Bölüm 9: Geri İzleme Algoritmaları

Algoritmalar

P, NP ve NP-zor | Geri dönüp dene | Durum uzayı ağacı | Budama | n-vezir problemi | En az çakışma | Sudoku | Çizge boyama ve Hamilton yolu | Kelime arama | Karmaşıklık

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Kavram

Geri izlemeyi (backtracking) kısmi çözüm, aday, kısıt ve budama kavramlarıyla açıklamak; durum uzayı ağacını çizmek; P, NP ve NP-zor sınıflarını ayırt etmek.

Şablon

"seç → keşfet → geri al" özyinelemeli şablonunu Java ile yazmak ve yeni bir probleme uyarlamak.

Uygulamalar

n-vezir, Sudoku, labirent, çizge boyama, Hamilton yolu ve kelime arama problemlerini adım adım çözmek.

Analiz

En kötü durumun neden üstel olduğunu göstermek; budama ve sezgisellerin (MRV, en az çakışma) etkisini değerlendirmek.

GİRİŞ

Zor Problemler: P, NP ve NP-Zor

P

Polinom zamanda **çözülebilen** karar problemleri (ör. en kısa yol, MST).

NP

Önerilen bir çözümü polinom zamanda **doğrulanabilen** karar problemleri.

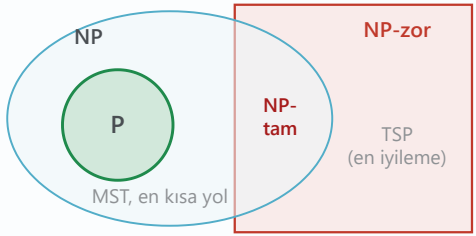
NP-zor (NP-hard)

NP'deki her problem buna polinom zamanda indirgenebilir; en az NP'deki en zor problem kadar zordur.

NP-tam (NP-complete)

Hem NP'de hem NP-zor. Örnekler: SAT, 3-SAT, Hamilton yolu ve döngüsü, çizge 3-boyama.

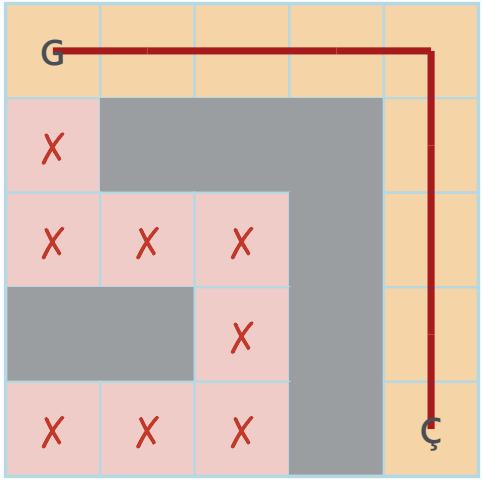
- Hamilton yolu, çizge boyama, SAT ve alt küme toplamının karar sürümleri NP-tamdır; TSP ve sırt çantasının **en iyileme** sürümleri NP-zordur.
- P ≠ NP ise bunlar için polinom zamanlı **kesin** algoritma yoktur. Kesin çözüm için üstel arama kaçınılmazdır; amaç aramayı **budamayla** küçültmektir (geri izleme, dal sınır). Polinom süre şartsa **yaklaşık** çözümle yetinilir.



TEMEL FİKİR

Geri Dönüp Dene

- Bazen bir problemi çözmek için **farklı yollar denemek** ve **yanlış yollardan geri dönmek** gerekir.
- **Geri izleme (backtracking)**, karmaşık problemleri çözmek için bu stratejiyi sistemli biçimde kullanır.
- Çözüm, adım adım **kısmi çözüm** olarak kurulur; bir adım kısıtları bozuyorsa o dal **terk edilir**.
- Tüm olasılıkları körü körüne üreten **kaba kuvvetten** (brute force) farkı budur: geçersiz kısmi çözümlerin devamı hiç üretilmez.

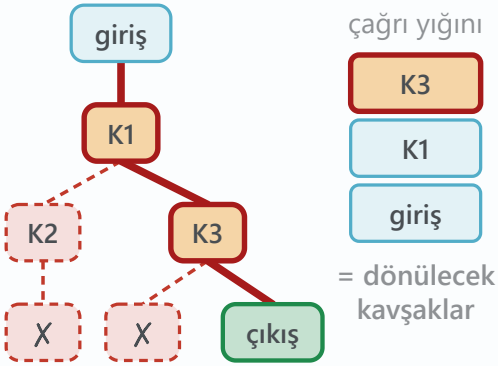


Bulunan yol (turuncu) ve denenip bırakılan çıkmazlar (X)

TEMEL FİKİR

Labirentteki Yolcu

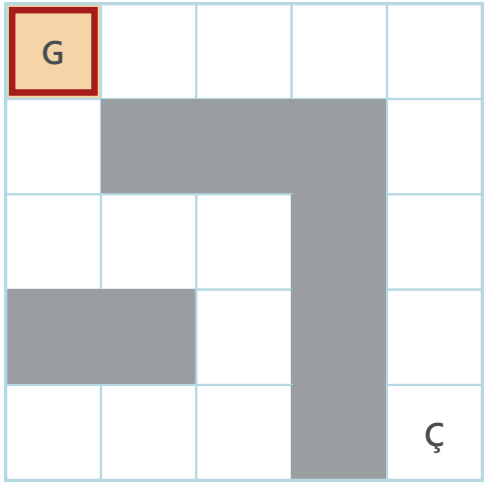
- Geri izleme algoritması, bir **labirentte yolculuk yapan** kişiye benzetilebilir.
- Kişi her **kavşak** noktasında bir karar vererek ilerler.
- Seçtiği yol çıkışa götürmüyorsa **geri dönüp** başka bir yol dener.
- Her adımda bir karar verilir ve kararın **doğru olup olmadığı** kontrol edilir.
- Bu **deneme-yanılma** süreci problem çözülene (ya da tüm seçenekler tükenene) kadar sürer.
- Labirent, Sudoku ve satranç gibi bulmaca ve yapay zekâ oyunlarında kullanılır.



Önemli: Geri dönmek için nereden gelindiğini hatırlamak gerekir. Özyineleme bunu bedavaya sağlar: **çağrı yığını** kavşakların listesidir.

TEMEL FİKİR

Labirentte Geri Dönüş Adım Adım

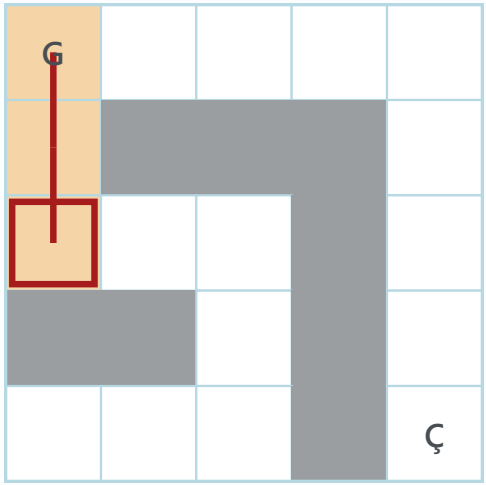


Giriş (G) hücresinden başlanır. Yön sırası: **aşağı, sağ, yukarı, sol**. Her hücre ziyaret edildiğinde işaretlenir; böylece aynı hücreye ikinci kez girilmez.

Adım 1 / 8

TEMEL FİKİR

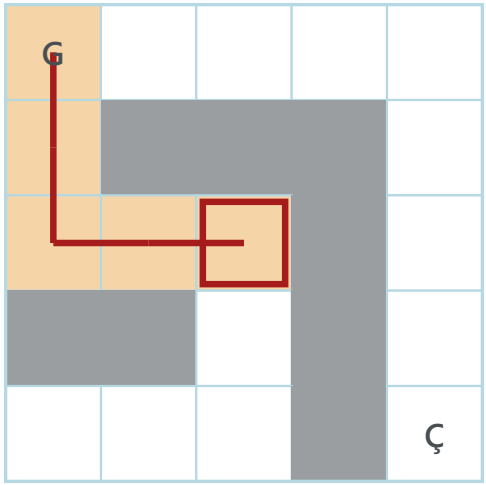
Labirentte Geri Dönüş Adım Adım



Aşağı iki adım: (2,0). Bir daha aşağısı duvar, bu yüzden bir sonraki yön (sağ) denenir.

TEMEL FİKİR

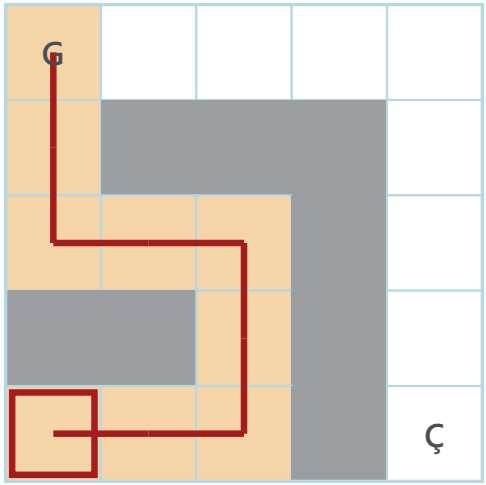
Labirentte Geri Dönüş Adım Adım



Sağa iki adım: (2,2) bir **kavşak**. Önce aşağı denenecek; sağ ve yukarı duvar.

TEMEL FİKİR

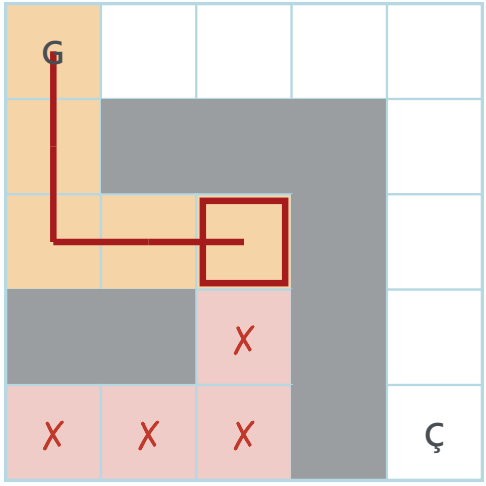
Labirentte Geri Dönüş Adım Adım



Aşağı ve sola ilerleyen koridor (4,0) hücresinde **çıkmaza** girer: dört yön de duvar, sınır ya da ziyaret edilmiş.

TEMEL FİKİR

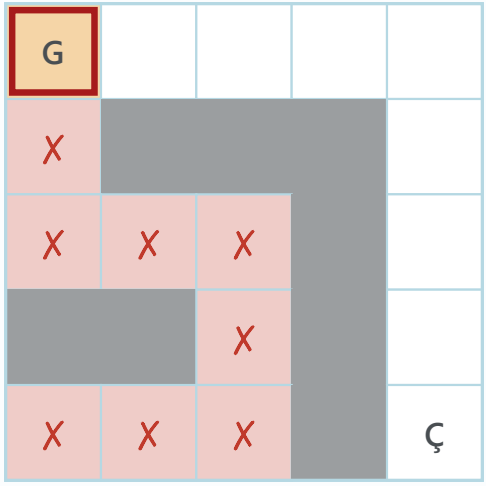
Labirentte Geri Dönüş Adım Adım



Geri dönüş: (4,0), (4,1), (4,2), (3,2) yoldan çıkarılır (*X*). Kavşak (2,2) yeniden denenir ama kalan yönleri de kapalı.

TEMEL FİKİR

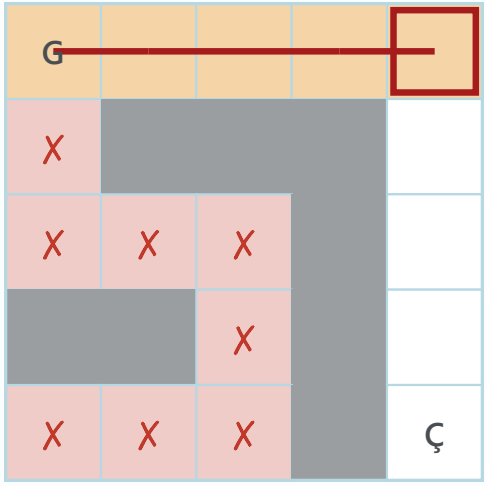
Labirentte Geri Dönüş Adım Adım



Geri dönüş girişe kadar sürer: (2,2), (2,1), (2,0), (1,0) de çıkmaz. Girişte sıradaki yön **sağ** denenir.

TEMEL FİKİR

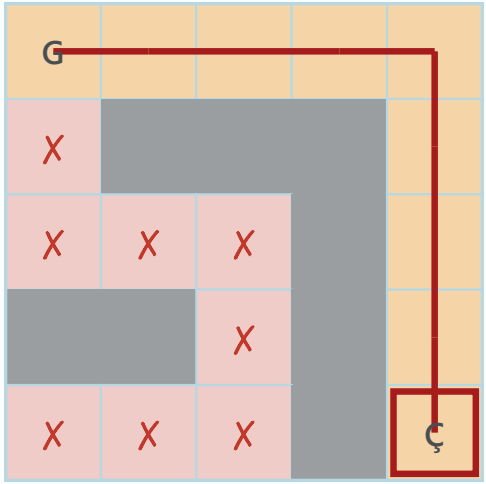
Labirentte Geri Dönüş Adım Adım



Üst koridor boyunca sağa: (0,4). Aşağısı açık.

TEMEL FİKİR

Labirentte Geri Dönüş Adım Adım

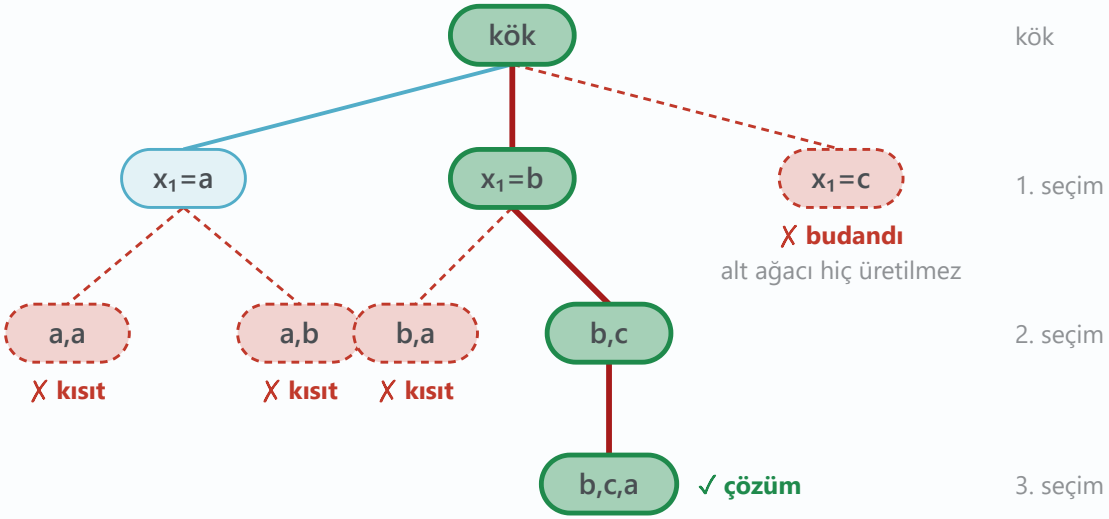


(1,4), (2,4), (3,4) üzerinden **çıkışa** (Ç) ulaşıldı. Yol 9 hücre; 8 hücre çıkmaz sokak olarak denendi ve bırakıldı.

Adım 8 / 8

TEMEL FİKİR

Durum Uzayı Ağacı (State-Space Tree)



Kısmi çözüm
Kökten bir düğüme giden yol: o ana kadar yapılan seçimler.

Aday
Bir sonraki düzeyde yapılabilecek seçimler: düğümün çocukları.

Kısıt
Kısmi çözüm geçersizse o düğümün **alt ağacı budanır**.

Hedef testi
Tam ve geçerli bir çözüme ulaşıldı mı? (yaprak)

TEMEL FİKİR

Genel Şablon: Seç → Keşfet → Geri Al

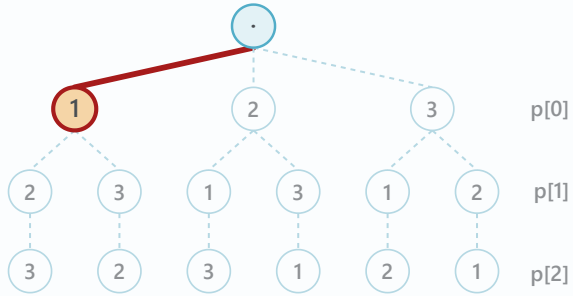
```
void geriDon(Durum kismi) {
  if (cozumMu(kismi)) {
    kaydet(kismi);
    return;
  }
  for (Aday a : adaylar(kismi)) {
    if (gecerliMi(kismi, a)) {
      ekle(kismi, a);    // seç
      geriDon(kismi);    // keşfet
      cikar(kismi, a);   // geri al
    }
  }
}
```

1. **Hedef testi:** kısmi çözüm tamamsa kaydedilir (ya da durulur).
2. **Seç:** sıradaki aday kısmi çözüme eklenir.
3. **Keşfet:** özyinelemeyle bir düzey derine inilir.
4. **Geri al:** aday çıkarılır; durum çağrıdan önceki hâline döner, sıradaki aday denenir.

Budama `gecerliMi` içinde olur: geçersiz aday için özyinelemeli çağrı **hiç yapılmaz**, böylece o adayın altındaki tüm alt ağaç atlanır. Tek çözüm yetiyorsa `geriDon` `boolean` döndürür ve ilk `true` 'da durulur.

Şablonun Uygulaması: 1, 2, 3 Permütasyonları

```
1  void permutasyon(int[] p, boolean[] kullan, int k) {
2    if (k == p.length) {                // çözüm
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!kullan[a]) {                 // geçerli
▶        p[k] = a; kullan[a] = true;     // seç
▶        permutasyon(p, kullan, k + 1); // keşfet
10       kullan[a] = false;              // geri al
11     }
12   }
13 }
```



p[0..2] boş, hiçbir sayı kullanılmadı (k = 0). Adaylar 1, 2, 3 sırayla denenir: **a = 1** seçilir, kullan[1] = true, bir düzey derine inilir.

Adım 1 / 11

ÇAĞRI YIĞINI

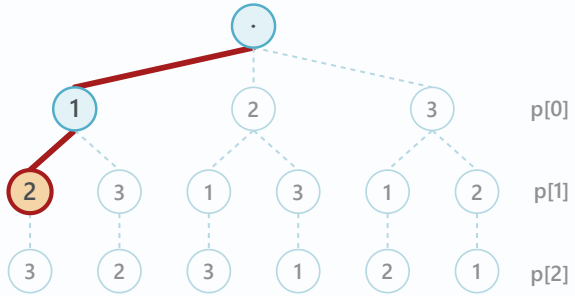
- **permutasyon(k=1)**
- permutasyon(k=0)
- main()

DEĞİŞKENLER

k = **0** | a = **1** | p = **[1]** | kullan = **{1}**

Şablonun Uygulaması: 1, 2, 3 Permütasyonları

```
1  void permutasyon(int[] p, boolean[] kullan, int k) {
2    if (k == p.length) {                // çözüm
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!kullan[a]) {                 // geçerli
▶        p[k] = a; kullan[a] = true;     // seç
▶        permutasyon(p, kullan, k + 1); // keşfet
10       kullan[a] = false;              // geri al
11     }
12   }
13 }
```



k = 1: **a = 2** kullanılmamış → seçilir, kullan[2] = true.

Adım 2 / 11

ÇAĞRI YIĞINI

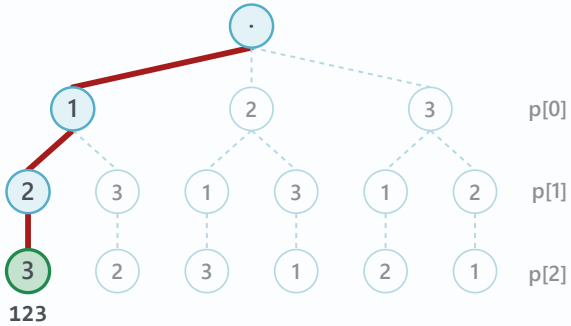
- **permutasyon(k=2)**
- permutasyon(k=1)
- permutasyon(k=0)
- main()

DEĞİŞKENLER

`k = 1` `a = 2` `p = [1,2]` `kullan = {1, 2}`

Şablonun Uygulaması: 1, 2, 3 Permütasyonları

```
1  void permutasyon(int[] p, boolean[] kullan, int k) {
▶    if (k == p.length) {                // çözüm
▶      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
7      if (!kullan[a]) {                 // geçerli
8        p[k] = a; kullan[a] = true;     // seç
9        permutasyon(p, kullan, k + 1); // keşfet
10       kullan[a] = false;              // geri al
11     }
12   }
13 }
```



k = 2: yalnız 3 kullanılmamış → seçilir. k = 3 = p.length: **çözüm**, [1, 2, 3] yazdırılır.

Adım 3 / 11

ÇAĞRI YIĞINI

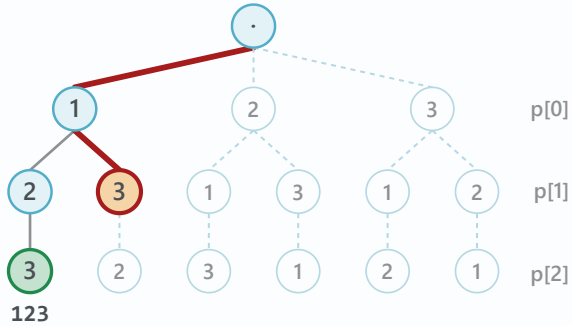
- `permutasyon(k=3)`
- `permutasyon(k=2)`
- `permutasyon(k=1)`
- `permutasyon(k=0)`
- `main()`

DEĞİŞKENLER

`k = 2` `a = 3` `p = [1,2,3]` `kullan = {1, 2, 3}`

Şablonun Uygulaması: 1, 2, 3 Permütasyonları

```
1  void permutasyon(int[] p, boolean[] kullan, int k) {
2    if (k == p.length) {                // çözüm
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!kullan[a]) {                 // geçerli
▶        p[k] = a; kullan[a] = true;     // seç
▶        permutasyon(p, kullan, k + 1); // keşfet
▶        kullan[a] = false;              // geri al
11     }
12   }
13 }
```



Geri al: kullan[3] = false, kullan[2] = false. k = 1: **a = 3** kullanılmamış → seçilir, kullan[3] = true.

Adım 4 / 11

ÇAĞRI YIĞINI

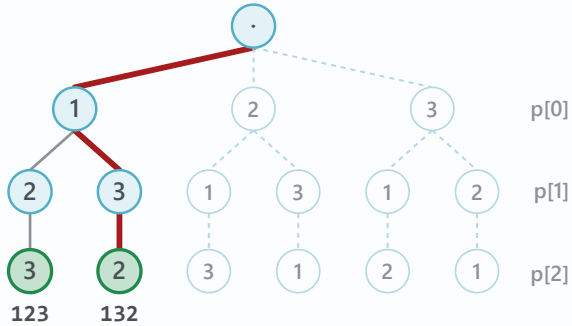
- **permutasyon(k=2)**
- permutasyon(k=1)
- permutasyon(k=0)
- main()

DEĞİŞKENLER

`k = 1` `a = 3` `p = [1,3]` `kullan = {1, 3}`

Şablonun Uygulaması: 1, 2, 3 Permütasyonları

```
1  void permutasyon(int[] p, boolean[] kullan, int k) {
▶    if (k == p.length) {                // çözüm
▶      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
7      if (!kullan[a]) {                 // geçerli
8        p[k] = a; kullan[a] = true;     // seç
9        permutasyon(p, kullan, k + 1); // keşfet
10       kullan[a] = false;              // geri al
11     }
12   }
13 }
```



k = 2: yalnız 2 kullanılmamış → seçilir. k = 3 = p.length: **çözüm**, [1, 3, 2] yazdırılır.

Adım 5 / 11

ÇAĞRI YIĞINI

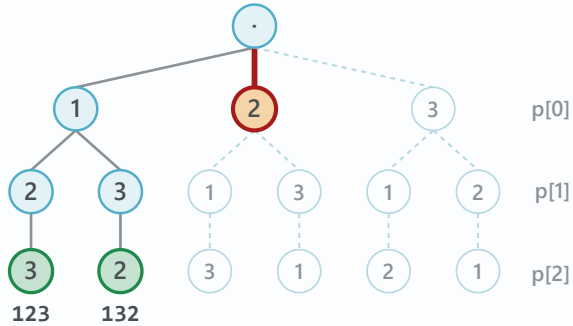
- `permutasyon(k=3)`
- `permutasyon(k=2)`
- `permutasyon(k=1)`
- `permutasyon(k=0)`
- `main()`

DEĞİŞKENLER

`k = 2` `a = 2` `p = [1,3,2]` `kullan = {1, 2, 3}`

Şablonun Uygulaması: 1, 2, 3 Permütasyonları

```
1  void permutasyon(int[] p, boolean[] kullan, int k) {
2    if (k == p.length) {                // çözüm
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!kullan[a]) {                 // geçerli
▶        p[k] = a; kullan[a] = true;     // seç
▶        permutasyon(p, kullan, k + 1); // keşfet
▶        kullan[a] = false;              // geri al
11     }
12   }
13 }
```



Geri al: kullan[2] = false, kullan[3] = false, kullan[1] = false. k = 0: **a = 2** kullanılmamış → seçilir, kullan[2] = true.

Adım 6 / 11

ÇAĞRI YIĞINI

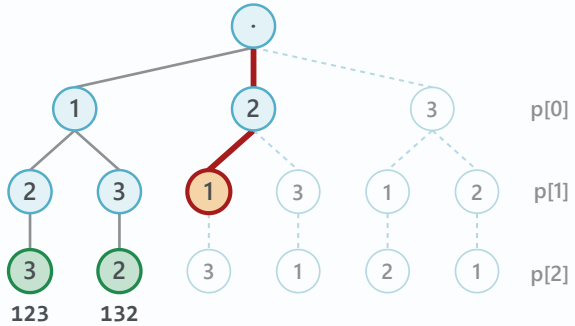
- `permutasyon(k=1)`
- `permutasyon(k=0)`
- `main()`

DEĞİŞKENLER

`k = 0` `a = 2` `p = [2]` `kullan = {2}`

Şablonun Uygulaması: 1, 2, 3 Permütasyonları

```
1  void permutasyon(int[] p, boolean[] kullan, int k) {
2    if (k == p.length) {                // çözüm
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!kullan[a]) {                 // geçerli
▶        p[k] = a; kullan[a] = true;     // seç
▶        permutasyon(p, kullan, k + 1); // keşfet
10       kullan[a] = false;              // geri al
11     }
12   }
13 }
```



k = 1: **a = 1** kullanılmamış → seçilir, kullan[1] = true.

Adım 7 / 11

ÇAĞRI YIĞINI

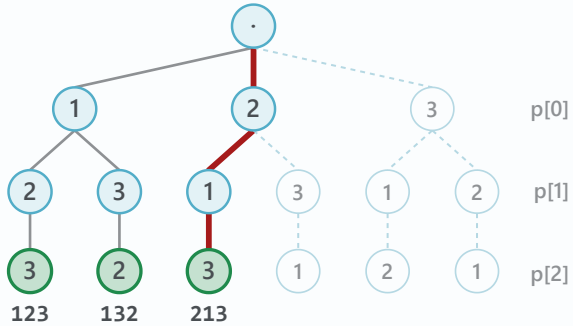
- **permutasyon(k=2)**
- permutasyon(k=1)
- permutasyon(k=0)
- main()

DEĞİŞKENLER

`k = 1` `a = 1` `p = [2,1]` `kullan = {1, 2}`

Şablonun Uygulaması: 1, 2, 3 Permütasyonları

```
1  void permutasyon(int[] p, boolean[] kullan, int k) {
▶    if (k == p.length) {                // çözüm
▶      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
7      if (!kullan[a]) {                 // geçerli
8        p[k] = a; kullan[a] = true;     // seç
9        permutasyon(p, kullan, k + 1); // keşfet
10       kullan[a] = false;              // geri al
11     }
12   }
13 }
```



k = 2: yalnız 3 kullanılmamış → seçilir. k = 3 = p.length: **çözüm**, [2, 1, 3] yazdırılır.

Adım 8 / 11

ÇAĞRI YIĞINI

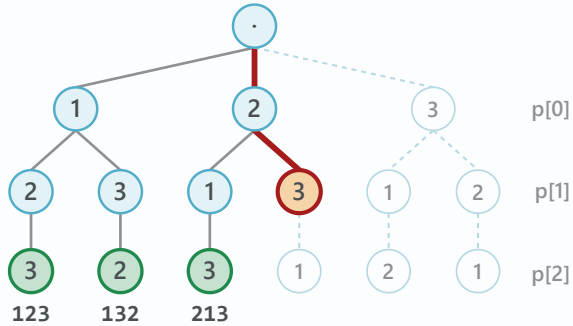
- `permutasyon(k=3)`
- `permutasyon(k=2)`
- `permutasyon(k=1)`
- `permutasyon(k=0)`
- `main()`

DEĞİŞKENLER

`k = 2` `a = 3` `p = [2,1,3]` `kullan = {1, 2, 3}`

Şablonun Uygulaması: 1, 2, 3 Permütasyonları

```
1  void permutasyon(int[] p, boolean[] kullan, int k) {
2    if (k == p.length) {                // çözüm
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
▶      if (!kullan[a]) {                 // geçerli
▶        p[k] = a; kullan[a] = true;     // seç
▶        permutasyon(p, kullan, k + 1); // keşfet
▶        kullan[a] = false;              // geri al
11     }
12   }
13 }
```



Geri al: kullan[3] = false, kullan[1] = false. k = 1: **a = 3** kullanılmamış → seçilir, kullan[3] = true.

Adım 9 / 11

ÇAĞRI YIĞINI

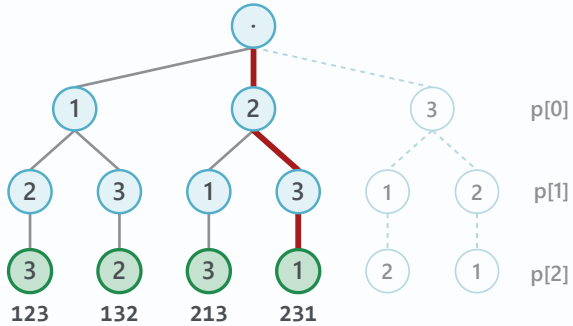
- **permutasyon(k=2)**
- permutasyon(k=1)
- permutasyon(k=0)
- main()

DEĞİŞKENLER

`k = 1` `a = 3` `p = [2,3]` `kullan = {2, 3}`

Şablonun Uygulaması: 1, 2, 3 Permütasyonları

```
1  void permutasyon(int[] p, boolean[] kullan, int k) {
▶    if (k == p.length) {              // çözüm
▶      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
7      if (!kullan[a]) {               // geçerli
8        p[k] = a; kullan[a] = true;   // seç
9        permutasyon(p, kullan, k + 1); // keşfet
10       kullan[a] = false;            // geri al
11     }
12   }
13 }
```



k = 2: yalnız 1 kullanılmamış → seçilir. k = 3 = p.length: **çözüm**, [2, 3, 1] yazdırılır.

Adım 10 / 11

ÇAĞRI YIĞINI

- **permutasyon(k=3)**
- permutasyon(k=2)
- permutasyon(k=1)
- permutasyon(k=0)
- main()

DEĞİŞKENLER

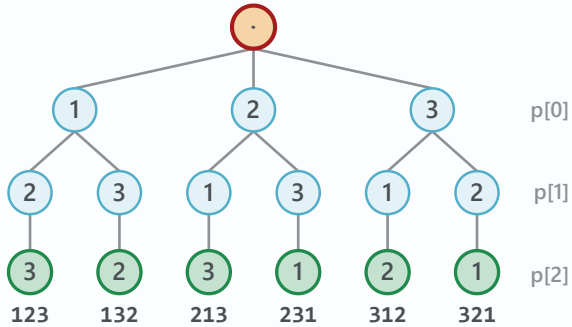
`k = 2` `a = 1` `p = [2,3,1]` `kullan = {1, 2, 3}`

Şablonun Uygulaması: 1, 2, 3 Permütasyonları

```
1  void permutasyon(int[] p, boolean[] kullan, int k) {
2    if (k == p.length) {                // çözüm
3      System.out.println(Arrays.toString(p));
4      return;
5    }
6    for (int a = 1; a <= p.length; a++) {
7      if (!kullan[a]) {                 // geçerli
8        p[k] = a; kullan[a] = true;     // seç
9        permutasyon(p, kullan, k + 1); // keşfet
▶        kullan[a] = false;              // geri al
11     }
12   }
13 }
```

ÇAĞRI YIĞINI

- `permutasyon(k=0)`
- `main()`



Geri alınır, 3 ile başlayan dal aynı biçimde gezilir: [3, 1, 2] ve [3, 2, 1]. **3! = 6** permütasyon, 15 seç / 15 geri al; kullan[] her dönüşte eski hâline döner.

Adım 11 / 11

DEĞİŞKENLER

`k = 0` `a = 3` `p = –` `kullan = {}`

TEMEL FİKİR

Kaba Kuvvet ve Budama: 8 Vezir

Yaklaşım	İncelenen aday	Açıklama	log ölçek
64 kareden 8'ini seç	4 426 165 368	C(64, 8): her yerleşimi üret, sonra test et	
Her satıra bir vezir	16 777 216	8^8 tam yerleşim	
Her satıra bir vezir, sütunlar farklı	40 320	8! permütasyon	
Geri izleme, **tüm** çözümler	2056	güvenli yerleştirme sayısı (92 çözüm)	
Geri izleme, **ilk** çözüm	113	güvenli yerleştirme; 105 geri alma	

Budamanın gücü: Kısmi yerleşim geçersiz olur olmaz geri dönüldüğü için, 16,7 milyon tam yerleşim yerine yalnızca birkaç bin **kısmi** yerleşim üretilir. En kötü durum yine üsteldir, ama pratikte ağacın çok küçük bir kısmı gezilir.

TEMEL FİKİR

Avantajlar ve Dezavantajlar

Avantajlar

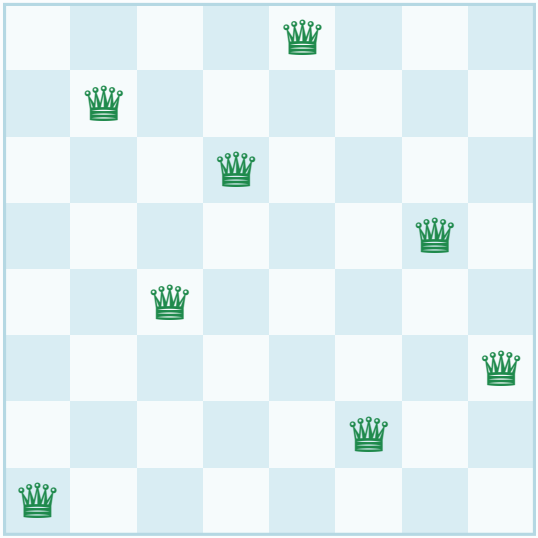
+ **Çok çeşitli** problemlere uygulanabilen genel bir yöntemdir.
+ Çözümleri **ağaç** olarak ifade edilebilen problemlerde çoğu zaman verimlidir.
+ Anlaması ve uygulaması **kolaydır**: kısa, özyinelemeli kod.
+ **Tüm** çözümleri ya da hiç çözüm olmadığını kesin olarak bulur (tam arama).

Dezavantajlar

– En kötü durumda zaman karmaşıklığı **üsteldir** (2^n, $n!$, 9^m ...).
– Olası çözüm sayısı çok büyük olduğunda **yavaşlar**; iyi budama ve sezgisel gerekir.
– Derin özyineleme yığın belleği kullanır: O(derinlik).
– Aynı alt problem farklı yollardan tekrar tekrar çözülebilir (DP'deki gibi saklanmaz).

8 Vezir Problemi (Eight Queens)

- Bir 8×8 satranç tahtasına **8 vezirin** yerleştirilmesi.
- Amaç: **hiçbir vezirin diğerini tehdit etmediği** bir yerleşim bulmak. Vezir aynı **satır**, **sütun** ve **çaprazdaki** taşları tehdit eder.
- Genelleştirilmiş hâli: n×n tahtaya **n vezir** (N-Queens). n = 2 ve n = 3 dışında her n ≥ 1 için çözüm vardır.
- 8 vezirin **92** çözümü vardır (simetriler ayıklanınca 12 temel çözüm).
- Geri izleme algoritması bu tür problemleri çözmek için kullanılır.

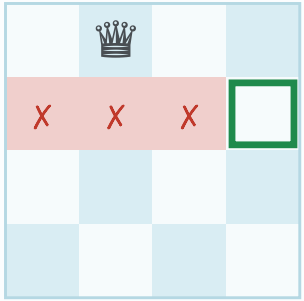


Bir çözüm: a1, b7, c4, d6, e8, f2, g5, h3

N-VEZİR PROBLEMİ

Geri İzleme Çözümü: Adımlar

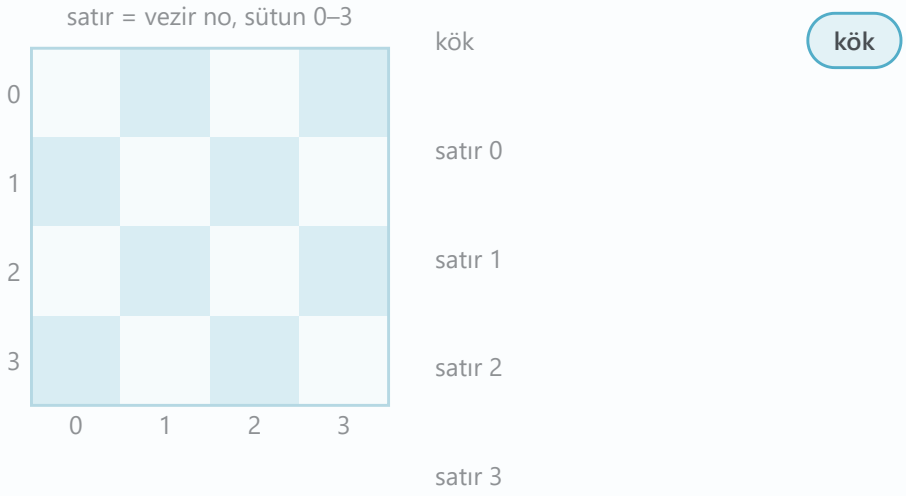
1. Tahtanın her hücresi başlangıçta **boş**.
2. İlk vezir ilk satırda bir sütuna yerleştirilir.
3. Yerleştirilen vezirin önceki vezirlerle **çakışıp çakışmadığı** kontrol edilir (sütun ve iki çapraz).
4. Çakışma yoksa konum **geçerli** kabul edilir; bir sonraki vezir bir sonraki satıra yerleştirilir.
5. Bir satırda **tüm sütunlar** denenip güvenli yer bulunamazsa, bir önceki vezir **yer değiştirilir** (geri dönüş).
6. Tüm 8 vezir yerleştirildiğinde **çözüm** bulunmuş olur.



Neden satır satır? Her satıra tam olarak bir vezir gelmek zorunda. Bu gözlem satır kısıtını **yapının içine gömer**: geriye yalnızca sütun ve çapraz denetimi kalır, arama uzayı 64^8'den 8^8'e iner.

N-VEZİR PROBLEMİ

4 Vezir: Tahta ve Durum Uzayı Ağacı

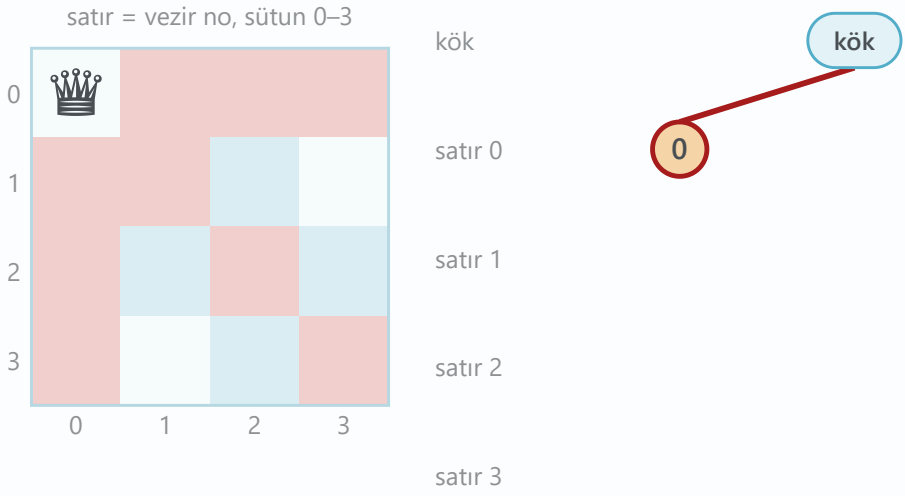


Boş tahta. Vezirler **satır satır** yerleştirilir; her satırda sütun 0, 1, 2, 3 sırayla denenir. Ağaçtaki düğüm, o satırda seçilen **sütunu** gösterir.

Adım 1 / 13

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

4 Vezir: Tahta ve Durum Uzayı Ağacı

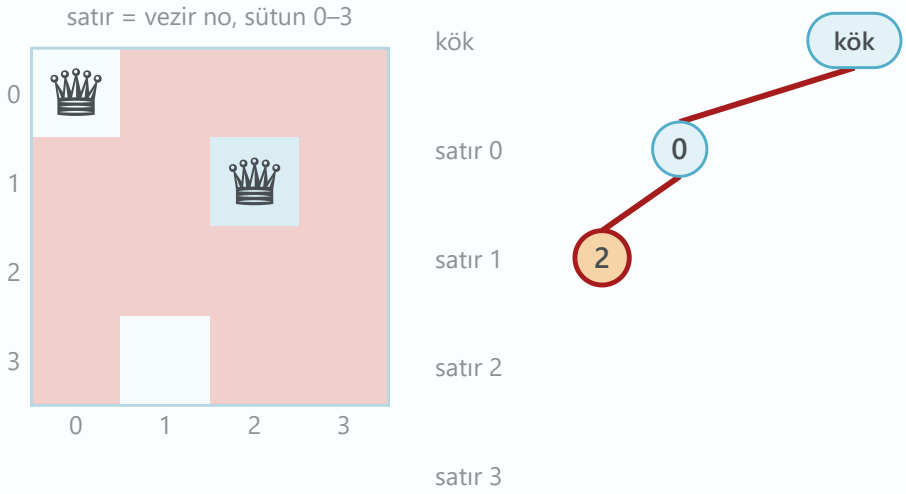


Satır 0: **sütun 0 güvenli** → vezir yerleştirilir, bir alt düzeye (satır 1) inilir.

Adım 2 / 13

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

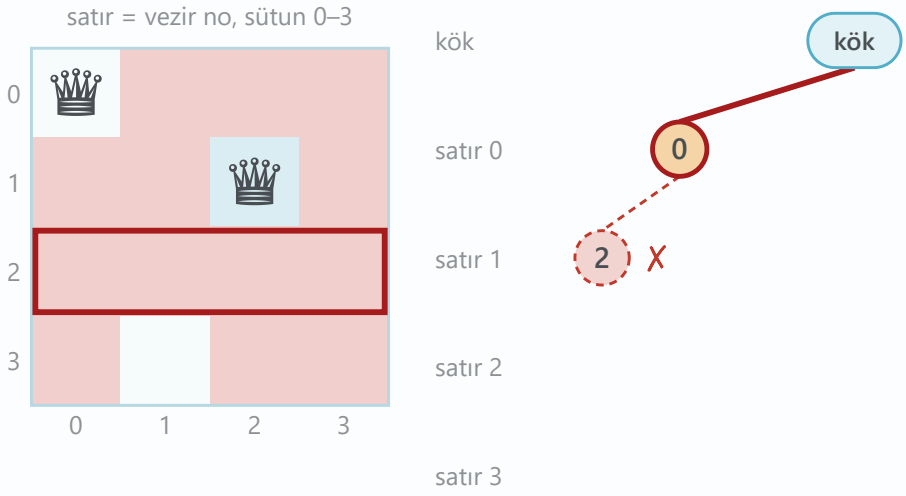
4 Vezir: Tahta ve Durum Uzayı Ağacı



Satır 1: sütun 0, 1 tehdit altında; **sütun 2 güvenli** → vezir yerleştirilir, bir alt düzeye (satır 2) inilir.

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

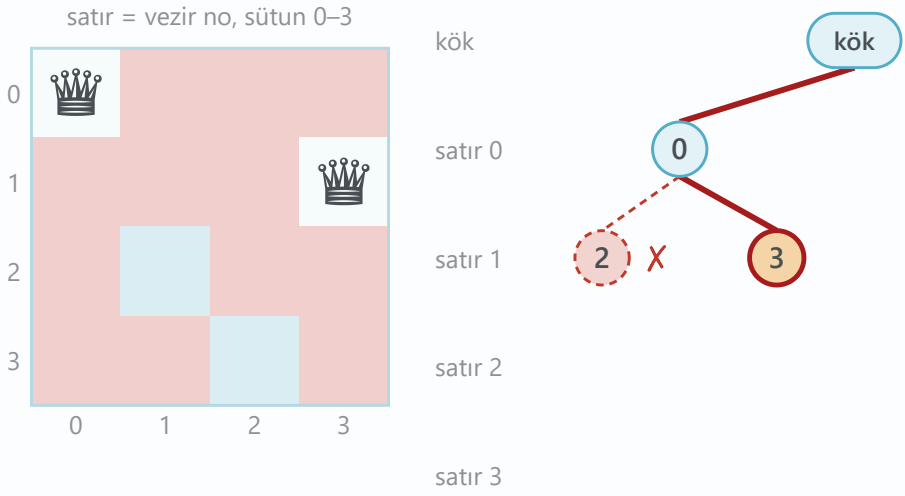
4 Vezir: Tahta ve Durum Uzayı Ağacı



Satır 2: sütun 0, 1, 2, 3 tehdit altında → **güvenli sütun yok** (çıkmaz). **Geri dön**: satır 1'deki vezir (sütun 2) kaldırılır ve o satırda sıradaki sütun denenir.

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

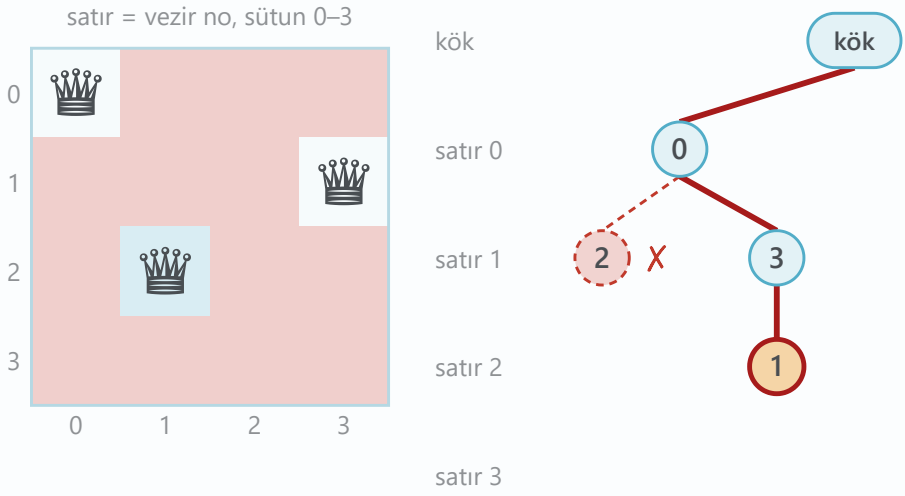
4 Vezir: Tahta ve Durum Uzayı Ağacı



Satır 1: **sütun 3 güvenli** → vezir yerleştirilir, bir alt düzeye (satır 2) inilir.

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli *X*: geri dönülen dal · yeşil: çözüm.

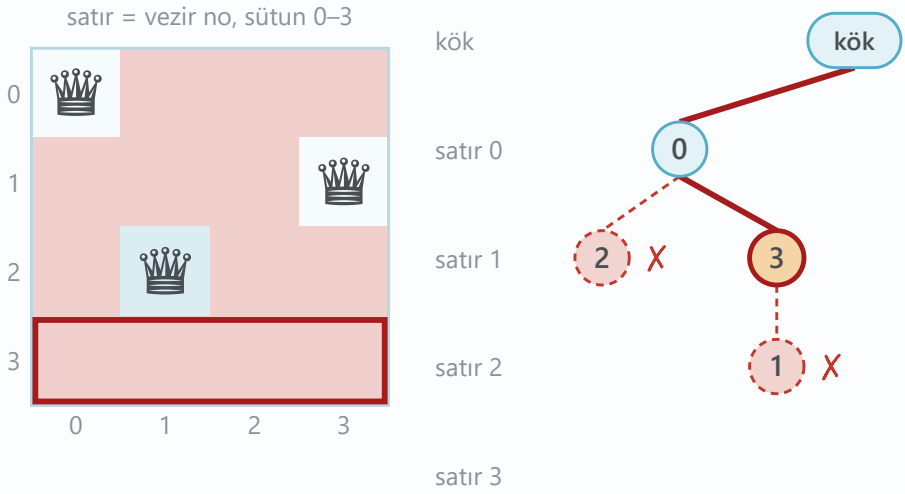
4 Vezir: Tahta ve Durum Uzayı Ağacı



Satır 2: sütun 0 tehdit altında; **sütun 1 güvenli** → vezir yerleştirilir, bir alt düzeye (satır 3) inilir.

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

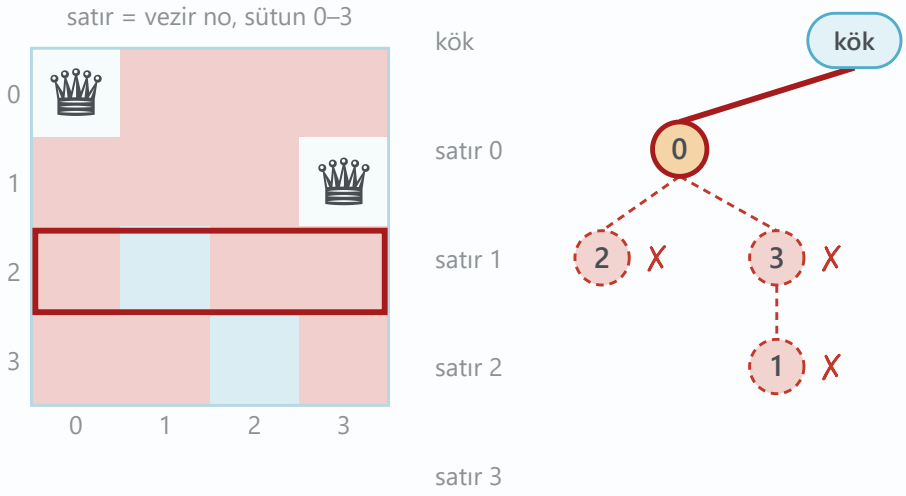
4 Vezir: Tahta ve Durum Uzayı Ağacı



Satır 3: sütun 0, 1, 2, 3 tehdit altında → **güvenli sütun yok** (çıkmaz). **Geri dön**: satır 2'deki vezir (sütun 1) kaldırılır ve o satırda sıradaki sütun denenir.

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

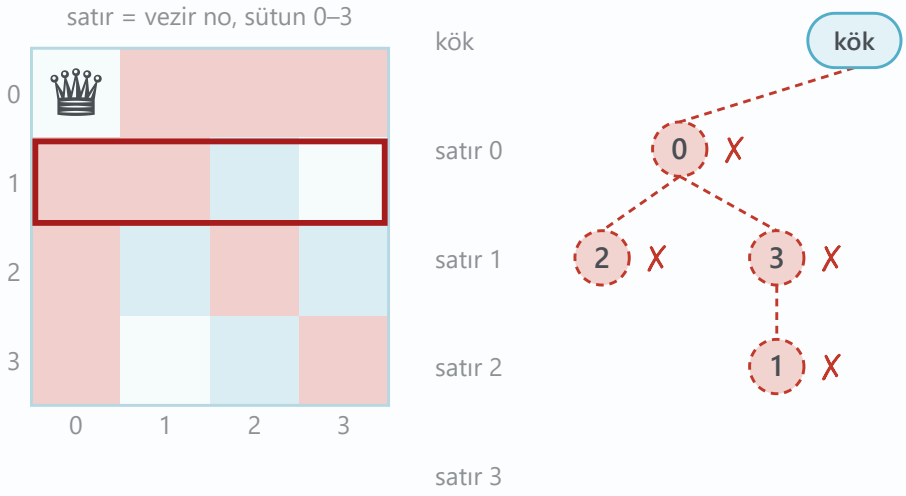
4 Vezir: Tahta ve Durum Uzayı Ağacı



Satır 2: sütun 2, 3 tehdit altında → **güvenli sütun yok** (çıkmaz). **Geri dön**: satır 1'deki vezir (sütun 3) kaldırılır ve o satırda sıradaki sütun denenir.

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

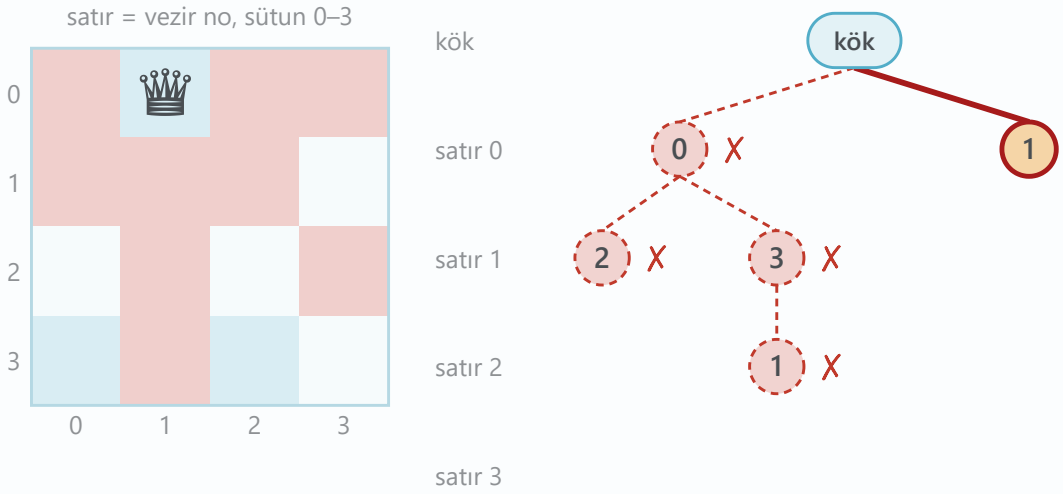
4 Vezir: Tahta ve Durum Uzayı Ağacı



Satır 1'in denenecek başka sütunu kalmadı. **Geri dön**: satır 0'daki vezir (sütun 0) da kaldırılır.

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

4 Vezir: Tahta ve Durum Uzayı Ağacı

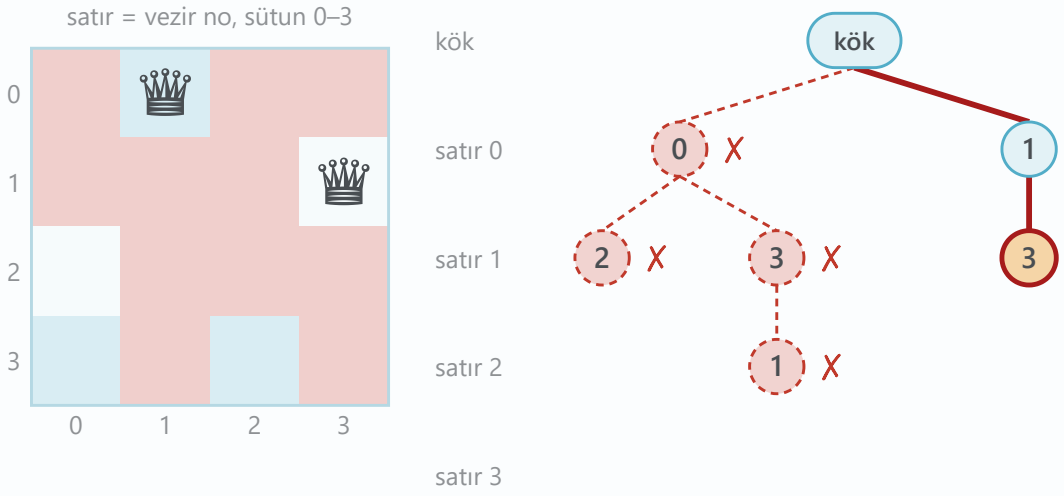


Satır 0: **sütun 1 güvenli** → vezir yerleştirilir, bir alt düzeye (satır 1) inilir.

Adım 10 / 13

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

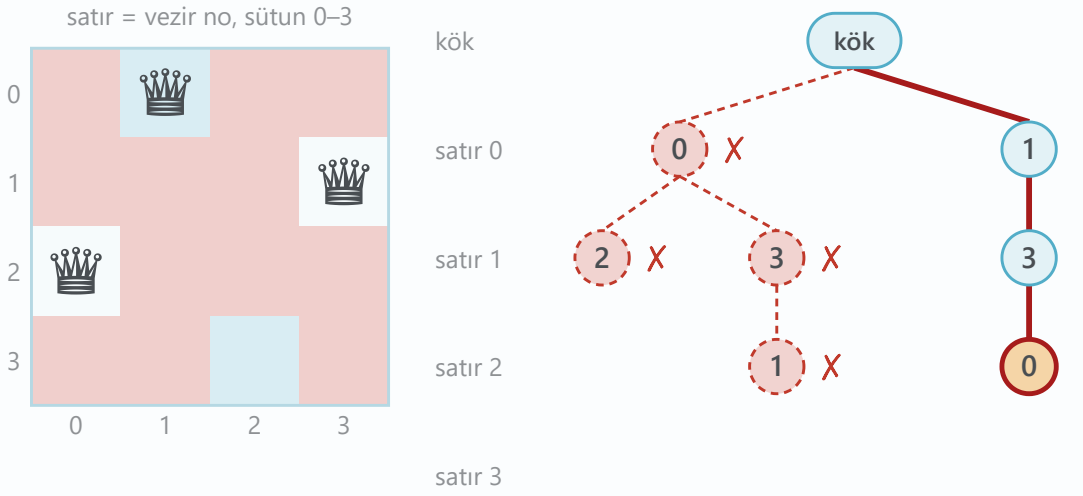
4 Vezir: Tahta ve Durum Uzayı Ağacı



Satır 1: sütun 0, 1, 2 tehdit altında; **sütun 3 güvenli** → vezir yerleştirilir, bir alt düzeye (satır 2) inilir.

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

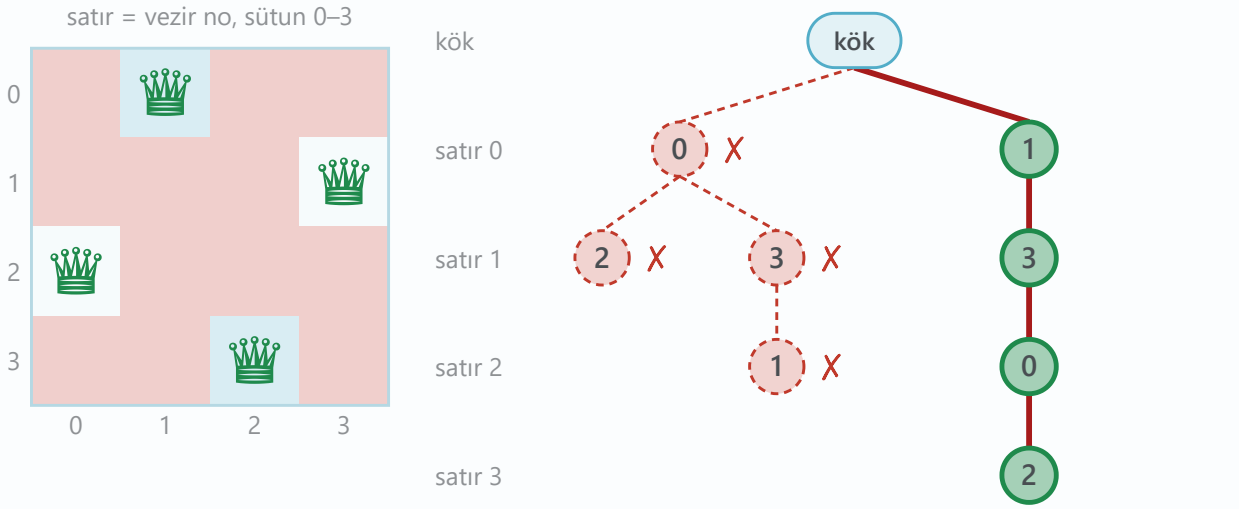
4 Vezir: Tahta ve Durum Uzayı Ağacı



Satır 2: **sütun 0 güvenli** → vezir yerleştirilir, bir alt düzeye (satır 3) inilir.

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

4 Vezir: Tahta ve Durum Uzayı Ağacı



Satır 3: sütun 0, 1 tehdit altında; **sütun 2 güvenli** → dördüncü vezir yerleşti. `satir == n` : **çözüm** bulundu: sütunlar [1, 3, 0, 2]. Toplam 8 yerleştirme, 4 geri dönüş.

Adım 13 / 13

Kırmızı hücre: mevcut vezirlerce tehdit edilen kare · turuncu düğüm: son yerleştirilen · kesikli ✗: geri dönülen dal · yeşil: çözüm.

n-Vezir: Java Kodu

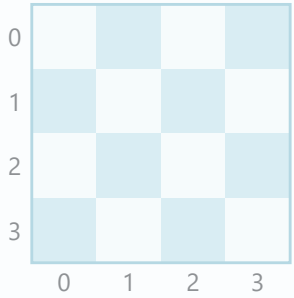
```java
boolean guvenliMi(int[][] t,
        int s, int k) {
  // aynı sütun
  for (int i = 0; i < s; i++)
    if (t[i][k] == 1) return false;
  // sol üst çapraz
  for (int i = s-1, j = k-1;
       i >= 0 && j >= 0; i--, j--)
    if (t[i][j] == 1) return false;
  // sağ üst çapraz
  for (int i = s-1, j = k+1;
       i >= 0 && j < n; i--, j++)
    if (t[i][j] == 1) return false;
  return true;
}
```

```java
boolean coz(int[][] t, int satir) {
  if (satir == n)
    return true;
  for (int sutun = 0; sutun < n;
       sutun++) {
    if (guvenliMi(t, satir, sutun)) {
      t[satir][sutun] = 1; // yerleştir
      if (coz(t, satir + 1))
        return true;
      t[satir][sutun] = 0; // geri al
    }
  }
  return false;
}
```

- `t[i][j] == 1` : (i, j)'de vezir var; `n` sınıfın bir alanı. Vezirler yalnızca **üst satırlarda** olduğundan `guvenliMi` üç yöne bakar: yukarı, sol üst ve sağ üst çapraz.
- `coz` satır satır ilerler; **ilk** çözümde `true` döner ve yığın boyunca yukarı taşınır. `guvenliMi` O(n); `coz` en kötü durumda O(n!).

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
4      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



`coz(t, 0)` çağrıldı: tahta boş, `satir` = 0.

Adım 1 / 36

DEĞİŞKENLER

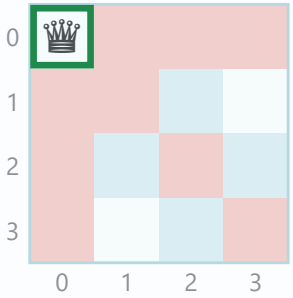
- `satir = 0`
- `sutun = –`
- `guvenliMi = –`
- `yer = [–,–,–,–]`

ÇAĞRI YIĞINI

`coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
▶        t[satir][sutun] = 1;
▶        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(0, 0) güvenli → `t[0][0] = 1`, ardından `coz(t, 1)` çağrılır.

Adım 2 / 36

DEĞİŞKENLER

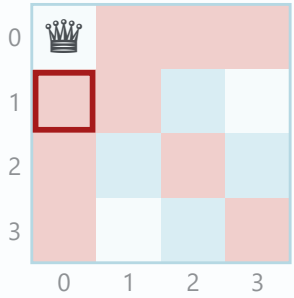
- `satir = 0`
- `sutun = 0`
- `guvenliMi = true`
- `yer = [0,-,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(1, 0) tehdit altında (aynı sütun: (0, 0)) → `guvenliMi` false, sonraki sütun.

Adım 3 / 36

DEĞİŞKENLER

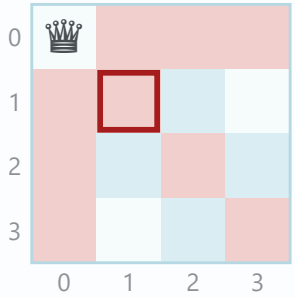
- `satir = 1`
- `sutun = 0`
- `guvenliMi = false`
- `yer = [0,-,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(1, 1) tehdit altında (çapraz: (0, 0)) → `guvenliMi` false, sonraki sütun.

Adım 4 / 36

DEĞİŞKENLER

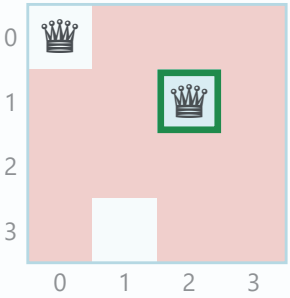
- `satir = 1`
- `sutun = 1`
- `guvenliMi = false`
- `yer = [0,-,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
▶        t[satir][sutun] = 1;
▶        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(1, 2) güvenli → `t[1][2] = 1`, ardından `coz(t, 2)` çağrılır.

Adım 5 / 36

DEĞİŞKENLER

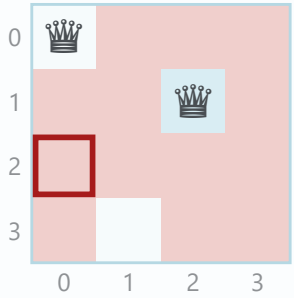
- `satir = 1`
- `sutun = 2`
- `guvenliMi = true`
- `yer = [0,2,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(2, 0) tehdit altında (aynı sütun: (0, 0)) → `guvenliMi` false, sonraki sütun.

Adım 6 / 36

DEĞİŞKENLER

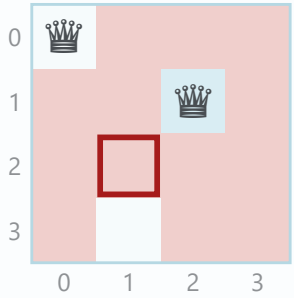
- `satir = 2`
- `sutun = 0`
- `guvenliMi = false`
- `yer = [0,2,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(2, 1) tehdit altında (çapraz: (1, 2)) → `guvenliMi` false, sonraki sütun.

Adım 7 / 36

DEĞİŞKENLER

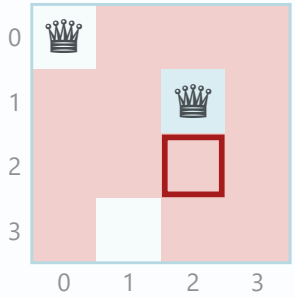
- `satir = 2`
- `sutun = 1`
- `guvenliMi = false`
- `yer = [0,2,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(2, 2) tehdit altında (aynı sütun: (1, 2)) → `guvenliMi` false, sonraki sütun.

Adım 8 / 36

DEĞİŞKENLER

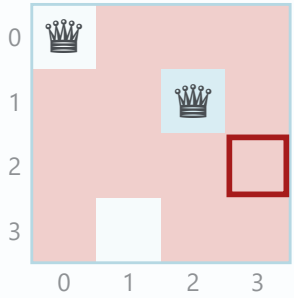
- `satir = 2`
- `sutun = 2`
- `guvenliMi = false`
- `yer = [0,2,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(2, 3) tehdit altında (çapraz: (1, 2)) → `guvenliMi` false, sonraki sütun.

Adım 9 / 36

DEĞİŞKENLER

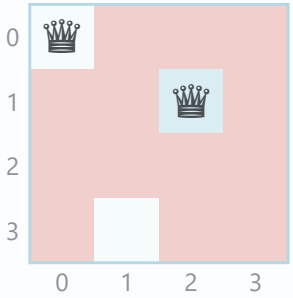
- `satir = 2`
- `sutun = 3`
- `guvenliMi = false`
- `yer = [0,2,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
4      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
▶    return false;
11 }
```



Satır 2 için döngü bitti, güvenli sütun yok → `return false`.

Adım 10 / 36

DEĞİŞKENLER

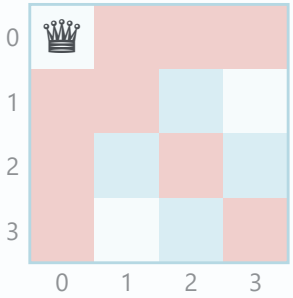
- `satir = 2`
- `sutun = 4`
- `guvenliMi = –`
- `yer = [0,2,–,–]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
4      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
▶        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



`coz(t, 2)` false döndü → **geri al**: `t[1][2] = 0` ; döngü sütun 3 ile sürer.

Adım 11 / 36

DEĞİŞKENLER

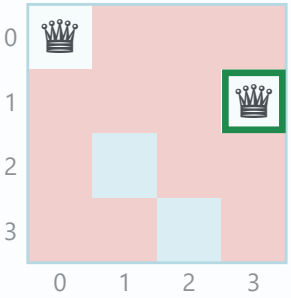
- `satir = 1`
- `sutun = 2`
- `guvenliMi = true`
- `yer = [0,-,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
▶        t[satir][sutun] = 1;
▶        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(1, 3) güvenli → `t[1][3] = 1`, ardından `coz(t, 2)` çağrılır.

Adım 12 / 36

DEĞİŞKENLER

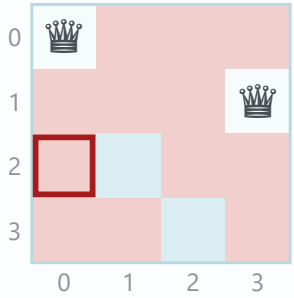
- `satir = 1`
- `sutun = 3`
- `guvenliMi = true`
- `yer = [0,3,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(2, 0) tehdit altında (aynı sütun: (0, 0)) → `guvenliMi` false, sonraki sütun.

Adım 13 / 36

DEĞİŞKENLER

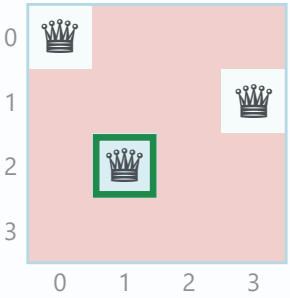
- `satir = 2`
- `sutun = 0`
- `guvenliMi = false`
- `yer = [0,3,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
▶        t[satir][sutun] = 1;
▶        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(2, 1) güvenli → `t[2][1] = 1`, ardından `coz(t, 3)` çağrılır.

Adım 14 / 36

DEĞİŞKENLER

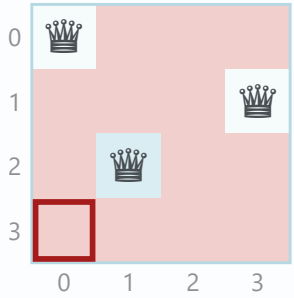
- `satir = 2`
- `sutun = 1`
- `guvenliMi = true`
- `yer = [0,3,1,-]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(3, 0) tehdit altında (aynı sütun: (0, 0)) → `guvenliMi` false, sonraki sütun.

Adım 15 / 36

DEĞİŞKENLER

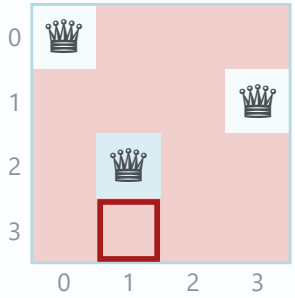
- `satir = 3`
- `sutun = 0`
- `guvenliMi = false`
- `yer = [0,3,1,-]`

ÇAĞRI YIĞINI

- `coz(t, 3)`
- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(3, 1) tehdit altında (aynı sütun: (2, 1)) → `guvenliMi` false, sonraki sütun.

Adım 16 / 36

DEĞİŞKENLER

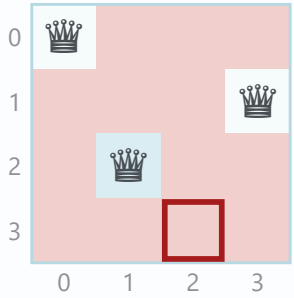
- `satir = 3`
- `sutun = 1`
- `guvenliMi = false`
- `yer = [0,3,1,-]`

ÇAĞRI YIĞINI

- `coz(t, 3)`
- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(3, 2) tehdit altında (çapraz: (2, 1)) → `guvenliMi` false, sonraki sütun.

Adım 17 / 36

DEĞİŞKENLER

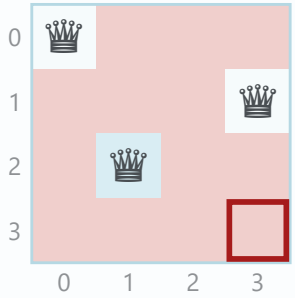
- `satir = 3`
- `sutun = 2`
- `guvenliMi = false`
- `yer = [0,3,1,-]`

ÇAĞRI YIĞINI

- `coz(t, 3)`
- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(3, 3) tehdit altında (aynı sütun: (1, 3)) → `guvenliMi` false, sonraki sütun.

Adım 18 / 36

DEĞİŞKENLER

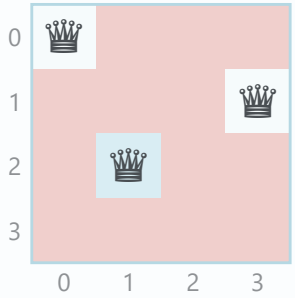
- `satir = 3`
- `sutun = 3`
- `guvenliMi = false`
- `yer = [0,3,1,-]`

ÇAĞRI YIĞINI

- `coz(t, 3)`
- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
4      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
▶    return false;
11 }
```



Satır 3 için döngü bitti, güvenli sütun yok → `return false`.

Adım 19 / 36

DEĞİŞKENLER

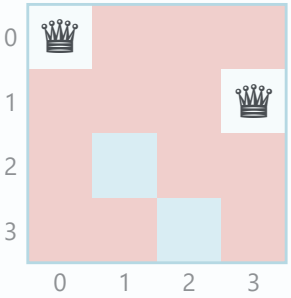
- `satir = 3`
- `sutun = 4`
- `guvenliMi = –`
- `yer = [0,3,1,–]`

ÇAĞRI YIĞINI

- `coz(t, 3)`
- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
4      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
▶        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



`coz(t, 3)` false döndü → **geri al**: `t[2][1] = 0` ; döngü sütun 2 ile sürer.

Adım 20 / 36

DEĞİŞKENLER

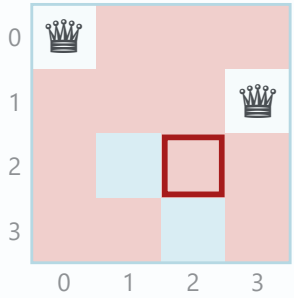
- `satir = 2`
- `sutun = 1`
- `guvenliMi = true`
- `yer = [0,3,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(2, 2) tehdit altında (çapraz: (0, 0)) → `guvenliMi` false, sonraki sütun.

Adım 21 / 36

DEĞİŞKENLER

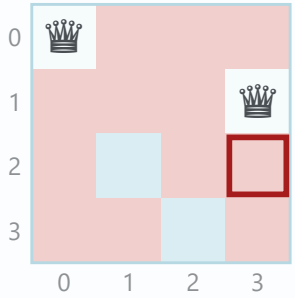
- `satir = 2`
- `sutun = 2`
- `guvenliMi = false`
- `yer = [0,3,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(2, 3) tehdit altında (aynı sütun: (1, 3)) → `guvenliMi` false, sonraki sütun.

Adım 22 / 36

DEĞİŞKENLER

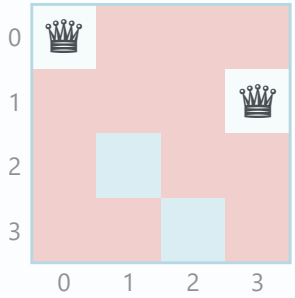
- `satir = 2`
- `sutun = 3`
- `guvenliMi = false`
- `yer = [0,3,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
4      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
▶    return false;
11 }
```



Satır 2 için döngü bitti, güvenli sütun yok → `return false`.

Adım 23 / 36

DEĞİŞKENLER

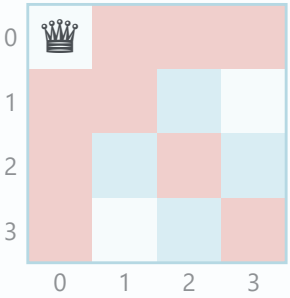
- `satir = 2`
- `sutun = 4`
- `guvenliMi = –`
- `yer = [0,3,–,–]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
4      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
▶        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



`coz(t, 2)` false döndü → **geri al**: `t[1][3] = 0` ; sütun 4 = n olduğundan döngü sona erer.

Adım 24 / 36

DEĞİŞKENLER

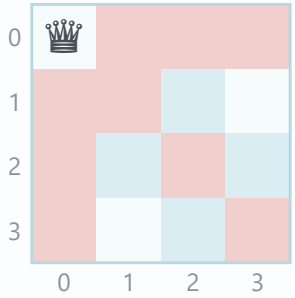
- `satir = 1`
- `sutun = 3`
- `guvenliMi = true`
- `yer = [0,-,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
4      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
▶    return false;
11 }
```



Satır 1 için döngü bitti, güvenli sütun yok → `return false`.

Adım 25 / 36

DEĞİŞKENLER

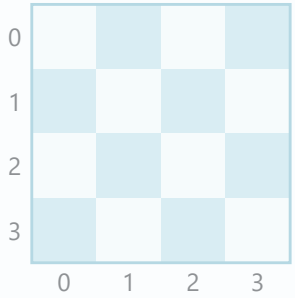
- `satir = 1`
- `sutun = 4`
- `guvenliMi = –`
- `yer = [0,–,–,–]`

ÇAĞRI YIĞINI

- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
4      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
▶        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



`coz(t, 1)` false döndü → **geri al**: `t[0][0] = 0` ; döngü sütun 1 ile sürer.

Adım 26 / 36

DEĞİŞKENLER

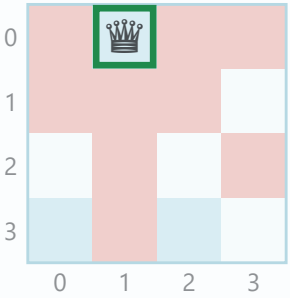
- `satir = 0`
- `sutun = 0`
- `guvenliMi = true`
- `yer = [-,-,-,-]`

ÇAĞRI YIĞINI

`coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
▶        t[satir][sutun] = 1;
▶        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(0, 1) güvenli → `t[0][1] = 1`, ardından `coz(t, 1)` çağrılır.

Adım 27 / 36

DEĞİŞKENLER

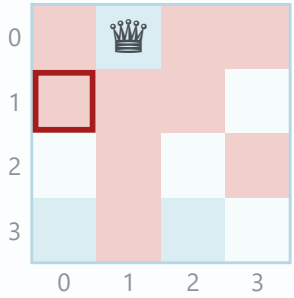
- `satir = 0`
- `sutun = 1`
- `guvenliMi = true`
- `yer = [1,-,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(1, 0) tehdit altında (çapraz: (0, 1)) → `guvenliMi` false, sonraki sütun.

Adım 28 / 36

DEĞİŞKENLER

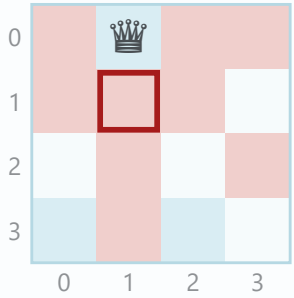
- `satir = 1`
- `sutun = 0`
- `guvenliMi = false`
- `yer = [1,-,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(1, 1) tehdit altında (aynı sütun: (0, 1)) → `guvenliMi` false, sonraki sütun.

Adım 29 / 36

DEĞİŞKENLER

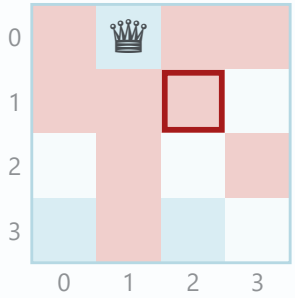
- `satir = 1`
- `sutun = 1`
- `guvenliMi = false`
- `yer = [1,-,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(1, 2) tehdit altında (çapraz: (0, 1)) → `guvenliMi` false, sonraki sütun.

Adım 30 / 36

DEĞİŞKENLER

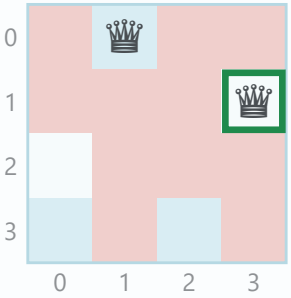
- `satir = 1`
- `sutun = 2`
- `guvenliMi = false`
- `yer = [1,-,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
▶        t[satir][sutun] = 1;
▶        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(1, 3) güvenli → `t[1][3] = 1`, ardından `coz(t, 2)` çağrılır.

Adım 31 / 36

DEĞİŞKENLER

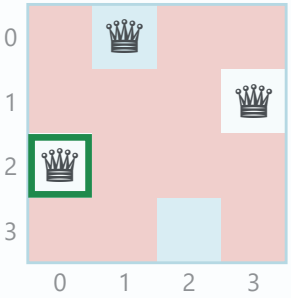
- `satir = 1`
- `sutun = 3`
- `guvenliMi = true`
- `yer = [1,3,-,-]`

ÇAĞRI YIĞINI

- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
▶        t[satir][sutun] = 1;
▶        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(2, 0) güvenli → `t[2][0] = 1`, ardından `coz(t, 3)` çağrılır.

Adım 32 / 36

DEĞİŞKENLER

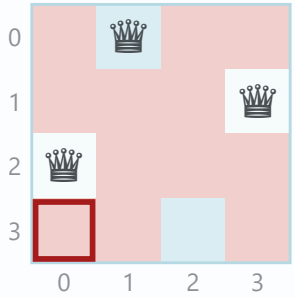
- `satir = 2`
- `sutun = 0`
- `guvenliMi = true`
- `yer = [1,3,0,-]`

ÇAĞRI YIĞINI

- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(3, 0) tehdit altında (aynı sütun: (2, 0)) → `guvenliMi` false, sonraki sütun.

Adım 33 / 36

DEĞİŞKENLER

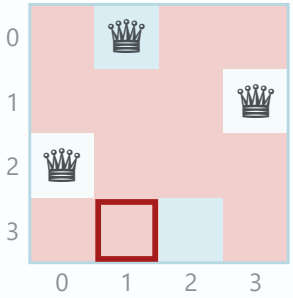
- `satir = 3`
- `sutun = 0`
- `guvenliMi = false`
- `yer = [1,3,0,-]`

ÇAĞRI YIĞINI

- `coz(t, 3)`
- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
▶    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(3, 1) tehdit altında (aynı sütun: (0, 1)) → `guvenliMi` false, sonraki sütun.

Adım 34 / 36

DEĞİŞKENLER

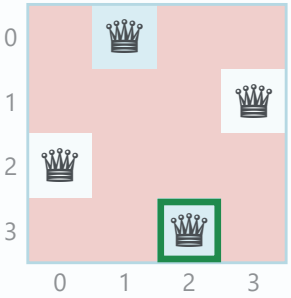
- `satir = 3`
- `sutun = 1`
- `guvenliMi = false`
- `yer = [1,3,0,-]`

ÇAĞRI YIĞINI

- `coz(t, 3)`
- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
2    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
▶      if (guvenliMi(t, satir, sutun)) {
▶        t[satir][sutun] = 1;
▶        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



(3, 2) güvenli → `t[3][2] = 1`, ardından `coz(t, 4)` çağrılır.

Adım 35 / 36

DEĞİŞKENLER

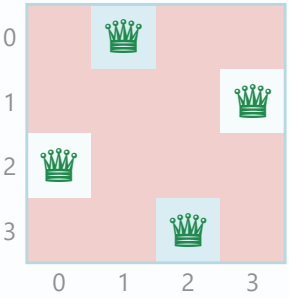
- `satir = 3`
- `sutun = 2`
- `guvenliMi = true`
- `yer = [1,3,0,2]`

ÇAĞRI YIĞINI

- `coz(t, 3)`
- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

coz(t, 0), n = 4: Kodu Adım Adım İzle

```
1  boolean coz(int[][] t, int satir) {
▶    if (satir == n) return true;
3    for (int sutun = 0; sutun < n; sutun++) {
4      if (guvenliMi(t, satir, sutun)) {
5        t[satir][sutun] = 1;
6        if (coz(t, satir + 1)) return true;
7        t[satir][sutun] = 0;  // geri al
8      }
9    }
10   return false;
11 }
```



`satir == n` → `true`. true değeri yığın boyunca yukarı taşınır: çözüm [1, 3, 0, 2].

Adım 36 / 36

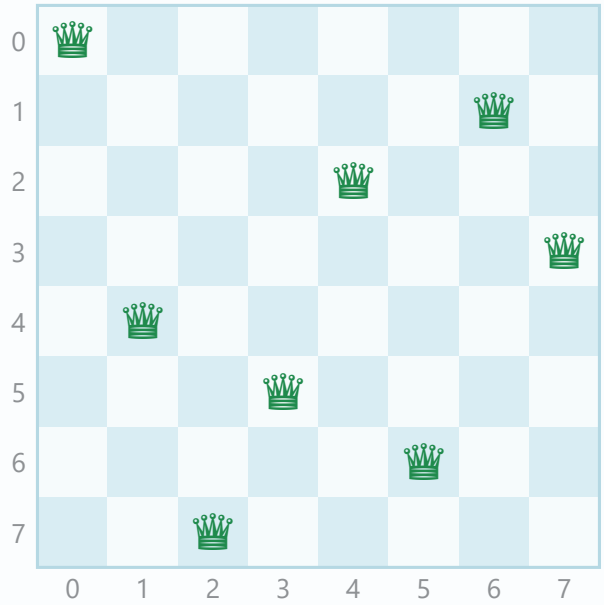
DEĞİŞKENLER

- `satir = 4`
- `sutun = –`
- `guvenliMi = –`
- `yer = [1,3,0,2]`

ÇAĞRI YIĞINI

- `coz(t, 4)`
- `coz(t, 3)`
- `coz(t, 2)`
- `coz(t, 1)`
- `coz(t, 0)`

8 Vezir: Algoritmanın İlk Çözümü



Satır 0–7 için sütunlar: [0, 4, 7, 5, 2, 6, 1, 3]

- `coz(t, 0)` (n = 8) sütunları küçükten büyüğe denediği için her zaman **sözlüksel sırada ilk** çözümü bulur.
- Bu çözüme ulaşana kadar **876** kez `guvenliMi` çağrılır, **113** vezir yerleştirilir ve **105** kez geri dönülür.
- (a1, b7, c4, d6, e8, f2, g5, h3) yerleşimi de bir çözümdür; 92 çözümün hepsi geçerlidir.
- Tüm çözümleri bulmak için `satir == n` olduğunda çözümü kaydedip `false` döndürmek (aramayı sürdürmek) yeterlidir.

N-VEZİR PROBLEMİ

n-Vezir: Çözüm ve Yerleştirme Sayıları

n	Çözüm	n^n (kaba)	Yerleştirme (tümü)	İlk çözüme kadar
1	1	1	1	1
2	0	4	2	—
3	0	27	5	—
4	2	256	16	8
5	10	3 125	53	5
6	4	46 656	152	31
7	40	823 543	551	9
8	92	16 777 216	2056	113
9	352	387 420 489	8393	41
10	724	10 000 000 000	35538	102

- n = 2 ve 3 için çözüm yoktur: algoritma tüm ağacı gezip `false` döner.
- "Yerleştirme": güvenli bulunup tahtaya konan vezir sayısı (ağaçtaki kök dışı düğüm).
- Tüm çözümleri bulma maliyeti n^n'den çok küçük, ama yine de hızla (yaklaşık n! gibi) büyür.
- İlk çözüme kadar geçen iş n ile düzgün artmaz: n = 6'da 31, n = 7'de yalnız 9 yerleştirme.

Daha Hızlı Güvenlik Testi: O(1)

```
boolean[] sut = new boolean[n];
boolean[] k1 = new boolean[2*n-1]; // r+c
boolean[] k2 = new boolean[2*n-1]; // r-c+n-1

void coz(int r) {
  if (r == n) { sayac++; return; }
  for (int c = 0; c < n; c++) {
    int a = r + c, b = r - c + n - 1;
    if (sut[c] || k1[a] || k2[b])
      continue;
    sut[c] = k1[a] = k2[b] = true;
    coz(r + 1);
    sut[c] = k1[a] = k2[b] = false;
  }
}
```

k1: r + c

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

k2: r − c + n − 1

3	2	1	0
4	3	2	1
5	4	3	2
6	5	4	3

- Aynı **ters köşegendeki** hücrelerde r + c sabittir (0 … 2n − 2).
- Aynı **köşegendeki** hücrelerde r − c sabittir; + n − 1 ile indis 0'dan başlar.

Her vezir üç bayrak tutar: sütun, ters köşegen, köşegen. `guvenliMi` O(n) yerine **O(1)** olur; tahta dizisine bile gerek kalmaz (yalnızca `yer[satir] = sutun`).

N-VEZİR PROBLEMİ

Deneyin: n-Vezirde Geri İzleme

n 6 adım oynat sonraki çözüm sıfırla

Boş tahta (n = 6). Vezirler satır satır yerleştirilir; her satırda sütunlar soldan sağa denenir. · yerleştirme: 0, geri dönüş: 0, çözüm: 0

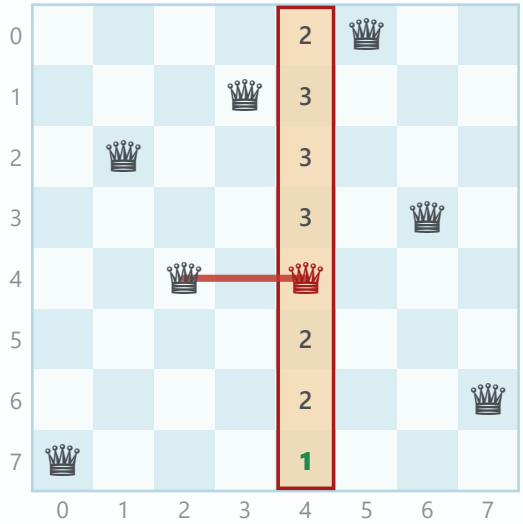
Kırmızı kare: tehdit altında · yeşil çerçeve: güvenli deneme · kırmızı çerçeve: reddedilen deneme · yeşil vezirler: çözüm.

En Az Çakışma Sezgiseli (Min-Conflicts)

- Farklı bir fikir: önce **tüm vezirler** yerleştirilir (her sütuna bir tane), sonra çakışmalar **yerel değişikliklerle** onarılır.
- Her hücreye yazılan sayı: o satıra konacak vezirin **kaç vezirle çakışacağı** (satır + çaprazlar).
- **Başlangıç:** sütun sütun, en az çakışmalı satır seçilir (eşitlikte rastgele).
- **Onarım:** çakışan bir vezir seçilir ve sütunu içinde en az çakışmalı satıra taşınır. Çakışma kalmayınca durulur.
- Bu bir **yerel arama** (local search) yöntemidir: durum tam ama geçersiz olabilen bir yerleşimdir, geri dönüş yığını yoktur. Tam değildir: platoda dolaşabilir, adım sınırı gerekir, yalnız bir çözüm bulur ve çözüm olmadığını kanıtlayamaz. Buna karşılık n = 1 000 000 için bile ortalama birkaç onlu adımda çözer.

```
enAzCakisma(n, maksAdim):
  // başlangıç: açgözlü yerleşim
  for k = 0 .. n-1:
    yer[k] = en az çakışmalı satır
  for adim = 1 .. maksAdim:
    if çakışma yok: return yer
    k = rastgele çakışan sütun
    yer[k] = argmin_r cakisma(k, r)
             // eşitlikte rastgele
  return başarısız
```

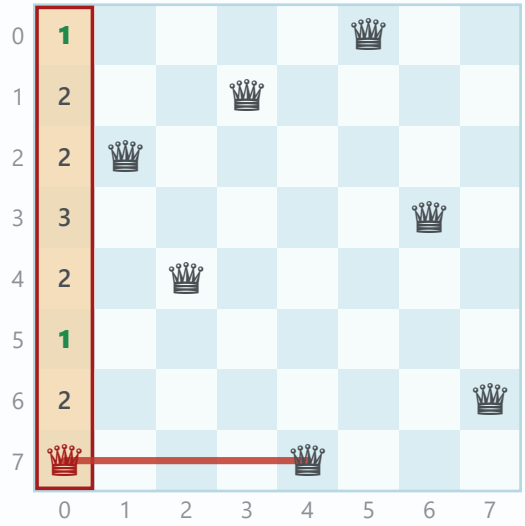
En Az Çakışma: Onarım Adımları



Açgözlü başlangıç [7, 2, 4, 1, 4, 0, 3, 6]: yalnız sütun 2 ve 4 (satır 4) çakışıyor. Sütun 4 seçilir; en küçük **1** (satır 7) → vezir **satır 7**'ye.

Vurgulu sütundaki sayılar, kırmızı vezirin o satıra taşınması hâlinde kaç vezirle çakışacağını gösterir (vezirin kendi satırı boş bırakıldı).

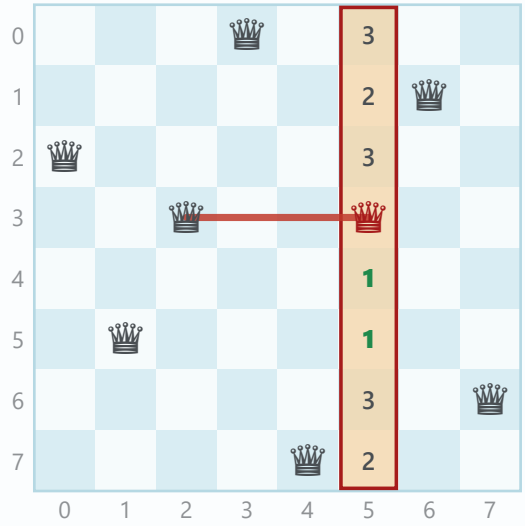
En Az Çakışma: Onarım Adımları



Taşıma yeni bir çakışma doğurdu: sütun 0 ile sütun 4 artık aynı satırda (7). Sütun 0'ın seçenekleri: en küçük 1 (satır 0, 5). Toplam çakışan çift: 1.

Vurgulu sütundaki sayılar, kırmızı vezirin o satıra taşınması hâlinde kaç vezirle çakışacağını gösterir (vezirin kendi satırı boş bırakıldı).

En Az Çakışma: Onarım Adımları

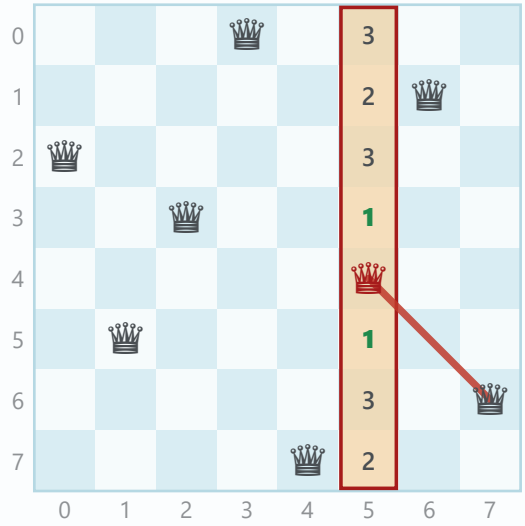


Birkaç onarım adımı sonra: sütun 5'teki vezir satır 3'te, sütun 2 ile çakışıyor. En küçük değer 1 (satır 4, 5) → satır 4.

Adım 3 / 9

Vurgulu sütundaki sayılar, kırmızı vezirin o satıra taşınması hâlinde kaç vezirle çakışacağını gösterir (vezirin kendi satırı boş bırakıldı).

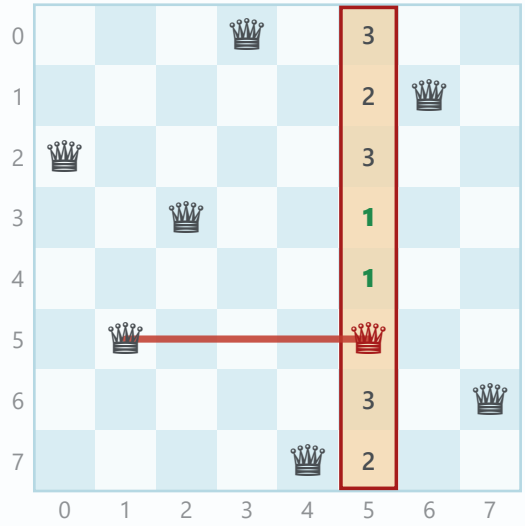
En Az Çakışma: Onarım Adımları



Satır 4'te bu kez (6, 7) ile çapraz çakışma var (1). Satır 3, 4, 5'in hepsi 1 çakışmalı: bir **plato**. Rastgele seçimle satır 5'e geçer.

Vurgulu sütundaki sayılar, kırmızı vezirin o satıra taşınması hâlinde kaç vezirle çakışacağını gösterir (vezirin kendi satırı boş bırakıldı).

En Az Çakışma: Onarım Adımları

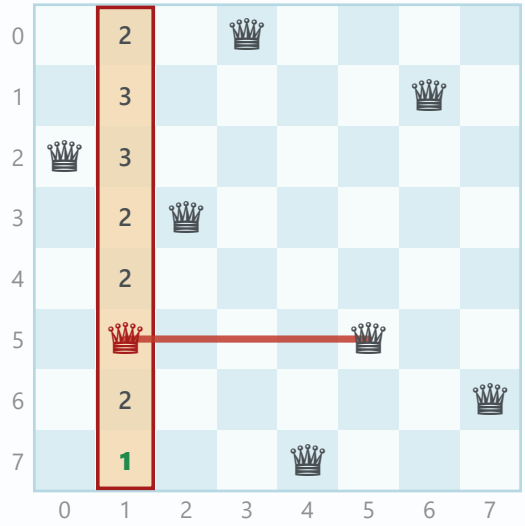


Satır 5'te sütun 1 ile aynı satırda. Sütun 5 platoda 3 ↔ 4 ↔ 5 arasında birkaç kez gidip gelir; rastgele seçim sayesinde sonunda başka bir vezir seçilir.

Adım 5 / 9

Vurgulu sütundaki sayılar, kırmızı vezirin o satıra taşınması hâlinde kaç vezirle çakışacağını gösterir (vezirin kendi satırı boş bırakıldı).

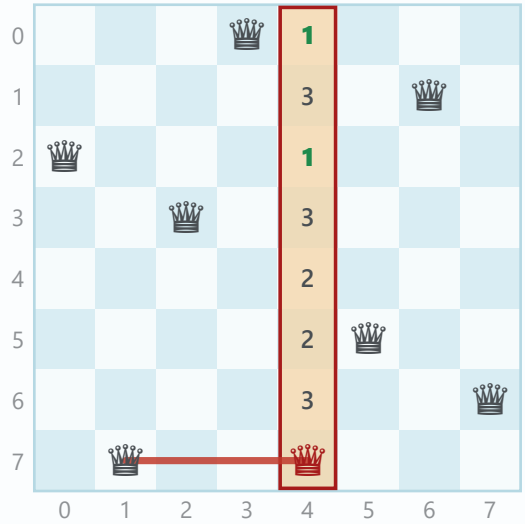
En Az Çakışma: Onarım Adımları



Sütun 1 seçildi (satır 5, 1 çakışma). En küçük değer **1** (satır 7) → satır 7.

Vurgulu sütundaki sayılar, kırmızı vezirin o satıra taşınması hâlinde kaç vezirle çakışacağını gösterir (vezirin kendi satırı boş bırakıldı).

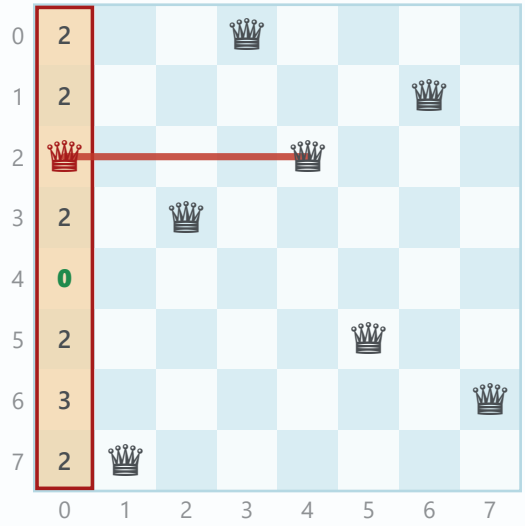
En Az Çakışma: Onarım Adımları



Şimdi sütun 1 ile sütun 4 satır 7'de çakışıyor. Sütun 4'ün en küçüğü 1 (satır 0, 2) → satır 2.

Vurgulu sütundaki sayılar, kırmızı vezirin o satıra taşınması hâlinde kaç vezirle çakışacağını gösterir (vezirin kendi satırı boş bırakıldı).

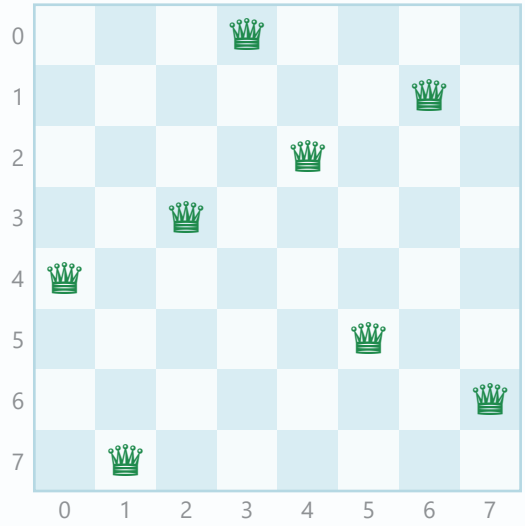
En Az Çakışma: Onarım Adımları



Sütun 0 ile sütun 4 satır 2'de çakışıyor. Sütun 0'da satır 4 **0 çakışmalı** → vezir oraya taşınır.

Vurgulu sütundaki sayılar, kırmızı vezirin o satıra taşınması hâlinde kaç vezirle çakışacağını gösterir (vezirin kendi satırı boş bırakıldı).

En Az Çakışma: Onarım Adımları



Çakışan vezir kalmadı: **çözüm** [4, 7, 3, 0, 2, 5, 1, 6]. Algoritma 8 vezirin hepsini baştan yerleştirdi, sonra yalnızca **yerel değişikliklerle** çakışmaları giderdi.

Vurgulu sütundaki sayılar, kırmızı vezirin o satıra taşınması hâlinde kaç vezirle çakışacağını gösterir (vezirin kendi satırı boş bırakıldı).

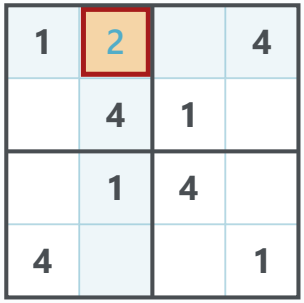
Sudoku

- Sudoku'nun amacı, 9×9'luk bir karede **her satır**, **her sütun** ve **her 3×3'lük küçük karede** 1'den 9'a kadar olan rakamları **birer kez** kullanmaktır.
- Bazı hücreler baştan verilir (**ipucu**); geri kalanlar doldurulur.
- Bu bir **kısıt sağlama problemi**dir (constraint satisfaction): değişkenler boş hücreler, alan {1, …, 9}, kısıtlar "aynı birimde farklı".

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Sudoku: Geri İzleme Algoritması

1. **Boş** bir hücreye bir rakam atanır.
2. Atanan rakam **geçerliyse** (satır, sütun ve kutuda yoksa) bir sonraki boş hücreye **geçilir**.
3. Geçerli değilse **sıradaki rakam** denenir.
4. Bir hücrede **tüm rakamlar** denenmişse: **önceki hücreye geri dönülür** (backtrack) ve orada farklı bir rakam denenir.
5. Adımlar tüm hücreler dolana kadar tekrarlanır.



Bu, genel şablonun doğrudan uygulamasıdır: **kısmi çözüm** = doldurulmuş hücreler, **aday** = 1–9, **kısıt** = `gecerliMi` , **hedef** = boş hücre kalmaması.

Sudoku: Java Kodu

```java
boolean coz(int[][] t) {
  for (int s = 0; s < 9; s++)
    for (int k = 0; k < 9; k++)
      if (t[s][k] == 0) {
        for (int r = 1; r <= 9; r++)
          if (gecerliMi(t, s, k, r)) {
            t[s][k] = r;
            if (coz(t)) return true;
            t[s][k] = 0;   // geri al
          }
        return false;  // rakam yok
      }
  return true;  // boş hücre yok
}
```

```java
boolean gecerliMi(int[][] t,
    int s, int k, int r) {
  for (int i = 0; i < 9; i++)
    if (t[s][i] == r || t[i][k] == r)
      return false;
  int bs = s / 3 * 3;
  int bk = k / 3 * 3;
  for (int i = bs; i < bs + 3; i++)
    for (int j = bk; j < bk + 3; j++)
      if (t[i][j] == r)
        return false;
  return true;
}
```

`coz` her çağrıda ilk boş hücreyi baştan arar (O(81)); bir hücrede hiçbir rakam olmazsa `return false` geri dönüşü tetikler.

SUDOKU

Sudoku: Geri İzleme Çözümü Adım Adım

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Bulmaca: 30 ipucu, **51 boş hücre**. Boş hücreler satır satır, soldan sağa doldurulur; her hücrede 1–9 sırayla denenir.

Adım 1 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı *X*: rakam kalmayan (çıkmaz) hücre.

Sudoku: Geri İzleme Çözümü Adım Adım

5	3	1	2	7	4	8	9	✗
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

İlk beş boş hücreye 1, 2, 4, 8, 9 girildi. (0, 8) için **hiçbir rakam geçerli değil**: satırda yalnız 6 eksik, o da sütunda var → çıkmaz. (şu ana kadar 5 yerleştirme, 0 geri alma)

Adım 2 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı ✗: rakam kalmayan (çıkmaz) hücre.

Sudoku: Geri İzleme Çözümü Adım Adım

5	3	1	2	7	6			
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Geri dönüş: (0, 7)'deki 9 silinir, (0, 6) 9 ile denenir, yine çıkmaz; geri dönüş (0, 5)'e kadar sürer ve orada 4 yerine **6** denenir. (şu ana kadar 7 yerleştirme, 4 geri alma)

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı ✗: rakam kalmayan (çıkmaz) hücre.

Sudoku: Geri İzleme Çözümü Adım Adım

5	3	1	2	7	6	4	9	8
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Satır 0 ilk kez tamamlandı: 5 3 1 2 7 6 4 9 8. Bu satır **kendi içinde** geçerli, ama çözüme götürüp götürmediği henüz bilinmiyor. (şu ana kadar 10 yerleştirme, 4 geri alma)

Adım 4 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı ✗: rakam kalmayan (çıkmaz) hücre.

Sudoku: Geri İzleme Çözümü Adım Adım

5	3	1	2	7	6	4	9	X
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Satır 1'deki tüm denemeler çıkmaza girdi. Geri dönüş **satır 0'a** kadar uzanır: (0, 8)'deki 8 silinir. Erken verilen bir karar ancak çok sonra yanlış çıkabilir. (şu ana kadar 31 yerleştirme, 26 geri alma)

Adım 5 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı X: rakam kalmayan (çıkmaz) hücre.

Sudoku: Geri İzleme Çözümü Adım Adım

5	3	1	6	7	8	4	9	2
6	2	4	1	9	5	7	3	8
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Satır 1 ilk kez tamamlandı. (şu ana kadar 229 yerleştirme, 218 geri alma)

Adım 6 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı X: rakam kalmayan (çıkmaz) hücre.

Sudoku: Geri İzleme Çözümü Adım Adım

5	3	2	6	7	8	1	9	4
6	4	7	1	9	5	3	2	8
1	9	8	2	3	4	5	6	7
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Satır 2 ilk kez tamamlandı; ilk üç satır son çözümdekiyle aynı. (şu ana kadar 377 yerleştirme, 360 geri alma)

Adım 7 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı X: rakam kalmayan (çıkmaz) hücre.

Sudoku: Geri İzleme Çözümü Adım Adım

5	3	2	6	7	8	1	9	4
6	4	7	1	9	5	3	2	8
1	9	8	2	3	4	5	6	7
8	2	1	5	6	7	9	4	3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

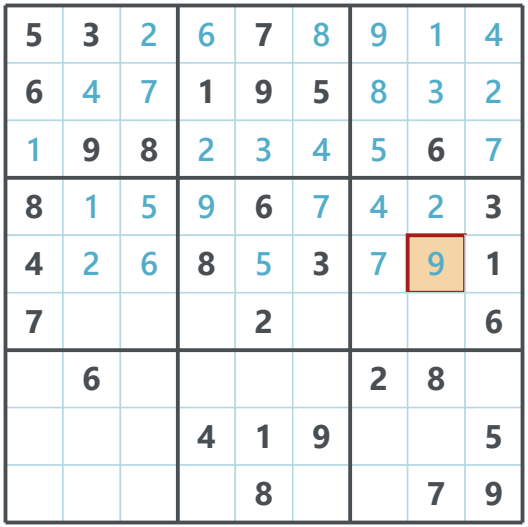
Satır 3 ilk kez tamamlandı. (şu ana kadar 395 yerleştirme, 372 geri alma)

Adım 8 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı X: rakam kalmayan (çıkmaz) hücre.

SUDOKU

Sudoku: Geri İzleme Çözümü Adım Adım



Satır 4 ilk kez tamamlandı. (şu ana kadar 1520 yerleştirme, 1492 geri alma)

Adım 9 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı X: rakam kalmayan (çıkmaz) hücre.

Sudoku: Geri İzleme Çözümü Adım Adım

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
	6					2	8	
			4	1	9			5
				8			7	9

Satır 5 ilk kez tamamlandı. (şu ana kadar 4185 yerleştirme, 4151 geri alma)

Adım 10 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı X: rakam kalmayan (çıkmaz) hücre.

Sudoku: Geri İzleme Çözümü Adım Adım

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
			4	1	9			5
				8			7	9

Satır 6 ilk kez tamamlandı. (şu ana kadar 4197 yerleştirme, 4157 geri alma)

Adım 11 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı X: rakam kalmayan (çıkmaz) hücre.

Sudoku: Geri İzleme Çözümü Adım Adım

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
				8			7	9

Satır 7 ilk kez tamamlandı. (şu ana kadar 4202 yerleştirme, 4157 geri alma)

Adım 12 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı X: rakam kalmayan (çıkmaz) hücre.

SUDOKU

Sudoku: Geri İzleme Çözümü Adım Adım

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Çözüm bulundu: 51 boş hücrenin hepsi dolu, tüm kısıtlar sağlanıyor. Toplam **4208 yerleştirme** ve **4157 geri alma**.

Adım 13 / 13

Siyah: ipucu · mavi: algoritmanın yazdığı rakam · turuncu: son yazılan hücre · kırmızı ✗: rakam kalmayan (çıkmaz) hücre.

Sudoku: Çözüm ve Maliyet

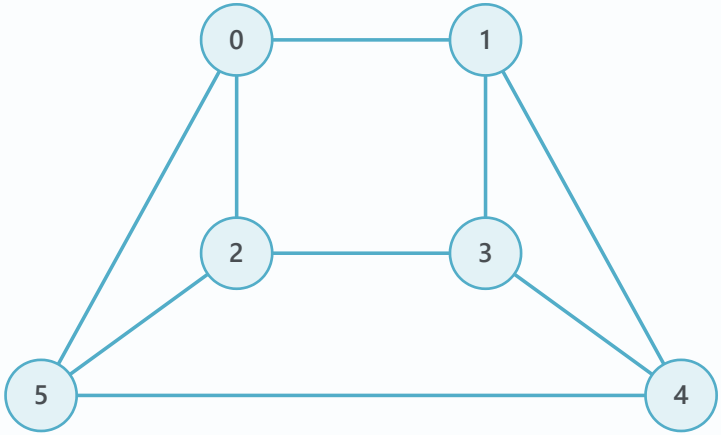
5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Strateji	Yerleştirme	Geri alma
Satır satır, ilk boş hücre	4208	4157
MRV: en az adaylı hücre	51	0

- Kaba kuvvet: 51 boş hücre için $9^{51} \approx 4{,}6 \cdot 10^{48}$ atama. Budama bunu birkaç bine indirir.
- **MRV** (minimum remaining values): her adımda **en az adayı kalan** hücre doldurulur. Bu bulmacada her seferinde tek adaylı hücre bulunur: hiç geri dönüş gerekmez.

Çizge Boyama (m-Coloring)

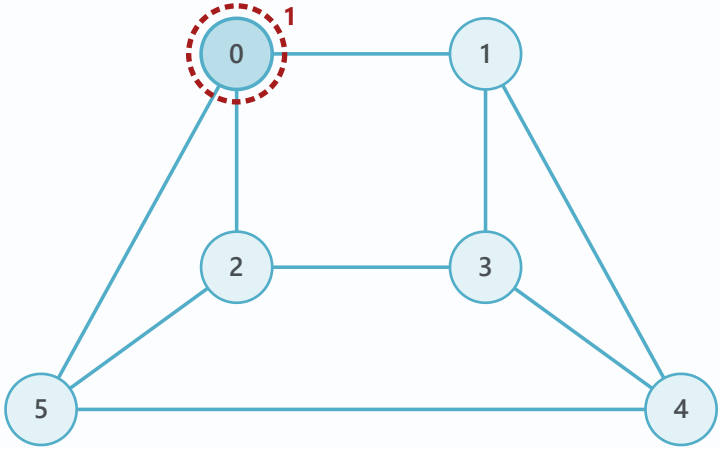
Komşu iki düğüm aynı rengi almayacak biçimde düğümlere en fazla **m** renk atanır. Kısmi çözüm: ilk k düğümün renkleri; kısıt: komşuyla aynı renk olmamak.



Altı düğümlü çizge, **m = 3** renk (1 mavi, 2 yeşil, 3 turuncu). Düğümler 0, 1, 2, … sırasıyla boyanır; her düğümde renk 1, 2, 3 sırayla denenir.

Çizge Boyama (m-Coloring)

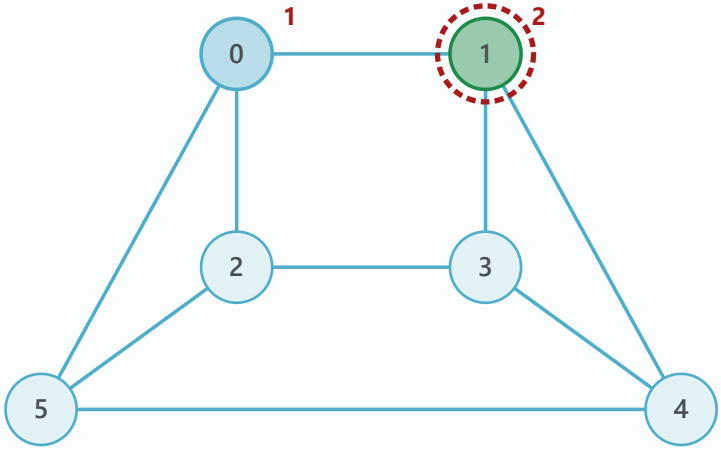
Komşu iki düğüm aynı rengi almayacak biçimde düğümlere en fazla **m** renk atanır. Kısmi çözüm: ilk k düğümün renkleri; kısıt: komşuyla aynı renk olmamak.



Düğüm 0: **renk 1 (mavi)** uygun → atanır.

Çizge Boyama (m-Coloring)

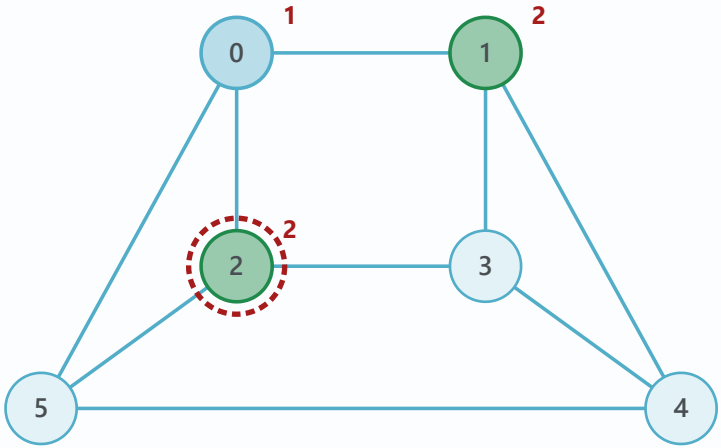
Komşu iki düğüm aynı rengi almayacak biçimde düğümlere en fazla **m** renk atanır. Kısmi çözüm: ilk k düğümün renkleri; kısıt: komşuyla aynı renk olmamak.



Düğüm 1: renk 1 komşuda var; **renk 2 (yeşil)** uygun → atanır.

Çizge Boyama (m-Coloring)

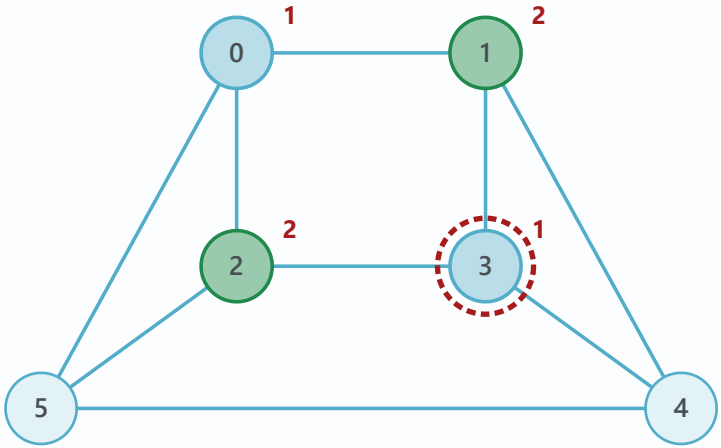
Komşu iki düğüm aynı rengi almayacak biçimde düğümlere en fazla **m** renk atanır. Kısmi çözüm: ilk k düğümün renkleri; kısıt: komşuyla aynı renk olmamak.



Düğüm 2: renk 1 komşuda var; **renk 2 (yeşil)** uygun → atanır.

Çizge Boyama (m-Coloring)

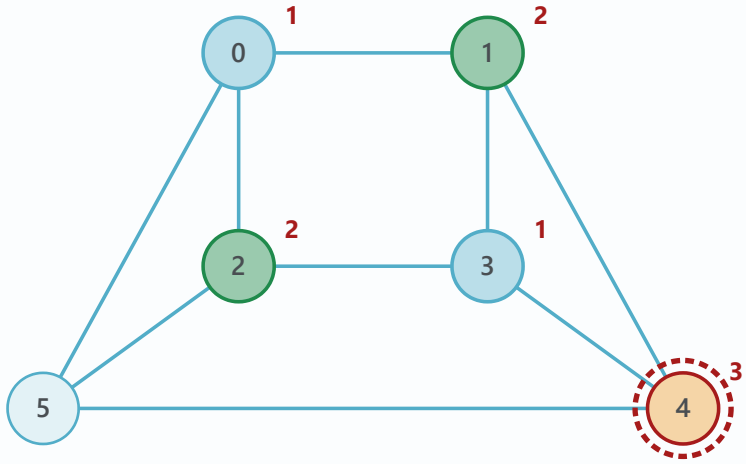
Komşu iki düğüm aynı rengi almayacak biçimde düğümlere en fazla **m** renk atanır. Kısmi çözüm: ilk k düğümün renkleri; kısıt: komşuyla aynı renk olmamak.



Düğüm 3: **renk 1 (mavi)** uygun → atanır.

Çizge Boyama (m-Coloring)

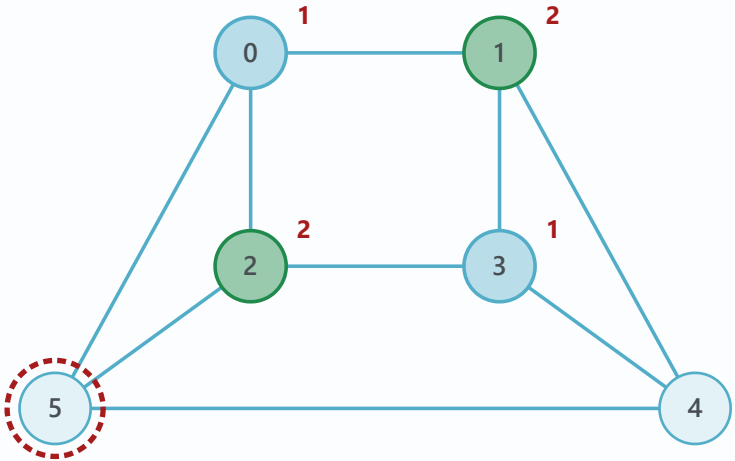
Komşu iki düğüm aynı rengi almayacak biçimde düğümlere en fazla **m** renk atanır. Kısmi çözüm: ilk k düğümün renkleri; kısıt: komşuyla aynı renk olmamak.



Düğüm 4: renk 1, 2 komşuda var; **renk 3 (turuncu)** uygun → atanır.

Çizge Boyama (m-Coloring)

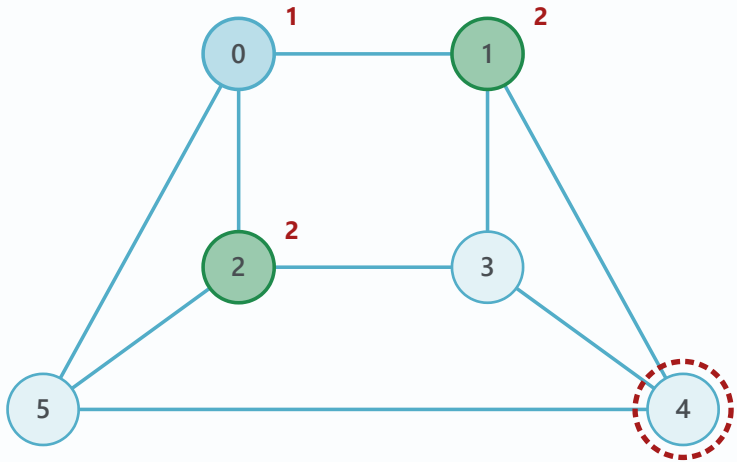
Komşu iki düğüm aynı rengi almayacak biçimde düğümlere en fazla **m** renk atanır. Kısmi çözüm: ilk k düğümün renkleri; kısıt: komşuyla aynı renk olmamak.



Düğüm 5: renk 1, 2, 3 komşularda kullanılıyor → **hiçbir renk uygun değil** (çıkmaz). **Geri dön**: düğüm 4'ün rengi (3) silinir, sıradaki renk denenecek.

Çizge Boyama (m-Coloring)

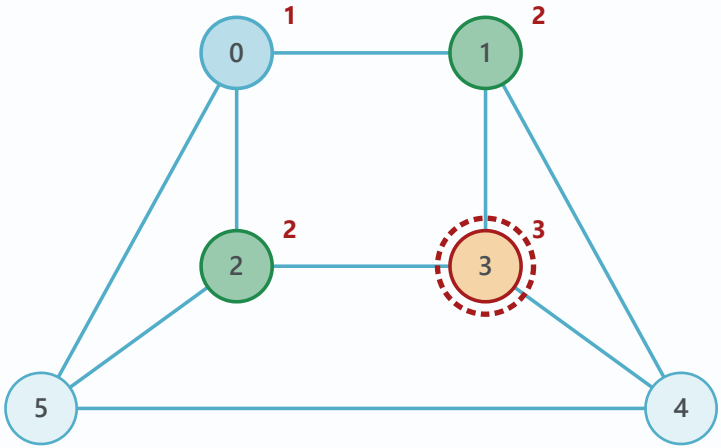
Komşu iki düğüm aynı rengi almayacak biçimde düğümlere en fazla **m** renk atanır. Kısmi çözüm: ilk k düğümün renkleri; kısıt: komşuyla aynı renk olmamak.



Düğüm 4: denenecek başka renk kalmadı (çıkmaz). **Geri dön**: düğüm 3'ün rengi (1) silinir, sıradaki renk denenecek.

Çizge Boyama (m-Coloring)

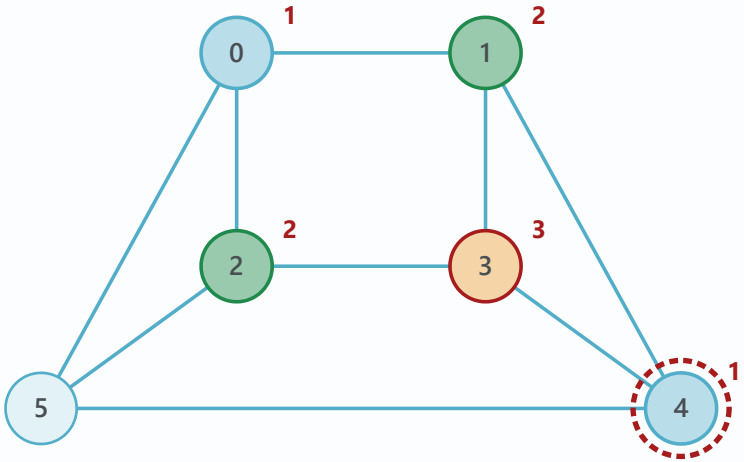
Komşu iki düğüm aynı rengi almayacak biçimde düğümlere en fazla **m** renk atanır. Kısmi çözüm: ilk k düğümün renkleri; kısıt: komşuyla aynı renk olmamak.



Düğüm 3: renk 2 komşuda var; **renk 3 (turuncu)** uygun → atanır.

Çizge Boyama (m-Coloring)

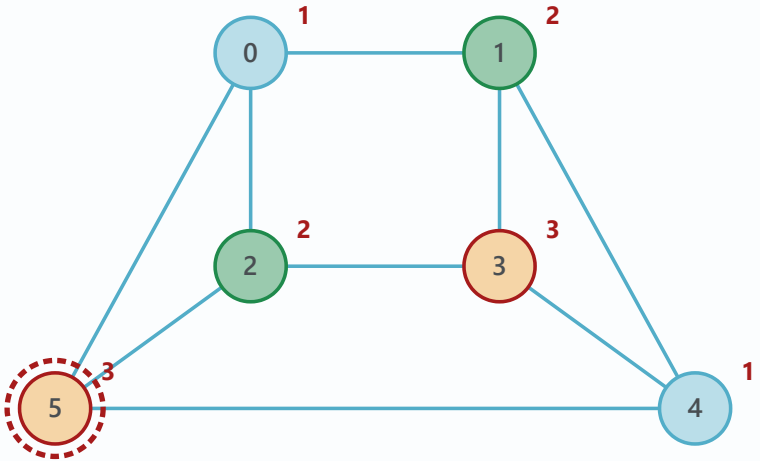
Komşu iki düğüm aynı rengi almayacak biçimde düğümlere en fazla **m** renk atanır. Kısmi çözüm: ilk k düğümün renkleri; kısıt: komşuyla aynı renk olmamak.



Düğüm 4: **renk 1 (mavi)** uygun → atanır.

Çizge Boyama (m-Coloring)

Komşu iki düğüm aynı rengi almayacak biçimde düğümlere en fazla **m** renk atanır. Kısmi çözüm: ilk k düğümün renkleri; kısıt: komşuyla aynı renk olmamak.



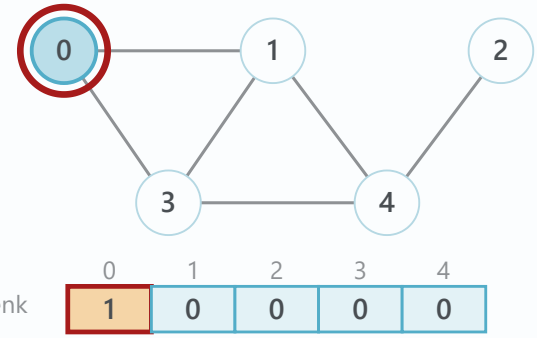
Düğüm 5: renk 1, 2 komşuda var; **renk 3 (turuncu)** uygun → atanır. Altı düğüm de boyandı: **geçerli 3-boyama** bulundu.

boya(0): m = 3 ile Kod İzleme

```
1  boolean boya(int v) {
2    if (v == n) return true;          // bitti
3    for (int c = 1; c <= m; c++) {
▶      if (uygunMu(v, c)) {
▶        renk[v] = c;                  // seç
▶        if (boya(v + 1)) return true; // keşfet
7        renk[v] = 0;                  // geri al
8      }
9    }
10   return false;                     // çıkmaz
11 }
12 // uygunMu: v'nin c renkli komşusu yoksa true
```

ÇAĞRI YIĞINI

- `boya(0)`
- `main()`



Beş düğüm, m = 3 renk (1 mavi, 2 yeşil, 3 turuncu); düğümler 0, 1, 2, … sırasıyla, renkler 1, 2, 3 sırasıyla denenir. Düğüm 0: **renk 1** atanır.

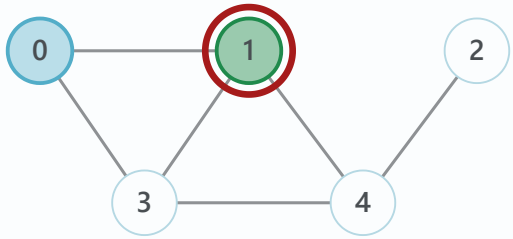
Adım 1 / 9

DEĞİŞKENLER

`v = 0` `c = 1` `renk = [1,0,0,0,0]`

boya(0): m = 3 ile Kod İzleme

```
1  boolean boya(int v) {
2    if (v == n) return true;          // bitti
3    for (int c = 1; c <= m; c++) {
▶      if (uygunMu(v, c)) {
▶        renk[v] = c;                  // seç
▶        if (boya(v + 1)) return true; // keşfet
7        renk[v] = 0;                  // geri al
8      }
9    }
10   return false;                     // çıkmaz
11 }
12 // uygunMu: v'nin c renkli komşusu yoksa true
```



	0	1	2	3	4
renk	1	2	0	0	0

Renk 1 bir komşuda var; düğüm 1: **renk 2 (yeşil)** uygun → atanır.

Adım 2 / 9

ÇAĞRI YIĞINI

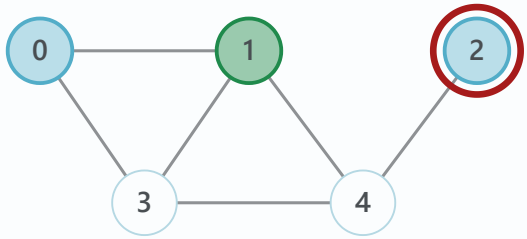
- `boya(1)`
- `boya(0)`
- `main()`

DEĞİŞKENLER

`v = 1` `c = 2` `renk = [1,2,0,0,0]`

boya(0): m = 3 ile Kod İzleme

```
1  boolean boya(int v) {
2    if (v == n) return true;            // bitti
3    for (int c = 1; c <= m; c++) {
▶      if (uygunMu(v, c)) {
▶        renk[v] = c;                    // seç
▶        if (boya(v + 1)) return true; // keşfet
7        renk[v] = 0;                    // geri al
8      }
9    }
10   return false;                       // çıkmaz
11 }
12 // uygunMu: v'nin c renkli komşusu yoksa true
```



düğüm 2: **renk 1 (mavi)** uygun → atanır.

ÇAĞRI YIĞINI

- `boya(2)`
- `boya(1)`
- `boya(0)`
- `main()`

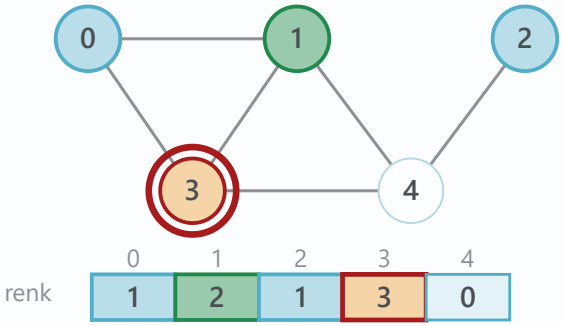
Adım 3 / 9

DEĞİŞKENLER

`v = 2` `c = 1` `renk = [1,2,1,0,0]`

boya(0): m = 3 ile Kod İzleme

```
1  boolean boya(int v) {
2    if (v == n) return true;            // bitti
3    for (int c = 1; c <= m; c++) {
▶      if (uygunMu(v, c)) {
▶        renk[v] = c;                    // seç
▶        if (boya(v + 1)) return true; // keşfet
7        renk[v] = 0;                    // geri al
8      }
9    }
10   return false;                       // çıkmaz
11 }
12 // uygunMu: v'nin c renkli komşusu yoksa true
```



Renk 1, 2 bir komşuda var; düğüm 3: **renk 3 (turuncu)** uygun → atanır.

Adım 4 / 9

ÇAĞRI YIĞINI

- `boya(3)`
- `boya(2)`
- `boya(1)`
- `boya(0)`
- `main()`

DEĞİŞKENLER

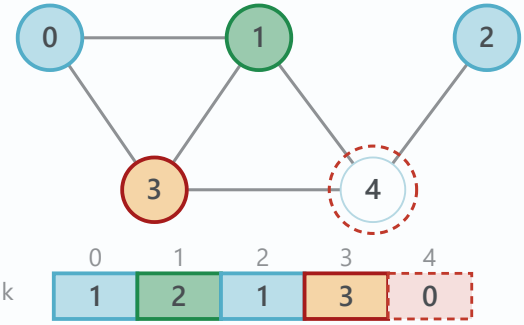
`v = 3` `c = 3` `renk = [1,2,1,3,0]`

boya(0): m = 3 ile Kod İzleme

```
1  boolean boya(int v) {
2    if (v == n) return true;          // bitti
3    for (int c = 1; c <= m; c++) {
4      if (uygunMu(v, c)) {
5        renk[v] = c;                  // seç
6        if (boya(v + 1)) return true; // keşfet
▶        renk[v] = 0;                  // geri al
8      }
9    }
▶    return false;                     // çıkmaz
11 }
12 // uygunMu: v'nin c renkli komşusu yoksa true
```

ÇAĞRI YIĞINI

- **boya(3)**
- boya(2)
- boya(1)
- boya(0)
- main()



Düğüm 4: renk 1, 2, 3 komşularında var → **çıkmaz**, `return false`. Geri dönülür: renk[3] = 0, düğüm 3 için sıradaki renk denenecek.

Adım 5 / 9

DEĞİŞKENLER

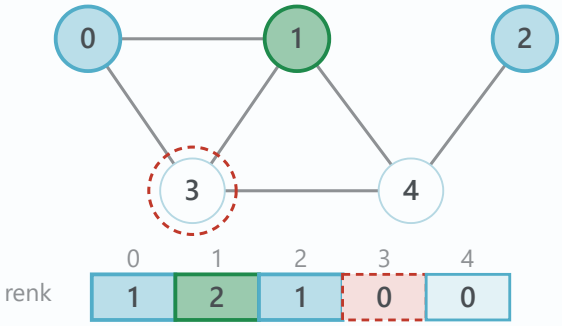
`v = 3` `c = 3` `renk = [1,2,1,0,0]`

boya(0): m = 3 ile Kod İzleme

```
1  boolean boya(int v) {
2    if (v == n) return true;           // bitti
3    for (int c = 1; c <= m; c++) {
4      if (uygunMu(v, c)) {
5        renk[v] = c;                   // seç
6        if (boya(v + 1)) return true; // keşfet
▶        renk[v] = 0;                   // geri al
8      }
9    }
▶    return false;                      // çıkmaz
11 }
12 // uygunMu: v'nin c renkli komşusu yoksa true
```

ÇAĞRI YIĞINI

- **boya(2)**
- boya(1)
- boya(0)
- main()



Düğüm 3: denenecek renk kalmadı → **çıkmaz**. Geri dönülür: renk[2] = 0 (renk 1 siliniyor).

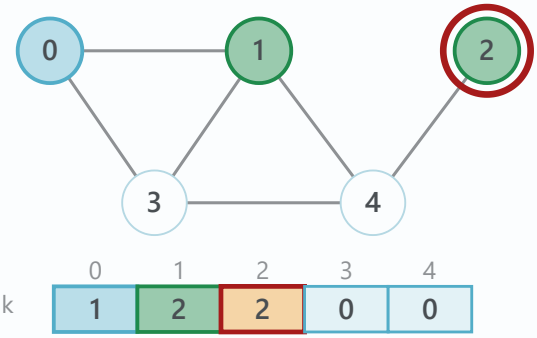
Adım 6 / 9

DEĞİŞKENLER

`v = 2` `c = 1` `renk = [1,2,0,0,0]`

boya(0): m = 3 ile Kod İzleme

```
1  boolean boya(int v) {
2    if (v == n) return true;            // bitti
3    for (int c = 1; c <= m; c++) {
▶      if (uygunMu(v, c)) {
▶        renk[v] = c;                    // seç
▶        if (boya(v + 1)) return true;   // keşfet
7        renk[v] = 0;                    // geri al
8      }
9    }
10   return false;                       // çıkmaz
11 }
12 // uygunMu: v'nin c renkli komşusu yoksa true
```



düğüm 2: **renk 2 (yeşil)** uygun → atanır.

ÇAĞRI YIĞINI

- **boya(2)**
- boya(1)
- boya(0)
- main()

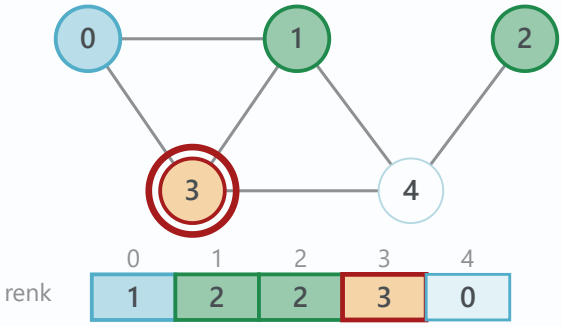
Adım 7 / 9

DEĞİŞKENLER

`v = 2` `c = 2` `renk = [1,2,2,0,0]`

boya(0): m = 3 ile Kod İzleme

```
1  boolean boya(int v) {
2    if (v == n) return true;          // bitti
3    for (int c = 1; c <= m; c++) {
▶      if (uygunMu(v, c)) {
▶        renk[v] = c;                  // seç
▶        if (boya(v + 1)) return true; // keşfet
7        renk[v] = 0;                  // geri al
8      }
9    }
10   return false;                     // çıkmaz
11 }
12 // uygunMu: v'nin c renkli komşusu yoksa true
```



Renk 1, 2 bir komşuda var; düğüm 3: **renk 3 (turuncu)** uygun → atanır.

ÇAĞRI YIĞINI

- boya(3)
- boya(2)
- boya(1)
- boya(0)
- main()

Adım 8 / 9

DEĞİŞKENLER

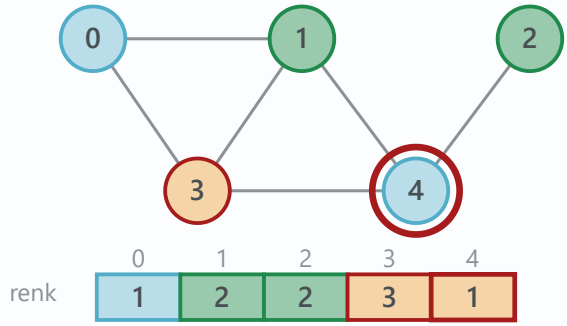
`v = 3` `c = 3` `renk = [1,2,2,3,0]`

boya(0): m = 3 ile Kod İzleme

```
1  boolean boya(int v) {
▶    if (v == n) return true;          // bitti
3    for (int c = 1; c <= m; c++) {
▶      if (uygunMu(v, c)) {
▶        renk[v] = c;                  // seç
▶        if (boya(v + 1)) return true; // keşfet
7        renk[v] = 0;                  // geri al
8      }
9    }
10   return false;                     // çıkmaz
11 }
12 // uygunMu: v'nin c renkli komşusu yoksa true
```

ÇAĞRI YIĞINI

- **boya(4)**
- boya(3)
- boya(2)
- boya(1)
- boya(0)
- main()



düğüm 4: **renk 1 (mavi)** uygun → atanır. v = 5 = n: tüm düğümler boyandı, **geçerli 3-boyama** [1, 2, 2, 3, 1] bulundu.

Adım 9 / 9

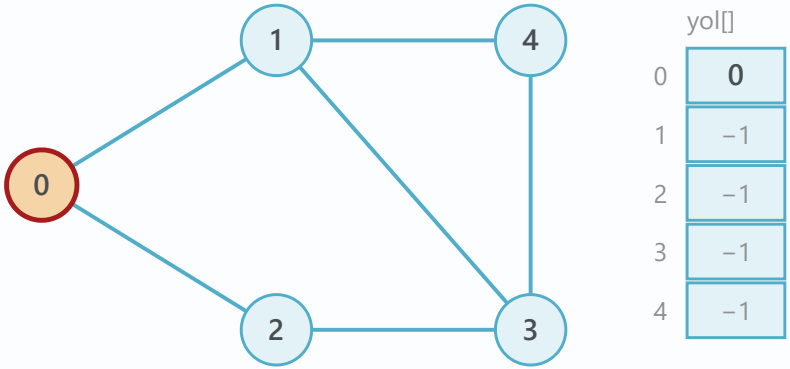
DEĞİŞKENLER

`v = 4` `c = 1` `renk = [1,2,2,3,1]`

DİĞER PROBLEMLER

Hamilton Yolu (Hamiltonian Path)

Çizgenin **her düğümünü tam bir kez** ziyaret eden yol. Kısmi çözüm: `yol[0..konum-1]`; kısıt: yeni düğüm son düğüme komşu ve yolda yok.



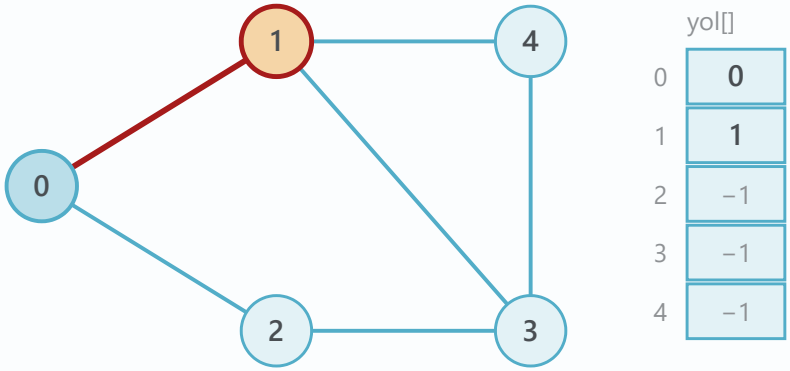
`yol[0] = 0`; `konum = 1`. Her konumda düğümler 0, 1, 2, … sırasıyla denenir: önceki düğüme **komşu** ve yolda **olmayan** düğüm eklenebilir.

Adım 1 / 11

DİĞER PROBLEMLER

Hamilton Yolu (Hamiltonian Path)

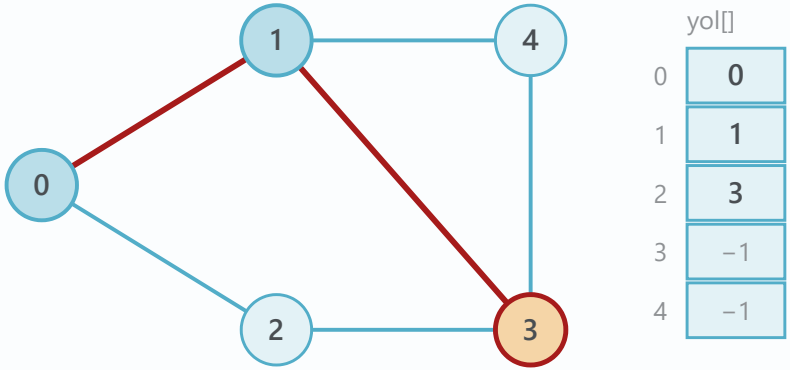
Çizgenin **her düğümünü tam bir kez** ziyaret eden yol. Kısmi çözüm: `yol[0..konum-1]`; kısıt: yeni düğüm son düğüme komşu ve yolda yok.



Konum 1: düğüm **1** uygun → `yol[1] = 1`.

Hamilton Yolu (Hamiltonian Path)

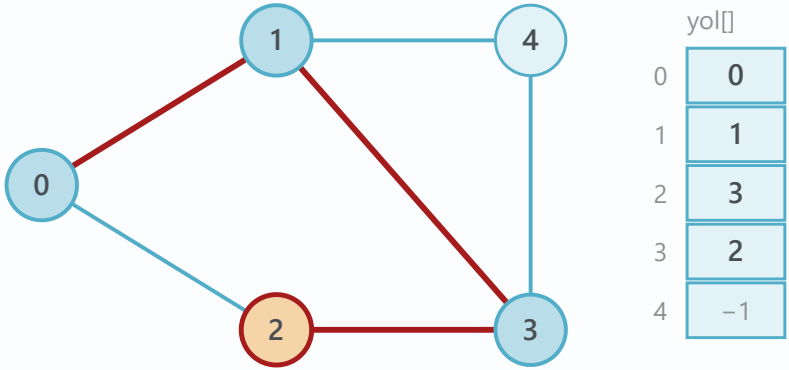
Çizgenin **her düğümünü tam bir kez** ziyaret eden yol. Kısmi çözüm: `yol[0..konum-1]`; kısıt: yeni düğüm son düğüme komşu ve yolda yok.



Konum 2: düğüm **3** uygun → `yol[2] = 3`.

Hamilton Yolu (Hamiltonian Path)

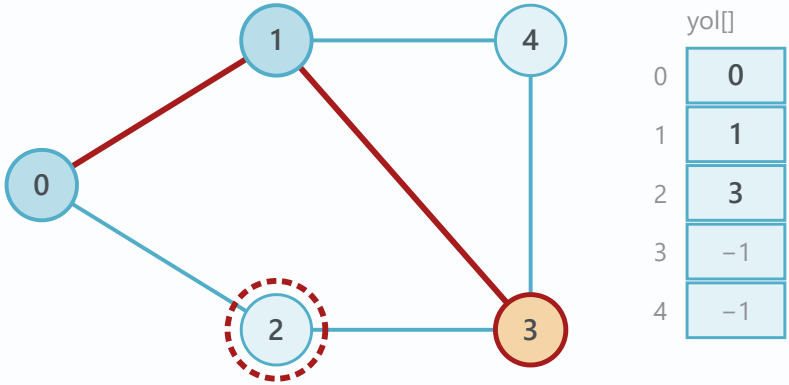
Çizgenin **her düğümünü tam bir kez** ziyaret eden yol. Kısmi çözüm: `yol[0..konum-1]`; kısıt: yeni düğüm son düğüme komşu ve yolda yok.



Konum 3: düğüm **2** uygun → `yol[3] = 2`.

Hamilton Yolu (Hamiltonian Path)

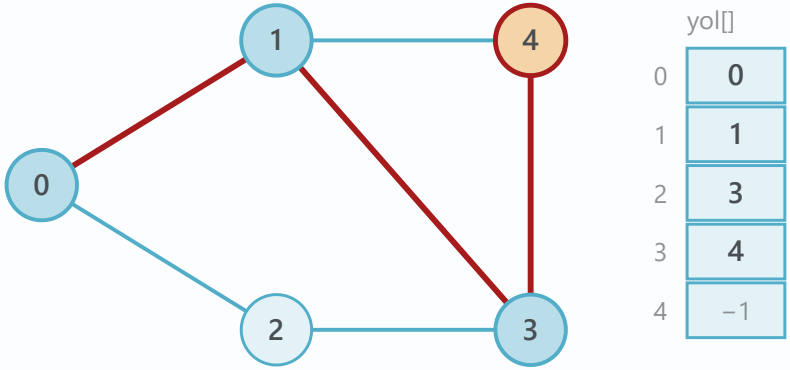
Çizgenin **her düğümünü tam bir kez** ziyaret eden yol. Kısmi çözüm: `yol[0..konum-1]`; kısıt: yeni düğüm son düğüme komşu ve yolda yok.



Konum 4: son düğüm 2'nin yolda olmayan (denenmemiş) komşusu kalmadı → **çıkmaz**.
Geri dön: `yol[3] = -1` (düğüm 2 çıkarılır).

Hamilton Yolu (Hamiltonian Path)

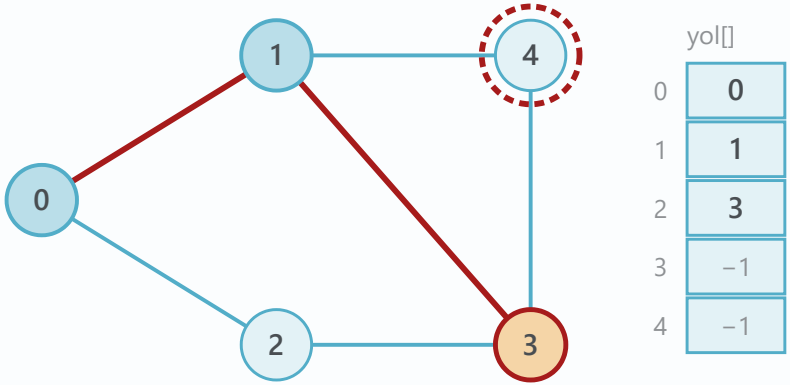
Çizgenin **her düğümünü tam bir kez** ziyaret eden yol. Kısmi çözüm: `yol[0..konum-1]`; kısıt: yeni düğüm son düğüme komşu ve yolda yok.



Konum 3: düğüm **4** uygun → `yol[3] = 4`.

Hamilton Yolu (Hamiltonian Path)

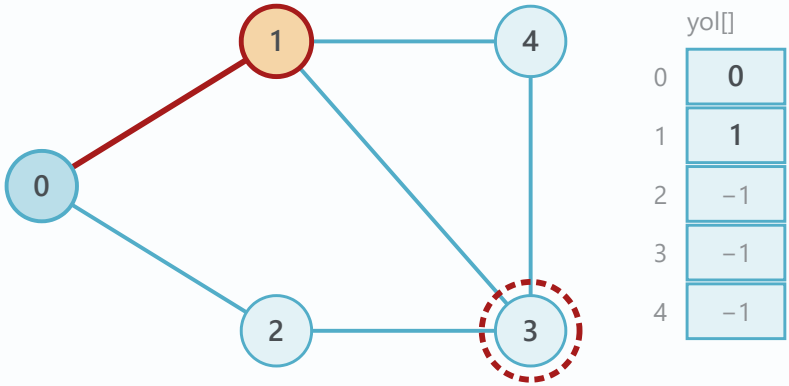
Çizgenin **her düğümünü tam bir kez** ziyaret eden yol. Kısmi çözüm: `yol[0..konum-1]`; kısıt: yeni düğüm son düğüme komşu ve yolda yok.



Konum 4: son düğüm 4'ün yolda olmayan (denenmemiş) komşusu kalmadı → **çıkmaz**. Geri dön: `yol[3] = -1` (düğüm 4 çıkarılır).

Hamilton Yolu (Hamiltonian Path)

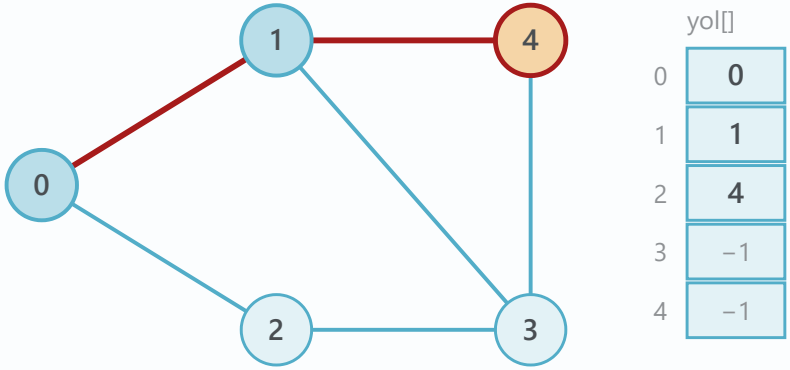
Çizgenin **her düğümünü tam bir kez** ziyaret eden yol. Kısmi çözüm: `yol[0..konum-1]`; kısıt: yeni düğüm son düğüme komşu ve yolda yok.



Konum 3: son düğüm 3'ün yolda olmayan (denenmemiş) komşusu kalmadı → **çıkmaz**. Geri dön: `yol[2] = -1` (düğüm 3 çıkarılır).

Hamilton Yolu (Hamiltonian Path)

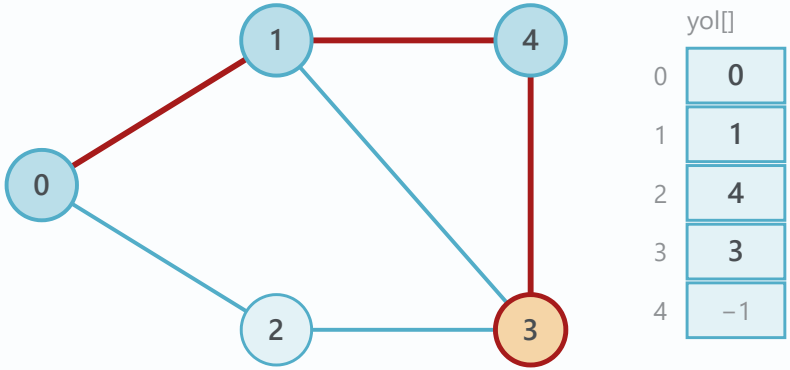
Çizgenin **her düğümünü tam bir kez** ziyaret eden yol. Kısmi çözüm: `yol[0..konum-1]`; kısıt: yeni düğüm son düğüme komşu ve yolda yok.



Konum 2: düğüm **4** uygun → `yol[2] = 4`.

Hamilton Yolu (Hamiltonian Path)

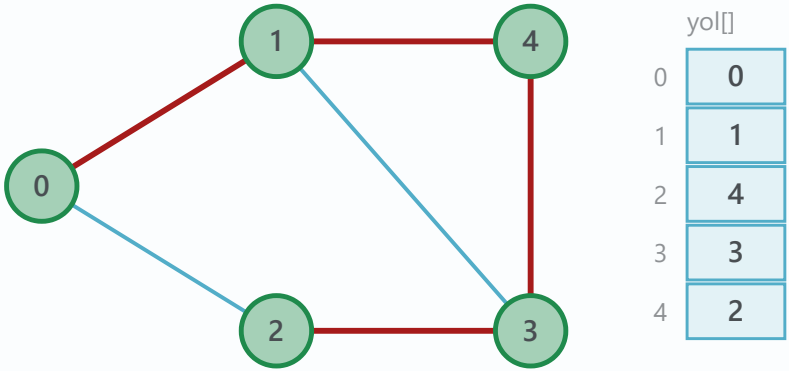
Çizgenin **her düğümünü tam bir kez** ziyaret eden yol. Kısmi çözüm: `yol[0..konum-1]`; kısıt: yeni düğüm son düğüme komşu ve yolda yok.



Konum 3: düğüm **3** uygun → `yol[3] = 3`.

Hamilton Yolu (Hamiltonian Path)

Çizgenin **her düğümünü tam bir kez** ziyaret eden yol. Kısmi çözüm: `yol[0..konum-1]`; kısıt: yeni düğüm son düğüme komşu ve yolda yok.



Konum 4: düğüm **2** uygun → `yol[4] = 2`. `konum == 5`: tüm düğümler yolda, **Hamilton yolu** bulundu: 0 → 1 → 4 → 3 → 2.

Hamilton Yolu: Java Kodu

```
boolean bul(int[][] graf, int[] yol, int konum) {
  if (konum == graf.length) return true;
  for (int v = 0; v < graf.length; v++) {
    if (uygunMu(v, graf, yol, konum)) {
      yol[konum] = v;
      if (bul(graf, yol, konum + 1)) return true;
      yol[konum] = -1;     // geri al
    }
  }
  return false;
}

boolean uygunMu(int v, int[][] graf,
                int[] yol, int konum) {
  if (graf[yol[konum - 1]][v] == 0) return false;
  for (int i = 0; i < konum; i++)
    if (yol[i] == v) return false;
  return true;
}
```

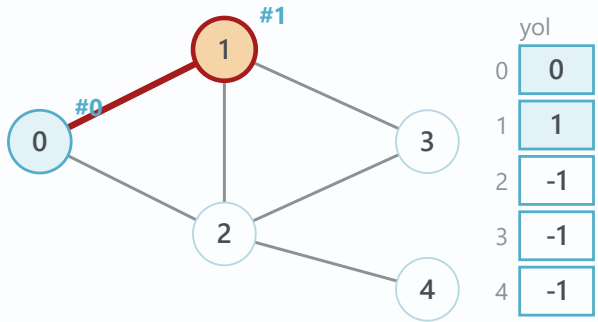
- `graf` : komşuluk matrisi; `yol[0] = 0` ile başlanır, `bul(graf, yol, 1)` çağrılır.
- Örnekteki 0 → 1 → 4 → 3 → 2 yolunda son düğüm 2 ile başlangıç 0 da komşudur; böyle bir yol aynı zamanda bir **Hamilton döngüsüdür**.
- En kötü durum O(n!) yol; `uygunMu` O(n) → O(n · n!).

bul(graf, yol, 1): Kod İzleme

```
1  boolean bul(int[][] graf, int[] yol, int konum) {
2    if (konum == graf.length) return true;
3    for (int v = 0; v < graf.length; v++) {
▶      if (uygunMu(v, graf, yol, konum)) {
▶        yol[konum] = v;
▶        if (bul(graf, yol, konum + 1)) return true;
7        yol[konum] = -1;      // geri al
8      }
9    }
10   return false;
11 }
```

ÇAĞRI YIĞINI

- **bul(konum=1)**
- main()



Yeni çizge (5 düğüm); yol[0] = 0 ile `bul(graf, yol, 1)`. Konum 1: düğüm 0 yolda, 0'a komşu ilk düğüm **1** → yol[1] = 1.

Adım 1 / 10

DEĞİŞKENLER

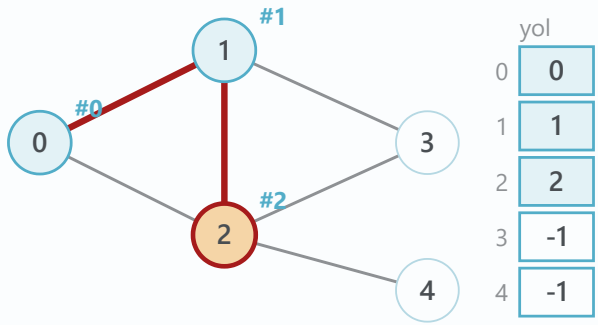
`konum = 1` `v = 1` `yol = [0,1,-1,-1,-1]`

bul(graf, yol, 1): Kod İzleme

```
1  boolean bul(int[][] graf, int[] yol, int konum) {
2    if (konum == graf.length) return true;
3    for (int v = 0; v < graf.length; v++) {
▶      if (uygunMu(v, graf, yol, konum)) {
▶        yol[konum] = v;
▶        if (bul(graf, yol, konum + 1)) return true;
7        yol[konum] = -1;      // geri al
8      }
9    }
10   return false;
11 }
```

ÇAĞRI YIĞINI

- **bul(konum=2)**
- bul(konum=1)
- main()



Konum 2: 1'e komşu ve yolda olmayan ilk düğüm **2** → yol[2] = 2.

Adım 2 / 10

DEĞİŞKENLER

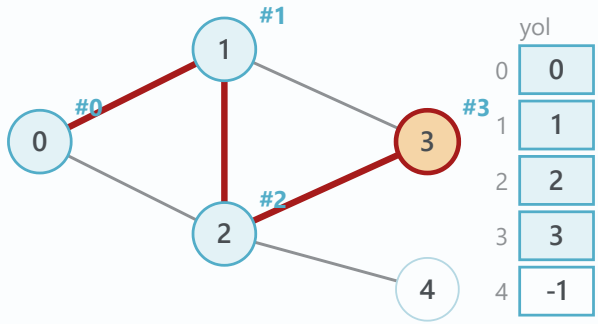
`konum = 2` `v = 2` `yol = [0,1,2,-1,-1]`

bul(graf, yol, 1): Kod İzleme

```
1  boolean bul(int[][] graf, int[] yol, int konum) {
2    if (konum == graf.length) return true;
3    for (int v = 0; v < graf.length; v++) {
▶      if (uygunMu(v, graf, yol, konum)) {
▶        yol[konum] = v;
▶        if (bul(graf, yol, konum + 1)) return true;
7        yol[konum] = -1;      // geri al
8      }
9    }
10   return false;
11 }
```

ÇAĞRI YIĞINI

- **bul(konum=3)**
- bul(konum=2)
- bul(konum=1)
- main()



Konum 3: 2'ye komşu ve yolda olmayan ilk düğüm **3** → yol[3] = 3.

Adım 3 / 10

DEĞİŞKENLER

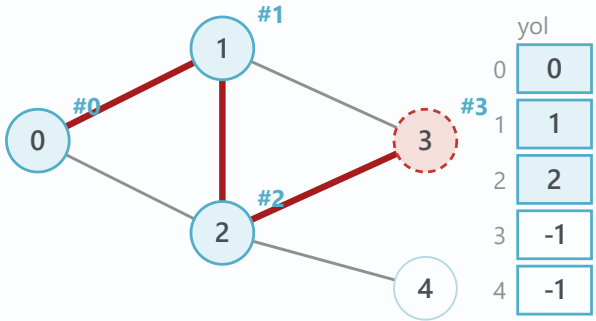
`konum = 3` `v = 3` `yol = [0,1,2,3,-1]`

bul(graf, yol, 1): Kod İzleme

```
1  boolean bul(int[][] graf, int[] yol, int konum) {
2    if (konum == graf.length) return true;
3    for (int v = 0; v < graf.length; v++) {
4      if (uygunMu(v, graf, yol, konum)) {
5        yol[konum] = v;
6        if (bul(graf, yol, konum + 1)) return true;
▶        yol[konum] = -1;      // geri al
8      }
9    }
▶    return false;
11 }
```

ÇAĞRI YIĞINI

- **bul(konum=3)**
- bul(konum=2)
- bul(konum=1)
- main()



Konum 4: son düğüm 3'ün yolda olmayan komşusu yok → **çıkmaz**, false. Geri al: yol[3] = −1 (düğüm 3 çıkarılır).

Adım 4 / 10

DEĞİŞKENLER

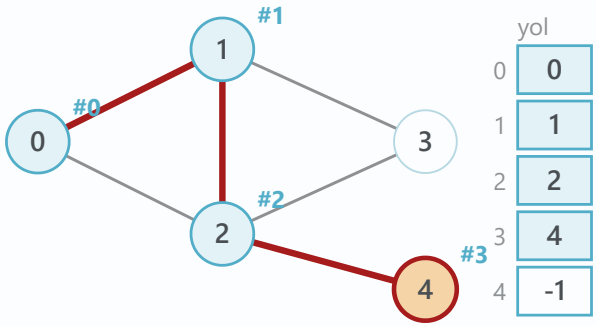
`konum = 3` `v = 3` `yol = [0,1,2,-1,-1]`

bul(graf, yol, 1): Kod İzleme

```
1  boolean bul(int[][] graf, int[] yol, int konum) {
2    if (konum == graf.length) return true;
3    for (int v = 0; v < graf.length; v++) {
▶      if (uygunMu(v, graf, yol, konum)) {
▶        yol[konum] = v;
▶        if (bul(graf, yol, konum + 1)) return true;
7        yol[konum] = -1;      // geri al
8      }
9    }
10   return false;
11 }
```

ÇAĞRI YIĞINI

- **bul(konum=3)**
- bul(konum=2)
- bul(konum=1)
- main()



Konum 3: 2'ye komşu ve yolda olmayan ilk düğüm **4** → yol[3] = 4.

Adım 5 / 10

DEĞİŞKENLER

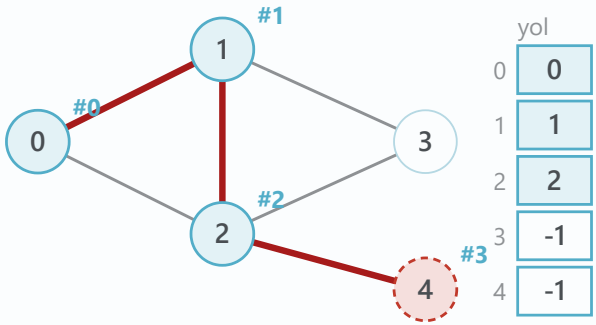
`konum = 3` `v = 4` `yol = [0,1,2,4,-1]`

bul(graf, yol, 1): Kod İzleme

```
1  boolean bul(int[][] graf, int[] yol, int konum) {
2    if (konum == graf.length) return true;
3    for (int v = 0; v < graf.length; v++) {
4      if (uygunMu(v, graf, yol, konum)) {
5        yol[konum] = v;
6        if (bul(graf, yol, konum + 1)) return true;
▶        yol[konum] = -1;      // geri al
8      }
9    }
▶    return false;
11 }
```

ÇAĞRI YIĞINI

- **bul(konum=3)**
- bul(konum=2)
- bul(konum=1)
- main()



Konum 4: son düğüm 4'ün yolda olmayan komşusu yok → **çıkmaz**, false. Geri al: yol[3] = −1 (düğüm 4 çıkarılır).

Adım 6 / 10

DEĞİŞKENLER

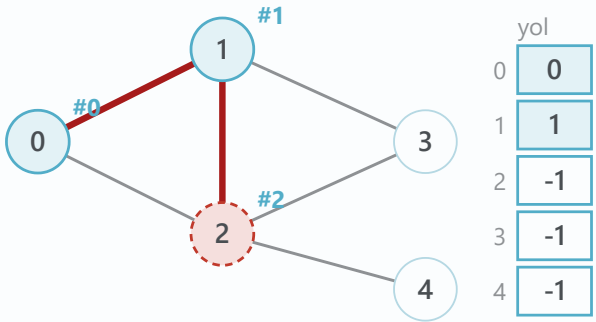
`konum = 3` `v = 4` `yol = [0,1,2,-1,-1]`

bul(graf, yol, 1): Kod İzleme

```
1  boolean bul(int[][] graf, int[] yol, int konum) {
2    if (konum == graf.length) return true;
3    for (int v = 0; v < graf.length; v++) {
4      if (uygunMu(v, graf, yol, konum)) {
5        yol[konum] = v;
6        if (bul(graf, yol, konum + 1)) return true;
▶        yol[konum] = -1;      // geri al
8      }
9    }
▶    return false;
11 }
```

ÇAĞRI YIĞINI

- **bul(konum=2)**
- bul(konum=1)
- main()



Konum 3: son düğüm 2'nin yolda olmayan komşusu yok → **çıkmaz**, false. Geri al: yol[2] = −1 (düğüm 2 çıkarılır).

Adım 7 / 10

DEĞİŞKENLER

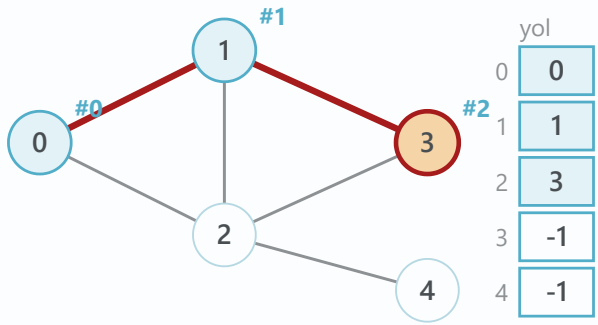
`konum = 2` `v = 2` `yol = [0,1,-1,-1,-1]`

bul(graf, yol, 1): Kod İzleme

```
1  boolean bul(int[][] graf, int[] yol, int konum) {
2    if (konum == graf.length) return true;
3    for (int v = 0; v < graf.length; v++) {
▶      if (uygunMu(v, graf, yol, konum)) {
▶        yol[konum] = v;
▶        if (bul(graf, yol, konum + 1)) return true;
7        yol[konum] = -1;      // geri al
8      }
9    }
10   return false;
11 }
```

ÇAĞRI YIĞINI

- **bul(konum=2)**
- bul(konum=1)
- main()



Konum 2: 1'e komşu ve yolda olmayan ilk düğüm **3** → yol[2] = 3.

Adım 8 / 10

DEĞİŞKENLER

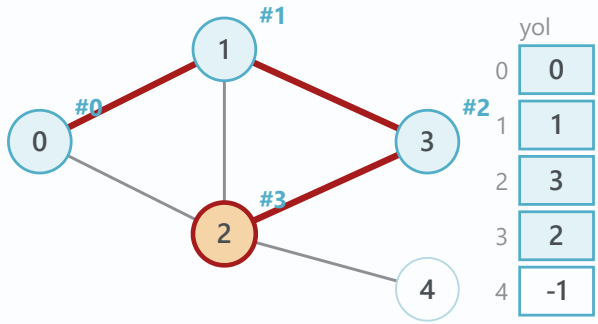
`konum = 2` `v = 3` `yol = [0,1,3,-1,-1]`

bul(graf, yol, 1): Kod İzleme

```
1  boolean bul(int[][] graf, int[] yol, int konum) {
2    if (konum == graf.length) return true;
3    for (int v = 0; v < graf.length; v++) {
▶      if (uygunMu(v, graf, yol, konum)) {
▶        yol[konum] = v;
▶        if (bul(graf, yol, konum + 1)) return true;
7        yol[konum] = -1;      // geri al
8      }
9    }
10   return false;
11 }
```

ÇAĞRI YIĞINI

- **bul(konum=3)**
- bul(konum=2)
- bul(konum=1)
- main()



Konum 3: 3'e komşu ve yolda olmayan ilk düğüm **2** → yol[3] = 2.

Adım 9 / 10

DEĞİŞKENLER

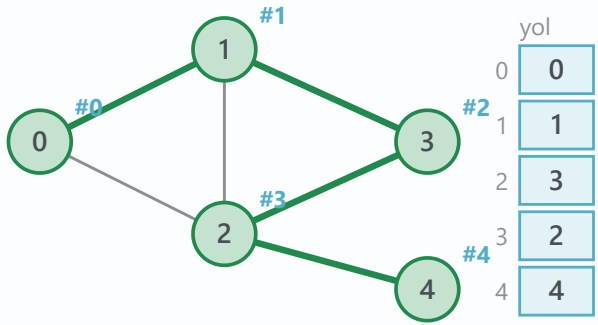
`konum = 3` `v = 2` `yol = [0,1,3,2,-1]`

bul(graf, yol, 1): Kod İzleme

```
1  boolean bul(int[][] graf, int[] yol, int konum) {
2    if (konum == graf.length) return true;
3    for (int v = 0; v < graf.length; v++) {
4      if (uygunMu(v, graf, yol, konum)) {
5        yol[konum] = v;
6        if (bul(graf, yol, konum + 1)) return true;
7        yol[konum] = -1;      // geri al
8      }
9    }
10   return false;
11 }
```

ÇAĞRI YIĞINI

- **bul(konum=5)**
- bul(konum=4)
- bul(konum=3)
- bul(konum=2)
- bul(konum=1)
- main()



Konum 4: 2'ye komşu ve yolda olmayan ilk düğüm **4** → yol[4] = 4. konum = 5: tüm düğümler yolda → **Hamilton yolu 0 → 1 → 3 → 2 → 4**, true yukarı taşınır.

Adım 10 / 10

DEĞİŞKENLER

`konum = 4` `v = 4` `yol = [0,1,3,2,4]`

DİĞER PROBLEMLER

Kelime Arama (Word Search)

Harf ızgarasında, komşu (yukarı/aşağı/sol/sağ) hücrelerden geçerek kelimeyi oluşturan bir yol var mı? Her hücre bir yolda en fazla bir kez kullanılır.

A	B	C	E
S	F	C	S
A	D	E	E

Aranan kelime **"SEE"**. Her hücre başlangıç olarak denenir (satır satır); komşu yönler: yukarı, aşağı, sol, sağ. Bir hücre aynı yolda iki kez kullanılamaz.

Kelime Arama (Word Search)

Harf ızgarasında, komşu (yukarı/aşağı/sol/sağ) hücrelerden geçerek kelimeyi oluşturan bir yol var mı? Her hücre bir yolda en fazla bir kez kullanılır.

A	B	C	E
0 S	F	C	S
A	D	E	E

(1, 0) = 'S' kelimenin 0. harfiyle eşleşti → ziyaret edildi olarak işaretlenir.

Kelime Arama (Word Search)

Harf ızgarasında, komşu (yukarı/aşağı/sol/sağ) hücrelerden geçerek kelimeyi oluşturan bir yol var mı? Her hücre bir yolda en fazla bir kez kullanılır.

A	B	C	E
S	F	C	S
A	D	E	E

(1, 0) hücresinden hiçbir komşu 'E' ile devam etmiyor → **geri dön**, işaret kaldırılır.

Kelime Arama (Word Search)

Harf ızgarasında, komşu (yukarı/aşağı/sol/sağ) hücrelerden geçerek kelimeyi oluşturan bir yol var mı? Her hücre bir yolda en fazla bir kez kullanılır.

A	B	C	E
S	F	C	0 S
A	D	E	E

(1, 3) = 'S' kelimenin 0. harfiyle eşleşti → ziyaret edildi olarak işaretlenir.

Kelime Arama (Word Search)

Harf ızgarasında, komşu (yukarı/aşağı/sol/sağ) hücrelerden geçerek kelimeyi oluşturan bir yol var mı? Her hücre bir yolda en fazla bir kez kullanılır.

A	B	C	1 E
S	F	C	0 S
A	D	E	E

(0, 3) = 'E' kelimenin 1. harfiyle eşleşti → ziyaret edildi olarak işaretlenir.

Kelime Arama (Word Search)

Harf ızgarasında, komşu (yukarı/aşağı/sol/sağ) hücrelerden geçerek kelimeyi oluşturan bir yol var mı? Her hücre bir yolda en fazla bir kez kullanılır.

A	B	C	E
S	F	C	0 S
A	D	E	E

(0, 3) hücresinden hiçbir komşu 'E' ile devam etmiyor → **geri dön**, işaret kaldırılır.

Kelime Arama (Word Search)

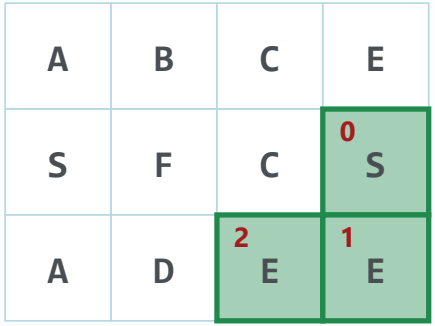
Harf ızgarasında, komşu (yukarı/aşağı/sol/sağ) hücrelerden geçerek kelimeyi oluşturan bir yol var mı? Her hücre bir yolda en fazla bir kez kullanılır.

A	B	C	E
S	F	C	0 S
A	D	E	1 E

(2, 3) = 'E' kelimenin 1. harfiyle eşleşti → ziyaret edildi olarak işaretlenir.

Kelime Arama (Word Search)

Harf ızgarasında, komşu (yukarı/aşağı/sol/sağ) hücrelerden geçerek kelimeyi oluşturan bir yol var mı? Her hücre bir yolda en fazla bir kez kullanılır.



(2, 2) = 'E' kelimenin 2. harfiyle eşleşti → ziyaret edildi olarak işaretlenir. Son harf de eşleşti: **"SEE" bulundu**.

Kelime Arama: Java Kodu

```
boolean ara(char[][] iz, String kelime, int i,
            int s, int k, boolean[][] ziyaret) {
  if (i == kelime.length()) return true;
  if (s < 0 || s >= iz.length || k < 0
      || k >= iz[0].length || ziyaret[s][k]
      || iz[s][k] != kelime.charAt(i))
    return false;
  ziyaret[s][k] = true;
  for (int d = 0; d < 4; d++)
    if (ara(iz, kelime, i + 1, s + ds[d],
            k + dk[d], ziyaret))
      return true;
  ziyaret[s][k] = false;   // geri al
  return false;
}
```

Kelime	Sonuç
ABCCED	var
SEE	var
ABCB	yok: B iki kez kullanılamaz
SADE	var

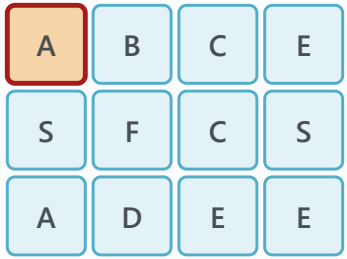
Izgara: ABCE / SFCS / ADEE. Geri dönüşte `ziyaret` işareti **kaldırılmalıdır**; yoksa hücre başka bir yolda kullanılamaz.

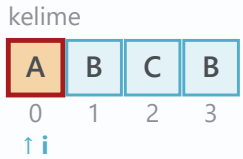
ara(..., "ABCB", 0, 0, 0, ...): Kod İzleme

```
1  boolean ara(char[][] iz, String kelime, int i,
2              int s, int k, boolean[][] ziyaret) {
3    if (i == kelime.length()) return true;
4    if (s < 0 || s >= iz.length || k < 0
5        || k >= iz[0].length || ziyaret[s][k]
6        || iz[s][k] != kelime.charAt(i))
7      return false;
▶    ziyaret[s][k] = true;
▶    for (int d = 0; d < 4; d++)
▶      if (ara(iz, kelime, i + 1, s + ds[d],
11             k + dk[d], ziyaret))
12       return true;
13   ziyaret[s][k] = false;   // geri al
14   return false;
15 }
```

ÇAĞRI YIĞINI

- **ara(i=0, 0, 0)**
- main()





Izgara ABCE / SFCS / ADEE, kelime **"ABCB"**; (0, 0)'dan başlanır. (0, 0) = 'A' = kelime[0] → ziyaret işaretlenir; yönler: yukarı, aşağı, sol, sağ.

Adım 1 / 8

DEĞİŞKENLER

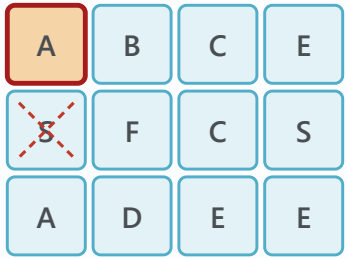
`i = 0` `s = 0` `k = 0`

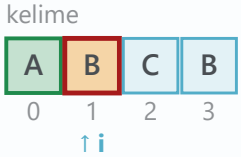
ara(..., "ABCB", 0, 0, 0, ...): Kod İzleme

```
1  boolean ara(char[][] iz, String kelime, int i,
2              int s, int k, boolean[][] ziyaret) {
3    if (i == kelime.length()) return true;
▶    if (s < 0 || s >= iz.length || k < 0
▶        || k >= iz[0].length || ziyaret[s][k]
▶        || iz[s][k] != kelime.charAt(i))
▶      return false;
8    ziyaret[s][k] = true;
9    for (int d = 0; d < 4; d++)
10     if (ara(iz, kelime, i + 1, s + ds[d],
11             k + dk[d], ziyaret))
12       return true;
13   ziyaret[s][k] = false;   // geri al
14   return false;
15 }
```

ÇAĞRI YIĞINI

- `ara(i=0, 0, 0)`
- `main()`





kelime[1] = 'B' aranıyor: yukarı: ızgara dışı; aşağı: 'S' ≠ 'B'; sol: ızgara dışı → false.

Adım 2 / 8

DEĞİŞKENLER

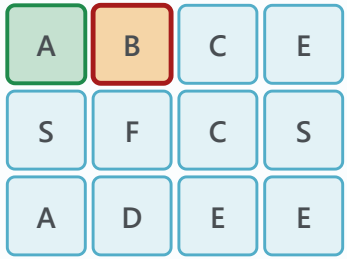
`i = 1` `s = 0` `k = 0`

ara(..., "ABCB", 0, 0, 0, ...): Kod İzleme

```
1  boolean ara(char[][] iz, String kelime, int i,
2              int s, int k, boolean[][] ziyaret) {
3    if (i == kelime.length()) return true;
4    if (s < 0 || s >= iz.length || k < 0
5        || k >= iz[0].length || ziyaret[s][k]
6        || iz[s][k] != kelime.charAt(i))
7      return false;
▶    ziyaret[s][k] = true;
▶    for (int d = 0; d < 4; d++)
▶      if (ara(iz, kelime, i + 1, s + ds[d],
11             k + dk[d], ziyaret))
12       return true;
13   ziyaret[s][k] = false;   // geri al
14   return false;
15 }
```

ÇAĞRI YIĞINI

- `ara(i=1, 0, 1)`
- `ara(i=0, 0, 0)`
- `main()`





(0, 1) = 'B' = kelime[1] → ziyaret edildi olarak işaretlenir.

Adım 3 / 8

DEĞİŞKENLER

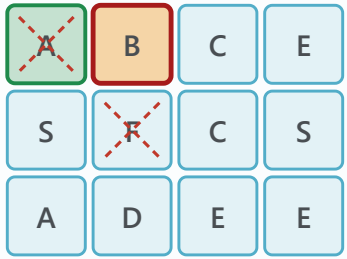
`i = 1` `s = 0` `k = 1`

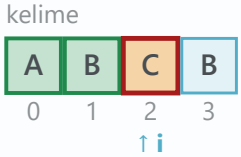
ara(..., "ABCB", 0, 0, 0, ...): Kod İzleme

```
1  boolean ara(char[][] iz, String kelime, int i,
2              int s, int k, boolean[][] ziyaret) {
3    if (i == kelime.length()) return true;
▶    if (s < 0 || s >= iz.length || k < 0
▶        || k >= iz[0].length || ziyaret[s][k]
▶        || iz[s][k] != kelime.charAt(i))
▶      return false;
8    ziyaret[s][k] = true;
9    for (int d = 0; d < 4; d++)
10     if (ara(iz, kelime, i + 1, s + ds[d],
11             k + dk[d], ziyaret))
12       return true;
13   ziyaret[s][k] = false;   // geri al
14   return false;
15 }
```

ÇAĞRI YIĞINI

- `ara(i=1, 0, 1)`
- `ara(i=0, 0, 0)`
- `main()`





kelime[2] = 'C' aranıyor: yukarı: ızgara dışı; aşağı: 'F' ≠ 'C'; sol: (0, 0) **ziyaretli** → false.

Adım 4 / 8

DEĞİŞKENLER

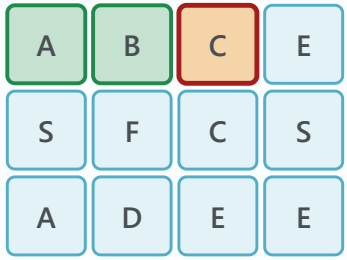
`i = 2` `s = 0` `k = 1`

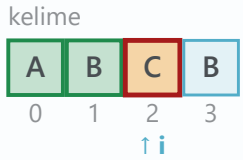
ara(..., "ABCB", 0, 0, 0, ...): Kod İzleme

```
1  boolean ara(char[][] iz, String kelime, int i,
2              int s, int k, boolean[][] ziyaret) {
3    if (i == kelime.length()) return true;
4    if (s < 0 || s >= iz.length || k < 0
5        || k >= iz[0].length || ziyaret[s][k]
6        || iz[s][k] != kelime.charAt(i))
7      return false;
▶    ziyaret[s][k] = true;
▶    for (int d = 0; d < 4; d++)
▶      if (ara(iz, kelime, i + 1, s + ds[d],
11             k + dk[d], ziyaret))
12       return true;
13   ziyaret[s][k] = false;   // geri al
14   return false;
15 }
```

ÇAĞRI YIĞINI

- **ara(i=2, 0, 2)**
- ara(i=1, 0, 1)
- ara(i=0, 0, 0)
- main()





(0, 2) = 'C' = kelime[2] → ziyaret edildi olarak işaretlenir.

Adım 5 / 8

DEĞİŞKENLER

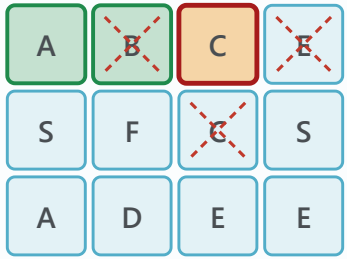
`i = 2` `s = 0` `k = 2`

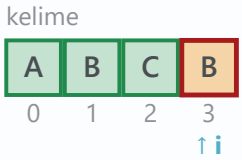
ara(..., "ABCB", 0, 0, 0, ...): Kod İzleme

```
1  boolean ara(char[][] iz, String kelime, int i,
2              int s, int k, boolean[][] ziyaret) {
3    if (i == kelime.length()) return true;
▶    if (s < 0 || s >= iz.length || k < 0
▶        || k >= iz[0].length || ziyaret[s][k]
▶        || iz[s][k] != kelime.charAt(i))
▶      return false;
8    ziyaret[s][k] = true;
9    for (int d = 0; d < 4; d++)
10     if (ara(iz, kelime, i + 1, s + ds[d],
11             k + dk[d], ziyaret))
12       return true;
13   ziyaret[s][k] = false;   // geri al
14   return false;
15 }
```

ÇAĞRI YIĞINI

- `ara(i=2, 0, 2)`
- `ara(i=1, 0, 1)`
- `ara(i=0, 0, 0)`
- `main()`





kelime[3] = 'B' aranıyor: yukarı: ızgara dışı; aşağı: 'C' ≠ 'B'; sol: (0, 1) **ziyaretli**; sağ: 'E' ≠ 'B' → false. B yalnız (0, 1)'de var ve o hücre bu yolda kullanıldı.

Adım 6 / 8

DEĞİŞKENLER

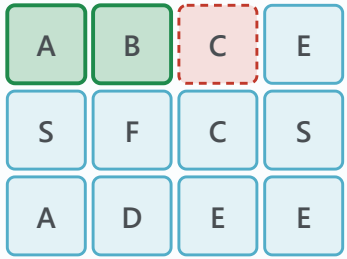
`i = 3` `s = 0` `k = 2`

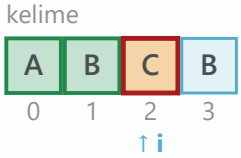
ara(..., "ABCB", 0, 0, 0, ...): Kod İzleme

```
 1 boolean ara(char[][] iz, String kelime, int i,
 2             int s, int k, boolean[][] ziyaret) {
 3   if (i == kelime.length()) return true;
 4   if (s < 0 || s >= iz.length || k < 0
 5       || k >= iz[0].length || ziyaret[s][k]
 6       || iz[s][k] != kelime.charAt(i))
 7     return false;
 8   ziyaret[s][k] = true;
 9   for (int d = 0; d < 4; d++)
10     if (ara(iz, kelime, i + 1, s + ds[d],
11             k + dk[d], ziyaret))
12       return true;
▶    ziyaret[s][k] = false;   // geri al
▶    return false;
15 }
```

ÇAĞRI YIĞINI

- `ara(i=2, 0, 2)`
- `ara(i=1, 0, 1)`
- `ara(i=0, 0, 0)`
- `main()`





(0, 2) için dört yön de başarısız → **geri al**: ziyaret[0][2] = false, false döner.

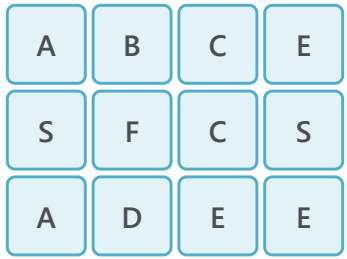
Adım 7 / 8

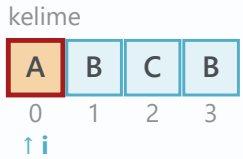
DEĞİŞKENLER

`i = 2` `s = 0` `k = 2`

ara(..., "ABCB", 0, 0, 0, ...): Kod İzleme

```
1  boolean ara(char[][] iz, String kelime, int i,
2              int s, int k, boolean[][] ziyaret) {
3    if (i == kelime.length()) return true;
4    if (s < 0 || s >= iz.length || k < 0
5        || k >= iz[0].length || ziyaret[s][k]
6        || iz[s][k] != kelime.charAt(i))
7      return false;
8    ziyaret[s][k] = true;
9    for (int d = 0; d < 4; d++)
10     if (ara(iz, kelime, i + 1, s + ds[d],
11             k + dk[d], ziyaret))
12       return true;
▶    ziyaret[s][k] = false;   // geri al
▶    return false;
15 }
```





Kalan yönler de başarısız: (0, 1) ve (0, 0) sırayla geri alınır, işaretler kaldırılır. `ara(…, 0, 0, 0, …)` = **false**. Dış döngü (2, 0)'daki A'yı da dener; orada B komşusu yok: "ABCB" **yok**.

Adım 8 / 8

ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

`i = 0` `s = 0` `k = 0`

DİĞER PROBLEMLER

Geri Dönüşle Çözülen Diğer Problemler

Problem	Her düzeydeki seçim	Budama (kısıt)
Alt küme toplamı / sırt çantası	i. öğe alınsın mı? (2 dal)	toplam / ağırlık sınırı aşıldı
Gezgin satıcı (TSP)	sıradaki şehir	yol, bilinen en iyi turdan uzun
Kutu yerleştirme (bin packing)	eşya hangi kutuya?	kutu kapasitesi aşıldı
Maksimum kapsama	küme seçilsin mi?	kalan kümelerle iyileşme mümkün değil
Mantıksal doyurulabilirlik (SAT)	değişken = doğru / yanlış	bir cümle (clause) yanlış oldu
Mayın tarlası	hücre mayın mı?	açılmış sayılarla çelişki
Permütasyon / kombinasyon üretimi	sıradaki öğe	öğe zaten kullanıldı

Eniyileme problemlerinde (TSP, sırt çantası) "bilinen en iyiden kötü" dalı kesmek **dal sınır** (branch and bound) yönteminin temelidir.

ANALİZ

Karmaşıklık

Problem	En kötü zaman	Açıklama
n-vezir	$O(n!)$	satır başına bir vezir, sütunlar farklı; dizili test ile $O(n \cdot n!)$
Sudoku	$O(9^m)$	m: boş hücre sayısı; her hücrede 9 aday
Çizge m-boyama	$O(n \cdot m^n)$	n düğüm, her biri m renk; denetim O(derece)
Hamilton yolu	$O(n \cdot n!)$	yoldaki her konum için kalan düğümler
Kelime arama	$O(R \cdot C \cdot 3^L)$	ilk harften sonra en fazla 3 yeni yön (geri gelinen hücre ziyaretli)
Alt kümeler / permütasyonlar	$O(2^n)$ / $O(n \cdot n!)$	ağaçtaki yaprak sayısı

Zaman = düğüm sayısı × düğüm başı iş

Budama düğüm sayısını küçültür ama en kötü durumu (çoğu zaman) değiştirmez.

Bellek: O(derinlik)

Yalnızca tek bir kısmi çözüm ve yığındaki çerçeveler tutulur (ör. n-vezirde O(n)).

ANALİZ

Budamayı Güçlendirme: Sezgiseller

Değişken sıralaması (MRV)

En az seçeneği kalan değişken önce doldurulur. Sudoku'da geri dönüşü tamamen ortadan kaldırabilir.

Değer sıralaması

Önce en az kısıtlayan değer denenir (en az çakışma fikri). Çözüm daha erken bulunur.

İleri denetim (forward checking)

Atamadan sonra komşu değişkenlerin alanlarından çelişen değerler silinir; alanı boşalan varsa hemen geri dönülür.

Hızlı kısıt testi

n-vezirde sütun/köşegen bayrakları, Sudoku'da satır/sütun/kutu bit maskeleri: test O(1).

Simetri kırma

n-vezirde ilk vezir yalnızca sol yarıda denenir, sonuç yansıtılır: iş yaklaşık yarıya iner.

Sınır (bound)

Eniyilemede kısmi çözümün en iyi olası değeri, bilinen en iyi çözümden kötüyse dal kesilir: dal sınır.

ANALİZ

Geri İzlemeyi Ayıran Ne?

Paradigma	Fikir	Geri izlemeden farkı
Böl ve fethet	Bağımsız alt problemlere böl, birleştir	Seçim ve deneme yok; her alt problem tam bir kez çözülür
Açgözlü	Her adımda yerel en iyi; **geri dönmez**	Tek dal izlenir; yanlış seçim düzeltilemez
Dinamik programlama	Örtüşen alt problemleri bir kez çöz, sakla	Tüm seçenekler denenir ama sonuçlar tabloda; alt problem sayısı polinom olmalı
Geri izleme	Kısmi çözümü genişlet, geçersizse geri al	Seçimler denenir ve geri alınır; kısıt sağlama, tüm çözümler, karar problemleri

Geri izleme, durum uzayı ağacında **derinlik öncelikli arama**dır (DFS); ağaç önceden kurulmaz, gezilirken üretilir ve geçersiz dallar hiç üretilmez.

Özet

Konu	Hatırlanacaklar
Zor problemler	NP-tam / NP-zor: P ≠ NP ise polinom zamanlı kesin algoritma yok
Fikir	kısmi çözümü adım adım kur; kısıt bozulursa son kararı geri al, sıradakini dene
Şablon	hedef testi → her aday için: geçerliyse seç, özyinele, geri al
Durum uzayı	DFS ile gezilen ağaç; budama geçersiz alt ağaçları atlar
n-vezir	satır satır; sütun + 2 çapraz denetimi; 8 için 92 çözüm, ilk: [0, 4, 7, 5, 2, 6, 1, 3]
En az çakışma	tam yerleşim + yerel onarım; hızlı ama tam değil
Sudoku	boş hücreye 1–9; satır/sütun/kutu denetimi; MRV ile çok daha hızlı
Diğerleri	çizge boyama, Hamilton yolu, kelime arama (ziyaret işaretini geri al!)
Maliyet	en kötü üstel (2^n, $n!$, 9^m); bellek O(derinlik)

ÖZET

Sık Yapılan Hatalar

Geri almayı unutmak

Özyinelemeden dönünce `t[s][k] = 0`, `ziyaret = false` yapılmazsa eski seçim sonraki dalları bozar.

Sonucu yaymamak

`if (coz(...)) return true;` yerine yalnızca `coz(...)` yazmak: çözüm bulunsa da arama sürer ve tahta geri alınır.

Erken `return false`

İlk geçersiz adayda döngüden çıkmak; sıradaki adaylar hiç denenmez.

Paylaşılan listeyi kaydetmek

Tüm çözümleri toplarken `sonuclar.add(yol)` yerine kopya (`new ArrayList<>(yol)`) eklenmeli.

Eksik kısıt denetimi

n-vezirde yalnız bir çaprazı, Sudoku'da 3×3 kutuyu unutmak: geçersiz "çözümler".

Yerel aramayı tam sanmak

En az çakışma çözüm olmadığını **kanıtlayamaz**; adım sınırına ulaşmak "çözüm yok" demek değildir.

ÖZET

Örnek Problemler

1. 4 vezir: `coz(t, 0)` ilk çözümü bulana kadar kaç vezir yerleştirir, kaç kez geri döner?
2. 5 vezir için algoritmanın ilk çözümü nedir? Kaç kez geri dönülür?
3. {3, 5, 6, 7} içinde toplamı 15 olan alt küme (önce "al" dalı; toplam aşılınca budama). İlk bulunan?
4. 5 düğümlü döngü çizgesi (C_5) 2 renkle boyanabilir mi? En az kaç renk gerekir?
5. ABCE / SFCS / ADEE ızgarasında "ABCB" neden bulunamaz, "SADE" hangi yolla bulunur?
6. 8 vezir: 8^8 tam yerleşimi test etmek ile geri izlemeyi (tüm çözümler) karşılaştırın.

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. n-vezir kodunu **tüm** çözümleri sayacak biçimde değiştirin (n = 8).
2. {1, 2, 3}'ün permütasyonlarını geri dönüşle üretin. Ağaçta kök dahil kaç düğüm vardır?
3. Hamilton yolu kodunu **Hamilton döngüsü** arayacak biçimde değiştirin; Hamilton yolu örneğindeki 5 düğümlü çizgede sonuç?
4. Sudoku çözücüsüne MRV (en az adaylı hücre önce) ekleyin.
5. Labirent örneğinde girişten çıkışa kaç farklı basit yol vardır?
6. Alt küme toplamına "kalan öğeler yetmiyorsa kes" budamasını ekleyin.

▶ **Cevaplar / ipuçları**

Kaynaklar

- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
- Levitin, *Introduction to the Design and Analysis of Algorithms*, 3. baskı, Pearson, 2012 (Bölüm 12: Backtracking).
- Russell ve Norvig, *Artificial Intelligence: A Modern Approach*, 4. baskı, 2020 (Bölüm 6: kısıt sağlama, en az çakışma).
- S. Minton, M. Johnston, A. Philips ve P. Laird, "Minimizing Conflicts: A Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems", *Artificial Intelligence* 58, 1992.
- D. E. Knuth, *The Art of Computer Programming*, Cilt 4B, 2022 (Backtrack programming, Dancing Links).

Son

Bölüm 9: Geri İzleme Algoritmaları

Sonraki bölüm: Bölüm 10: Dal Sınır Algoritmaları · Terim sözlüğü