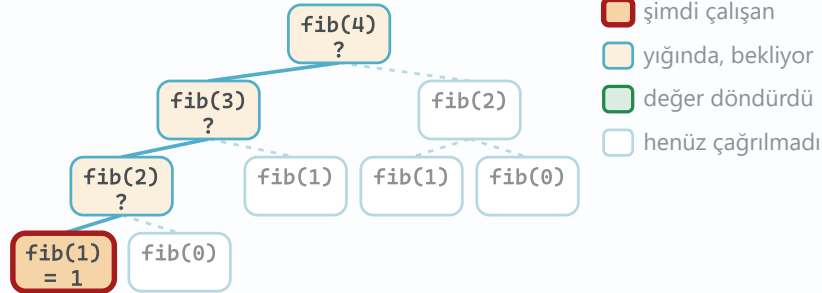
fibonacci(3)

fibonacci(4)

DEĞİŞKENLER

$n = 2$

fibonacci(4): Çağrı Ağacı ve Yiğın



`fibonacci(1)` : `n <= 1` → temel durum, **1** döner.

Adım 4 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

`fibonacci(1)`

`fibonacci(2)`

`fibonacci(3)`

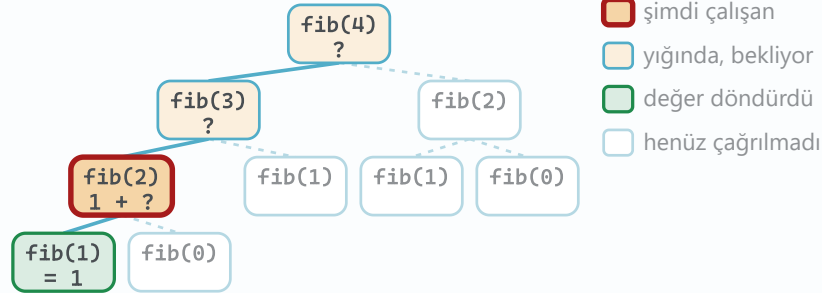
`fibonacci(4)`

DEĞİŞKENLER

`n = 1`

`dönüş = 1`

fibonacci(4): Çağrı Ağacı ve Yiğın



Sol terim 1 geldi. Şimdi **sağ** terim: `fibonacci(0)` çağrılır.

Adım 5 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

`fibonacci(2)`

`fibonacci(3)`

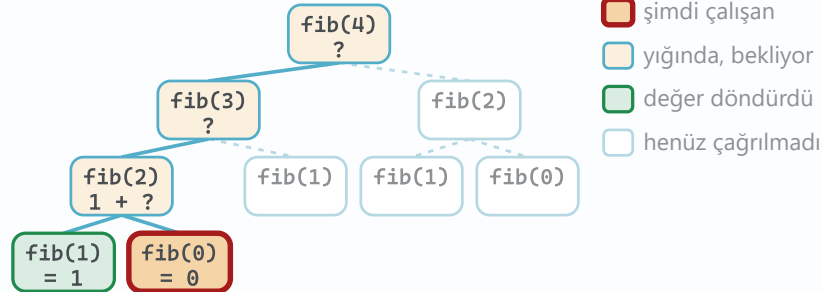
`fibonacci(4)`

DEĞİŞKENLER

`n = 2`

`f(n-1) = 1`

fibonacci(4): Çağrı Ağacı ve Yiğın



`fibonacci(0)` : `n <= 1` → temel durum, **0** döner.

Adım 6 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

`fibonacci(0)`

`fibonacci(2)`

`fibonacci(3)`

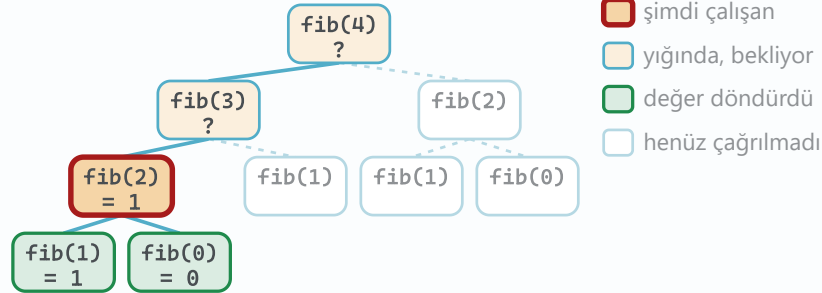
`fibonacci(4)`

DEĞİŞKENLER

`n = 0`

`dönüş = 0`

fibonacci(4): Çağrı Ağacı ve Yiğın



`1 + 0 = 1` döner.

Adım 7 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

fibonacci(2)

fibonacci(3)

fibonacci(4)

DEĞİŞKENLER

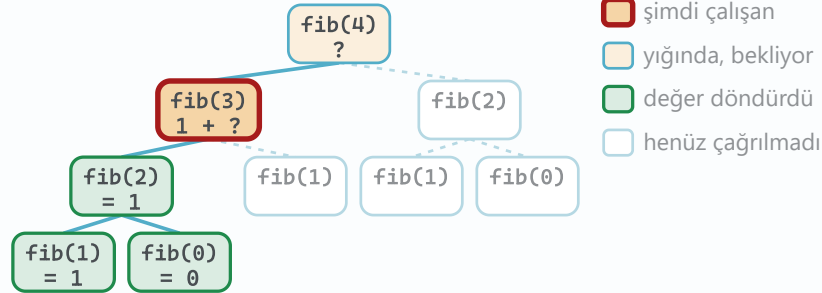
`n = 2`

`f(n-1) = 1`

`f(n-2) = 0`

`dönüş = 1`

fibonacci(4): Çağrı Ağacı ve Yiğın



Sol terim 1 geldi. Şimdi **sağ** terim: `fibonacci(1)` çağrılır.

Adım 8 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         ▶ return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }
  
```

ÇAĞRI YIĞINI

fibonacci(3)

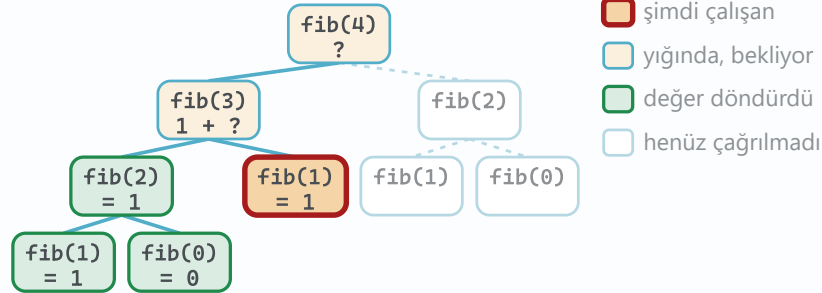
fibonacci(4)

DEĞİŞKENLER

$n = 3$

$f(n-1) = 1$

fibonacci(4): Çağrı Ağacı ve Yiğın



`fibonacci(1)` : `n <= 1` → temel durum, **1** döner.

Adım 9 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

`fibonacci(1)`

`fibonacci(3)`

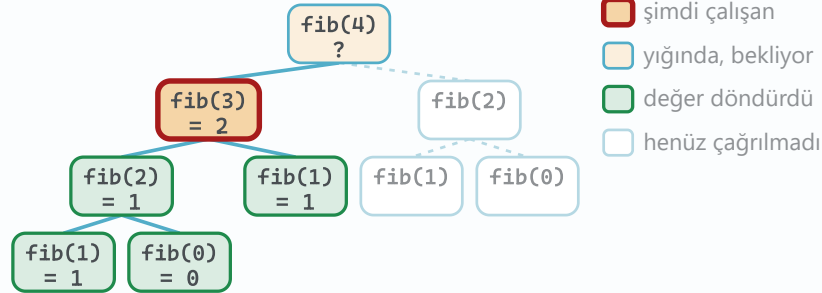
`fibonacci(4)`

DEĞİŞKENLER

`n = 1`

`dönüş = 1`

fibonacci(4): Çağrı Ağacı ve Yiğın



- şimdi çalışan
- yığında, bekliyor
- değer döndürdü
- henüz çağrılmadı

`1 + 1 = 2` döner.

Adım 10 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

`fibonacci(3)`

`fibonacci(4)`

DEĞİŞKENLER

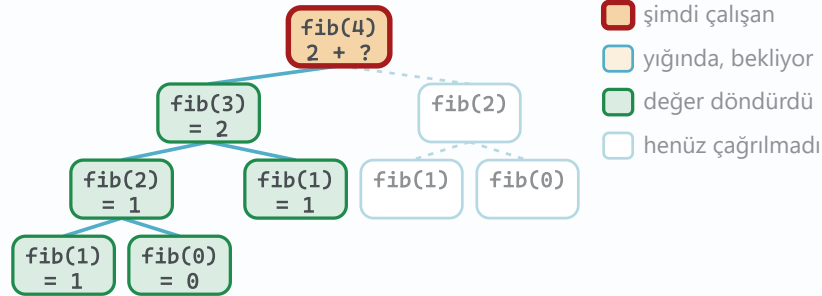
`n = 3`

`f(n-1) = 1`

`f(n-2) = 1`

`dönüş = 2`

fibonacci(4): Çağrı Ağacı ve Yiğın



- şimdi çalışan
- yığında, bekliyor
- değer döndürdü
- henüz çağrılmadı

Sol terim 2 geldi. Şimdi **sağ** terim: `fibonacci(2)` çağrılır.

Adım 11 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         ▶ return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

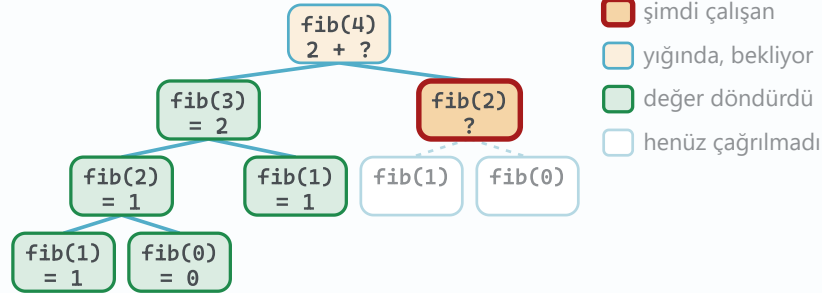
`fibonacci(4)`

DEĞİŞKENLER

`n = 4`

`f(n-1) = 2`

fibonacci(4): Çağrı Ağacı ve Yiğın



`fibonacci(2)` : $n > 1$. Toplamın önce **sol** terimi hesaplanır: `fibonacci(1)` çağrılır.

Adım 12 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

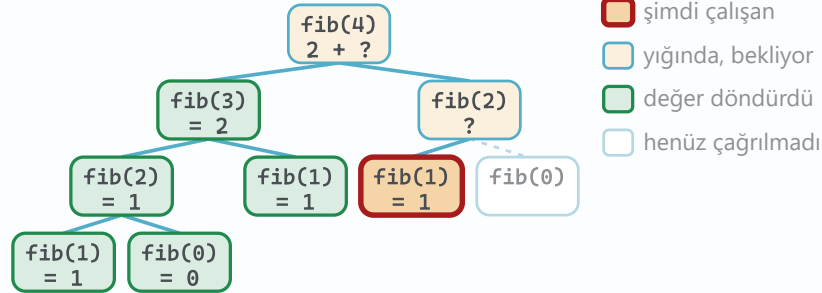
`fibonacci(2)`

`fibonacci(4)`

DEĞİŞKENLER

`n = 2`

fibonacci(4): Çağrı Ağacı ve Yiğın



`fibonacci(1)` : `n <= 1` → temel durum, 1 döner.

Adım 13 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

`fibonacci(1)`

`fibonacci(2)`

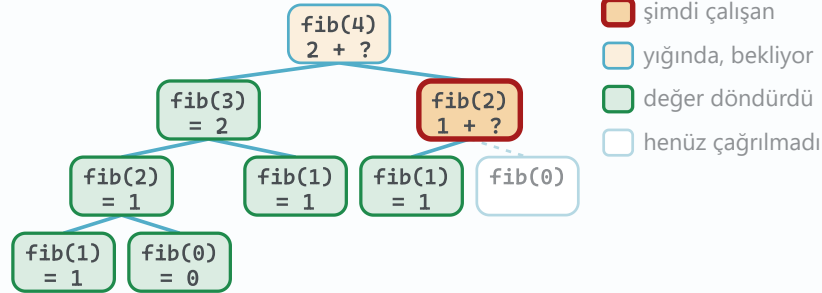
`fibonacci(4)`

DEĞİŞKENLER

`n = 1`

`dönüş = 1`

fibonacci(4): Çağrı Ağacı ve Yiğın



Sol terim 1 geldi. Şimdi **sağ** terim: `fibonacci(0)` çağrılır.

Adım 14 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

`fibonacci(2)`

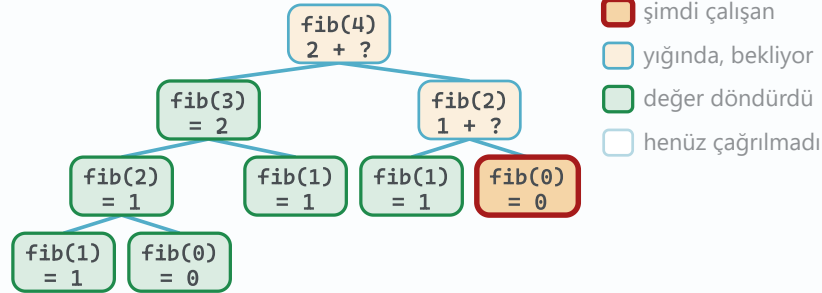
`fibonacci(4)`

DEĞİŞKENLER

`n = 2`

`f(n-1) = 1`

fibonacci(4): Çağrı Ağacı ve Yiğın



`fibonacci(0)` : `n <= 1` → temel durum, **0** döner.

Adım 15 / 17

```
1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }
```

ÇAĞRI YIĞINI

`fibonacci(0)`

`fibonacci(2)`

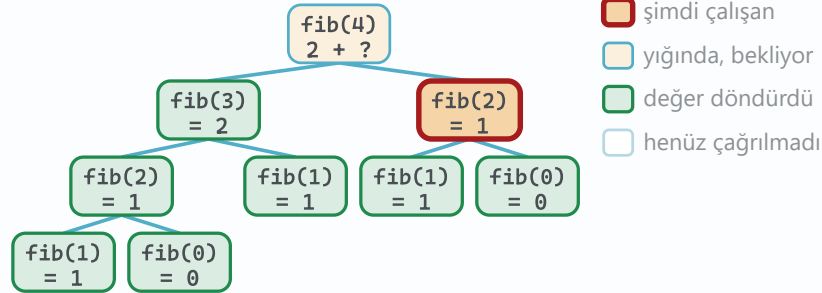
`fibonacci(4)`

DEĞİŞKENLER

`n = 0`

`dönüş = 0`

fibonacci(4): Çağrı Ağacı ve Yiğın



1 + 0 = 1 döner.

Adım 16 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

fibonacci(2)

fibonacci(4)

DEĞİŞKENLER

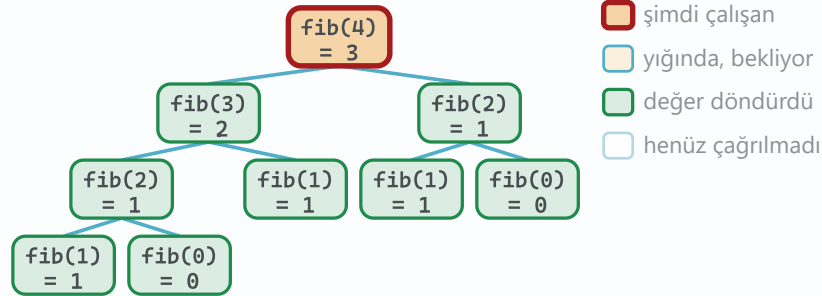
n = 2

f(n-1) = 1

f(n-2) = 0

dönüş = 1

fibonacci(4): Çağrı Ağacı ve Yiğın



$2 + 1 = 3$ döner. Toplam **9 çağrı**; `fibonacci(2)` iki kez hesaplandı.

Adım 17 / 17

```

1 static int fibonacci(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fibonacci(n - 1) + fibonacci(n - 2);
6     }
7 }

```

ÇAĞRI YIĞINI

`fibonacci(4)`

DEĞİŞKENLER

`n = 4`

`f(n-1) = 2`

`f(n-2) = 1`

`dönüş = 3`

SERİLER

Lucas Serisi

- 2 ile başlar, ardından 1 gelir: $L(0) = 2, L(1) = 1$.
- Genel formül Fibonacci ile aynı: $L(n) = L(n - 1) + L(n - 2)$.
- Yalnızca başlangıç değerleri farklıdır → **iki ayrı temel durum** gerekir.

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

n	0	1	2	3	4	5
L(n)	2	1	3	4	7	11
n	6	7	8	9	10	
L(n)	18	29	47	76	123	

İki seri birbirine bağlıdır: $L(n) = F(n - 1) + F(n + 1)$. Örn. $L(5) = F(4) + F(6) = 3 + 8 = 11$.

Catalan Serisi

- 1 ile başlar: $C(0) = 1, C(1) = 1$.
- Kapalı formül: $C(n) = (2n)! / ((n + 1)! \cdot n!)$.
- Kodun kullandığı özyinelemeli formül: $C(n) = \sum C(i) \cdot C(n - 1 - i), i = 0 \dots n - 1$.
- Özyineleme bir **döngünün içinde** yapılır: her $C(n)$, daha küçük tüm C değerlerinden kurulur.

```
long catalan(int n) {  
    if (n <= 1) {  
        return 1;  
    }  
    long sonuc = 0;  
    for (int i = 0; i < n; i++) {  
        sonuc += catalan(i) * catalan(n - i - 1);  
    }  
    return sonuc;  
}
```

Değerler:

1	1	2	5	14	42	132
429	1430					

Ne sayar? n çift parantezin doğru dizilişlerini. $n = 3 \rightarrow C(3) = 5$:

((()))	((()()))	((())())
()((()))	()()()()	

Ayrıca $n + 1$ sayının çarpım sırasını belirleme ve n düğümlü ikili ağaç sayısı.

Bell Serisi

- 1 ile başlar: $B(0) = 1, B(1) = 1$.
- Genel formül: $B(n) = \sum C(n-1, k) \cdot B(k)$, $k = 0 \dots n-1$;
burada $C(n, k)$ kombinasyondur.

```
int bell(int n) {
    if (n == 0) { return 1; }
    int sayi = 0;
    for (int k = 0; k < n; k++) {
        sayi += katsayi(n - 1, k) * bell(k);
    }
    return sayi;
}

int katsayi(int n, int k) {          // C(n, k)
    return faktoriyel(n) / (faktoriyel(k) * faktoriyel(n - k));
}
```

Değerler:

1	1	2	5	15	52
203	877				

Ne sayar? n elemanlı bir kümeyi gruplara ayırma yollarını. $\{a, b, c\} \rightarrow B(3) = 5$:

abc	a bc	b ac	c ab
a b c			

katsayi, int döndüren faktoriyel 'i kullanır; 13! taşar. Bu yüzden kod yalnızca $n \leq 13$ için doğrudur.

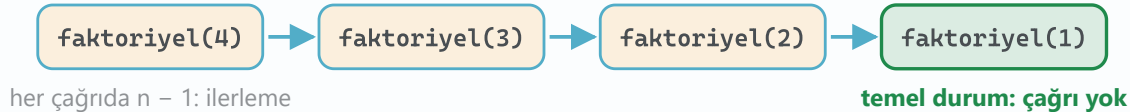
İki Temel Kural

Kural 1: Temel durum (Base Case)

Sonsuz özyinelemeye girmemek için en az bir temel duruma ihtiyaç vardır. Örneğin n sayısı 0 veya 1 olduğunda metot kendini çağırmadan sonlanmalıdır.

Kural 2: İlerleme sağlama

Her özyinelemeli çağrı temel duruma doğru **ilerlemelidir**: parametre her çağrıda azaltılır (ya da artırılır), dizgi kısalır, aralık daralır.



- **Sonsuz özyineleme** (temel durum yok ya da hiç ulaşılamıyor) ve **yetersiz ilerleme** (parametre değişmiyor) gibi hatalardan kaçınin.
- Her özyinelemeli çağrı problemin **daha küçük bir alt kümesine** odaklanmalıdır.
- Temel durumu her zaman özyinelemeli çağrıdan **önce** kontrol edin.
- Tüm olası girdileri düşünün: negatif sayı, 0, boş dizgi temel duruma ulaşıyor mu?

Sonsuz Özyineleme

```
public class Toplam {  
    static int toplam(int n) {  
        if (n == 1) {  
            return 1;  
        } else {  
            return n + toplam(n - 1);    // satır 6  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println(toplam(0));  
    }  
}
```

```
Exception in thread "main" java.lang.StackOverflowError  
    at Toplam.toplam(Toplam.java:6)  
    at Toplam.toplam(Toplam.java:6)  
    at Toplam.toplam(Toplam.java:6)  
    ...
```

`toplam(0)` : $n = 0, -1, -2, \dots$ hiçbir zaman 1 olmaz. Her çağrı yeni bir çerçeve açar; yaklaşık on binlerce çağrıdan sonra yığın dolar ve JVM programı durdurur.

Derleyici bu hatayı **yakalamaz**; çalışma zamanında ortaya çıkar. Yığın izindeki (stack trace) **aynı satırın tekrar tekrar** görünmesi sonsuz özyinelemenin işaretidir.

Çözüm: temel durumu genişletin (`n <= 1`), girdiyi kontrol edin ya da derinliği çok büyük olan problemler için döngü kullanın.

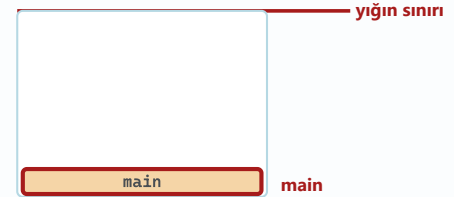
toplam(0): Yığın Nasıl Dolar?

```
1 public class Toplam {  
2     static int toplam(int n) {  
3         if (n == 1) {  
4             return 1;  
5         } else {  
6             return n + toplam(n - 1);    // satır 6  
7         }  
8     }  
9     public static void main(String[] args) {  
10        ▶ System.out.println(toplam(0));  
11    }  
12 }
```

DEĞİŞKENLER

n = -

ÇIKTI



main, toplam(0) 'ı çağırır. Temel durum yalnızca n == 1 .

Adım 1 / 6

ÇAĞRI YIĞINI

main()

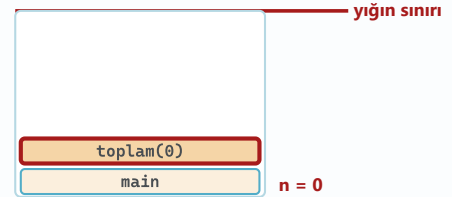
toplam(0): Yığın Nasıl Dolar?

```
1 public class Toplam {  
2     static int toplam(int n) {  
3         if (n == 1) {  
4             return 1;  
5         } else {  
6             return n + toplam(n - 1); // satır 6  
7         }  
8     }  
9     public static void main(String[] args) {  
10        System.out.println(toplam(0));  
11    }  
12 }
```

DEĞİŞKENLER

n = 0

ÇIKTI



n = 0 : 0 == 1 yanlış → 0 + toplam(-1)
hesaplanmalı; yeni çağrı.

Adım 2 / 6

ÇAĞRI YIĞINI

toplam(0)

main()

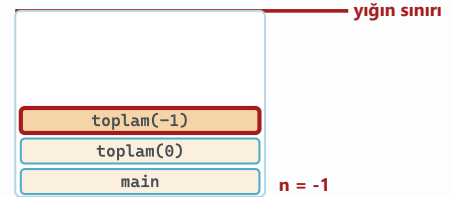
toplam(0): Yığın Nasıl Dolar?

```
1 public class Toplam {  
2     static int toplam(int n) {  
3         if (n == 1) {  
4             return 1;  
5         } else {  
6             return n + toplam(n - 1); // satır 6  
7         }  
8     }  
9     public static void main(String[] args) {  
10        System.out.println(toplam(0));  
11    }  
12 }
```

DEĞİŞKENLER

n = -1

ÇIKTI



n = -1 : yine 1 değil. n her çağrıda **uzaklaşıyor**: temel duruma hiç ulaşılmayacak.

Adım 3 / 6

ÇAĞRI YIĞINI

toplam(-1)

toplam(0)

main()

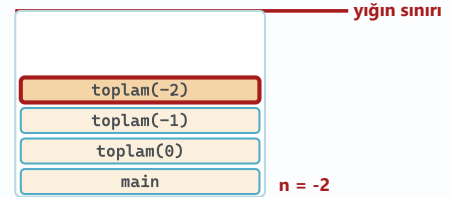
toplam(0): Yığın Nasıl Dolar?

```
1 public class Toplam {  
2     static int toplam(int n) {  
3         if (n == 1) {  
4             return 1;  
5         } else {  
6             return n + toplam(n - 1); // satır 6  
7         }  
8     }  
9     public static void main(String[] args) {  
10        System.out.println(toplam(0));  
11    }  
12 }
```

DEĞİŞKENLER

n = -2

ÇIKTI



Her çağrı yığına yeni bir çerçeve ekler; hiçbiri dönemez, çünkü hepsi bir sonrakinin sonucunu bekler.

Adım 4 / 6

ÇAĞRI YIĞINI

toplam(-2)

toplam(-1)

toplam(0)

main()

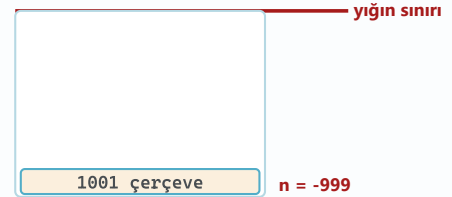
toplam(0): Yığın Nasıl Dolar?

```
1 public class Toplam {  
2     static int toplam(int n) {  
3         if (n == 1) {  
4             return 1;  
5         } else {  
6             return n + toplam(n - 1); // satır 6  
7         }  
8     }  
9     public static void main(String[] args) {  
10        System.out.println(toplam(0));  
11    }  
12 }
```

DEĞİŞKENLER

n = -999

ÇIKTI



Bin çağrı sonra yığında 1001 çerçeve var; hâlâ hata yok.

Adım 5 / 6

ÇAĞRI YIĞINI

toplam(-999)

...

toplam(0)

main()

toplam(0): Yığın Nasıl Dolar?

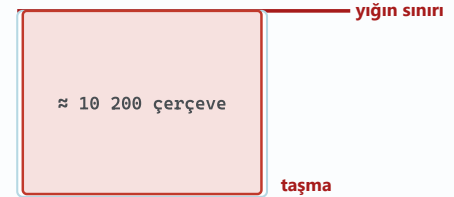
```
1 public class Toplam {  
2     static int toplam(int n) {  
3         if (n == 1) {  
4             return 1;  
5         } else {  
6             return n + toplam(n - 1); // satır 6  
7         }  
8     }  
9     public static void main(String[] args) {  
10        System.out.println(toplam(0));  
11    }  
12 }
```

DEĞİŞKENLER

—

ÇIKTI

```
Exception in thread "main"  
java.lang.StackOverflowError  
    at Toplam.toplam(Toplam.java:6)  
    ...
```



Yığın için ayrılan bellek doldu: JVM `StackOverflowError` fırlatır. Bu makinede ölçülen derinlik $\approx 10\,200$ çağrıdır; sayı JVM ayarına göre değişir.

Adım 6 / 6

ÇAĞRI YIĞINI

toplam(-10200)

...

toplam(0)

main()

Özyinelemeli Düşünme

1. **Küçült:** problemi kendisinin daha küçük bir kopyası cinsinden ifade et.
2. **Temel durumu bul:** cevabı doğrudan bilinen en küçük girdi.
3. **Güven:** küçük kopyanın doğru çözüldüğünü **varsay** (inanç sıçraması, leap of faith); çağrının içini kafanda açma.
4. **Birleştir:** küçük cevaptan büyük cevabı kur.

Örnek: bir sayının basamak sayısı.

- 48729'un basamak sayısı = 1 + 4872'nin basamak sayısı.
- Tek basamaklı sayı ($n < 10$) $\rightarrow 1$.

```
int basamakSayisi(int n) {  
    if (n < 10) {                                // temel durum  
        return 1;  
    }  
    return 1 + basamakSayisi(n / 10); // küçült + birleştir  
}
```

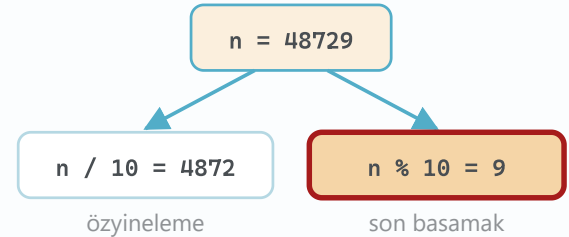
`basamakSayisi(48729)` $\rightarrow 5 \cdot$ `basamakSayisi(7)` $\rightarrow 1$

ÖRNEKLER

Rakamlar Toplamı

```
int rakamlarToplami(int n) {  
    if (n < 0) {  
        return rakamlarToplami(-n);  
    } else if (n < 10) {  
        return n;  
    } else {  
        return rakamlarToplami(n / 10) + (n % 10);  
    }  
}
```

- Negatif sayı önce pozitif e çevrilir (satır 2–3).
- Tek basamaklı sayı kendisidir: temel durum (satır 4–5).
- Aksi halde: son basamak + kalan basamakların toplamı.



$n / 10$ tam sayı bölmesi son basamağı **atar**; $n \% 10$ son basamağı **verir**. Her çağrıda sayı bir basamak kısalır → ilerleme garantilidir.

ÖRNEKLER

rakamlarToplami(-48729) Açılımı

```
rakamlarToplami(-48729)
= rakamlarToplami(48729)
= rakamlarToplami(4872) + 9
= (rakamlarToplami(487) + 2) + 9
= ((rakamlarToplami(48) + 7) + 2) + 9
= (((rakamlarToplami(4) + 8) + 7) + 2) + 9
= (((4 + 8) + 7) + 2) + 9
= ((12 + 7) + 2) + 9
= (19 + 2) + 9
= 21 + 9
= 30
```

İlk çağrı işareti düzeltir; sonra her adımda son basamak (9, 2, 7, 8) **bekleyen toplama** olarak dışarıda kalır.

Temel durum `rakamlarToplami(4)`
→ 4. Dönüşte toplamlar içten dışa yapılır: $4 + 8 + 7 + 2 + 9 = 30$.

Toplam 6 çağrı yapılır (1 işaret + 5 basamak).

rakamlarToplami(-357) : Kodu Adım Adım İzle

r(-357)

r(357)

r(35)

r(3)

main , rakamlarToplami(-357) 'yi çağırır (şekilde kısaca **r**).

Adım 1 / 9

```
1 static int rakamlarToplami(int n) {  
2     if (n < 0) {  
3         return rakamlarToplami(-n);  
4     } else if (n < 10) {  
5         return n;  
6     } else {  
7         return rakamlarToplami(n / 10) + (n % 10);  
8     }  
9 }  
10  
11 public static void main(String[] args) {  
12     ▶ System.out.println(rakamlarToplami(-357));  
13 }
```

ÇAĞRI YIĞINI

main()

DEĞİŞKENLER

n = -

ÇIKTI

rakamlarToplami(-357) : Kodu Adım Adım İzle

r(-357)
n < 0

r(357)

r(35)

r(3)

n = -357 : negatif → ilk dal. İşaret düzeltilip r(357) çağrılır; bu çerçeveye yalnızca sonucu aynen döndürecek.

Adım 2 / 9

```
1 static int rakamlarToplami(int n) {  
2     if (n < 0) {  
3         return rakamlarToplami(-n);  
4     } else if (n < 10) {  
5         return n;  
6     } else {  
7         return rakamlarToplami(n / 10) + (n % 10);  
8     }  
9 }  
10  
11 public static void main(String[] args) {  
12     System.out.println(rakamlarToplami(-357));  
13 }
```

ÇAĞRI YIĞINI

r(-357)

main()

DEĞİŞKENLER

n = -357

ÇIKTI

rakamlarToplami(-357) : Kodu Adım Adım İzle

r(-357)
→ r(357)

r(357)
r(35) + 7

r(35)

r(3)

n = 357 : 10'dan büyük. $357 / 10 = 35$ ile çağrı yapılır; son basamak **7** bekleyen toplama olarak kalır.

Adım 3 / 9

```

1 static int rakamlarToplami(int n) {
2     if (n < 0) {
3         return rakamlarToplami(-n);
4     } else if (n < 10) {
5         return n;
6     } else {
7         ▶ return rakamlarToplami(n / 10) + (n % 10);
8     }
9 }
10
11 public static void main(String[] args) {
12     System.out.println(rakamlarToplami(-357));
13 }

```

ÇAĞRI YIĞINI

r(357)

r(-357)

main()

DEĞİŞKENLER

n = 357

ÇIKTI

rakamlarToplami(-357) : Kodu Adım Adım İzle

r(-357)
→ r(357)

r(357)
r(35) + 7

r(35)
r(3) + 5

r(3)

n = 35 : r(3) çağrılır, 5 bekler.

Adım 4 / 9

```
1 static int rakamlarToplami(int n) {  
2     if (n < 0) {  
3         return rakamlarToplami(-n);  
4     } else if (n < 10) {  
5         return n;  
6     } else {  
7         ▶ return rakamlarToplami(n / 10) + (n % 10);  
8     }  
9 }  
10  
11 public static void main(String[] args) {  
12     System.out.println(rakamlarToplami(-357));  
13 }
```

ÇAĞRI YIĞINI

r(35)

r(357)

r(-357)

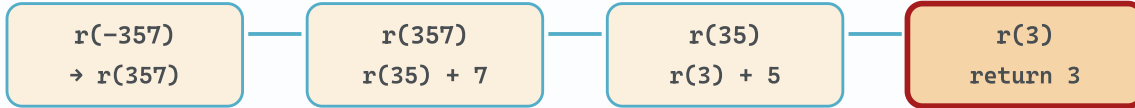
main()

DEĞİŞKENLER

n = 35

ÇIKTI

rakamlarToplami(-357): Kodu Adım Adım İzle



`n = 3 < 10` : **temel durum**, 3 döner. Artık çerçeveler geri dönmeye başlar.

Adım 5 / 9

```
1 static int rakamlarToplami(int n) {  
2     if (n < 0) {  
3         return rakamlarToplami(-n);  
4     } else if (n < 10) {  
5         return n;  
6     } else {  
7         return rakamlarToplami(n / 10) + (n % 10);  
8     }  
9 }  
10  
11 public static void main(String[] args) {  
12     System.out.println(rakamlarToplami(-357));  
13 }
```

ÇAĞRI YIĞINI

r(3)

r(35)

r(357)

r(-357)

main()

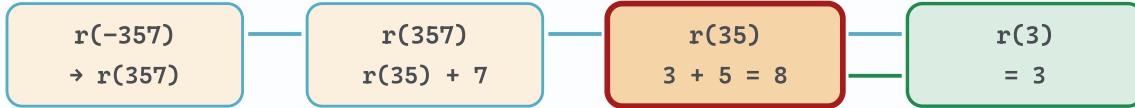
DEĞİŞKENLER

n = 3

dönüş = 3

ÇIKTI

rakamlarToplami(-357) : Kodu Adım Adım İzle



`r(35)` 3'ü aldı: `3 + 5 = 8` döner.

Adım 6 / 9

```
1 static int rakamlarToplami(int n) {  
2     if (n < 0) {  
3         return rakamlarToplami(-n);  
4     } else if (n < 10) {  
5         return n;  
6     } else {  
7         ▶ return rakamlarToplami(n / 10) + (n % 10);  
8     }  
9 }  
10  
11 public static void main(String[] args) {  
12     System.out.println(rakamlarToplami(-357));  
13 }
```

ÇAĞRI YIĞINI

`r(35)`

`r(357)`

`r(-357)`

`main()`

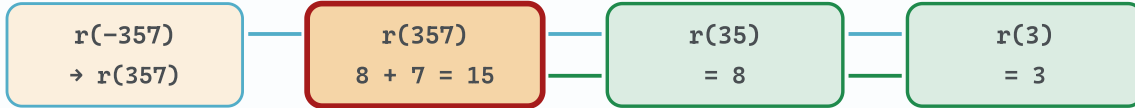
DEĞİŞKENLER

`n = 35`

`dönüş = 8`

ÇIKTI

rakamlarToplami(-357): Kodu Adım Adım İzle



`r(357)` 8'i aldı: `8 + 7 = 15` döner.

Adım 7 / 9

```
1 static int rakamlarToplami(int n) {  
2     if (n < 0) {  
3         return rakamlarToplami(-n);  
4     } else if (n < 10) {  
5         return n;  
6     } else {  
7         ▶ return rakamlarToplami(n / 10) + (n % 10);  
8     }  
9 }  
10  
11 public static void main(String[] args) {  
12     System.out.println(rakamlarToplami(-357));  
13 }
```

ÇAĞRI YIĞINI

`r(357)`

`r(-357)`

`main()`

DEĞİŞKENLER

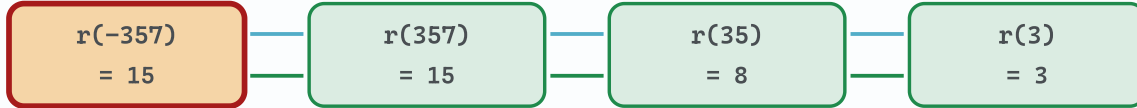
`n = 357`

`dönüş = 15`

ÇIKTI

`15`

rakamlarToplami(-357): Kodu Adım Adım İzle



`r(-357)` gelen 15'i değiştirmeden döndürür.

Adım 8 / 9

```
1 static int rakamlarToplami(int n) {  
2     if (n < 0) {  
3         return rakamlarToplami(-n);  
4     } else if (n < 10) {  
5         return n;  
6     } else {  
7         return rakamlarToplami(n / 10) + (n % 10);  
8     }  
9 }  
10  
11 public static void main(String[] args) {  
12     System.out.println(rakamlarToplami(-357));  
13 }
```

ÇAĞRI YIĞINI

`r(-357)`

`main()`

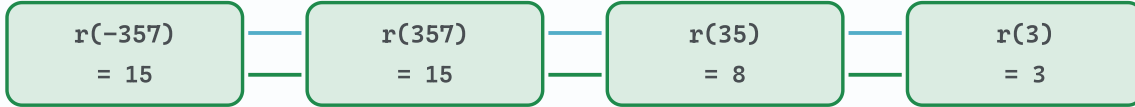
DEĞİŞKENLER

`n = -357`

`dönüş = 15`

ÇIKTI

rakamlarToplami(-357): Kodu Adım Adım İzle



15 yazdırılır: $3 + 5 + 7 = 15$. Dört çağrı yapıldı (1 işaret + 3 basamak).

Adım 9 / 9

```
1 static int rakamlarToplami(int n) {  
2     if (n < 0) {  
3         return rakamlarToplami(-n);  
4     } else if (n < 10) {  
5         return n;  
6     } else {  
7         return rakamlarToplami(n / 10) + (n % 10);  
8     }  
9 }  
10  
11 public static void main(String[] args) {  
12     ▶ System.out.println(rakamlarToplami(-357));  
13 }
```

ÇAĞRI YIĞINI

main()

DEĞİŞKENLER

n = -

ÇIKTI

15

ÖRNEKLER

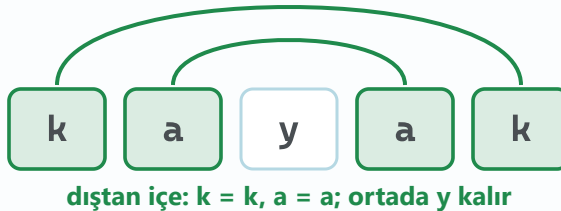
Palindrom Kontrolü

```
static boolean palindromMu(String s) {  
    if (s.length() <= 1) {  
        return true;  
    } else if (s.charAt(0) == s.charAt(s.length() - 1)) {  
        return palindromMu(s.substring(1, s.length() - 1));  
    }  
    return false;  
}
```

palindromMu("kayak") → **true** .

palindromMu("abba") → **true** .

palindromMu("java") → **false**



- **Palindrom:** tersten okunuşu aynı olan dizgi (kayak, ey edip adanada pide ye).
- `s.charAt(i)` : i. karakter (0'dan başlar).
- `s.substring(1, s.length() - 1)` : ilk ve son harfi atılmış dizgi.
- İki temel durum: uzunluk ≤ 1 → **true**; uçlar farklı → **false**.
- Büyük-küçük harfe duyarlıdır: "Kayak" → false.

palindromMu("racecar"): Kodu Adım Adım İzle

0	1	2	3	4	5	6
r	a	c	e	c	a	r

s = "racecar" (uzunluk 7)

s = "racecar": uzunluk 7 > 1. İlk harf 'r' ile son harf 'r' karşılaştırılır → eşit.

Adım 1 / 10

```
1 static boolean palindromMu(String s) {  
2     if (s.length() <= 1) {  
3         return true;  
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {  
5         return palindromMu(s.substring(1, s.length() - 1));  
6     }  
7     return false;  
8 }
```

ÇAĞRI YIĞINI

palindromMu("racecar")

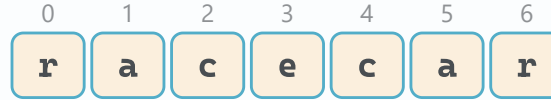
DEĞİŞKENLER

s = "racecar"

s.charAt(0) = 'r'

son harf = 'r'

palindromMu("racecar"): Kodu Adım Adım İzle



s = "racecar" (uzunluk 7)

Uçlar eşleşti; iç kısım `s.substring(1, 6)` = "aceca" için yeni çağrı yapılır.

Adım 2 / 10

```
1 static boolean palindromMu(String s) {  
2     if (s.length() <= 1) {  
3         return true;  
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {  
5         return palindromMu(s.substring(1, s.length() - 1));  
6     }  
7     return false;  
8 }
```

ÇAĞRI YIĞINI

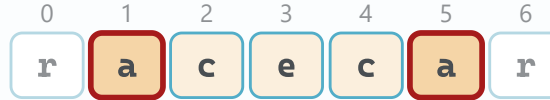
palindromMu("racecar")

DEĞİŞKENLER

s = "racecar"

substring = "aceca"

palindromMu("racecar"): Kodu Adım Adım İzle



s = "aceca" (uzunluk 5)

s = "aceca": uzunluk 5 > 1. İlk harf 'a' ile son harf 'a' karşılaştırılır → eşit.

Adım 3 / 10

```
1 static boolean palindromMu(String s) {  
2     if (s.length() <= 1) {  
3         return true;  
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {  
5         return palindromMu(s.substring(1, s.length() - 1));  
6     }  
7     return false;  
8 }
```

ÇAĞRI YIĞINI

palindromMu("aceca")

palindromMu("racecar")

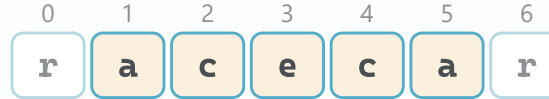
DEĞİŞKENLER

s = "aceca"

s.charAt(0) = 'a'

son harf = 'a'

palindromMu("racecar"): Kodu Adım Adım İzle



s = "aceca" (uzunluk 5)

Uçlar eşleşti; iç kısım `s.substring(1, 4)` = "cec" için yeni çağrı yapılır.

Adım 4 / 10

```
1 static boolean palindromMu(String s) {  
2     if (s.length() <= 1) {  
3         return true;  
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {  
5         return palindromMu(s.substring(1, s.length() - 1));  
6     }  
7     return false;  
8 }
```

ÇAĞRI YIĞINI

palindromMu("aceca")

palindromMu("racecar")

DEĞİŞKENLER

s = "aceca"

substring = "cec"

palindromMu("racecar"): Kodu Adım Adım İzle



s = "cec" (uzunluk 3)

s = "cec": uzunluk 3 > 1. İlk harf 'c' ile son harf 'c' karşılaştırılır → **eşit**.

Adım 5 / 10

```
1 static boolean palindromMu(String s) {  
2     if (s.length() <= 1) {  
3         return true;  
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {  
5         return palindromMu(s.substring(1, s.length() - 1));  
6     }  
7     return false;  
8 }
```

ÇAĞRI YIĞINI

palindromMu("cec")

palindromMu("aceca")

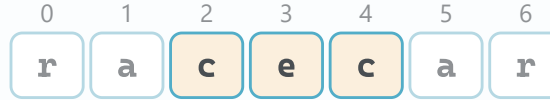
palindromMu("racecar")

DEĞİŞKENLER

s = "cec" s.charAt(0) = 'c'

son harf = 'c'

palindromMu("racecar"): Kodu Adım Adım İzle



s = "cec" (uzunluk 3)

Uçlar eşleşti; iç kısım `s.substring(1, 2)` = "e" için yeni çağrı yapılır.

Adım 6 / 10

```
1 static boolean palindromMu(String s) {  
2     if (s.length() <= 1) {  
3         return true;  
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {  
5         return palindromMu(s.substring(1, s.length() - 1));  
6     }  
7     return false;  
8 }
```

ÇAĞRI YIĞINI

palindromMu("cec")

palindromMu("aceca")

palindromMu("racecar")

DEĞİŞKENLER

s = "cec"

substring = "e"

palindromMu("racecar"): Kodu Adım Adım İzle



`s = "e"` (uzunluk 1)

Uzunluk $1 \leq 1 \rightarrow$ **temel durum**: tek harfli dizgi palindromdur, **true** döner.

Adım 7 / 10

```
1 static boolean palindromMu(String s) {  
2     if (s.length() <= 1) {  
3         return true;  
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {  
5         return palindromMu(s.substring(1, s.length() - 1));  
6     }  
7     return false;  
8 }
```

ÇAĞRI YIĞINI

palindromMu("e")

palindromMu("cec")

palindromMu("aceca")

palindromMu("racecar")

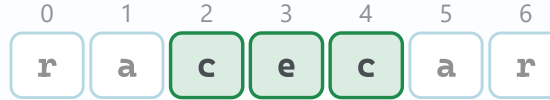
DEĞİŞKENLER

`s = "e"`

`s.length() = 1`

`dönüş = true`

palindromMu("racecar"): Kodu Adım Adım İzle



s = "cec" (uzunluk 3)

palindromMu("cec") çerçevesi true aldı ve aynen **true** döndürür.

Adım 8 / 10

```
1 static boolean palindromMu(String s) {  
2     if (s.length() <= 1) {  
3         return true;  
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {  
5         return palindromMu(s.substring(1, s.length() - 1));  
6     }  
7     return false;  
8 }
```

ÇAĞRI YIĞINI

palindromMu("cec")

palindromMu("aceca")

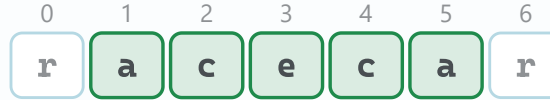
palindromMu("racecar")

DEĞİŞKENLER

s = "cec"

dönüş = true

palindromMu("racecar"): Kodu Adım Adım İzle



s = "aceca" (uzunluk 5)

palindromMu("aceca") çerçevesi true aldı ve aynen **true** döndürür.

Adım 9 / 10

```
1 static boolean palindromMu(String s) {  
2     if (s.length() <= 1) {  
3         return true;  
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {  
5         return palindromMu(s.substring(1, s.length() - 1));  
6     }  
7     return false;  
8 }
```

ÇAĞRI YIĞINI

palindromMu("aceca")

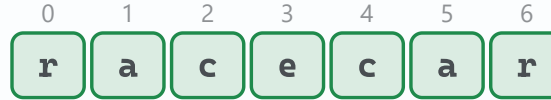
palindromMu("racecar")

DEĞİŞKENLER

s = "aceca"

dönüş = true

palindromMu("racecar"): Kodu Adım Adım İzle



s = "racecar" (uzunluk 7)

palindromMu("racecar") çerçevesi true aldı ve aynen **true** döndürür. Sonuç: "racecar" bir **palindromdur**.

Adım 10 / 10

```
1 static boolean palindromMu(String s) {  
2     if (s.length() <= 1) {  
3         return true;  
4     } else if (s.charAt(0) == s.charAt(s.length() - 1)) {  
5         return palindromMu(s.substring(1, s.length() - 1));  
6     }  
7     return false;  
8 }
```

ÇAĞRI YIĞINI

palindromMu("racecar")

DEĞİŞKENLER

s = "racecar"

dönüş = true

ÖRNEKLER

En Büyük Ortak Bölen

```
static int gcd(int x, int y) {  
    if (y == 0) {  
        return x;  
    }  
    return gcd(y, x % y);  
}
```

- **Öklid algoritması:** $EBOB(x, y) = EBOB(y, x \bmod y)$.
- Temel durum: $y = 0$ ise $EBOB = x$.
- Kalan her adımda küçülür $\rightarrow$ mutlaka 0'a ulaşır.
- `gcd(93, 444)` : ilk adım `gcd(444, 93)` olur; sıra önemsizdir.

```
gcd(444, 93)  
  = gcd(93, 72)    // 444 % 93 = 72  
  = gcd(72, 21)    // 93 % 72 = 21  
  = gcd(21, 9)     // 72 % 21 = 9  
  = gcd(9, 3)      // 21 % 9 = 3  
  = gcd(3, 0)      // 9 % 3 = 0  
  = 3
```

gcd (greatest common divisor) = **EBOB**.

`gcd(48, 18)` $\rightarrow$ 6.

gcd(444, 93): Kodu Adım Adım İzle

```
1 static int gcd(int x, int y) {  
▶   if (y == 0) {  
3       return x;  
4   }  
▶   return gcd(y, x % y);  
6 }
```

ÇAĞRI YIĞINI

gcd(444, 93)

çağrı

	x	y	x % y
gcd(444, 93)	444	93	72

y = 93 $\neq$ 0 . Kalan 444 % 93 = 72 ; yeni çağrı gcd(93, 72) .

Adım 1 / 7

DEĞİŞKENLER

x = 444

y = 93

x % y = 72

gcd(444, 93): Kodu Adım Adım İzle

```
1 static int gcd(int x, int y) {  
▶   if (y == 0) {  
3       return x;  
4   }  
▶   return gcd(y, x % y);  
6 }
```

çağrı

	x	y	x % y
gcd(444, 93)	444	93	72
gcd(93, 72)	93	72	21

$y = 72 \neq 0$. Kalan $93 \% 72 = 21$; yeni çağrı `gcd(72, 21)` .

Adım 2 / 7

ÇAĞRI YIĞINI

gcd(93, 72)

gcd(444, 93)

DEĞİŞKENLER

x = 93

y = 72

x % y = 21

gcd(444, 93): Kodu Adım Adım İzle

```

1 static int gcd(int x, int y) {
2     if (y == 0) {
3         return x;
4     }
5     return gcd(y, x % y);
6 }

```

çağrı	x	y	x % y
gcd(444, 93)	444	93	72
gcd(93, 72)	93	72	21
gcd(72, 21)	72	21	9

y = 21 ≠ 0 . Kalan 72 % 21 = 9 ; yeni çağrı gcd(21, 9) .

Adım 3 / 7

ÇAĞRI YIĞINI

gcd(72, 21)

gcd(93, 72)

gcd(444, 93)

DEĞİŞKENLER

x = 72

y = 21

x % y = 9

gcd(444, 93): Kodu Adım Adım İzle

```
1 static int gcd(int x, int y) {  
▶   if (y == 0) {  
3       return x;  
4   }  
▶   return gcd(y, x % y);  
6 }
```

çağrı	x	y	x % y
gcd(444, 93)	444	93	72
gcd(93, 72)	93	72	21
gcd(72, 21)	72	21	9
gcd(21, 9)	21	9	3

y = 9 ≠ 0 . Kalan 21 % 9 = 3 ; yeni çağrı gcd(9, 3) .

Adım 4 / 7

ÇAĞRI YIĞINI

gcd(21, 9)

gcd(72, 21)

gcd(93, 72)

gcd(444, 93)

DEĞİŞKENLER

x = 21

y = 9

x % y = 3

gcd(444, 93): Kodu Adım Adım İzle

```

1 static int gcd(int x, int y) {
2     if (y == 0) {
3         return x;
4     }
5     return gcd(y, x % y);
6 }

```

çağrı	x	y	x % y
gcd(444, 93)	444	93	72
gcd(93, 72)	93	72	21
gcd(72, 21)	72	21	9
gcd(21, 9)	21	9	3
gcd(9, 3)	9	3	0

$y = 3 \neq 0$. Kalan $9 \% 3 = 0$; yeni çağrı `gcd(3, 0)` .

Adım 5 / 7

ÇAĞRI YIĞINI

gcd(9, 3)

gcd(21, 9)

gcd(72, 21)

gcd(93, 72)

gcd(444, 93)

DEĞİŞKENLER

x = 9

y = 3

x % y = 0

gcd(444, 93): Kodu Adım Adım İzle

```

1 static int gcd(int x, int y) {
2     if (y == 0) {
3         return x;
4     }
5     return gcd(y, x % y);
6 }

```

çağrı	x	y	x % y
gcd(444, 93)	444	93	72
gcd(93, 72)	93	72	21
gcd(72, 21)	72	21	9
gcd(21, 9)	21	9	3
gcd(9, 3)	9	3	0
gcd(3, 0)	3	0	—

y == 0 → temel durum: x = 3 döner. EBOB(444, 93) = 3.

Adım 6 / 7

ÇAĞRI YIĞINI

gcd(3, 0)

gcd(9, 3)

gcd(21, 9)

gcd(72, 21)

gcd(93, 72)

gcd(444, 93)

DEĞİŞKENLER

x = 3 y = 0 dönüş = 3

gcd(444, 93): Kodu Adım Adım İzle

```
1 static int gcd(int x, int y) {  
2     if (y == 0) {  
3         return x;  
4     }  
5     return gcd(y, x % y);  
6 }
```

çağrı	x	y	x % y
gcd(444, 93)	444	93	72
gcd(93, 72)	93	72	21
gcd(72, 21)	72	21	9
gcd(21, 9)	21	9	3
gcd(9, 3)	9	3	0
gcd(3, 0)	3	0	—

Son çağrıdaki `return gcd(...)` ifadesinden sonra yapılacak iş yoktur: 3 değeri her çerçeveden **değişmeden** geri taşınır (5 dönüş). `main`, sonuç olarak **3** alır.

Adım 7 / 7

ÇAĞRI YIĞINI

gcd(444, 93)

DEĞİŞKENLER

dönüş = 3

ÖRNEKLER

Hanoi Kuleleri

Üç çubuk (A, B, C) ve A'da büyükten küçüğe dizili n disk var. Hepsi C'ye taşınacak:

- Her hamlede **yalnızca bir** disk, en üstteki taşınır.
- Büyük disk **asla** küçük diskin üstüne konmaz.

Özyinelemeli fikir: üstteki $n - 1$ diski yardımcıya taşı, en büyüğü hedefe taşı, $n - 1$ diski yardımcıdan hedefe taşı.

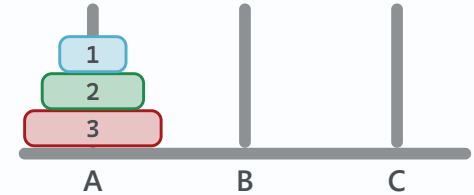
Hamle sayısı $2^n - 1$: 3 disk $\rightarrow$ 7, 10 disk $\rightarrow$ 1023, 64 disk $\rightarrow$ yaklaşık $1,8 \times 10^{19}$.

```
public class HanoiKuleleri {
    static void coz(int n, char kaynak,
                    char hedef, char yordimci) {
        if (n == 1) { // tek disk: doğrudan taşı
            System.out.println(n + ": " + kaynak + "->" + hedef);
        } else {
            coz(n - 1, kaynak, yordimci, hedef);
            System.out.println(n + ": " + kaynak + "->" + hedef);
            coz(n - 1, yordimci, hedef, kaynak);
        }
    }
    public static void main(String[] args) {
        coz(3, 'A', 'C', 'B');
    }
}
```

Döngüyle yazması zor olan bu problem, özyinelemeyle birkaç satırdır.

Hanoi Kuleleri: `coz(3, 'A', 'C', 'B')`

```
1 static void coz(int n, char kaynak,
2                 char hedef, char yardimci) {
3     if (n == 1) {
4         System.out.println(n + ": " + kaynak + "->" + hedef);
5     } else {
6         coz(n - 1, kaynak, yardimci, hedef);
7         System.out.println(n + ": " + kaynak + "->" + hedef);
8         coz(n - 1, yardimci, hedef, kaynak);
9     }
10 }
```



Başlangıç: 3 disk A'da. `coz(3, 'A', 'C', 'B')` : hepsini B'yi yardımcı kullanarak C'ye taşı.

Adım 1 / 8

ÇIKTI

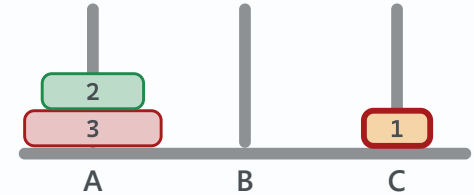
ÇAĞRI YİĞİNİ

`coz(3, A, C, B)`

`main()`

Hanoi Kuleleri: `coz(3, 'A', 'C', 'B')`

```
1 static void coz(int n, char kaynak,  
2                 char hedef, char yardimci) {  
3     if (n == 1) {  
4         System.out.println(n + ": " + kaynak + "->" + hedef);  
5     } else {  
6         coz(n - 1, kaynak, yardimci, hedef);  
7         System.out.println(n + ": " + kaynak + "->" + hedef);  
8         coz(n - 1, yardimci, hedef, kaynak);  
9     }  
10 }
```



En derindeki `coz(1, A, C, B)` :
temel durum, 1. disk doğrudan
A'dan C'ye.

Adım 2 / 8

ÇIKTI

1: A->C

ÇAĞRI YİĞİNİ

`coz(1, A, C, B)`

`coz(2, A, B, C)`

`coz(3, A, C, B)`

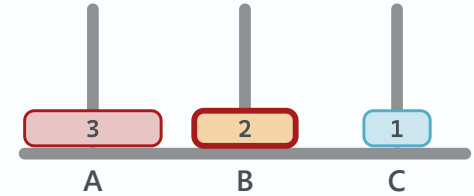
`main()`

Hanoi Kuleleri: `coz(3, 'A', 'C', 'B')`

```
1 static void coz(int n, char kaynak,
2                 char hedef, char yardimci) {
3     if (n == 1) {
4         System.out.println(n + ": " + kaynak + "->" + hedef);
5     } else {
6         coz(n - 1, kaynak, yardimci, hedef);
7         System.out.println(n + ": " + kaynak + "->" + hedef);
8         coz(n - 1, yardimci, hedef, kaynak);
9     }
10 }
```

ÇIKTI

```
1: A->C
2: A->B
```



`coz(2, A, B, C)` : üstündeki 1 disk kalktı; 2. disk A'dan B'ye.

Adım 3 / 8

ÇAĞRI YIĞINI

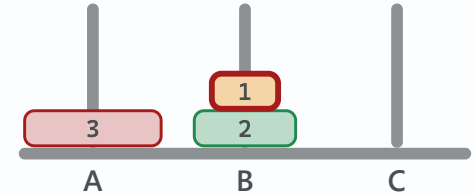
`coz(2, A, B, C)`

`coz(3, A, C, B)`

`main()`

Hanoi Kuleleri: `coz(3, 'A', 'C', 'B')`

```
1 static void coz(int n, char kaynak,  
2                 char hedef, char yardimci) {  
3     if (n == 1) {  
4         System.out.println(n + ": " + kaynak + "->" + hedef);  
5     } else {  
6         coz(n - 1, kaynak, yardimci, hedef);  
7         System.out.println(n + ": " + kaynak + "->" + hedef);  
8         coz(n - 1, yardimci, hedef, kaynak);  
9     }  
10 }
```



2. diskin üstüne 1. disk: `coz(1, C, B, A)`. Üstteki 2 disk artık B'de.

Adım 4 / 8

ÇIKTI

```
1: A->C  
2: A->B  
1: C->B
```

ÇAĞRI YİĞİNİ

`coz(1, C, B, A)`

`coz(2, A, B, C)`

`coz(3, A, C, B)`

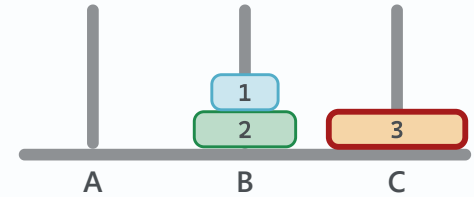
`main()`

Hanoi Kuleleri: `coz(3, 'A', 'C', 'B')`

```
1 static void coz(int n, char kaynak,  
2                 char hedef, char yardimci) {  
3     if (n == 1) {  
4         System.out.println(n + ": " + kaynak + "->" + hedef);  
5     } else {  
6         coz(n - 1, kaynak, yardimci, hedef);  
7         System.out.println(n + ": " + kaynak + "->" + hedef);  
8         coz(n - 1, yardimci, hedef, kaynak);  
9     }  
10 }
```

ÇIKTI

```
1: A->C  
2: A->B  
1: C->B  
3: A->C
```



`coz(3, ...)` ortadaki satıra geldi:
en büyük disk (3) A'dan C'ye.

Adım 5 / 8

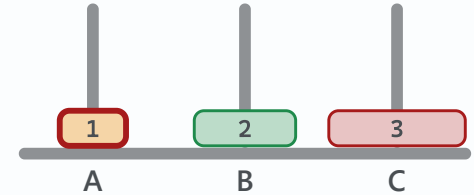
ÇAĞRI YIĞINI

`coz(3, A, C, B)`

`main()`

Hanoi Kuleleri: `coz(3, 'A', 'C', 'B')`

```
1 static void coz(int n, char kaynak,  
2                 char hedef, char yordimci) {  
3     if (n == 1) {  
4         System.out.println(n + ": " + kaynak + "->" + hedef);  
5     } else {  
6         coz(n - 1, kaynak, yordimci, hedef);  
7         System.out.println(n + ": " + kaynak + "->" + hedef);  
8         coz(n - 1, yordimci, hedef, kaynak);  
9     }  
10 }
```



Şimdi B'deki 2 disk C'ye taşınacak:

`coz(2, B, C, A)` → önce 1.
disk B'den A'ya.

Adım 6 / 8

ÇIKTI

```
1: A->C  
2: A->B  
1: C->B  
3: A->C  
1: B->A
```

ÇAĞRI YİĞİNİ

`coz(1, B, A, C)`

`coz(2, B, C, A)`

`coz(3, A, C, B)`

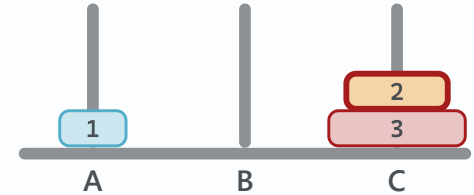
`main()`

Hanoi Kuleleri: `coz(3, 'A', 'C', 'B')`

```
1 static void coz(int n, char kaynak,  
2                 char hedef, char yordimci) {  
3     if (n == 1) {  
4         System.out.println(n + ": " + kaynak + "->" + hedef);  
5     } else {  
6         coz(n - 1, kaynak, yordimci, hedef);  
7         System.out.println(n + ": " + kaynak + "->" + hedef);  
8         coz(n - 1, yordimci, hedef, kaynak);  
9     }  
10 }
```

ÇIKTI

```
1: A->C  
2: A->B  
1: C->B  
3: A->C  
1: B->A  
2: B->C
```



2. disk B'den C'ye, 3. diskin üstüne.

Adım 7 / 8

ÇAĞRI YIĞINI

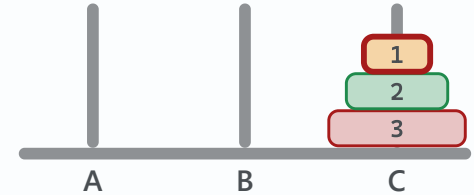
`coz(2, B, C, A)`

`coz(3, A, C, B)`

`main()`

Hanoi Kuleleri: `coz(3, 'A', 'C', 'B')`

```
1 static void coz(int n, char kaynak,  
2                 char hedef, char yordimci) {  
3     if (n == 1) {  
4         System.out.println(n + ": " + kaynak + "->" + hedef);  
5     } else {  
6         coz(n - 1, kaynak, yordimci, hedef);  
7         System.out.println(n + ": " + kaynak + "->" + hedef);  
8         coz(n - 1, yordimci, hedef, kaynak);  
9     }  
10 }
```



Son hamle: 1. disk A'dan C'ye. $2^3 - 1 = 7$ hamle; bulmaca çözüldü.

Adım 8 / 8

ÇIKTI

```
1: A->C  
2: A->B  
1: C->B  
3: A->C  
1: B->A  
2: B->C  
1: A->C
```

ÇAĞRI YIĞINI

`coz(1, A, C, B)`

`coz(2, B, C, A)`

`coz(3, A, C, B)`

`main()`

Verimlilik (Efficiency)

- Özyinelemeli metotlar bazen **yavaş** çalışabilir: her çağrının bir maliyeti (çerçeve açma, parametre kopyalama) vardır.
- Özyinelemeli çağrılar yapıldıkça sistem, bekleyen **kısmi sonuçları** çağrı yığnında biriktirir.
- Yığın aşırı dolarsa program `StackOverflowError` (yığın taşması) ile sonlanır.
- Sorunlu durumlarda çözüm, döngülerle çalışan **ilerlemeli** (iterative) çözümlerdir.
- İlerlemeli çözümler kısmi sonuçları biriktirmez: tek bir çerçeve ve birkaç değişken yeter.

Bellek

`faktoriyel(n)` özyinelemeli: n çerçeve, $O(n)$ bellek. Döngüyle: $O(1)$.

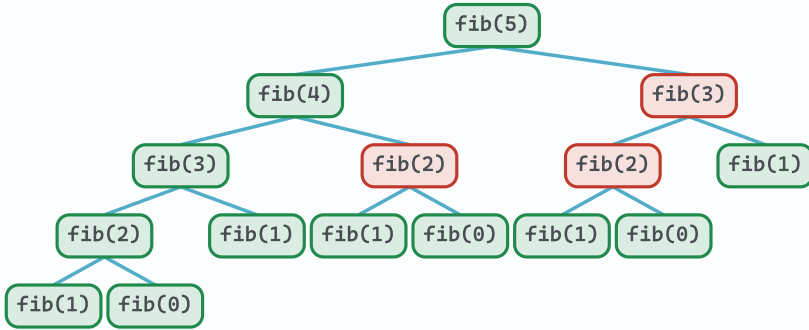
Derinlik sınırı

Varsayılan yığınla birkaç on bin çağrı derinliğinden sonra `StackOverflowError` alınır.

Tekrar

Ağaç özyinelemesi (Fibonacci) aynı değerleri defalarca hesaplayabilir.

Tekrarlanan Hesaplama



`fibonacci(5)` : 15 çağrı. Kırmızı düğümler daha önce hesaplanmış değerlerin **yeniden** hesaplanmasıdır.

n	çağrı sayısı	F(n)
10	177	55
20	21 891	6 765
30	2 692 537	832 040
40	331 160 281	102 334 155

n 1 artınca çağrı sayısı $\approx 1,6$ katına çıkar: **üstel** büyüme.

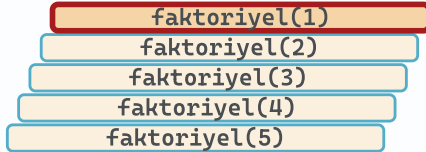
`fibonacci(40)` yarım saniye sürer; `fibonacci(50)` dakikalar alır.

Faktöriyel: Döngü ile

ÖZYİNELEMELİ

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

özyinelemeli: 5 çerçeve



İLERLEMELİ (DÖNGÜ)

```
int faktoriyel(int n) {  
    int carpim = 1;  
    for (int i = 1; i <= n; i++) {  
        carpim *= i;  
    }  
    return carpim;  
}
```

döngü: tek çerçeve

carpim: 1→1→2→6→24→120

Aynı sonuç ($\text{faktoriyel}(5) \rightarrow 120$). Döngülü sürüm tek çerçevede, sabit bellekle çalışır; çarpım `carpim` değişkeninde **biriktirilir**.

Rakamlar Toplamı ve Palindrom: Döngü ile

RAKAMLAR TOPLAMI

```
int rakamlarToplami(int n) {  
    int toplam = 0;  
    int kalan = Math.abs(n);  
    while (kalan > 0) {  
        toplam += kalan % 10;  
        kalan /= 10;  
    }  
    return toplam;  
}
```

`rakamlarToplami(-48729)` → **30**:
özyinelemeli sürümle aynı. Rakamların toplamı negatif olamaz; işaret yalnızca `Math.abs` ile atılır.

PALİNDROM

```
boolean palindromMu(String s) {  
    s = s.toLowerCase();  
    int uzunluk = s.length();  
    for (int i = 0; i < uzunluk / 2; i++) {  
        if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
            return false;  
        }  
    }  
    return true;  
}
```

`i` baştan, `uzunluk - 1 - i` sonndan ilerler.
`toLowerCase()` sayesinde `"Kayak"` → true.

Döngü ile Palindrom: Adım Adım

```
1 static boolean palindromMu(String s) {  
2     s = s.toLowerCase();  
3     int uzunluk = s.length();  
4     for (int i = 0; i < uzunluk / 2; i++) {  
5         if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
6             return false;  
7         }  
8     }  
9     return true;  
10 }  
11  
12 public static void main(String[] args) {  
13     ▶ System.out.println(palindromMu("Kepek"));  
14 }
```

DEĞİŞKENLER

s = -

ÇIKTI

palindromMu("Kepek")

main , palindromMu("Kepek") 'i çağırır.

Adım 1 / 11

ÇAĞRI YİĞİNİ

main()

Döngü ile Palindrom: Adım Adım

```
1 static boolean palindromMu(String s) {  
2     s = s.toLowerCase();  
3     int uzunluk = s.length();  
4     for (int i = 0; i < uzunluk / 2; i++) {  
5         if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
6             return false;  
7         }  
8     }  
9     return true;  
10 }  
11  
12 public static void main(String[] args) {  
13     System.out.println(palindromMu("Kepek"));  
14 }
```

DEĞİŞKENLER

s = "Kepek"

ÇIKTI



Parametre s = "Kepek" : ilk harf büyük.

Adım 2 / 11

ÇAĞRI YİĞİNİ

palindromMu("Kepek")

main()

Döngü ile Palindrom: Adım Adım

```
1 static boolean palindromMu(String s) {  
2     s = s.toLowerCase();  
3     int uzunluk = s.length();  
4     for (int i = 0; i < uzunluk / 2; i++) {  
5         if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
6             return false;  
7         }  
8     }  
9     return true;  
10 }  
11  
12 public static void main(String[] args) {  
13     System.out.println(palindromMu("Kepek"));  
14 }
```

DEĞİŞKENLER

s = "kepek"

ÇIKTI



toLowerCase() yeni bir dizgi döndürür:

s = "kepek" . Artık K ile k farkı yok.

Adım 3 / 11

ÇAĞRI YİĞİNİ

palindromMu("Kepek")

main()

Döngü ile Palindrom: Adım Adım

```
1 static boolean palindromMu(String s) {  
2     s = s.toLowerCase();  
3     int uzunluk = s.length();  
4     for (int i = 0; i < uzunluk / 2; i++) {  
5         if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
6             return false;  
7         }  
8     }  
9     return true;  
10 }  
11  
12 public static void main(String[] args) {  
13     System.out.println(palindromMu("Kepek"));  
14 }
```

DEĞİŞKENLER

s = "kepek"

uzunluk = 5

ÇIKTI

```
Kepek
```



uzunluk = 5 ; döngü $i < 5 / 2 = 2$ olduğu sürece döner.

Adım 4 / 11

ÇAĞRI YİĞİNİ

palindromMu("Kepek")

main()

Döngü ile Palindrom: Adım Adım

```
1 static boolean palindromMu(String s) {  
2     s = s.toLowerCase();  
3     int uzunluk = s.length();  
4     for (int i = 0; i < uzunluk / 2; i++) {  
5         if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
6             return false;  
7         }  
8     }  
9     return true;  
10 }  
11  
12 public static void main(String[] args) {  
13     System.out.println(palindromMu("Kepek"));  
14 }
```

DEĞİŞKENLER

`s = "kepek"``uzunluk = 5``i = 0`

ÇIKTI

```
Kepek
```



`i = 0` : baştan 0., sondan `5 - 1 - 0 = 4` . indis karşılaştırılacak.

Adım 5 / 11

ÇAĞRI YİĞİNİ

`palindromMu("Kepek")``main()`

Döngü ile Palindrom: Adım Adım

```
1 static boolean palindromMu(String s) {  
2     s = s.toLowerCase();  
3     int uzunluk = s.length();  
4     for (int i = 0; i < uzunluk / 2; i++) {  
5         if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
6             return false;  
7         }  
8     }  
9     return true;  
10 }  
11  
12 public static void main(String[] args) {  
13     System.out.println(palindromMu("Kepek"));  
14 }
```

DEĞİŞKENLER

`s = "kepek"``uzunluk = 5``i = 0`

ÇIKTI



'k' == 'k' : eşit, return false çalışmaz.

Adım 6 / 11

ÇAĞRI YİĞİNİ

`palindromMu("Kepek")``main()`

Döngü ile Palindrom: Adım Adım

```
1 static boolean palindromMu(String s) {  
2     s = s.toLowerCase();  
3     int uzunluk = s.length();  
4     for (int i = 0; i < uzunluk / 2; i++) {  
5         if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
6             return false;  
7         }  
8     }  
9     return true;  
10 }  
11  
12 public static void main(String[] args) {  
13     System.out.println(palindromMu("Kepek"));  
14 }
```

DEĞİŞKENLER

s = "kepek"

uzunluk = 5

i = 1

ÇIKTI



i = 1 : 1. ve 3. indis.

Adım 7 / 11

ÇAĞRI YİĞİNİ

palindromMu("Kepek")

main()

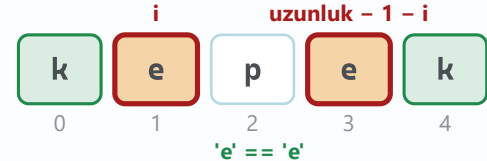
Döngü ile Palindrom: Adım Adım

```
1 static boolean palindromMu(String s) {  
2     s = s.toLowerCase();  
3     int uzunluk = s.length();  
4     for (int i = 0; i < uzunluk / 2; i++) {  
5         if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
6             return false;  
7         }  
8     }  
9     return true;  
10 }  
11  
12 public static void main(String[] args) {  
13     System.out.println(palindromMu("Kepek"));  
14 }
```

DEĞİŞKENLER

`s = "kepek"``uzunluk = 5``i = 1`

ÇIKTI



`'e' == 'e'` : eşit.

Adım 8 / 11

ÇAĞRI YİĞİNİ

`palindromMu("Kepek")``main()`

Döngü ile Palindrom: Adım Adım

```
1 static boolean palindromMu(String s) {  
2     s = s.toLowerCase();  
3     int uzunluk = s.length();  
4     for (int i = 0; i < uzunluk / 2; i++) {  
5         if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
6             return false;  
7         }  
8     }  
9     return true;  
10 }  
11  
12 public static void main(String[] args) {  
13     System.out.println(palindromMu("Kepek"));  
14 }
```

DEĞİŞKENLER

`s = "kepek"``uzunluk = 5``i = 2`

ÇIKTI



`i = 2` : `2 < 2` yanlış, döngü biter.
Ortadaki 'p' kendisiyle eşleşir, bakmaya gerek yok.

Adım 9 / 11

ÇAĞRI YIĞINI

`palindromMu("Kepek")``main()`

Döngü ile Palindrom: Adım Adım

```
1 static boolean palindromMu(String s) {  
2     s = s.toLowerCase();  
3     int uzunluk = s.length();  
4     for (int i = 0; i < uzunluk / 2; i++) {  
5         if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
6             return false;  
7         }  
8     }  
9     return true;  
10 }  
11  
12 public static void main(String[] args) {  
13     System.out.println(palindromMu("Kepek"));  
14 }
```



Hiç farklı çift bulunmadı: **true** döner.

Adım 10 / 11

DEĞİŞKENLER

s = "kepek"

uzunluk = 5

dönüş = true

ÇIKTI

ÇAĞRI YIĞINI

palindromMu("Kepek")

main()

Döngü ile Palindrom: Adım Adım

```
1 static boolean palindromMu(String s) {  
2     s = s.toLowerCase();  
3     int uzunluk = s.length();  
4     for (int i = 0; i < uzunluk / 2; i++) {  
5         if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {  
6             return false;  
7         }  
8     }  
9     return true;  
10 }  
11  
12 public static void main(String[] args) {  
13     ▶ System.out.println(palindromMu("Kepek"));  
14 }
```

DEĞİŞKENLER

s = -

ÇIKTI

true

palindromMu("Kepek")

true yazdırılır. Tek çerçeve kullanıldı; özyinelemeli sürümdeki gibi her adımda yeni dizgi de üretilmedi.

Adım 11 / 11

ÇAĞRI YIĞINI

main()

Özyineleme mi Döngü mü?

	Özyineleme	Döngü (ilerlemeli)
Durma	Temel durum	Döngü koşulu
İlerleme	Daha küçük argümanla çağrı	Sayaç / değişken güncelleme
Bellek	Her çağrı için bir çerçeve: $O(\text{derinlik})$	Genellikle $O(1)$
Hız	Çağrı maliyeti; ağaç özyinelemesinde tekrar	Genellikle daha hızlı
Hata	Temel durum yoksa <code>StackOverflowError</code>	Koşul bozuksa sonsuz döngü (program donar)
Okunabilirlik	Ağaç, böl ve yönet, Hanoi, seriler: çok doğal	Basit sayma ve biriktirme işleri için doğal

Her özyinelemeli çözüm döngüyle (gerekirse kendi yığını tutarak) yazılabilir; tersi de doğrudur. Seçim; açıklık, derinlik ve performansa göre yapılır.

Kuyruk Özyinelemesi (Tail Recursion)

- Metodun **son işlemi** özyinelemeli çağrının kendisidir; dönüşte yapılacak iş kalmaz.
- Sonuç, ek bir **biriktirici** (accumulator) parametrede taşınır.
- Bu biçim, derleyicinin **kuyruk çağrısı optimizasyonu** (TCO) yapmasını mümkün kılar: çağrı bir döngü gibi işlenir, çerçeveler yığılmaz → bellek verimliliği.
- **Java (JVM) bu optimizasyonu yapmaz:** kuyruk özyinelemeli Java kodu yine $O(n)$ yığın kullanır. Scheme, Scala (`@tailrec`), Kotlin (`tailrec`) gibi dillerde yapılır.

KUYRUK ÖZYİNELEMELİ DEĞİL

```
return n * faktoriyel(n - 1);  
// çağrı döndükten SONRA çarpma var
```

KUYRUK ÖZYİNELEMELİ

```
return faktoriyel(n - 1, n * birikim);  
// çarpma çağrıdan ÖNCE; son iş çağrı
```

n * faktoriyel(n - 1)

4 * ?

3 * ?

2 * ?

1

dönüşte çarpım bekler

faktoriyel(n - 1, n * b)

(4, 1)

(3, 4)

(2, 12)

(1, 24)

bekleyen iş yok

Kuyruk Özyinelemeli Örnekler

```
int rakamlarToplami(int sayi, int toplam) {  
    if (sayi == 0) {  
        return toplam;  
    } else {  
        return rakamlarToplami(sayi / 10,  
                                toplam + sayi % 10);  
    }  
}
```

Rakamlar toplamı: son basamak, çağrıdan önce `toplam` 'a eklenir.

`rakamlarToplami(48729, 0)` → **30** ($n \geq 0$ için).

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

Faktöriyel: çarpım, çağrıdan önce `birikim` 'e katılır. `faktoriyel(5, 1)` → **120**.

Biriktiricinin başlangıç değeri işlemin etkisiz elemanıdır: toplama için **0**, çarpma için **1**.
Kullanıcıyı bundan kurtarmak için sarmalayıcı metot yazılır: `int faktoriyel(int n) {
 return faktoriyel(n, 1); }`

faktoriyel(4, 1): Kuyruk Özyinelemesini İzle

faktoriyel(4, 1)
son iş: çağrı

faktoriyel(3, 4)

faktoriyel(2, 12)

faktoriyel(1, 24)

faktoriyel(0, 24)

Çarpım **çağrıdan önce** yapılır: $4 * 1 = 4$ yeni birikim olur; çağrı `faktoriyel(3, 4)`.

Adım 1 / 9

```
1 static int faktoriyel(int n, int birikim) {  
2     if (n == 0) {  
3         return birikim;  
4     } else {  
5         return faktoriyel(n - 1, n * birikim);  
6     }  
7 }
```

ÇAĞRI YIĞINI

faktoriyel(4, 1)

DEĞİŞKENLER

n = 4 birikim = 1

n * birikim = 4

faktoriyel(4, 1): Kuyruk Özyinelemesini İzle

faktoriyel(4, 1)
son iş: çağrı

faktoriyel(3, 4)
son iş: çağrı

faktoriyel(2, 12)

faktoriyel(1, 24)

faktoriyel(0, 24)

Çarpım **çağrıdan önce** yapılır: $3 * 4 = 12$ yeni birikim olur; çağrı `faktoriyel(2, 12)`.

Adım 2 / 9

```
1 static int faktoriyel(int n, int birikim) {  
2     if (n == 0) {  
3         return birikim;  
4     } else {  
5         return faktoriyel(n - 1, n * birikim);  
6     }  
7 }
```

ÇAĞRI YIĞINI

faktoriyel(3, 4)

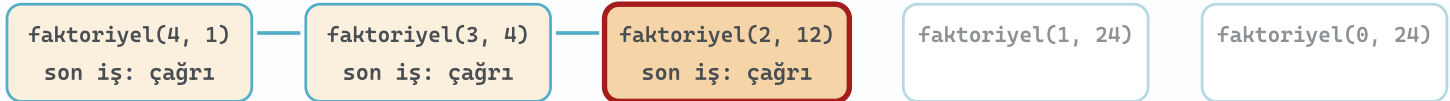
faktoriyel(4, 1)

DEĞİŞKENLER

n = 3 birikim = 4

n * birikim = 12

faktoriyel(4, 1): Kuyruk Özyinelemesini İzle



Çarpım **çağrıdan önce** yapılır: $2 * 12 = 24$ yeni birikim olur; çağrı `faktoriyel(1, 24)`.

Adım 3 / 9

```

1 static int faktoriyel(int n, int birikim) {
2     if (n == 0) {
3         return birikim;
4     } else {
5         return faktoriyel(n - 1, n * birikim);
6     }
7 }
    
```

ÇAĞRI YIĞINI

faktoriyel(2, 12)

faktoriyel(3, 4)

faktoriyel(4, 1)

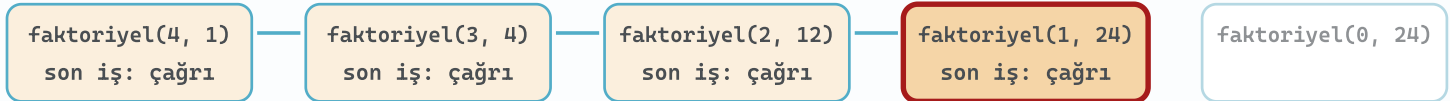
DEĞİŞKENLER

$n = 2$

$birikim = 12$

$n * birikim = 24$

faktoriyel(4, 1): Kuyruk Özyinelemesini İzle



Çarpım **çağrıdan önce** yapılır: $1 * 24 = 24$ yeni birikim olur; çağrı `faktoriyel(0, 24)`.

Adım 4 / 9

```
1 static int faktoriyel(int n, int birikim) {  
2     if (n == 0) {  
3         return birikim;  
4     } else {  
5         return faktoriyel(n - 1, n * birikim);  
6     }  
7 }
```

ÇAĞRI YIĞINI

faktoriyel(1, 24)

faktoriyel(2, 12)

faktoriyel(3, 4)

faktoriyel(4, 1)

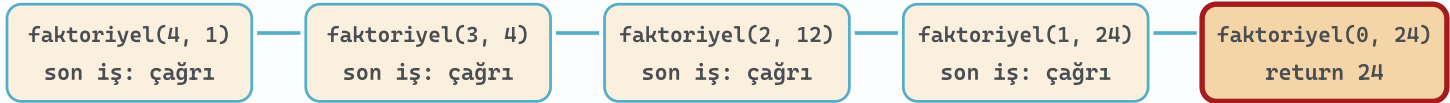
DEĞİŞKENLER

`n = 1`

`birikim = 24`

`n * birikim = 24`

faktoriyel(4, 1): Kuyruk Özyinelemesini İzle



`n == 0` : temel durum. Sonuç zaten `birikim` 'de hazır: **24** döner.

Adım 5 / 9

```
1 static int faktoriyel(int n, int birikim) {  
2     if (n == 0) {  
3         return birikim;  
4     } else {  
5         return faktoriyel(n - 1, n * birikim);  
6     }  
7 }
```

ÇAĞRI YIĞINI

faktoriyel(0, 24)

faktoriyel(1, 24)

faktoriyel(2, 12)

faktoriyel(3, 4)

faktoriyel(4, 1)

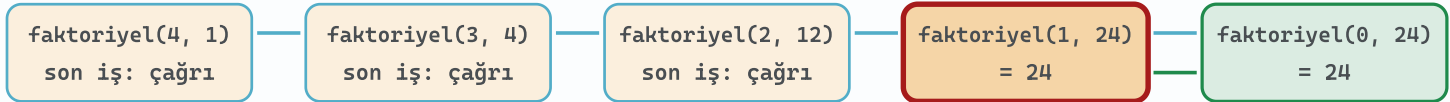
DEĞİŞKENLER

`n = 0`

`birikim = 24`

`dönüş = 24`

faktoriyel(4, 1): Kuyruk Özyinelemesini İzle



Dönüşte hiçbir hesap yapılmaz; 24 aynen yukarı aktarılır.

Adım 6 / 9

```
1 static int faktoriyel(int n, int birikim) {  
2     if (n == 0) {  
3         return birikim;  
4     } else {  
5         return faktoriyel(n - 1, n * birikim);  
6     }  
7 }
```

ÇAĞRI YIĞINI

faktoriyel(1, 24)

faktoriyel(2, 12)

faktoriyel(3, 4)

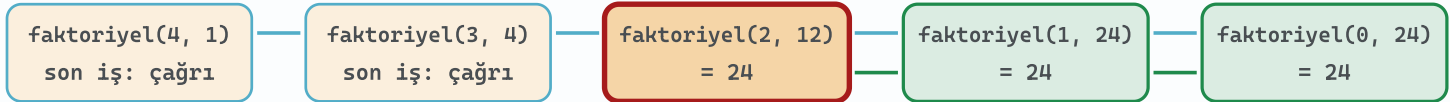
faktoriyel(4, 1)

DEĞİŞKENLER

n = 1

dönüş = 24

faktoriyel(4, 1): Kuyruk Özyinelemesini İzle



Dönüşte hiçbir hesap yapılmaz; 24 aynen yukarı aktarılır.

Adım 7 / 9

```
1 static int faktoriyel(int n, int birikim) {  
2     if (n == 0) {  
3         return birikim;  
4     } else {  
5         return faktoriyel(n - 1, n * birikim);  
6     }  
7 }
```

ÇAĞRI YIĞINI

faktoriyel(2, 12)

faktoriyel(3, 4)

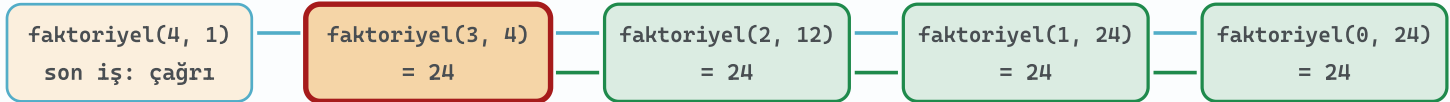
faktoriyel(4, 1)

DEĞİŞKENLER

n = 2

dönüş = 24

faktoriyel(4, 1): Kuyruk Özyinelemesini İzle



Dönüşte hiçbir hesap yapılmaz; 24 aynen yukarı aktarılır.

Adım 8 / 9

```
1 static int faktoriyel(int n, int birikim) {  
2     if (n == 0) {  
3         return birikim;  
4     } else {  
5         return faktoriyel(n - 1, n * birikim);  
6     }  
7 }
```

ÇAĞRI YIĞINI

faktoriyel(3, 4)

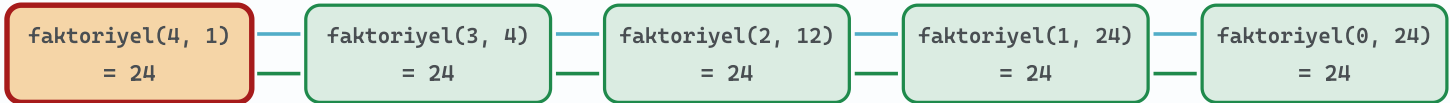
faktoriyel(4, 1)

DEĞİŞKENLER

n = 3

dönüş = 24

faktoriyel(4, 1): Kuyruk Özyinelemesini İzle



Sonuç 24. Java kuyruk çağrısını optimize etmez: 5 çerçeve yine de yığına girdi. TCO yapan dillerde tek çerçeve yeterdi.

Adım 9 / 9

```
1 static int faktoriyel(int n, int birikim) {  
2     if (n == 0) {  
3         return birikim;  
4     } else {  
5         return faktoriyel(n - 1, n * birikim);  
6     }  
7 }
```

ÇAĞRI YIĞINI

faktoriyel(4, 1)

DEĞİŞKENLER

n = 4

dönüş = 24

Hafızalama (Memoization)

- Tekrarlanan hesaplamaların sonuçlarını **saklayıp yeniden kullanma** yöntemidir (durum bilgisi taşıma).
- Sonuçlar bir önbellege (`Map`) alınır; aynı argüman tekrar gelince **hesaplamadan** döndürülür.
- Her n için hesap **bir kez** yapılır: üstel çağrı sayısı doğrusala iner.
- `Map<Integer, Long>` : anahtar n, değer F(n).

`fibonacci(90)` →
2880067194370816120
(anında).

```
import java.util.HashMap;
import java.util.Map;

static Map<Integer, Long> hafiza = new HashMap<>();

static long fibonacci(int n) {
    if (hafiza.containsKey(n)) {
        return hafiza.get(n);    // daha önce hesaplandı
    }
    if (n == 0 || n == 1) {
        return n;
    }
    long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
    hafiza.put(n, sonuc);        // sonucu sakla
    return sonuc;
}
```

Hafızalı fibonacci(4): Adım Adım

fib(4)
çağrı yapar

hafıza

2 → ?

3 → ?

4 → ?

fibonacci(4) çağrıldı. Hafıza boş: 4 orada yok, temel durum da değil → önce fibonacci(3) çağrılır.

Adım 1 / 10

```
1 static Map<Integer, Long> hafiza = new HashMap<>();
2 static long fibonacci(int n) {
3     if (hafiza.containsKey(n)) {
4         return hafiza.get(n);
5     }
6     if (n == 0 || n == 1) {
7         return n;
8     }
9     ▶ long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
10    hafiza.put(n, sonuc);
11    return sonuc;
12 }
```

ÇAĞRI YIĞINI

fibonacci(4)

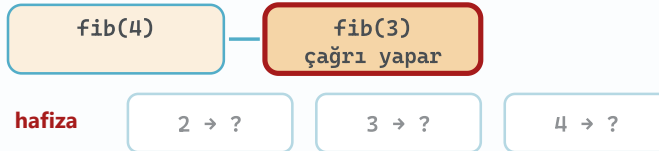
main()

DEĞİŞKENLER

n = 4

ÇIKTI

Hafızalı fibonacci(4): Adım Adım



fibonacci(3) : hafızada yok → fibonacci(2) çağrılır.

Adım 2 / 10

```
1 static Map<Integer, Long> hafiza = new HashMap<>();
2 static long fibonacci(int n) {
3     if (hafiza.containsKey(n)) {
4         return hafiza.get(n);
5     }
6     if (n == 0 || n == 1) {
7         return n;
8     }
9     ▶ long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
10    hafiza.put(n, sonuc);
11    return sonuc;
12 }
```

ÇAĞRI YIĞINI

fibonacci(3)

fibonacci(4)

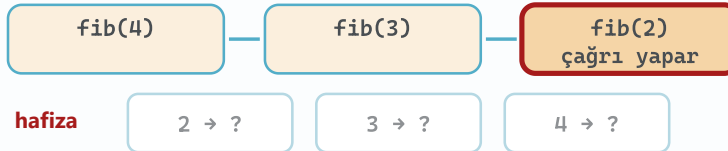
main()

DEĞİŞKENLER

n = 3

ÇIKTI

Hafızalı fibonacci(4): Adım Adım



fibonacci(2) : hafızada yok → fibonacci(1) çağrılır.

Adım 3 / 10

```
1 static Map<Integer, Long> hafiza = new HashMap<>();
2 static long fibonacci(int n) {
3     if (hafiza.containsKey(n)) {
4         return hafiza.get(n);
5     }
6     if (n == 0 || n == 1) {
7         return n;
8     }
9     ▶ long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
10    hafiza.put(n, sonuc);
11    return sonuc;
12 }
```

ÇAĞRI YIĞINI

fibonacci(2)

fibonacci(3)

fibonacci(4)

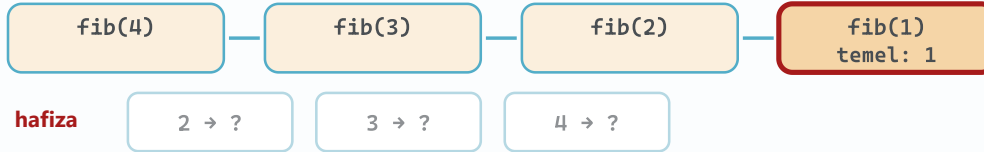
main()

DEĞİŞKENLER

n = 2

ÇIKTI

Hafızalı fibonacci(4): Adım Adım



n = 1 : temel durum, 1 döner.

Adım 4 / 10

```
1 static Map<Integer, Long> hafiza = new HashMap<>();
2 static long fibonacci(int n) {
3     if (hafiza.containsKey(n)) {
4         return hafiza.get(n);
5     }
6     ▶ if (n == 0 || n == 1) {
7     ▶     return n;
8     }
9     long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
10    hafiza.put(n, sonuc);
11    return sonuc;
12 }
```

ÇAĞRI YIĞINI

fibonacci(1)

fibonacci(2)

fibonacci(3)

fibonacci(4)

main()

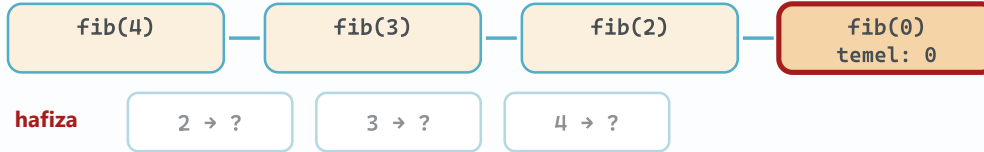
DEĞİŞKENLER

n = 1

dönüş = 1

ÇIKTI

Hafızalı fibonacci(4): Adım Adım



n = 0 : temel durum, 0 döner.

Adım 5 / 10

```
1 static Map<Integer, Long> hafiza = new HashMap<>();
2 static long fibonacci(int n) {
3     if (hafiza.containsKey(n)) {
4         return hafiza.get(n);
5     }
6     if (n == 0 || n == 1) {
7         return n;
8     }
9     long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
10    hafiza.put(n, sonuc);
11    return sonuc;
12 }
```

ÇAĞRI YIĞINI

fibonacci(0)

fibonacci(2)

fibonacci(3)

fibonacci(4)

main()

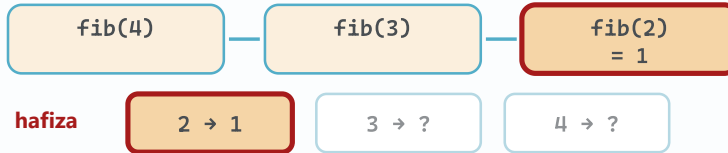
DEĞİŞKENLER

n = 0

dönüş = 0

ÇIKTI

Hafızalı fibonacci(4): Adım Adım



fibonacci(2) = 1 + 0 = 1 . Sonuç hafızaya yazılır: {2=1} .

Adım 6 / 10

```
1 static Map<Integer, Long> hafiza = new HashMap<>();
2 static long fibonacci(int n) {
3     if (hafiza.containsKey(n)) {
4         return hafiza.get(n);
5     }
6     if (n == 0 || n == 1) {
7         return n;
8     }
9     long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
10    ▶ hafiza.put(n, sonuc);
11    ▶ return sonuc;
12 }
```

ÇAĞRI YIĞINI

fibonacci(2)

fibonacci(3)

fibonacci(4)

main()

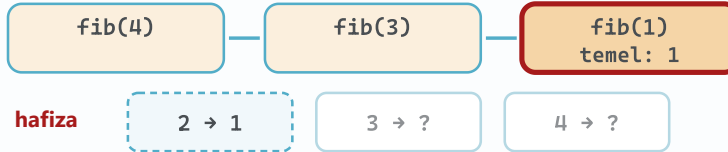
DEĞİŞKENLER

n = 2

dönüş = 1

ÇIKTI

Hafızalı fibonacci(4): Adım Adım



fibonacci(3) 'ün ikinci çağrısı fibonacci(1) : temel durum, 1.

Adım 7 / 10

```
1 static Map<Integer, Long> hafiza = new HashMap<>();
2 static long fibonacci(int n) {
3     if (hafiza.containsKey(n)) {
4         return hafiza.get(n);
5     }
6     if (n == 0 || n == 1) {
7         return n;
8     }
9     long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
10    hafiza.put(n, sonuc);
11    return sonuc;
12 }
```

ÇAĞRI YIĞINI

fibonacci(1)

fibonacci(3)

fibonacci(4)

main()

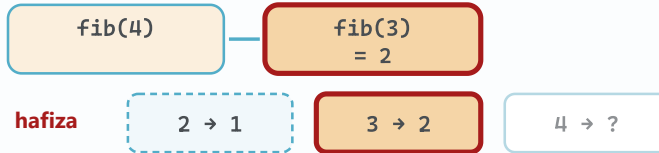
DEĞİŞKENLER

n = 1

dönüş = 1

ÇIKTI

Hafızalı fibonacci(4): Adım Adım



`fibonacci(3) = 1 + 1 = 2`, hafızaya yazılır.

Adım 8 / 10

```
1 static Map<Integer, Long> hafiza = new HashMap<>();
2 static long fibonacci(int n) {
3     if (hafiza.containsKey(n)) {
4         return hafiza.get(n);
5     }
6     if (n == 0 || n == 1) {
7         return n;
8     }
9     long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
10    ▶ hafiza.put(n, sonuc);
11    ▶ return sonuc;
12 }
```

ÇAĞRI YIĞINI

fibonacci(3)

fibonacci(4)

main()

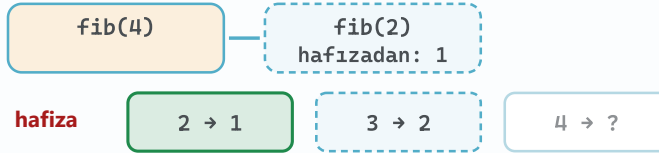
DEĞİŞKENLER

n = 3

dönüş = 2

ÇIKTI

Hafızalı fibonacci(4): Adım Adım



İkinci `fibonacci(2)` çağırısı: hafızada **var**, hesaplanmadan 1 döner (hafızasız sürüm burada 3 çağrı daha yapardı).

Adım 9 / 10

```
1 static Map<Integer, Long> hafiza = new HashMap<>();
2 static long fibonacci(int n) {
3     if (hafiza.containsKey(n)) {
4         return hafiza.get(n);
5     }
6     if (n == 0 || n == 1) {
7         return n;
8     }
9     long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
10    hafiza.put(n, sonuc);
11    return sonuc;
12 }
```

ÇAĞRI YIĞINI

fibonacci(2)

fibonacci(4)

main()

DEĞİŞKENLER

n = 2

dönüş = 1

ÇIKTI

Hafızalı fibonacci(4): Adım Adım

fib(4)
= 3

hafıza

2 → 1

3 → 2

4 → 3

fibonacci(4) = 2 + 1 = 3 , hafızaya yazılır. Toplam 7 çağrı (hafızasız: 9).

Adım 10 / 10

```
1 static Map<Integer, Long> hafiza = new HashMap<>();
2 static long fibonacci(int n) {
3     if (hafiza.containsKey(n)) {
4         return hafiza.get(n);
5     }
6     if (n == 0 || n == 1) {
7         return n;
8     }
9     long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
10    hafiza.put(n, sonuc);
11    return sonuc;
12 }
```

ÇAĞRI YIĞINI

fibonacci(4)

main()

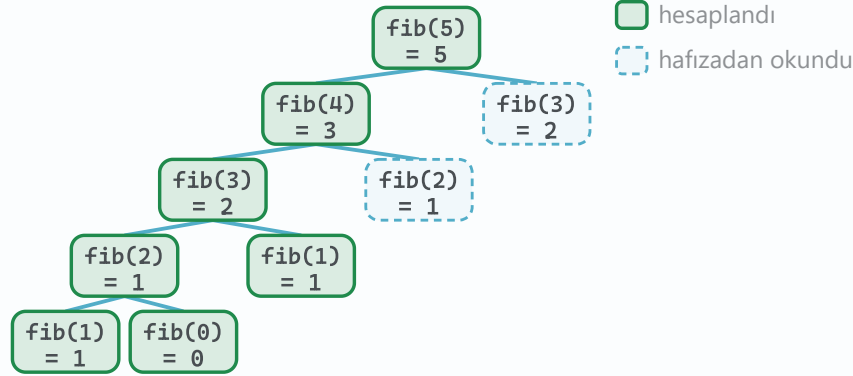
DEĞİŞKENLER

n = 4

dönüş = 3

ÇIKTI

Hafızalı Fibonacci: Çağrı Ağacı



Hafızalı **fibonacci(5)** : sol kol bir kez hesaplanır; sağdaki **fib(2)** ve **fib(3)** hafızadan gelir. 15 yerine 9 çağrı.

n	hafızasız çağrı	hafızalı çağrı	hafızasız süre (ölçülen)	hafızalı süre
20	21 891	39	≈ 0,5 ms	< 0,1 ms
40	331 160 281	79	≈ 0,5 s	< 1 ms

Hafızalı sürümde çağrı sayısı $2n - 1$ 'dir. Süreler makineye göre değişir; oran değişmez.

Hafızalı Faktöriyel

```
static Map<Integer, Long> hafiza = new HashMap<>();

static long faktoriyel(int n) {
    if (hafiza.containsKey(n)) {
        return hafiza.get(n);
    }
    if (n == 0 || n == 1) {
        return 1;
    }
    long sonuc = n * faktoriyel(n - 1);
    hafiza.put(n, sonuc);
    return sonuc;
}
```

- Aynı kalıp: önce hafızaya bak, yoksa hesapla, sakla, döndür.
- `long` ile 20!'e kadar doğru:
`faktoriyel(20)` = 2432902008176640000.
- Tek bir `faktoriyel(n)` çağrısında tekrar yoktur; hafıza bu çağrıyı **hızlandırmaz**.
- Kazanç, program **birçok kez** faktöriyel istediğinde ortaya çıkar:
`faktoriyel(10)` 'dan sonra `faktoriyel(12)` yalnızca 2 yeni çarpma yapar.
- Bedel: bellek. Hafızalama, hız için bellek harcamaktır.

ÖZET

Özet

Kavram	Özü
Özyineleme	Metodun kendini daha küçük bir girdiyle çağırması
Temel durum	Doğrudan çözülen en küçük durum; özyinelemeyi durdurur
İlerleme	Her çağrı temel duruma yaklaşmalı ($n - 1$, $n / 10$, substring, $x \% y$)
Çağrı yığını	Her çağrıya bir çerçeve; dönüşte LIFO sırasıyla kapanır. Çok derin → <code>StackOverflowError</code>
Ağaç özyinelemesi	Birden çok çağrı (Fibonacci, Hanoi); çağrı sayısı üstel büyüyebilir
İlerlemeli çözüm	Döngü + biriktirici değişken; sabit bellek, genellikle daha hızlı
Kuyruk özyinelemesi	Son iş çağrıdır; sonuç biriktirici parametrede. Java TCO yapmaz
Hafızalama	Hesaplanan sonuçları <code>Map</code> 'te saklayıp yeniden kullanmak

ÖZET

Sık Yapılan Hatalar

Temel durum yok / ulaşılamıyor

`toplam(0)` gibi: `Exception in thread "main" java.lang.StackOverflowError`.

İlerleme yok

`return n * faktoriyel(n);` parametre küçülmez; yine `StackOverflowError`.

Eksik `return`

`if ... else if` ile biten metot: `error: missing return statement`. Son dalı `else` yapın.

Çağrının sonucunu kullanmamak

`faktoriyel(n - 1);` tek başına yazılırsa değer kaybolur; `return n * faktoriyel(n - 1);` olmalı.

Taşma

`int` faktöriyel 13!'te, Bell katsayısı 13'ten sonra yanlıştır; hata mesajı yoktur. `long` kullanın.

Gereksiz ağaç özyinelemesi

Hafızasız `fibonacci(50)` dakikalar sürer. Döngü ya da hafızalama kullanın.

ÖZET

Örnek Problemler (1)

```
static void yaz(int n) {  
    if (n == 0) return;  
    System.out.print(n + " ");  
    yaz(n - 1);  
}  
static void yaz2(int n) {  
    if (n == 0) return;  
    yaz2(n - 1);  
    System.out.print(n + " ");  
}  
static int gizem(int a, int b) {  
    if (b == 0) return 0;  
    return a + gizem(a, b - 1);  
}
```

1. `yaz(3)` ne yazdırır?
2. `yaz2(3)` ne yazdırır? Fark nereden gelir?
3. `gizem(4, 3)` kaçtır? Metot ne hesaplar?
4. `gizem(4, -1)` çağrısında ne olur?

► Cevapları göster

ÖZET

Örnek Problemler (2)

```
static String ters(String s) {  
    if (s.isEmpty()) return "";  
    return ters(s.substring(1)) + s.charAt(0);  
}
```

1. `ters("java")` ne döndürür?
2. `ters` kuyruk özyinelemeli midir? Ya `gcd` ?
3. `fibonacci(5)` kaç kez çağrılır (ilk çağrı dahil)?
4. `gcd(48, 18)` çağrılarını yazın.

► Cevapları göster

ÖZET

Alıştırmalar

1. `long us(int taban, int us)` : tabanın üssünü özyinelemeyle hesaplayın. `us(2, 10)` = ?
2. `void ikili(int n)` : n'nin ikili (binary) gösterimini yazdırın. `ikili(13)` → ?
3. `int diziToplam(int[] a, int i)` : dizinin i. indisten sonuna kadar toplamı. `{3, 1, 4, 1, 5}` → ?
4. Fibonacci'yi döngüyle ve kuyruk özyinelemesiyle (`fibK(n, a, b)`) yazın.
5. `lucas(5)` kaç çağrı yapar? Hafızalama ile kaç iner?
6. Hanoi kulelerinde 4 disk için kaç hamle gerekir? İlk hamle hangisidir?

► Cevaplar / ipuçları

Kaynaklar

- Oracle, *The Java Tutorials*: Defining Methods; Returning a Value from a Method.
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21 Edition (15.12 Method Invocation Expressions).
- P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 18: Recursion).
- A. Downey ve C. Mayfield, *Think Java*, 2. baskı, O'Reilly (Bölüm 8: Recursive Methods).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson.
- R. Sedgewick ve K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley (2.3 Recursion).

Son

Bölüm 3: Özyineleme

Sonraki bölüm: Bölüm 4: Nesne ve Sınıflar · [Terim sözlüğü](#)