Bölüm 2: Diziler ve Yapılar

C ile Veri Yapıları ve Algoritmalar

Dizi ADT | Dizi ve işaretçi | Dinamik dizi | Çok boyutlu dizi · adres hesabı | Üçgen ve kuşak matris | struct · union | Dizgiler | Matris çarpımı

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Diziyi soyut ve somut görmek

Dizi ADT'sini tanımlayıp C dizisinin bellekteki ardışık yerleşimini açıklamak.

Adres hesaplamak

Satır/sütun öncelikli yerleşimde 2, 3 ve n boyutlu dizilerin öğe adresini bulmak.

Bellek tasarrufu

Üçgen ve kuşak matrisleri tek boyutlu dizide saklayıp O(1) erişim kurmak.

Dinamik bellek

malloc/realloc/free ile büyüyen dizi ve 2B dizi kurmak; amortize maliyeti açıklamak.

Kayıt türleri

struct, union ve dizgileri (char dizisi) doğru ve güvenli kullanmak.

Matris işlemleri

Matris toplama ve çarpımının adım sayısını çıkarıp $O(n^3)$ sonucunu göstermek.

Dizi: ⟨indis, değer⟩ çiftlerinden oluşan küme

Soyut bakışta dizi bir **eşleme**dir (mapping): tanımlı her indise bir değer karşılık gelir. "Ardışık bellek hücreleri" ise bu ADT'nin **en yaygın gerçeklemesi**dir, tanımı değil.

İşlem	Anlamı
`Create(j, liste)`	j boyutlu, indis kümesi `liste` olan boş dizi
`Retrieve(A, i)`	i geçerli indisse A[i] değerini, değilse hata döndür
`Store(A, i, x)`	i geçerliyse A'da i ile x'i eşleştirip yeni diziyi döndür

Neden hızlı?

Başlangıç adresi α ve öğe boyutu s biliniyorsa A[i] adresi **α + i·s**: tek çarpma + toplama, **O(1)**.

Bedeli

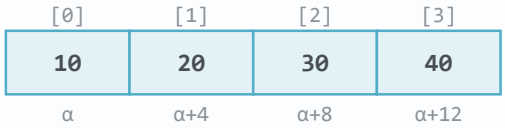
Boyut baştan sabittir; araya ekleme/silme sonraki öğeleri kaydırır: **O(n)**.

C'de dizi bildirimi ve adres

```c
int list[5], *plist[5];
/* list : 5 tamsayı       */
/* plist: 5 tamsayı işaretçisi */

list[i]  /* adresi: α + i * sizeof(int) */
sizeof list   /* 5 * sizeof(int) bayt */
```

Derleyici `list[0]`'ın adresini (α) tutar; `list[i]` için adres **çalışma anında** hesaplanır. İndis **denetlenmez**: `list[5]` yazmak tanımsız davranıştır.



sizeof(int) = 4 varsayımıyla adresler

İndisler 0'dan başlar: n öğeli dizinin geçerli indisleri **0 … n−1**.

Dizi adı ≈ ilk öğenin adresi

İfade	Eşdeğeri
`list`	`&list[0]`
`list + i`	`&list[i]` (i·sizeof(int) bayt ileri)
`*(list + i)`	`list[i]`
`p = list; p[i]`	`list[i]` ile aynı hücre

İşaretçi aritmetiği **öğe** sayar, bayt değil: `p + 1` bir sonraki `int`'i gösterir.

Fark 1: atanamaz

`list = p;` derlenmez; dizi adı bir değişken değil, sabit adrestir.

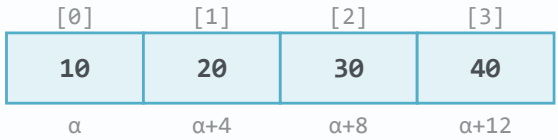
Fark 2: sizeof

Yerel dizide `sizeof list` tüm diziyi verir; parametrede işaretçi boyutunu.

Parametre

`void f(int a[])` ≡ `void f(int *a)`: dizi kopyalanmaz, adresi geçer; boyut ayrı bir parametreyle gönderilir.

İşaretçi aritmetiği: `*(p + i)`



`dizi` yığında 4 ardışık `int` olarak ayrılır; ilk öğenin adresine **α** diyelim.

Adım 1 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
▶    int dizi[] = {10, 20, 30, 40};
9    yazdir(dizi, 4);
10   return 0;
11 }
```

DEĞİŞKENLER

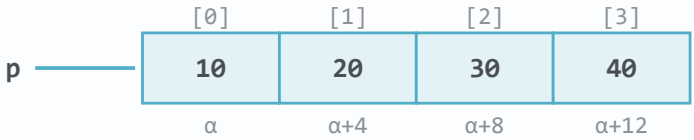
`dizi = [10, 20, 30, 40]`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

İşaretçi aritmetiği: `*(p + i)`



Çağrıda dizi **kopyalanmaz**: yalnızca `&dizi[0]` (α) değeri `p` 'ye geçer.

Adım 2 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int dizi[] = {10, 20, 30, 40};
▶    yazdir(dizi, 4);
10   return 0;
11 }
```

DEĞİŞKENLER

`dizi = [10, 20, 30, 40]`

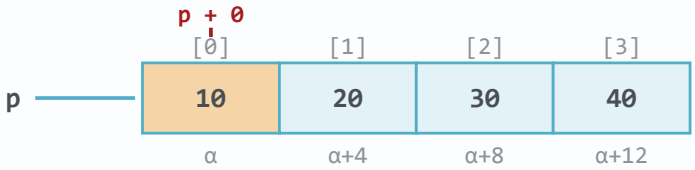
ÇAĞRI YIĞINI

`yazdir(α, 4)`

`main()`

ÇIKTI

İşaretçi aritmetiği: `*(p + i)`



0 < 4 → döngü sürer. `p + 0` = α + 0·4 = **α+0**.

Adım 3 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
▶   for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int dizi[] = {10, 20, 30, 40};
9    yazdir(dizi, 4);
10   return 0;
11 }
```

DEĞİŞKENLER

`p = α` `n = 4` `i = 0`

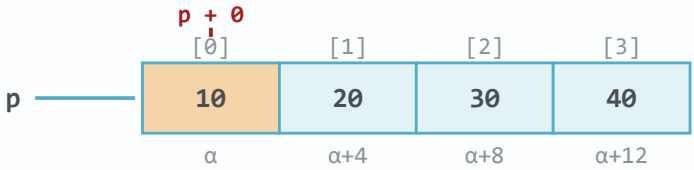
ÇAĞRI YIĞINI

`yazdir(α, 4)`

`main()`

ÇIKTI

İşaretçi aritmetiği: `*(p + i)`



`*(p + 0)` o adresteki değeri okur: **10** (≡ `p[0]`).

Adım 4 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
▶      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int dizi[] = {10, 20, 30, 40};
9    yazdir(dizi, 4);
10   return 0;
11 }
```

DEĞİŞKENLER

`p = α` `n = 4` `i = 0`

ÇAĞRI YIĞINI

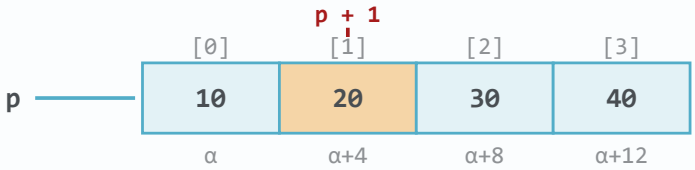
`yazdir(α, 4)`

`main()`

ÇIKTI

```
α 10
```

İşaretçi aritmetiği: `*(p + i)`



1 < 4 → döngü sürer. `p + 1` = α + 1·4 = **α+4**.

Adım 5 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
▶   for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int dizi[] = {10, 20, 30, 40};
9    yazdir(dizi, 4);
10   return 0;
11 }
```

DEĞİŞKENLER

`p = α` `n = 4` `i = 1`

ÇAĞRI YIĞINI

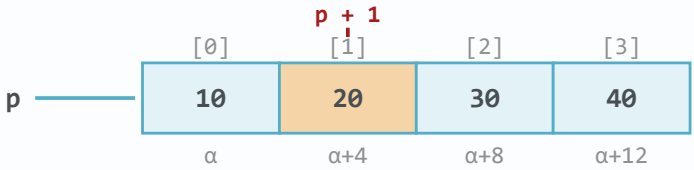
`yazdir(α, 4)`

`main()`

ÇIKTI

```
α 10
```

İşaretçi aritmetiği: `*(p + i)`



`*(p + 1)` o adresteki değeri okur: **20** (≡ `p[1]`).

Adım 6 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
▶      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int dizi[] = {10, 20, 30, 40};
9    yazdir(dizi, 4);
10   return 0;
11 }
```

DEĞİŞKENLER

`p = α` `n = 4` `i = 1`

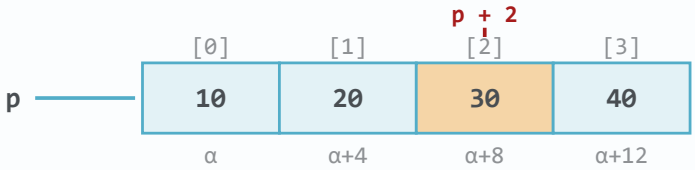
ÇAĞRI YIĞINI

- `yazdir(α, 4)`
- `main()`

ÇIKTI

```
α 10
α+4 20
```

İşaretçi aritmetiği: `*(p + i)`



2 < 4 → döngü sürer. `p + 2` = α + 2·4 = **α+8**.

Adım 7 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
▶   for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int dizi[] = {10, 20, 30, 40};
9    yazdir(dizi, 4);
10   return 0;
11 }
```

DEĞİŞKENLER

`p = α` `n = 4` `i = 2`

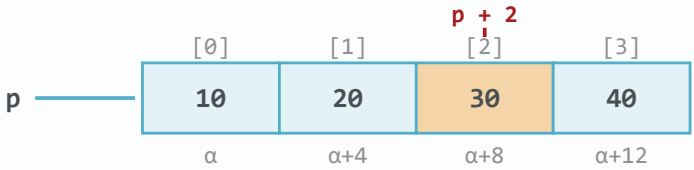
ÇAĞRI YIĞINI

- `yazdir(α, 4)`
- `main()`

ÇIKTI

```
α 10
α+4 20
```

İşaretçi aritmetiği: `*(p + i)`



`*(p + 2)` o adresteki değeri okur: **30** (≡ `p[2]`).

Adım 8 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
▶      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int dizi[] = {10, 20, 30, 40};
9    yazdir(dizi, 4);
10   return 0;
11 }
```

DEĞİŞKENLER

`p = α` `n = 4` `i = 2`

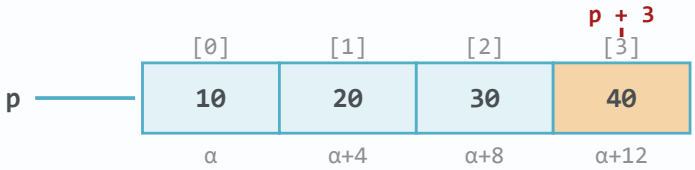
ÇAĞRI YIĞINI

- `yazdir(α, 4)`
- `main()`

ÇIKTI

```
α 10
α+4 20
α+8 30
```

İşaretçi aritmetiği: `*(p + i)`



3 < 4 → döngü sürer. `p + 3` = α + 3·4 = **α+12**.

Adım 9 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
▶   for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int dizi[] = {10, 20, 30, 40};
9    yazdir(dizi, 4);
10   return 0;
11 }
```

DEĞİŞKENLER

`p = α` `n = 4` `i = 3`

ÇAĞRI YIĞINI

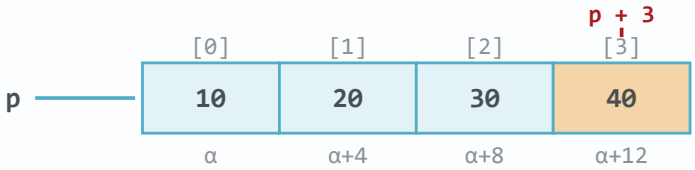
`yazdir(α, 4)`

`main()`

ÇIKTI

```
α 10
α+4 20
α+8 30
```

İşaretçi aritmetiği: `*(p + i)`



`*(p + 3)` o adresteki değeri okur: **40** (≡ `p[3]`).

Adım 10 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
▶      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int dizi[] = {10, 20, 30, 40};
9    yazdir(dizi, 4);
10   return 0;
11 }
```

DEĞİŞKENLER

`p = α` `n = 4` `i = 3`

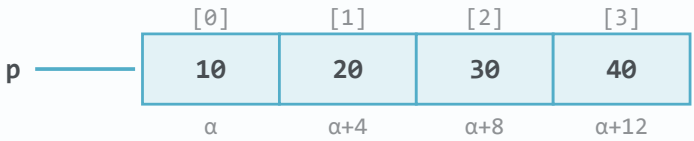
ÇAĞRI YIĞINI

- `yazdir(α, 4)`
- `main()`

ÇIKTI

```
α 10
α+4 20
α+8 30
α+12 40
```

İşaretçi aritmetiği: `*(p + i)`



4 < 4 yanlış → döngü biter; `yazdir` döner.

Adım 11 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
▶   for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int dizi[] = {10, 20, 30, 40};
9    yazdir(dizi, 4);
10   return 0;
11 }
```

DEĞİŞKENLER

`p = α` `n = 4` `i = 4`

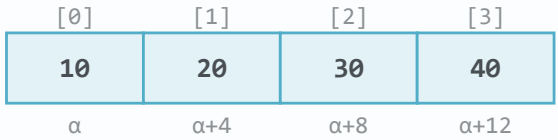
ÇAĞRI YIĞINI

- `yazdir(α, 4)`
- `main()`

ÇIKTI

```
α 10
α+4 20
α+8 30
α+12 40
```

İşaretçi aritmetiği: `*(p + i)`



Her satırda adres 4 bayt artar. (Gerçek `%p` çıktısı onaltılık bir adrestir; burada α ile gösterildi.)

Adım 12 / 12

```
1  void yazdir(int *p, int n) {
2    int i;
3    for (i = 0; i < n; i++)
4      printf("%p %d\n", (void *)(p + i), *(p + i));
5  }
6
7  int main(void) {
8    int dizi[] = {10, 20, 30, 40};
9    yazdir(dizi, 4);
▶    return 0;
11 }
```

DEĞİŞKENLER

`dizi = [10, 20, 30, 40]`

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
α 10
α+4 20
α+8 30
α+12 40
```

Boyutu çalışma anında belli olan dizi

```c
int n, i, *a;
scanf("%d", &n);
a = malloc(n * sizeof *a);   /* n int */
if (a == NULL) {
  fprintf(stderr, "Yetersiz bellek\n");
  exit(EXIT_FAILURE);
}
for (i = 0; i < n; i++)
  a[i] = i * i;              /* dizi gibi */
free(a);                     /* geri ver */
a = NULL;          /* sarkan işaretçi olmasın */
```

Yığın (stack) ve heap bellek

Yerel dizi fonksiyon bitince yok olur; `malloc` bloğu `free` edilene dek yaşar.

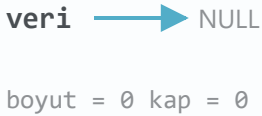
`sizeof *a` kalıbı

Tür değişse de doğru boyutu verir; `sizeof(int*)` ile karıştırma hatasını önler.

`calloc(n, s)`

Bloğu sıfırlayarak ayırır; `realloc` bloğu büyütür/küçültür.

Büyüyen dizi: kapasiteyi ikiye katla



Boş dinamik dizi: `{NULL, 0, 0}`. Sırayla 7, 3, 9, 4, 6 eklenecek.

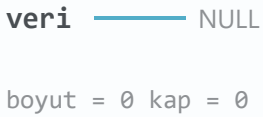
Adım 1 / 19

```
▶ typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3 int ekle(DinDizi *d, int x) {
4   if (d->boyut == d->kap) {              /* dolu */
5     int yeni = d->kap ? 2 * d->kap : 1;
6     int *t = realloc(d->veri, yeni * sizeof *t);
7     if (t == NULL) return 0;    /* eski blok sağlam */
8     d->veri = t;
9     d->kap = yeni;
10  }
11  d->veri[d->boyut++] = x;
12  return 1;
13 }
```

DEĞİŞKENLER

`boyut = 0` `kap = 0` `kopya = 0`

Büyüyen dizi: kapasiteyi ikiye katla



0 == 0 → dolu, büyütülmeli.


```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
▶    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 7` `boyut = 0` `kap = 0` `kopya = 0`

Büyüyen dizi: kapasiteyi ikiye katla



boyut = 0 kap = 1 +0 kopya

yeni = 1. `realloc` 1 hücrelik blok verir; eski 0 öğe kopyalanır (NULL'dan realloc = malloc). Önce `t` 'ye alınır: başarısızsa eski blok kaybolmaz.

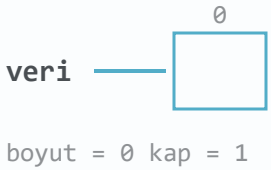
Adım 3 / 19

```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
▶      int yeni = d->kap ? 2 * d->kap : 1;
▶      int *t = realloc(d->veri, yeni * sizeof *t);
▶      if (t == NULL) return 0;     /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 7` `boyut = 0` `kap = 0`
`yeni = 1` `kopya = 0`

Büyüyen dizi: kapasiteyi ikiye katla



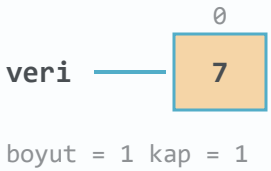
`t ≠ NULL` : `d→veri = t` , **kap = 1**.


```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
▶      d->veri = t;
▶      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 7` `boyut = 0` `kap = 1` `yeni = 1` `kopya = 0`

Büyüyen dizi: kapasiteyi ikiye katla



`veri[0] = 7`, sonra **boyut++** → 1. (Son ek: önce kullanılır, sonra artırılır.)

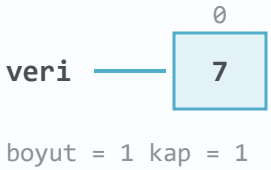
Adım 5 / 19

```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
▶    d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 7` `boyut = 1` `kap = 1` `kopya = 0`

Büyüyen dizi: kapasiteyi ikiye katla



1 == 1 → dolu, büyütülmeli.

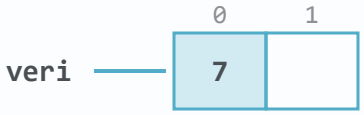
Adım 6 / 19

```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
▶    if (d->boyut == d->kap) {                /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 3` `boyut = 1` `kap = 1` `kopya = 0`

Büyüyen dizi: kapasiteyi ikiye katla



boyut = 1 kap = 2 +1 kopya

yeni = 2. `realloc` 2 hücrelik blok verir; eski 1 öğe kopyalanır (toplam kopya 1). Önce `t` 'ye alınır: başarısızsa eski blok kaybolmaz.

Adım 7 / 19

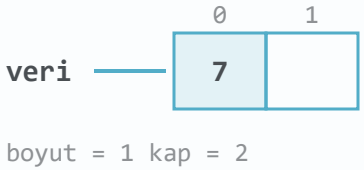
```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
▶      int yeni = d->kap ? 2 * d->kap : 1;
▶      int *t = realloc(d->veri, yeni * sizeof *t);
▶      if (t == NULL) return 0;     /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 3` `boyut = 1` `kap = 1`

`yeni = 2` `kopya = 1`

Büyüyen dizi: kapasiteyi ikiye katla



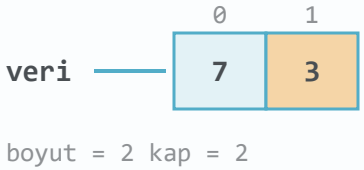
`t ≠ NULL` : `d→veri = t` , **kap = 2**.


```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
▶      d->veri = t;
▶      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 3` `boyut = 1` `kap = 2` `yeni = 2` `kopya = 1`

Büyüyen dizi: kapasiteyi ikiye katla



`veri[1] = 3`, sonra **boyut++** → 2. (Son ek: önce kullanılır, sonra artırılır.)

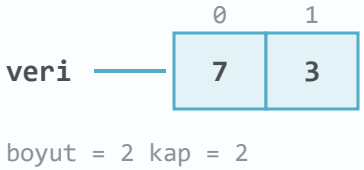
Adım 9 / 19

```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
▶    d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 3` `boyut = 2` `kap = 2` `kopya = 1`

Büyüyen dizi: kapasiteyi ikiye katla



2 == 2 → dolu, büyütülmeli.

Adım 10 / 19

```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
▶    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 9` `boyut = 2` `kap = 2` `kopya = 1`

Büyüyen dizi: kapasiteyi ikiye katla



boyut = 2 kap = 4 +2 kopya

yeni = 4. `realloc` 4 hücrelik blok verir; eski 2 öğe kopyalanır (toplam kopya 3). Önce `t` 'ye alınır: başarısızsa eski blok kaybolmaz.

Adım 11 / 19

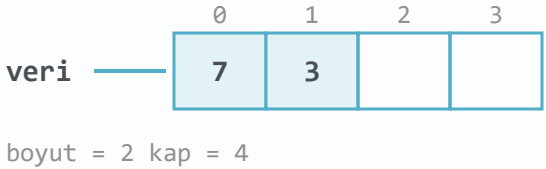
```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
▶      int yeni = d->kap ? 2 * d->kap : 1;
▶      int *t = realloc(d->veri, yeni * sizeof *t);
▶      if (t == NULL) return 0;     /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 9` `boyut = 2` `kap = 2`

`yeni = 4` `kopya = 3`

Büyüyen dizi: kapasiteyi ikiye katla



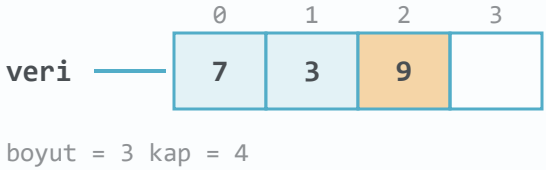
`t ≠ NULL` : `d→veri = t` , **kap = 4**.


```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
▶      d->veri = t;
▶      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 9` `boyut = 2` `kap = 4` `yeni = 4` `kopya = 3`

Büyüyen dizi: kapasiteyi ikiye katla



`veri[2] = 9`, sonra **boyut++** → 3. (Son ek: önce kullanılır, sonra artırılır.)

Adım 13 / 19

```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
▶    d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 9` `boyut = 3` `kap = 4` `kopya = 3`

Büyüyen dizi: kapasiteyi ikiye katla



boyut = 3 kap = 4

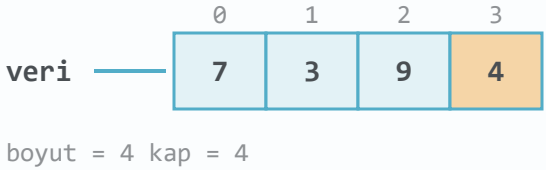
3 == 4 → yer var, doğrudan yazılır.


```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
▶    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

x = 4 | boyut = 3 | kap = 4 | kopya = 3

Büyüyen dizi: kapasiteyi ikiye katla



`veri[3] = 4`, sonra **boyut++** → 4. (Son ek: önce kullanılır, sonra artırılır.)

Adım 15 / 19

```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
▶    d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 4` `boyut = 4` `kap = 4` `kopya = 3`

Büyüyen dizi: kapasiteyi ikiye katla



boyut = 4 kap = 4

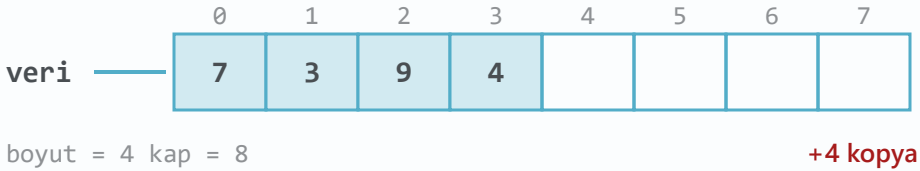
4 == 4 → dolu, büyütülmeli.


```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
▶    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

x = 6 | boyut = 4 | kap = 4 | kopya = 3

Büyüyen dizi: kapasiteyi ikiye katla



yeni = 8. `realloc` 8 hücrelik blok verir; eski 4 öğe kopyalanır (toplam kopya 7). Önce `t` 'ye alınır: başarısızsa eski blok kaybolmaz.

Adım 17 / 19

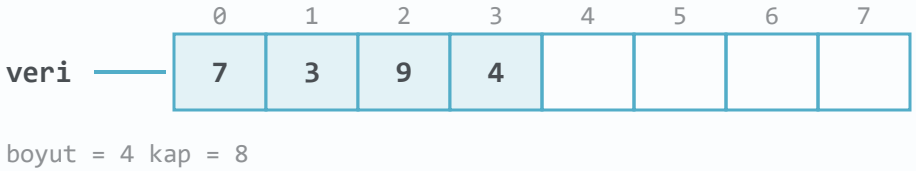
```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
▶      int yeni = d->kap ? 2 * d->kap : 1;
▶      int *t = realloc(d->veri, yeni * sizeof *t);
▶      if (t == NULL) return 0;     /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 6` `boyut = 4` `kap = 4`

`yeni = 8` `kopya = 7`

Büyüyen dizi: kapasiteyi ikiye katla



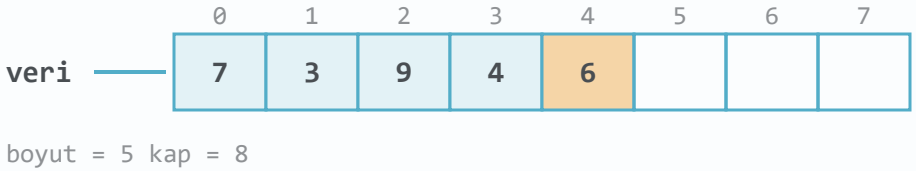
`t ≠ NULL` : `d→veri = t` , **kap = 8**.


```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
▶      d->veri = t;
▶      d->kap = yeni;
10   }
11   d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 6` `boyut = 4` `kap = 8` `yeni = 8` `kopya = 7`

Büyüyen dizi: kapasiteyi ikiye katla



`veri[4] = 6`, sonra **boyut++** → 5. 5 eklemede toplam yalnız **7 kopya** yapıldı.

Adım 19 / 19

```
1  typedef struct { int *veri; int boyut, kap; } DinDizi;
2
3  int ekle(DinDizi *d, int x) {
4    if (d->boyut == d->kap) {              /* dolu */
5      int yeni = d->kap ? 2 * d->kap : 1;
6      int *t = realloc(d->veri, yeni * sizeof *t);
7      if (t == NULL) return 0;    /* eski blok sağlam */
8      d->veri = t;
9      d->kap = yeni;
10   }
▶    d->veri[d->boyut++] = x;
12   return 1;
13 }
```

DEĞİŞKENLER

`x = 6` `boyut = 5` `kap = 8` `kopya = 7`

İkiye katlama: ekleme başına amortize O(1)

n ekleme	katlama: kopya	kopya / n	+1 büyütme: kopya	kopya / n
8	7	0.88	28	3.5
64	63	0.98	2016	31.5
1024	1023	1.00	523776	511.5

Katlamada kopyalar $1 + 2 + 4 + \ldots + 2^k < 2n \rightarrow$ toplam **O(n)** (n yazmayla birlikte toplam maliyet < 3n), ekleme başına **O(1)** (amortize). Her seferinde 1 (ya da sabit c) büyütmek toplam **$n(n-1)/2 = O(n^2)$** kopya demektir.

Tek ekleme yine O(n) olabilir

Büyütme anındaki ekleme pahalıdır; amortize analiz **uzun bir dizi işlemin ortalamasını** garanti eder.

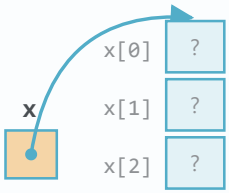
Bellek payı

Kapasite en çok 2·boyut olur; en fazla %50 boş hücre bedelidir.

`realloc` tuzağı

`a = realloc(a, …)` başarısız olursa NULL döner ve eski blok kaybolur (sızıntı). Önce geçici değişkene alınmalı.

2B dinamik dizi: işaretçi dizisi + satırlar



satır = 3
sütun = 4

Önce 3 hücrelik **işaretçi dizisi** ayrılır (`sizeof *x` = `sizeof(int *)`); içerikler henüz belirsiz.

Adım 1 / 10

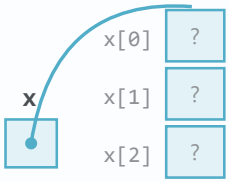
```
1  int **dizi2b(int satir, int sutun) {
▶    int i, **x = malloc(satir * sizeof *x);
3    if (x == NULL) return NULL;
4    for (i = 0; i < satir; i++) {
5      x[i] = malloc(sutun * sizeof **x);
6      if (x[i] == NULL) {                /* geri al */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

DEĞİŞKENLER

`satir = 3` `sutun = 4`

`x = &blok(3 int*)`

2B dinamik dizi: işaretçi dizisi + satırlar



satır = 3
sütun = 4

`x ≠ NULL` : ayırma başarılı, devam.

Adım 2 / 10

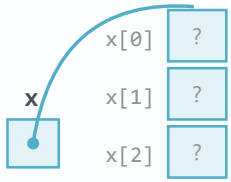
```
1  int **dizi2b(int satir, int sutun) {
2    int i, **x = malloc(satir * sizeof *x);
▶    if (x == NULL) return NULL;
4    for (i = 0; i < satir; i++) {
5      x[i] = malloc(sutun * sizeof **x);
6      if (x[i] == NULL) {                /* geri al */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

DEĞİŞKENLER

`satir = 3` `sutun = 4`

`x = &blok(3 int*)`

2B dinamik dizi: işaretçi dizisi + satırlar



satır = 3
sütun = 4

i = 0 < 3 → 0. satır ayrılacak.

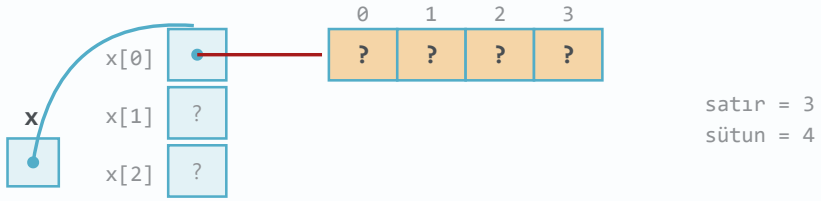
Adım 3 / 10

```
1  int **dizi2b(int satir, int sutun) {
2    int i, **x = malloc(satir * sizeof *x);
3    if (x == NULL) return NULL;
▶   for (i = 0; i < satir; i++) {
5      x[i] = malloc(sutun * sizeof **x);
6      if (x[i] == NULL) {                /* geri al */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

DEĞİŞKENLER

satir = **3** | sutun = **4** | i = **0**

2B dinamik dizi: işaretçi dizisi + satırlar



`x[0]` 4 `int` 'lik yeni bloğu gösterir (`sizeof **x` = `sizeof(int)`). NULL değil.

Adım 4 / 10

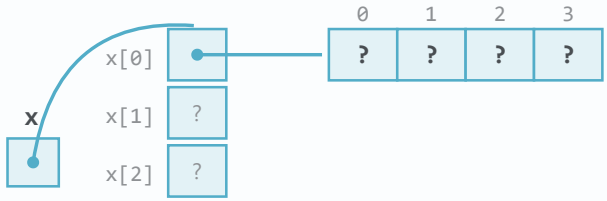
```
1  int **dizi2b(int satir, int sutun) {
2    int i, **x = malloc(satir * sizeof *x);
3    if (x == NULL) return NULL;
4    for (i = 0; i < satir; i++) {
▶      x[i] = malloc(sutun * sizeof **x);
▶      if (x[i] == NULL) {                /* geri al */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

DEĞİŞKENLER

`satir = 3` `sutun = 4` `i = 0`

`x[0] = &satir0(4 int)`

2B dinamik dizi: işaretçi dizisi + satırlar



satır = 3
sütun = 4

i = 1 < 3 → 1. satır ayrılacak.

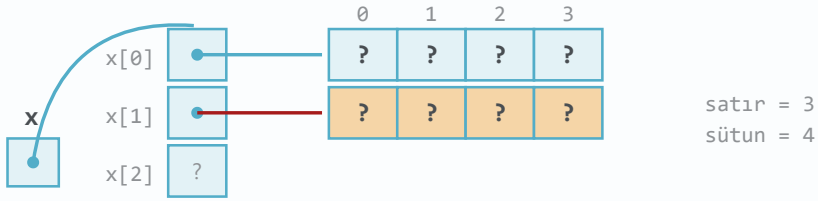
Adım 5 / 10

```
1  int **dizi2b(int satir, int sutun) {
2    int i, **x = malloc(satir * sizeof *x);
3    if (x == NULL) return NULL;
▶   for (i = 0; i < satir; i++) {
5      x[i] = malloc(sutun * sizeof **x);
6      if (x[i] == NULL) {                /* geri al */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

DEĞİŞKENLER

satir = **3** | sutun = **4** | i = **1**

2B dinamik dizi: işaretçi dizisi + satırlar



`x[1]` 4 `int` 'lik yeni bloğu gösterir (`sizeof **x` = `sizeof(int)`). NULL değil.

Adım 6 / 10

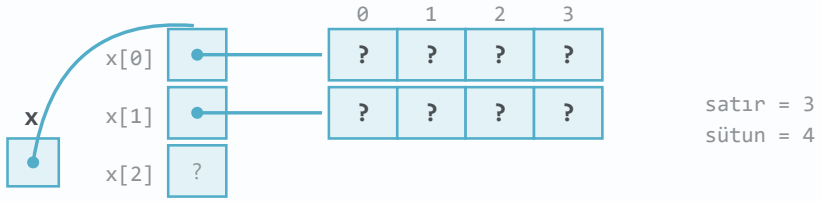
```
1  int **dizi2b(int satir, int sutun) {
2    int i, **x = malloc(satir * sizeof *x);
3    if (x == NULL) return NULL;
4    for (i = 0; i < satir; i++) {
▶      x[i] = malloc(sutun * sizeof **x);
▶      if (x[i] == NULL) {                /* geri al */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

DEĞİŞKENLER

`satir = 3` `sutun = 4` `i = 1`

`x[1] = &satir1(4 int)`

2B dinamik dizi: işaretçi dizisi + satırlar



i = 2 < 3 → 2. satır ayrılacak.

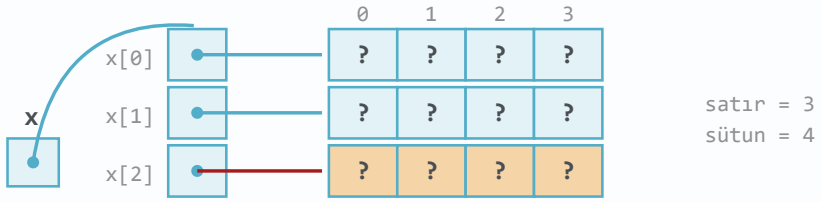
Adım 7 / 10

```
1  int **dizi2b(int satir, int sutun) {
2    int i, **x = malloc(satir * sizeof *x);
3    if (x == NULL) return NULL;
▶  for (i = 0; i < satir; i++) {
5      x[i] = malloc(sutun * sizeof **x);
6      if (x[i] == NULL) {                /* geri al */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

DEĞİŞKENLER

satir = 3 | sutun = 4 | i = 2

2B dinamik dizi: işaretçi dizisi + satırlar



`x[2]` 4 `int` 'lik yeni bloğu gösterir (`sizeof **x` = `sizeof(int)`). NULL değil.

Adım 8 / 10

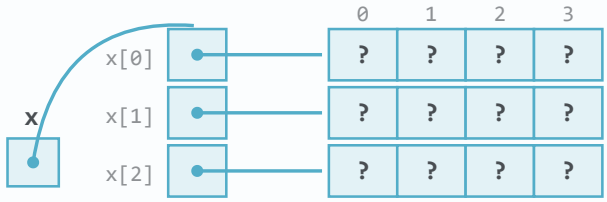
```
1  int **dizi2b(int satir, int sutun) {
2    int i, **x = malloc(satir * sizeof *x);
3    if (x == NULL) return NULL;
4    for (i = 0; i < satir; i++) {
▶      x[i] = malloc(sutun * sizeof **x);
▶      if (x[i] == NULL) {                /* geri al */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

DEĞİŞKENLER

`satir = 3` `sutun = 4` `i = 2`

`x[2] = &satir2(4 int)`

2B dinamik dizi: işaretçi dizisi + satırlar



satır = 3
sütun = 4

3 < 3 yanlış → döngü biter.

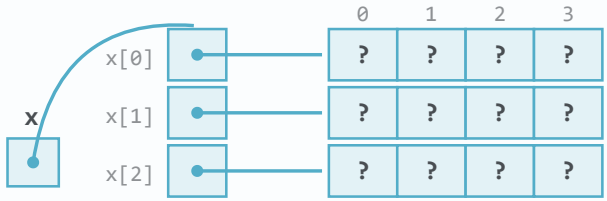
Adım 9 / 10

```
1  int **dizi2b(int satir, int sutun) {
2    int i, **x = malloc(satir * sizeof *x);
3    if (x == NULL) return NULL;
▶    for (i = 0; i < satir; i++) {
5      x[i] = malloc(sutun * sizeof **x);
6      if (x[i] == NULL) {                /* geri al */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
11   return x;
12 }
```

DEĞİŞKENLER

`satir = 3` `sutun = 4` `i = 3`

2B dinamik dizi: işaretçi dizisi + satırlar



`x` döner; `x[i][j]` ≡ `*(*(x + i) + j)`. Satırlar ardışık olmak zorunda değil. Bırakırken önce her `x[i]`, sonra `x` `free` edilir.

Adım 10 / 10

```
1  int **dizi2b(int satir, int sutun) {
2    int i, **x = malloc(satir * sizeof *x);
3    if (x == NULL) return NULL;
4    for (i = 0; i < satir; i++) {
5      x[i] = malloc(sutun * sizeof **x);
6      if (x[i] == NULL) {                /* geri al */
7        while (--i >= 0) free(x[i]);
8        free(x); return NULL;
9      }
10   }
▶    return x;
12 }
```

DEĞİŞKENLER

`satir = 3` `sutun = 4`

`x = &blok(3 int*)`

Bellek tek boyutlu, dizi çok boyutlu

`A[upper`$_0$`][upper`$_1$`]…[upper`$_{n-1}$`]` olarak tanımlı n boyutlu dizide $\prod \text{upper}_i$ öğe vardır. Ör. `A[10][10][10]` → 1000 öğe.

Bellek ise tek boyutlu bir bayt dizisidir; derleyici öğeleri bir **sıraya dizmek** zorundadır:

- **Satır öncelikli** (row major): önce 0. satırın tümü, sonra 1. satır… (C, C++, Python/NumPy varsayılanı)
- **Sütun öncelikli** (column major): önce 0. sütun… (Fortran, MATLAB)

Soru hep aynı: **A[i][j]** öğesinden önce kaç öğe var? Adres = α + (öncekilerin sayısı) · öğe boyutu.

C'de

`int x[3][4];` aslında 4'lük dizilerden oluşan 3 öğeli bir dizidir; `x[1]` 1. satırın başını gösterir.

`int x[3][4]` belleğe iki biçimde dizilebilir

	0	1	2	3
0	x00 0	x01 1	x02 2	x03 3
1	x10 4	x11 5	x12 6	x13 7
2	x20 8	x21 9	x22 10	x23 11

satır öncelikli: satır satır

0	1	2	3	4	5	6	7	8	9	10	11
x00	x01	x02	x03	x10	x11	x12	x13	x20	x21	x22	x23

α α + 11·s

Satır öncelikli: x[i][j]'den önce i tam satır (i·upper$_1$ öğe) ve j öğe vardır → **&x[i][j] = α + (i·upper$_1$ + j)·s**. C bunu kullanır.

Adım 1 / 2

`α + i·upper₁ + j` formülü öğe (hücre) birimindedir; bayt adresi için s = sizeof(öğe) ile çarpılır.

`int x[3][4]` belleğe iki biçimde dizilebilir

	0	1	2	3
0	x00 0	x01 3	x02 6	x03 9
1	x10 1	x11 4	x12 7	x13 10
2	x20 2	x21 5	x22 8	x23 11

sütun öncelikli: sütun sütun

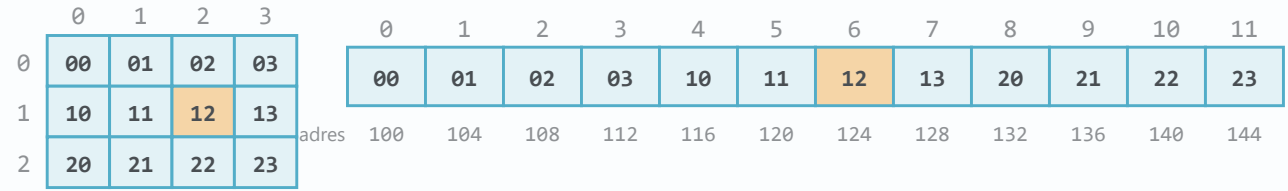
0	1	2	3	4	5	6	7	8	9	10	11
x00	x10	x20	x01	x11	x21	x02	x12	x22	x03	x13	x23

α α + 11·s

Sütun öncelikli: önce j tam sütun (j·upper$_0$ öğe), sonra i öğe → **&x[i][j] = α + (j·upper$_0$ + i)·s**. Aynı öğe farklı konumda: x12 → satırda 6, sütunda 7.

`α + i·upper$_1$ + j` formülü öğe (hücre) birimindedir; bayt adresi için s = sizeof(öğe) ile çarpılır.

Satır öncelikli adres hesabı



`x[1][2]` 'nin adresi istenir: α = 100, satır uzunluğu **upper$_1$ = 4**, **s = 4** bayt.

Adım 1 / 8

```
1  /* int x[3][4]: satır öncelikli bayt adresi */
2  long adres(long alfa, int i, int j,
3             int ust1, int s) {
4    long k = (long)i * ust1 + j;   /* öncekiler */
5    return alfa + k * s;
6  }
7
8  int main(void) {
▶    printf("%ld\n", adres(100, 1, 2, 4, 4));
10   printf("%ld\n", adres(100, 2, 3, 4, 4));
11   return 0;
12 }
```

DEĞİŞKENLER

`alfa = 100` `i = 1` `j = 2` `ust1 = 4` `s = 4`

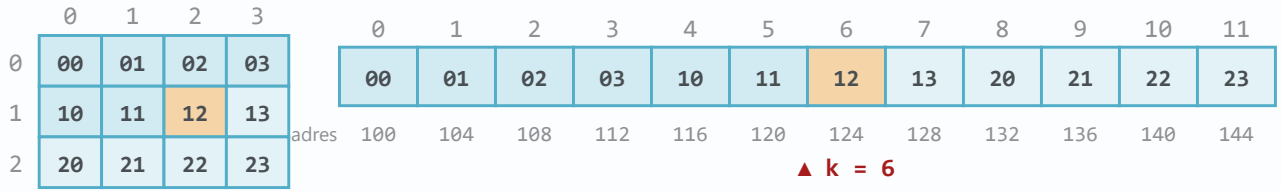
ÇAĞRI YIĞINI

`adres(100, 1, 2, 4, 4)`

`main()`

ÇIKTI

Satır öncelikli adres hesabı



Önce **1 tam satır** (1·4 = 4 öğe) ve satırında **2 öğe** var → **k = 6** (mavi hücreler).

Adım 2 / 8

```
1  /* int x[3][4]: satır öncelikli bayt adresi */
2  long adres(long alfa, int i, int j,
3             int ust1, int s) {
▶    long k = (long)i * ust1 + j;   /* öncekiler */
5    return alfa + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", adres(100, 1, 2, 4, 4));
10   printf("%ld\n", adres(100, 2, 3, 4, 4));
11   return 0;
12 }
```

DEĞİŞKENLER

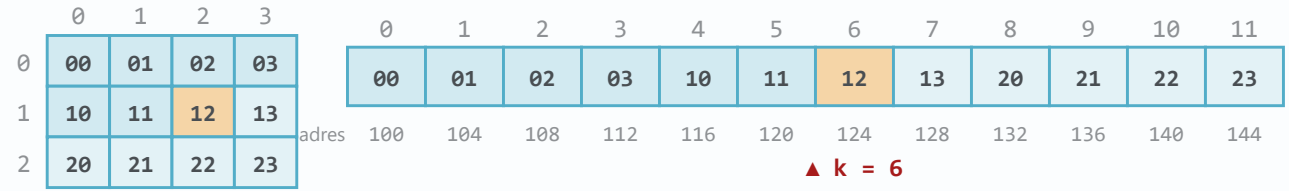
`alfa = 100` `i = 1` `j = 2`
`ust1 = 4` `s = 4` `k = 6`

ÇAĞRI YIĞINI

`adres(100, 1, 2, 4, 4)`
`main()`

ÇIKTI

Satır öncelikli adres hesabı



100 + 6·4 = 124 döner.

Adım 3 / 8

```
1  /* int x[3][4]: satır öncelikli bayt adresi */
2  long adres(long alfa, int i, int j,
3             int ust1, int s) {
4    long k = (long)i * ust1 + j;   /* öncekiler */
▶    return alfa + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", adres(100, 1, 2, 4, 4));
10   printf("%ld\n", adres(100, 2, 3, 4, 4));
11   return 0;
12 }
```

DEĞİŞKENLER

alfa = 100 | i = 1 | j = 2 | ust1 = 4 | s = 4 | k = 6

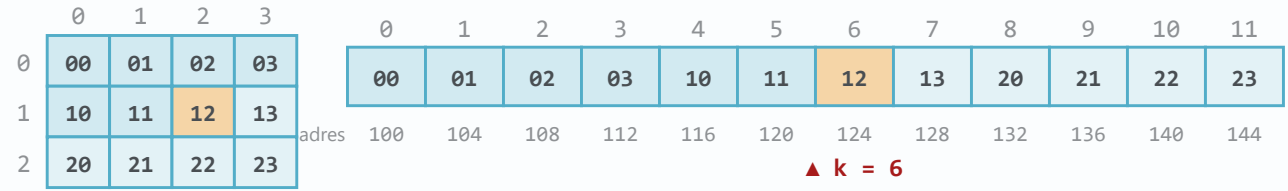
ÇAĞRI YIĞINI

adres(100, 1, 2, 4, 4)

main()

ÇIKTI

Satır öncelikli adres hesabı



`printf` 124 yazar. Sütun öncelikli olsaydı k = j·3 + i = 7 → 128 olurdu.

Adım 4 / 8

```
1  /* int x[3][4]: satır öncelikli bayt adresi */
2  long adres(long alfa, int i, int j,
3             int ust1, int s) {
4    long k = (long)i * ust1 + j;   /* öncekiler */
5    return alfa + k * s;
6  }
7
8  int main(void) {
▶    printf("%ld\n", adres(100, 1, 2, 4, 4));
10   printf("%ld\n", adres(100, 2, 3, 4, 4));
11   return 0;
12 }
```

DEĞİŞKENLER

—

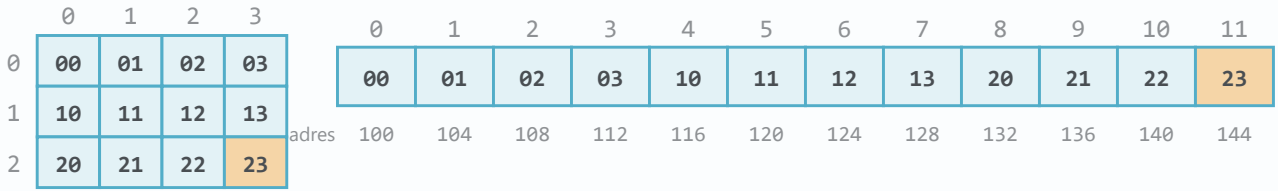
ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
124
```

Satır öncelikli adres hesabı



`x[2][3]` 'nin adresi istenir: α = 100, satır uzunluğu **upper$_1$ = 4**, **s = 4** bayt.

Adım 5 / 8

```
1  /* int x[3][4]: satır öncelikli bayt adresi */
2  long adres(long alfa, int i, int j,
3             int ust1, int s) {
4    long k = (long)i * ust1 + j;   /* öncekiler */
5    return alfa + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", adres(100, 1, 2, 4, 4));
▶    printf("%ld\n", adres(100, 2, 3, 4, 4));
11   return 0;
12 }
```

DEĞİŞKENLER

`alfa = 100` `i = 2` `j = 3` `ust1 = 4` `s = 4`

ÇAĞRI YIĞINI

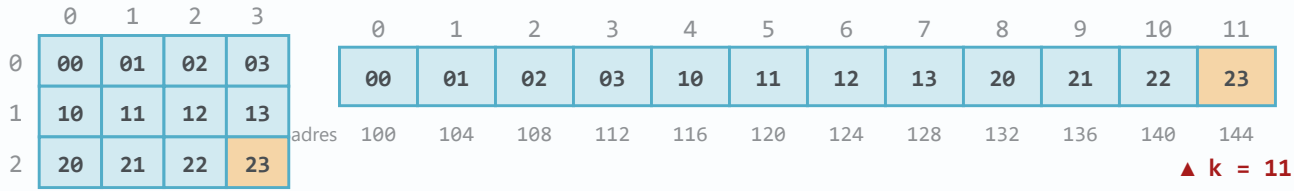
`adres(100, 2, 3, 4, 4)`

`main()`

ÇIKTI

```
124
```

Satır öncelikli adres hesabı



Önce **2 tam satır** (2·4 = 8 öğe) ve satırında **3 öğe** var → **k = 11** (mavi hücreler).

Adım 6 / 8

```
1  /* int x[3][4]: satır öncelikli bayt adresi */
2  long adres(long alfa, int i, int j,
3             int ust1, int s) {
▶    long k = (long)i * ust1 + j;   /* öncekiler */
5    return alfa + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", adres(100, 1, 2, 4, 4));
10   printf("%ld\n", adres(100, 2, 3, 4, 4));
11   return 0;
12 }
```

DEĞİŞKENLER

alfa = 100 | i = 2 | j = 3 | ust1 = 4 | s = 4 | k = 11

ÇAĞRI YIĞINI

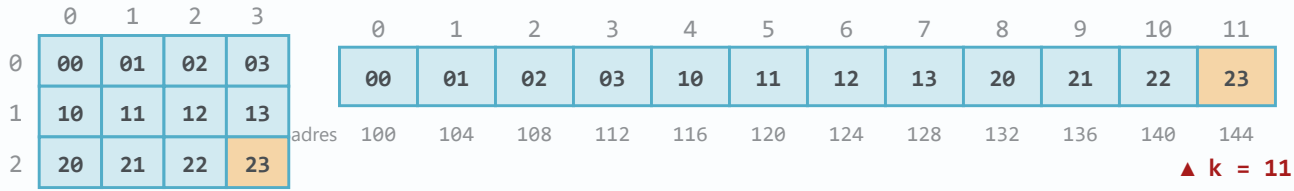
adres(100, 2, 3, 4, 4)

main()

ÇIKTI

```
124
```

Satır öncelikli adres hesabı



100 + 11·4 = 144 döner.

Adım 7 / 8

```
1  /* int x[3][4]: satır öncelikli bayt adresi */
2  long adres(long alfa, int i, int j,
3             int ust1, int s) {
4    long k = (long)i * ust1 + j;   /* öncekiler */
▶    return alfa + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", adres(100, 1, 2, 4, 4));
10   printf("%ld\n", adres(100, 2, 3, 4, 4));
11   return 0;
12 }
```

DEĞİŞKENLER

alfa = 100 | i = 2 | j = 3 | ust1 = 4 | s = 4 | k = 11

ÇAĞRI YIĞINI

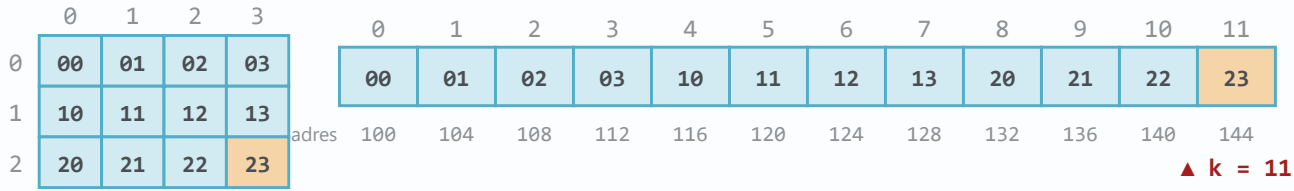
adres(100, 2, 3, 4, 4)

main()

ÇIKTI

```
124
```

Satır öncelikli adres hesabı



`printf` 144 yazar. Son öğe: 100 + 11·4 = 144.

Adım 8 / 8

```
1  /* int x[3][4]: satır öncelikli bayt adresi */
2  long adres(long alfa, int i, int j,
3             int ust1, int s) {
4    long k = (long)i * ust1 + j;   /* öncekiler */
5    return alfa + k * s;
6  }
7
8  int main(void) {
9    printf("%ld\n", adres(100, 1, 2, 4, 4));
▶    printf("%ld\n", adres(100, 2, 3, 4, 4));
11   return 0;
12 }
```

DEĞİŞKENLER

—

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
124
144
```

`G[2][2][3]`: dilim · satır · sütun

dilim 0

`[0][0][0]` +0	`[0][0][1]` +1	`[0][0][2]` +2
`[0][1][0]` +3	`[0][1][1]` +4	`[0][1][2]` +5

dilim 1

`[1][0][0]` +6	`[1][0][1]` +7	`[1][0][2]` +8
`[1][1][0]` +9	`[1][1][1]` +10	`[1][1][2]` +11

öğe	adres (öğe birimi)
`[0][0][0]`	α
`[i][0][0]`	$\alpha + i \cdot u_1 \cdot u_2$
`[i][j][0]`	$\alpha + i \cdot u_1 \cdot u_2 + j \cdot u_2$
`[i][j][k]`	$\alpha + i \cdot u_1 \cdot u_2 + j \cdot u_2 + k$

Son ifadede **3 çarpma + 3 toplama** var. Çarpma toplamadan pahalı olduğundan derleyici paranteze alır:

$$\alpha + (i \cdot u_1 + j) \cdot u_2 + k$$

→ **2 çarpma + 3 toplama**. Örnek: [1][0][2] → (1·2 + 0)·3 + 2 = **8**.

n boyutlu dizide adres formülü

$A[u_0][u_1]\ldots[u_{n-1}]$ için:

öğe	adres
A[0][0]…[0]	α
$A[i_0][0]\ldots[0]$	$\alpha + i_0 \cdot u_1 u_2 \ldots u_{n-1}$
$A[i_0][i_1]\ldots[0]$	$\ldots + i_1 \cdot u_2 \ldots u_{n-1}$
$A[i_0]\ldots[i_{n-1}]$	$\ldots + i_{n-1}$

$$\&A[i_0]\ldots[i_{n-1}] = \alpha + \sum_{j=0}^{n-1} i_j \cdot a_j$$

$$a_j = \prod_{k=j+1}^{n-1} u_k \quad (0 \le j < n-1),\ a_{n-1} = 1$$

a_j katsayıları yalnız boyutlara bağlıdır: derleyici onları **bir kez** hesaplar. Horner biçimiyle **n−1 çarpma** yeter: $t = i_0$; $t = t \cdot u_j + i_j$.

n boyutlu bağıl konum: Horner yöntemi

dilim 0

000	001	002
010	011	012

dilim 1

100	101	102
110	111	112

000	001	002	010	011	012	100	101	102	110	111	112
0	1	2	3	4	5	6	7	8	9	10	11

`G[1][0][2]` aranıyor. **t = ind[0] = 1**: 1 numaralı dilim.

Adım 1 / 7

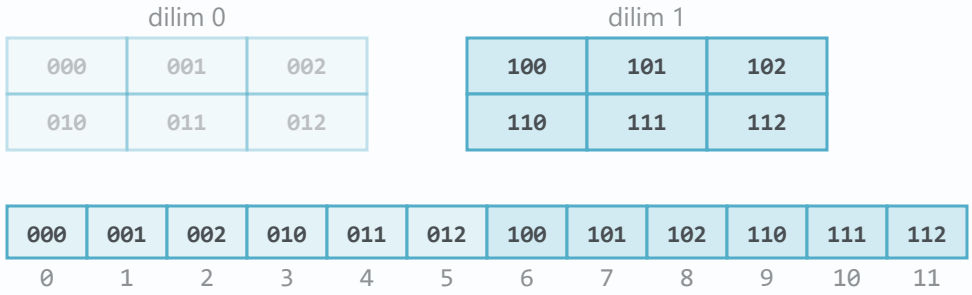
```
1 /* ind: indisler, ust: boyutlar, n: boyut sayısı */
2 int ofset(const int ind[], const int ust[], int n) {
▶   int j, t = ind[0];
4   for (j = 1; j < n; j++)
5     t = t * ust[j] + ind[j];            /* Horner */
6   return t;                 /* adres = alfa + t*s */
7 }
```

DEĞİŞKENLER

`ind = [1, 0, 2]`

`ust = [2, 2, 3]` `n = 3` `t = 1`

n boyutlu bağıl konum: Horner yöntemi



j = 1 < 3 → bir boyut daha içeri.

Adım 2 / 7

```
1 /* ind: indisler, ust: boyutlar, n: boyut sayısı */
2 int ofset(const int ind[], const int ust[], int n) {
3   int j, t = ind[0];
4   for (j = 1; j < n; j++)
5     t = t * ust[j] + ind[j];            /* Horner */
6   return t;                 /* adres = alfa + t*s */
7 }
```

DEĞİŞKENLER

`ind = [1, 0, 2]`

`ust = [2, 2, 3]` `n = 3` `t = 1`

`j = 1`

n boyutlu bağıl konum: Horner yöntemi

dilim 0

000	001	002
010	011	012

dilim 1

100	101	102
110	111	112

000	001	002	010	011	012	100	101	102	110	111	112
0	1	2	3	4	5	6	7	8	9	10	11

t = 1·2 + 0 = 2: bu, dilimler boyunca sayılan “genel satır” numarası (dilim 1, satır 0).

Adım 3 / 7

```
1 /* ind: indisler, ust: boyutlar, n: boyut sayısı */
2 int ofset(const int ind[], const int ust[], int n) {
3   int j, t = ind[0];
4   for (j = 1; j < n; j++)
▶     t = t * ust[j] + ind[j];            /* Horner */
6   return t;                 /* adres = alfa + t*s */
7 }
```

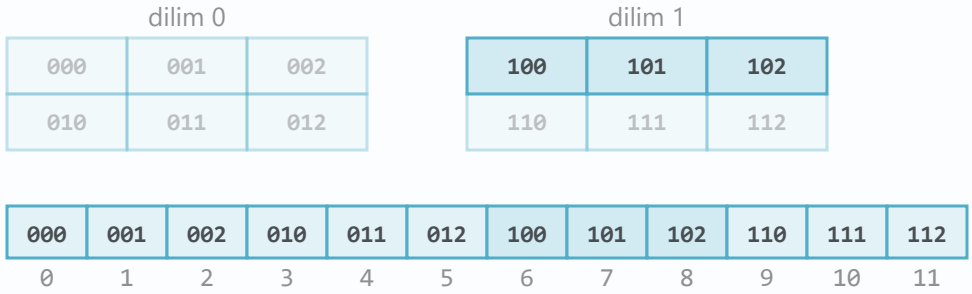
DEĞİŞKENLER

`ind = [1, 0, 2]`

`ust = [2, 2, 3]` `n = 3` `t = 2`

`j = 1`

n boyutlu bağıl konum: Horner yöntemi



j = 2 < 3 → bir boyut daha içeri.

Adım 4 / 7

```
1 /* ind: indisler, ust: boyutlar, n: boyut sayısı */
2 int ofset(const int ind[], const int ust[], int n) {
3   int j, t = ind[0];
▶   for (j = 1; j < n; j++)
5     t = t * ust[j] + ind[j];            /* Horner */
6   return t;               /* adres = alfa + t*s */
7 }
```

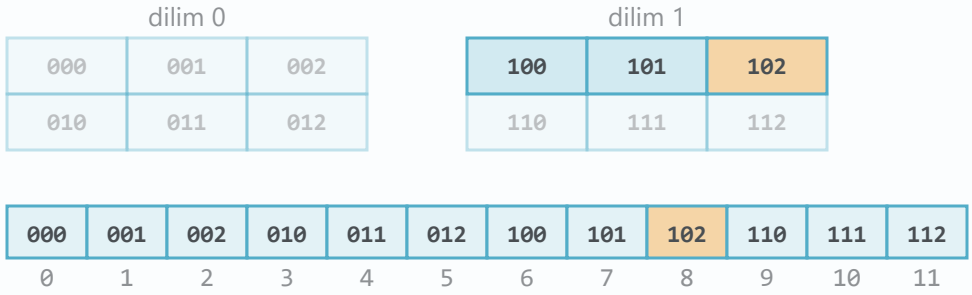
DEĞİŞKENLER

`ind = [1, 0, 2]`

`ust = [2, 2, 3]` `n = 3` `t = 2`

`j = 2`

n boyutlu bağıl konum: Horner yöntemi



t = 2·3 + 2 = 8: 8 öğe önde → hücre 8.

Adım 5 / 7

```
1 /* ind: indisler, ust: boyutlar, n: boyut sayısı */
2 int ofset(const int ind[], const int ust[], int n) {
3   int j, t = ind[0];
4   for (j = 1; j < n; j++)
▶     t = t * ust[j] + ind[j];            /* Horner */
6   return t;                 /* adres = alfa + t*s */
7 }
```

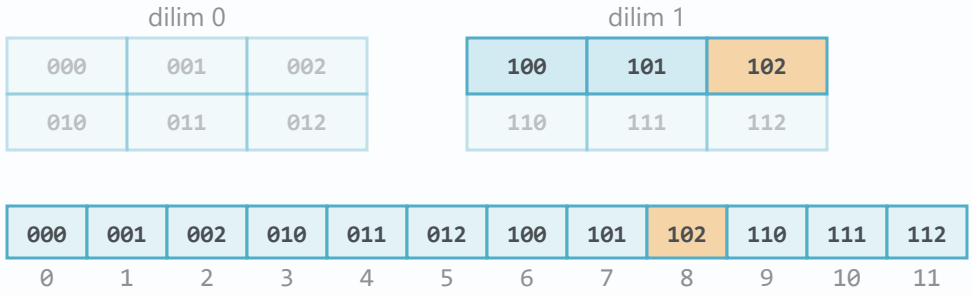
DEĞİŞKENLER

`ind = [1, 0, 2]`

`ust = [2, 2, 3]` `n = 3` `t = 8`

`j = 2`

n boyutlu bağıl konum: Horner yöntemi



3 < 3 yanlış → döngü biter.

Adım 6 / 7

```
1 /* ind: indisler, ust: boyutlar, n: boyut sayısı */
2 int ofset(const int ind[], const int ust[], int n) {
3   int j, t = ind[0];
▶   for (j = 1; j < n; j++)
5     t = t * ust[j] + ind[j];            /* Horner */
6   return t;               /* adres = alfa + t*s */
7 }
```

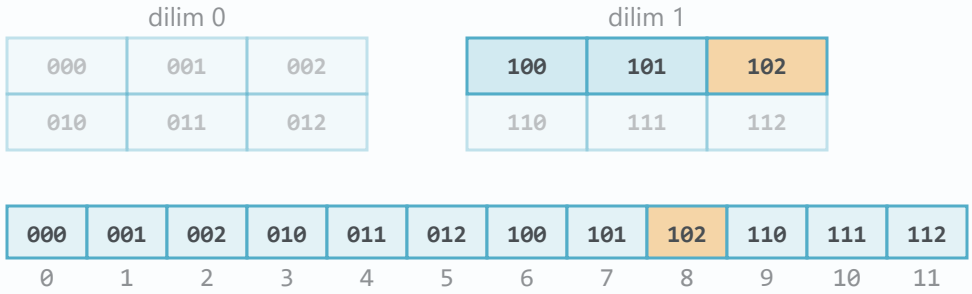
DEĞİŞKENLER

`ind = [1, 0, 2]`

`ust = [2, 2, 3]` `n = 3` `t = 8`

`j = 3`

n boyutlu bağıl konum: Horner yöntemi



8 döner. Toplam **2 çarpma** (n−1). Kontrol: 1·2·3 + 0·3 + 2 = 8. `ust[0]` hiç kullanılmadı!

Adım 7 / 7

```
1 /* ind: indisler, ust: boyutlar, n: boyut sayısı */
2 int ofset(const int ind[], const int ust[], int n) {
3   int j, t = ind[0];
4   for (j = 1; j < n; j++)
5     t = t * ust[j] + ind[j];            /* Horner */
▶   return t;                 /* adres = alfa + t*s */
7 }
```

DEĞİŞKENLER

`ind = [1, 0, 2]`

`ust = [2, 2, 3]` `n = 3` `t = 8`

Matrisin yarısını tek boyutlu dizide tutmak

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

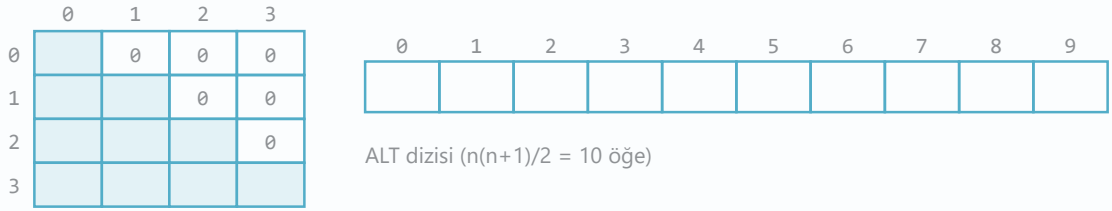
ALT dizisi (n(n+1)/2 = 10 öğe)

Kare matris köşegenle ikiye bölünür; bazen **bir yarısı yeter** (şehirler arası uzaklıklar, periyodik tablodaki atom ağırlıkları…). Veri tek boyutlu **ALT** dizisinde tutulur; kullanıcıya iki boyutlu bir matrisle çalışıyormuş **arayüzü** sunulur.

i. satırda **i+1** öğe var → toplam **n(n+1)/2**. a_{ij}, önceki satırların öğeleri + satırdaki sırası:

$$k = i(i+1)/2 + j \quad (j \le i)$$

Alt üçgeni okumak: satır başı `k`



4·5/2 = **10 ≤ 100**: ALT dizisine sığar. Girdi: `3 1 4 1 5 9 2 6 5 3` .

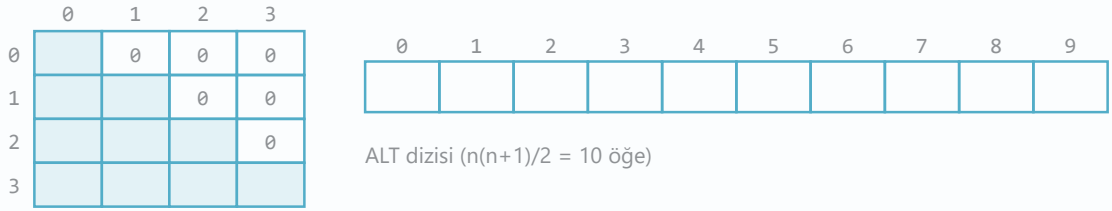
Adım 1 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
▶    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4`

Alt üçgeni okumak: satır başı `k`



k = 1·0/2 = 0: 0. satırdan önce 0 öğe yerleşti.

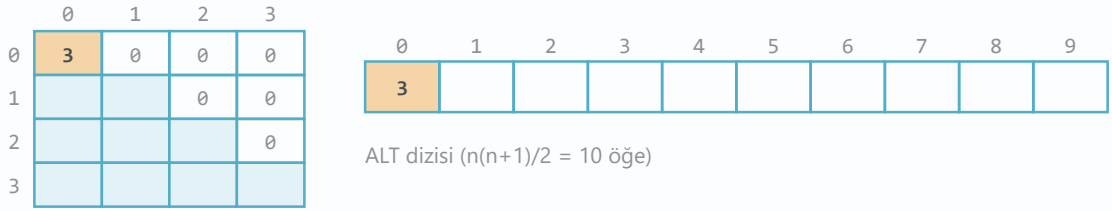
Adım 2 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
▶      k = (i + 1) * i / 2;      /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 0` `k = 0`

Alt üçgeni okumak: satır başı `k`



`alt[0+0] = alt[0]` ← 3 (a_{00}).

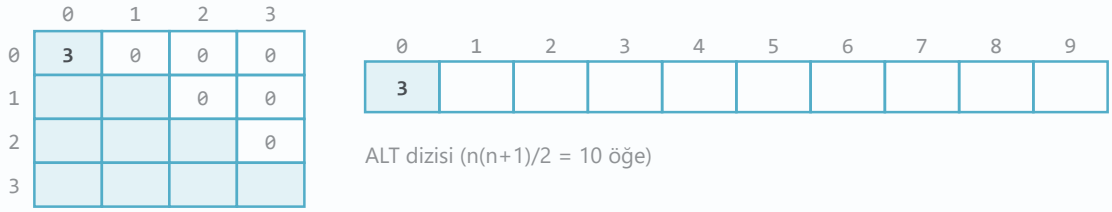
Adım 3 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;       /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 0` `k = 0` `j = 0`

Alt üçgeni okumak: satır başı `k`



k = 2·1/2 = 1: 1. satırdan önce 1 öğe yerleşti.

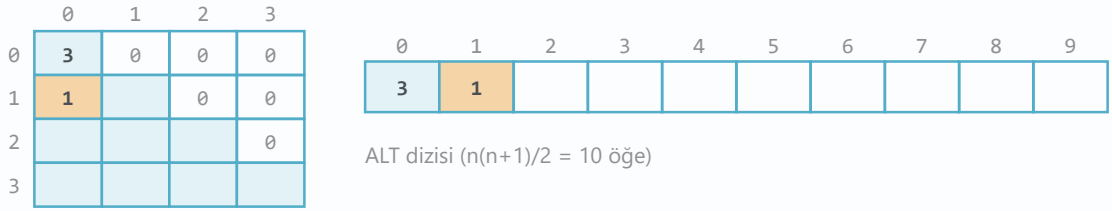
Adım 4 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
▶      k = (i + 1) * i / 2;      /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 1` `k = 1`

Alt üçgeni okumak: satır başı `k`



`alt[1+0] = alt[1]` ← 1 (a_{10}).

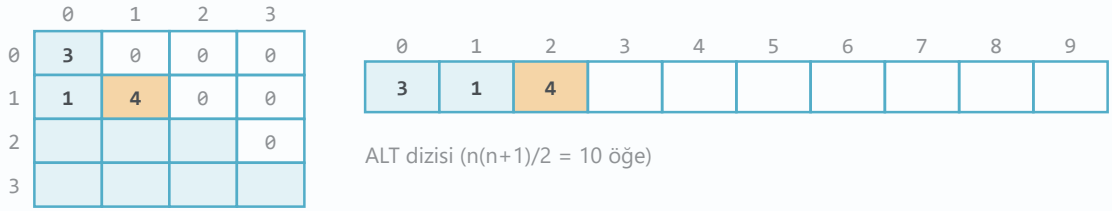
Adım 5 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;       /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 1` `k = 1` `j = 0`

Alt üçgeni okumak: satır başı `k`



`alt[1+1] = alt[2]` ← 4 (a_{11}).

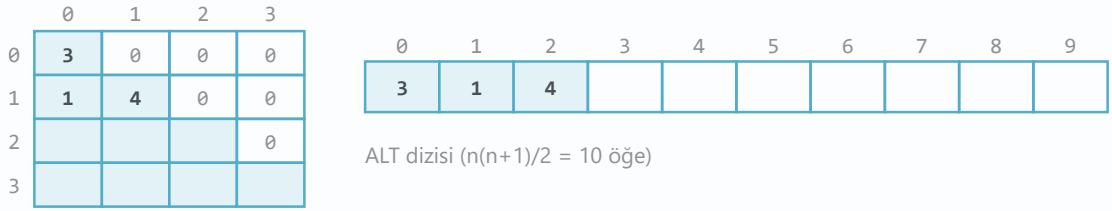
Adım 6 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;       /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 1` `k = 1` `j = 1`

Alt üçgeni okumak: satır başı `k`



k = 3·2/2 = 3: 2. satırdan önce 3 öğe yerleşti.


```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
▶      k = (i + 1) * i / 2;      /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 2` `k = 3`

Alt üçgeni okumak: satır başı `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1			0
3				

0	1	2	3	4	5	6	7	8	9
3	1	4	1						

ALT dizisi (n(n+1)/2 = 10 öğe)

`alt[3+0] = alt[3]` ← 1 (a_{20}).

Adım 8 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;       /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 2` `k = 3` `j = 0`

Alt üçgeni okumak: satır başı `k`

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

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5					

ALT dizisi (n(n+1)/2 = 10 öğe)

`alt[3+1] = alt[4]` ← 5 (a_{21}).


```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 2` `k = 3` `j = 1`

Alt üçgeni okumak: satır başı `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3				

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9				

ALT dizisi (n(n+1)/2 = 10 öğe)

`alt[3+2] = alt[5]` ← 9 (a_{22}).

Adım 10 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;       /* öncekiler */
9      for (j = 0; j <= i; j++)   /* j <= i */
▶        scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 2` `k = 3` `j = 2`

Alt üçgeni okumak: satır başı `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3				

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9				

ALT dizisi (n(n+1)/2 = 10 öğe)

k = 4·3/2 = 6: 3. satırdan önce 6 öğe yerleşti.

Adım 11 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
▶      k = (i + 1) * i / 2;      /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 3` `k = 6`

Alt üçgeni okumak: satır başı `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3	2			

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9	2			

ALT dizisi (n(n+1)/2 = 10 öğe)

`alt[6+0] = alt[6]` ← 2 (a_{30}).

Adım 12 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;       /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 3` `k = 6` `j = 0`

Alt üçgeni okumak: satır başı `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3	2	6		

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9	2	6		

ALT dizisi (n(n+1)/2 = 10 öğe)

`alt[6+1] = alt[7]` ← 6 (a_{31}).

Adım 13 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;       /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 3` `k = 6` `j = 1`

Alt üçgeni okumak: satır başı `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3	2	6	5	

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9	2	6	5	

ALT dizisi (n(n+1)/2 = 10 öğe)

`alt[6+2] = alt[8]` ← 5 (a_{32}).

Adım 14 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;       /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 3` `k = 6` `j = 2`

Alt üçgeni okumak: satır başı `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3	2	6	5	3

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9	2	6	5	3

ALT dizisi (n(n+1)/2 = 10 öğe)

`alt[6+3] = alt[9]` ← 3 (a_{33}).


```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;       /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
▶        scanf("%d", &alt[k + j]);
11   }
12 }
```

DEĞİŞKENLER

`n = 4` `i = 3` `k = 6` `j = 3`

Alt üçgeni okumak: satır başı `k`

	0	1	2	3
0	3	0	0	0
1	1	4	0	0
2	1	5	9	0
3	2	6	5	3

0	1	2	3	4	5	6	7	8	9
3	1	4	1	5	9	2	6	5	3

ALT dizisi (n(n+1)/2 = 10 öğe)

10 değer okundu; üst üçgendeki 6 sıfır **hiç saklanmadı**.

Adım 16 / 16

```
1  #define MAX_SIZE 100
2  void AltUcgenOku(int alt[], int n) {
3    int i, j, k;
4    if (n * (n + 1) / 2 > MAX_SIZE) {
5      printf("Alt dizi boyutu yetersiz!\n"); exit(1);
6    }
7    for (i = 0; i < n; i++) {
8      k = (i + 1) * i / 2;      /* öncekiler */
9      for (j = 0; j <= i; j++)  /* j <= i */
10       scanf("%d", &alt[k + j]);
11   }
▶ }
```

DEĞİŞKENLER

`n = 4`

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

a_{21} isteniyor.

Adım 1 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
5    } else if (i < j) return -1;    /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
▶  printf("%d\n", AltUcgenEris(2, 1, 4));
10 printf("%d\n", AltUcgenEris(1, 3, 4));
11 printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

`i = 2` `j = 1` `n = 4`

ÇIKTI

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

Sınır denetimi: indisler 0...3 aralığında.

Adım 2 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
▶    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
5    } else if (i < j) return -1;    /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
9  printf("%d\n", AltUcgenEris(2, 1, 4));
10 printf("%d\n", AltUcgenEris(1, 3, 4));
11 printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

`i = 2` `j = 1` `n = 4`

ÇIKTI

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

2 < 1? Hayır → alt üçgende.

Adım 3 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
▶    } else if (i < j) return -1;    /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
9  printf("%d\n", AltUcgenEris(2, 1, 4));
10 printf("%d\n", AltUcgenEris(1, 3, 4));
11 printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

`i = 2` `j = 1` `n = 4`

ÇIKTI

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

k = 3·2/2 + 1 = 4 → ALT[4] = a_{21}.

Adım 4 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
5    } else if (i < j) return -1;    /* üst üçgen: 0 */
▶    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
9  printf("%d\n", AltUcgenEris(2, 1, 4));
10 printf("%d\n", AltUcgenEris(1, 3, 4));
11 printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

`i = 2` `j = 1` `n = 4`

ÇIKTI

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

`printf` 4 yazar.

Adım 5 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
5    } else if (i < j) return -1;    /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
▶  printf("%d\n", AltUcgenEris(2, 1, 4));
10 printf("%d\n", AltUcgenEris(1, 3, 4));
11 printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

—

ÇIKTI

```
4
```

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

a_{13} isteniyor.

Adım 6 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
5    } else if (i < j) return -1;    /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
9  printf("%d\n", AltUcgenEris(2, 1, 4));
▶  printf("%d\n", AltUcgenEris(1, 3, 4));
11 printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

`i = 1` `j = 3` `n = 4`

ÇIKTI

```
4
```

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

Sınır denetimi: indisler 0...3 aralığında.

Adım 7 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
▶    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
5    } else if (i < j) return -1;    /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
9  printf("%d\n", AltUcgenEris(2, 1, 4));
10 printf("%d\n", AltUcgenEris(1, 3, 4));
11 printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

`i = 1` `j = 3` `n = 4`

ÇIKTI

```
4
```

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

1 < 3? Evet → üst üçgen (sıfırlı bölge): **−1**.

Adım 8 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
▶    } else if (i < j) return -1;    /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
9  printf("%d\n", AltUcgenEris(2, 1, 4));
10 printf("%d\n", AltUcgenEris(1, 3, 4));
11 printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

`i = 1` `j = 3` `n = 4`

ÇIKTI

```
4
```

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

`printf` -1 yazar. Hata kodları {−1, −2} ile geçerli dönüşler {0, 1, 2, …} **kesişmez**: çağıran durumu ayırt edebilir.

Adım 9 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
5    } else if (i < j) return -1;     /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
9  printf("%d\n", AltUcgenEris(2, 1, 4));
▶  printf("%d\n", AltUcgenEris(1, 3, 4));
11 printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

—

ÇIKTI

```
4
-1
```

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

a_{40} isteniyor.

Adım 10 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
5    } else if (i < j) return -1;    /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
9  printf("%d\n", AltUcgenEris(2, 1, 4));
10 printf("%d\n", AltUcgenEris(1, 3, 4));
▶  printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

`i = 4` `j = 0` `n = 4`

ÇIKTI

```
4
-1
```

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

Sınır denetimi: **i = 4 ≥ n** → geçersiz!

Adım 11 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
▶    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
5    } else if (i < j) return -1;    /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
9  printf("%d\n", AltUcgenEris(2, 1, 4));
10 printf("%d\n", AltUcgenEris(1, 3, 4));
11 printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

`i = 4` `j = 0` `n = 4`

ÇIKTI

```
4
-1
```

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

Hata iletisi basılır ve **−2** döner.

Adım 12 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
▶      printf("Gecersiz indis\n");
▶      return -2;
5    } else if (i < j) return -1;    /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
9  printf("%d\n", AltUcgenEris(2, 1, 4));
10 printf("%d\n", AltUcgenEris(1, 3, 4));
11 printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

`i = 4` `j = 0` `n = 4`

ÇIKTI

```
4
-1
Gecersiz indis
```

Erişim: indis mi, hata kodu mu?

	0	1	2	3
0	a00	0	0	0
1	a10	a11	0	0
2	a20	a21	a22	0
3	a30	a31	a32	a33

0	1	2	3	4	5	6	7	8	9
a00	a10	a11	a20	a21	a22	a30	a31	a32	a33

ALT dizisi (n(n+1)/2 = 10 öğe)

`printf` -2 yazar.

Adım 13 / 13

```
1  int AltUcgenEris(int i, int j, int n) {
2    if (i < 0 || i >= n || j < 0 || j >= n) {
3      printf("Gecersiz indis\n");
4      return -2;
5    } else if (i < j) return -1;    /* üst üçgen: 0 */
6    else return (i + 1) * i / 2 + j;   /* geçerli */
7  }
8  /* main: */
9  printf("%d\n", AltUcgenEris(2, 1, 4));
10 printf("%d\n", AltUcgenEris(1, 3, 4));
▶  printf("%d\n", AltUcgenEris(4, 0, 4));
```

DEĞİŞKENLER

—

ÇIKTI

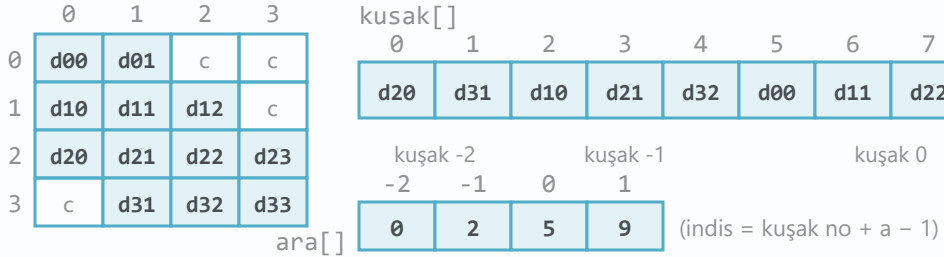
```
4
-1
Gecersiz indis
-2
```

ÇALIŞMA çözümü: dört üçgen türü ve simetrik matris

	saklanan bölge	satır i'deki öğe	k (satır öncelikli)
	j ≤ i (alt sol)	i + 1	**i(i+1)/2 + j**
	j ≥ i (üst sağ)	n − i	**i·n − i(i−1)/2 + (j − i)**
	i + j ≤ n−1 (sol üst)	n − i	**i·n − i(i−1)/2 + j**
	i + j ≥ n−1 (sağ alt)	i + 1	**i(i+1)/2 + j − (n−1−i)**

Gerçek simetrik matris ($a_{ij} = a_{ji}$): üst üçgene erişimde −1 döndürmek yerine indisler yer değiştirilir: i < j ise `k = j(j+1)/2 + i`. Formüller n = 1…8 için betikle doğrulandı.

Kuşak (band) matris: yalnız köşegen şeridi saklanır



Kuşak dışındaki öğeler kullanılmaz ya da hepsi aynı **sabit c** değerindedir. n: kenar (4), a: alt kuşak sayısı *köşegen dâhil* (3), b: üst kuşak sayısı *köşegen dâhil* (2).

```
öğe sayısı = n(a+b−1) − a(a−1)/2 −
             b(b−1)/2 = 12
```

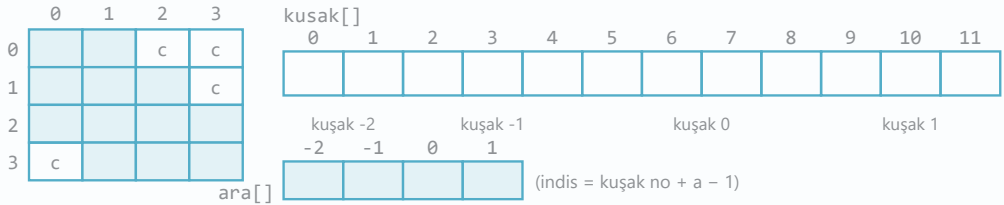
Hocadan *"Saklama sorun değil, ama hızlı erişilebiliyor mu?"*

Kuşak numarası j − i ve ARA dizisi

- Aynı kuşaktaki öğelerde **j − i** sabittir: d20 ve d31 → 0−2 = 1−3 = **−2**. Ana köşegen 0; negatifler alt, pozitifler üst kuşak.
- Geçerli kuşaklar **[1 − a, b − 1]** kapalı aralığında; dışı sabit bölge.
- **ARA** her kuşağın ilk öğesinin `kusak[]` içindeki konumunu tutar: a + b − 1 öğe. C'de negatif indis olmadığından **kuşak no + a − 1** ile kaydırılır.
- Kuşak içindeki sıra: alt kuşakta **j**, üst kuşakta **i** (kuşağın ilk öğesi 0. satır/0. sütundadır).

> **Hocadan** *"Her şey kısıtlı gibi tahlil ederek mekanizma belirliyorum."*

Kuşak matrisi kurmak (alt kuşaklar)



4·4 − 3 − 1 = **12 ≤ MAX_SIZE**. Girdi kuşak kuşak gelir: 6 10 · 3 7 11 · 1 4 8 12 · 2 5 9.

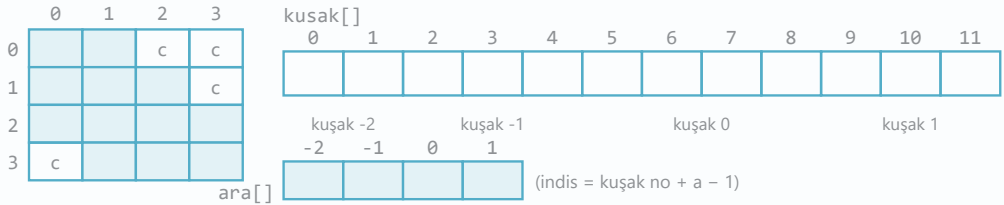
Adım 1 / 12

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
▶    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `toplamoge = 0`

Kuşak matrisi kurmak (alt kuşaklar)



Kuşak i = -2 (alt): 2 öğe.

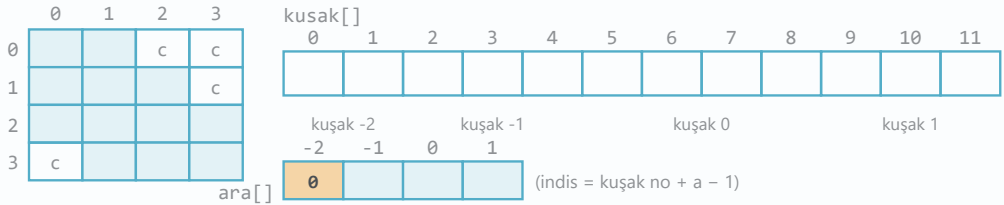
Adım 2 / 12

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
▶    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = -2` `toplamoge = 0`

Kuşak matrisi kurmak (alt kuşaklar)



`ara[-2+2] = ara[0] = 0` : kuşak `kusak[0]` 'den başlar.

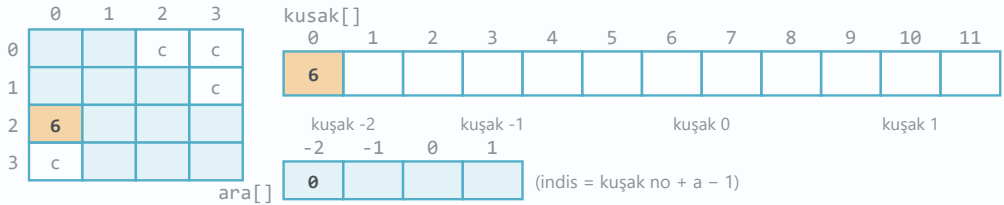
Adım 3 / 12

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
▶      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = -2` `toplamoge = 0`

Kuşak matrisi kurmak (alt kuşaklar)



`kusak[0+0]` ← 6 (d_{20}).

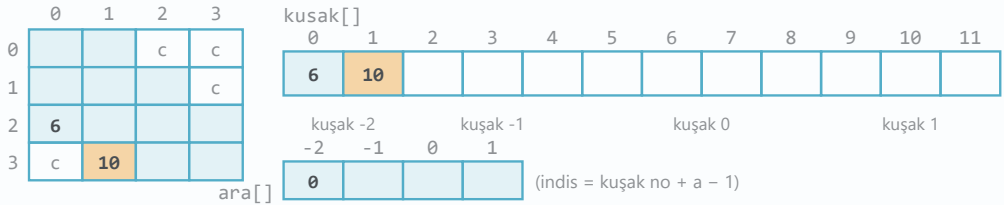
Adım 4 / 12

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = -2` `k = 0` `toplamoge = 0`

Kuşak matrisi kurmak (alt kuşaklar)



`kusak[0+1]` ← 10 (d_{31}).

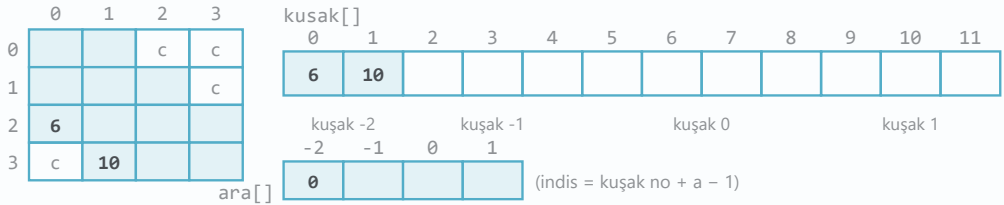
Adım 5 / 12

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = -2` `k = 1` `toplamoge = 0`

Kuşak matrisi kurmak (alt kuşaklar)



toplamoge += 2 → 2: sonraki kuşağın başı.

Adım 6 / 12

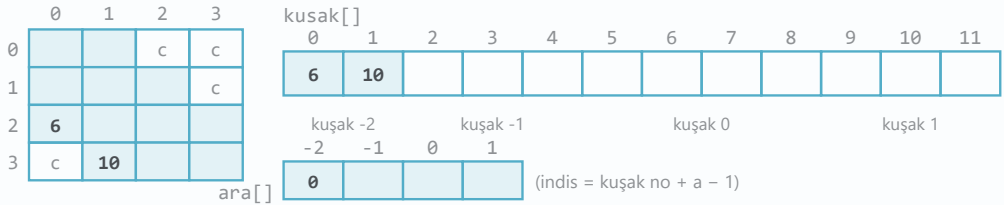
```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
▶      toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = -2`

`toplamoge = 2`

Kuşak matrisi kurmak (alt kuşaklar)



Kuşak i = -1 (alt): 3 öğe.

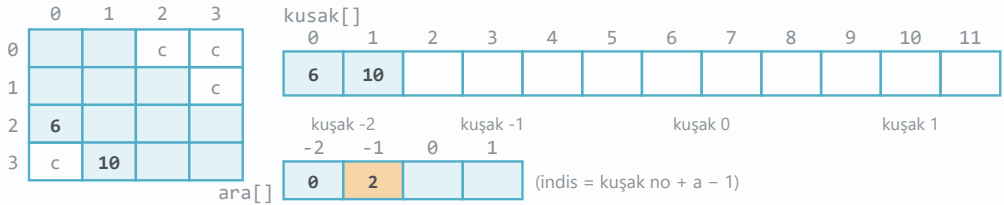
Adım 7 / 12

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
▶    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = -1` `toplamoge = 2`

Kuşak matrisi kurmak (alt kuşaklar)



`ara[-1+2] = ara[1] = 2` : kuşak `kusak[2]` 'den başlar.

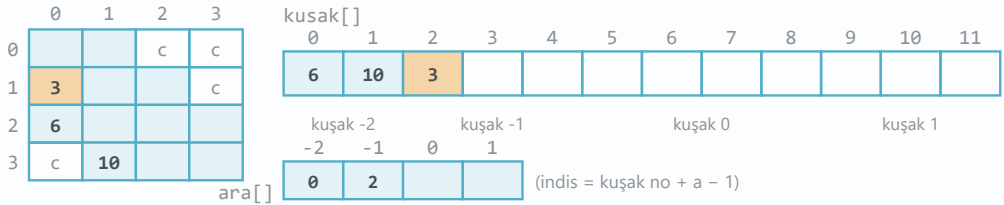
Adım 8 / 12

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
▶      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = -1` `toplamoge = 2`

Kuşak matrisi kurmak (alt kuşaklar)



`kusak[2+0]` ← 3 (d_{10}).

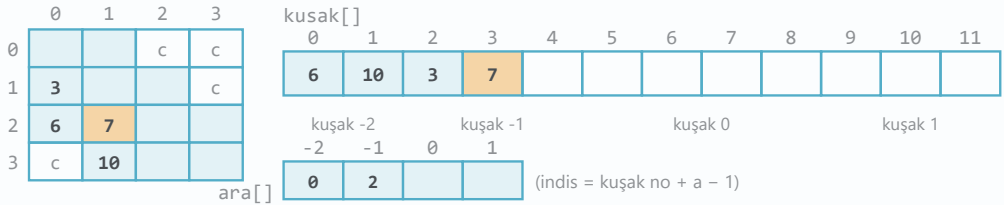
Adım 9 / 12

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = -1` `k = 0` `toplamoge = 2`

Kuşak matrisi kurmak (alt kuşaklar)



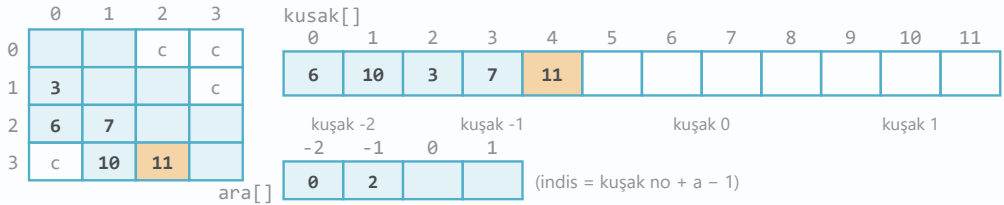
`kusak[2+1]` ← 7 (d_{21}).


```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = -1` `k = 1` `toplamoge = 2`

Kuşak matrisi kurmak (alt kuşaklar)



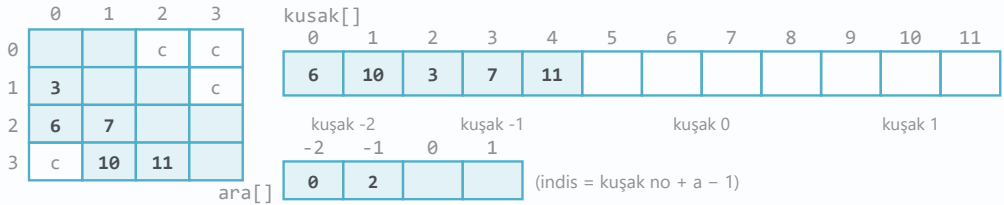
`kusak[2+2]` ← 11 (d_{32}).


```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = -1` `k = 2` `toplamoge = 2`

Kuşak matrisi kurmak (alt kuşaklar)



toplamoge += 3 → 5: sonraki kuşağın başı.

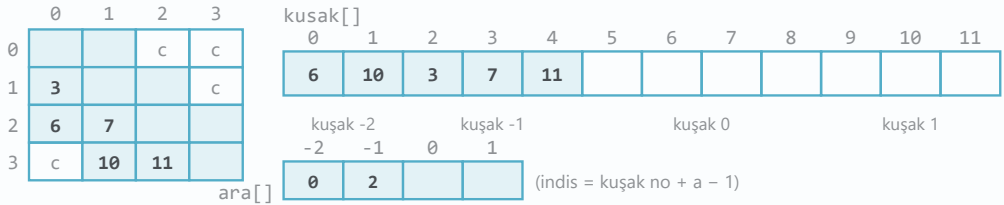

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
▶      toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = -1`

`toplamoge = 5`

Kuşak matrisi kurmak (devam: köşegen ve üst)



Kuşak i = 0 (ana köşegen): 4 öğe.

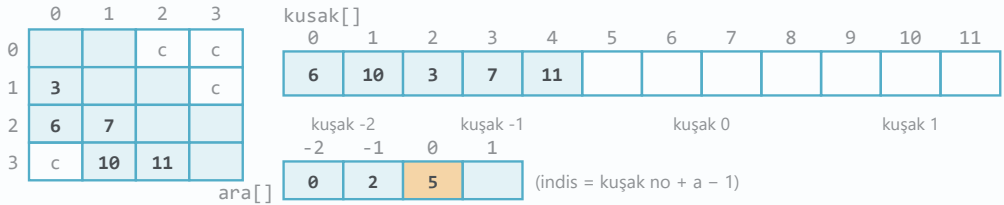
Adım 1 / 13

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
▶    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 0` `toplamoge = 5`

Kuşak matrisi kurmak (devam: köşegen ve üst)



`ara[0+2] = ara[2] = 5` : kuşak `kusak[5]` 'den başlar.

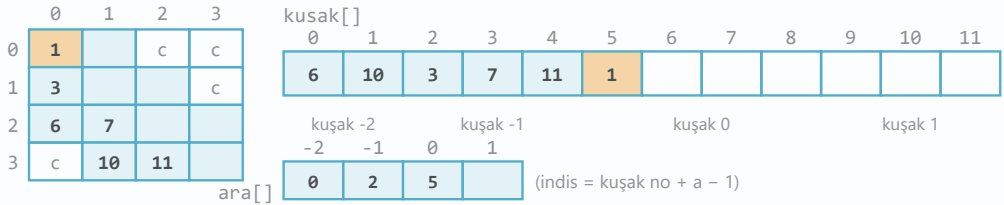
Adım 2 / 13

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
▶      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 0` `toplamoge = 5`

Kuşak matrisi kurmak (devam: köşegen ve üst)



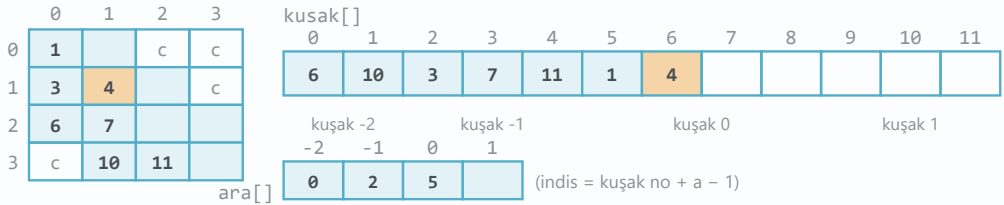
`kusak[5+0]` ← 1 (d_{00}).


```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 0` `k = 0` `toplamoge = 5`

Kuşak matrisi kurmak (devam: köşegen ve üst)



`kusak[5+1]` ← 4 (d_{11}).

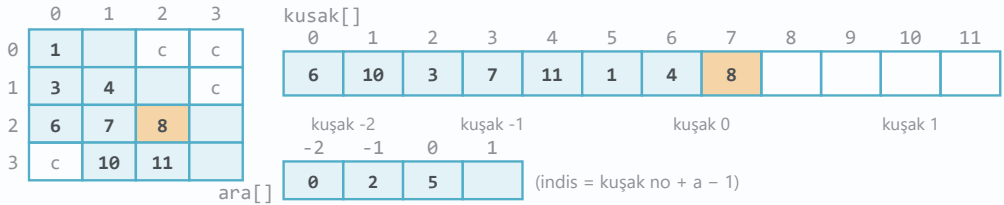
Adım 4 / 13

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 0` `k = 1` `toplamoge = 5`

Kuşak matrisi kurmak (devam: köşegen ve üst)



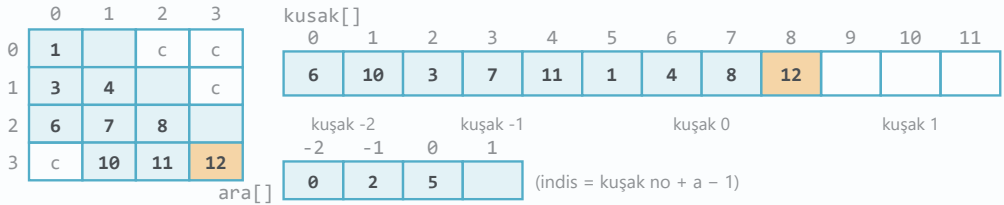
`kusak[5+2]` ← 8 (d_{22}).


```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 0` `k = 2` `toplamoge = 5`

Kuşak matrisi kurmak (devam: köşegen ve üst)



`kusak[5+3]` ← 12 (d_{33}).

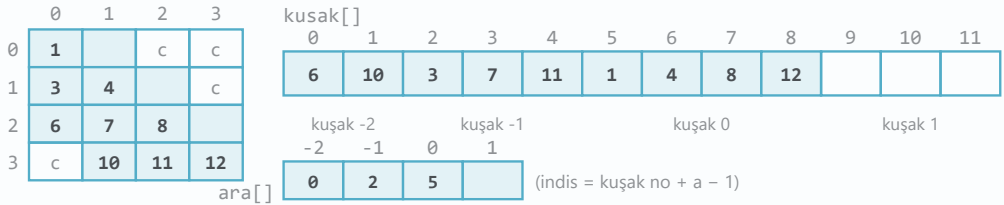
Adım 6 / 13

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 0` `k = 3` `toplamoge = 5`

Kuşak matrisi kurmak (devam: köşegen ve üst)



toplamoge += 4 → 9: sonraki kuşağın başı.

Adım 7 / 13

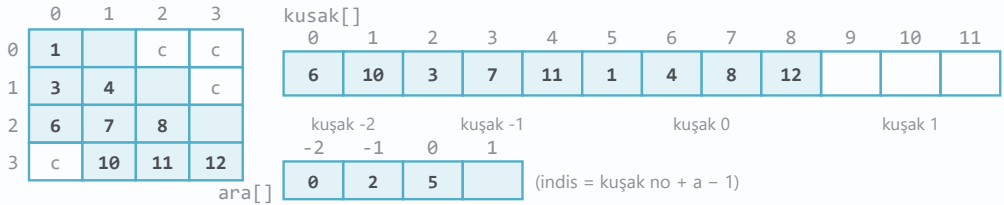
```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
▶      toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 0`

`toplamoge = 9`

Kuşak matrisi kurmak (devam: köşegen ve üst)



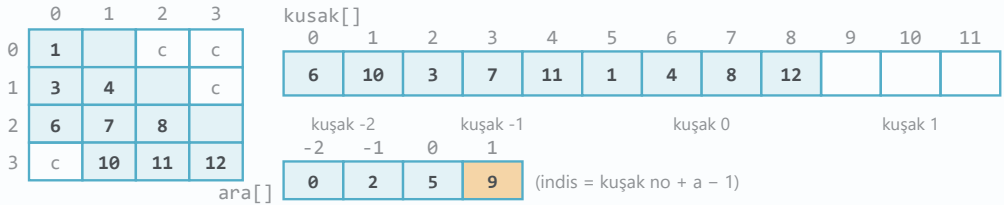
Kuşak i = 1 (üst): 3 öğe.


```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
▶    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 1` `toplamoge = 9`

Kuşak matrisi kurmak (devam: köşegen ve üst)



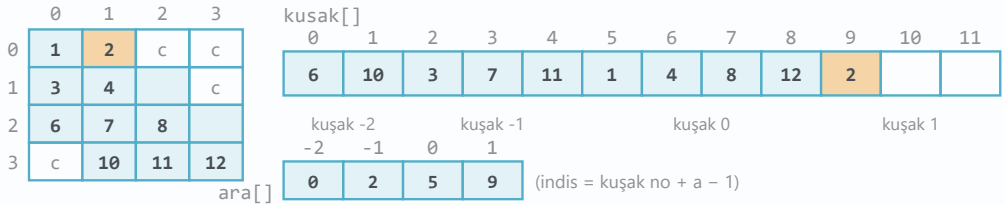
`ara[1+2] = ara[3] = 9` : kuşak `kusak[9]` 'den başlar.


```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
▶      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 1` `toplamoge = 9`

Kuşak matrisi kurmak (devam: köşegen ve üst)



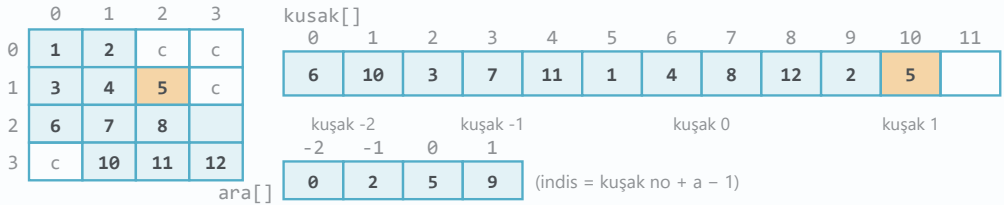
`kusak[9+0]` ← 2 (d_{01}).


```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 1` `k = 0` `toplamoge = 9`

Kuşak matrisi kurmak (devam: köşegen ve üst)



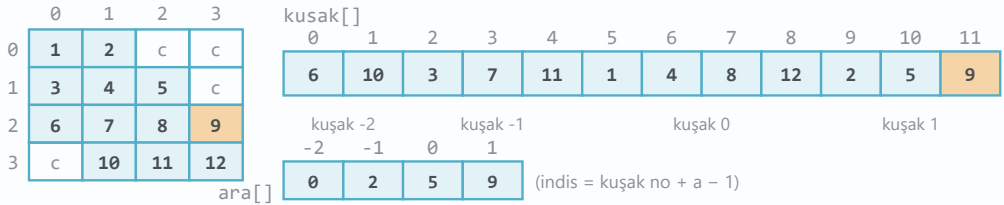
`kusak[9+1]` ← 5 (d_{12}).


```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 1` `k = 1` `toplamoge = 9`

Kuşak matrisi kurmak (devam: köşegen ve üst)



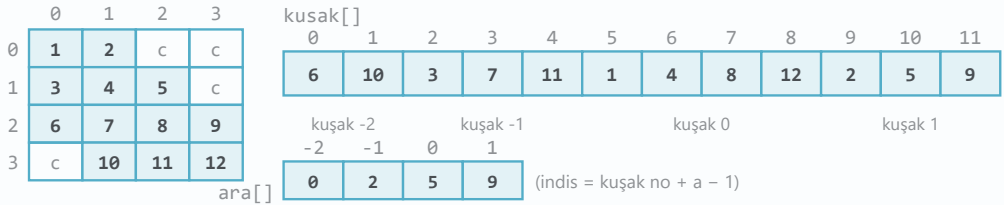
`kusak[9+2]` ← 9 (d_{23}).


```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
▶        scanf("%d", &kusak[ara[i + a - 1] + k]);
11     toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 1` `k = 2` `toplamoge = 9`

Kuşak matrisi kurmak (devam: köşegen ve üst)



toplamoge += 3 → 12: sonraki kuşağın başı.

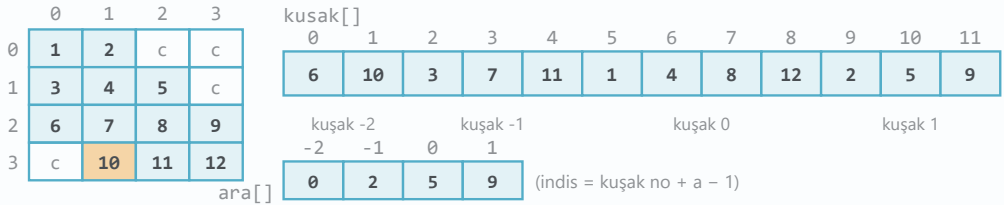

```
1  void KusakMatrisKur(int kusak[], int ara[],
2                      int n, int a, int b) {
3    int i, k, toplamoge = 0;   /* önceki kuşaklar */
4    if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
5      printf("Bellek yetersiz!\n"); exit(-1);
6    }
7    for (i = 1 - a; i <= b - 1; i++) {  /* kuşak kuşak */
8      ara[i + a - 1] = toplamoge;     /* kuşak başı */
9      for (k = 0; k <= n - abs(i) - 1; k++)
10       scanf("%d", &kusak[ara[i + a - 1] + k]);
▶      toplamoge += n - abs(i);
12   }
13 }
```

DEĞİŞKENLER

`n = 4` `a = 3` `b = 2` `i = 1`

`toplamoge = 12`

Kuşak matriste O(1) erişim



d_{31} isteniyor; indisler geçerli. Kuşak no **j − i = -2**.

Adım 1 / 12

```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
▶    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
5    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                 /* geçerli kuşak */
7        return ara[j - i + a - 1] + i;
8      else return -1;                /* üst sabit bölge */
9    } else if (i - j < a)            /* köşegen / alt */
10     return ara[j - i + a - 1] + j;
11   else
12     return -2;                     /* alt sabit bölge */
13 }
```

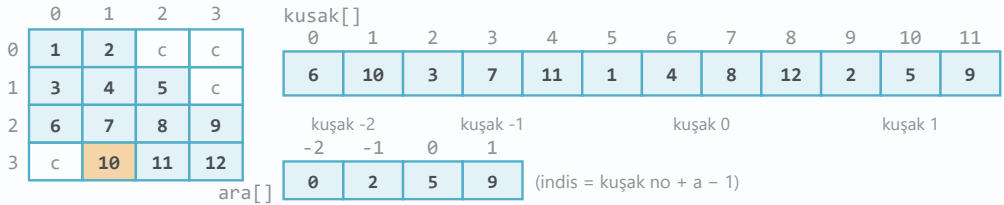
DEĞİŞKENLER

`i = 3` `j = 1` `j−i = −2`

ÇAĞRI YIĞINI

`KusakMatrisEris(3, 1, 4, 3, 2, ara)`

Kuşak matriste O(1) erişim



i − j = 2 < a = 3? Evet → alt kuşak.

Adım 2 / 12

```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
5    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                 /* geçerli kuşak */
7        return ara[j - i + a - 1] + i;
8      else return -1;                /* üst sabit bölge */
▶    } else if (i - j < a)            /* köşegen / alt */
10     return ara[j - i + a - 1] + j;
11   else
12     return -2;                     /* alt sabit bölge */
13 }
```

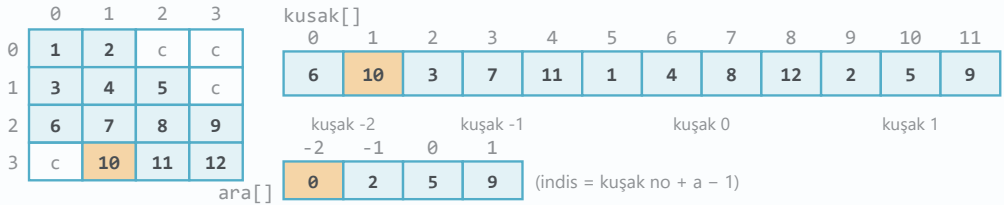
DEĞİŞKENLER

`i = 3` `j = 1` `j−i = −2`

ÇAĞRI YIĞINI

`KusakMatrisEris(3, 1, 4, 3, 2, ara)`

Kuşak matriste O(1) erişim



ara[-2+2] + j = 0 + 1 = **1** → kusak[1] = 10.

Adım 3 / 12

```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
5    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                 /* geçerli kuşak */
7        return ara[j - i + a - 1] + i;
8      else return -1;                /* üst sabit bölge */
9    } else if (i - j < a)            /* köşegen / alt */
▶      return ara[j - i + a - 1] + j;
11   else
12     return -2;                     /* alt sabit bölge */
13 }
```

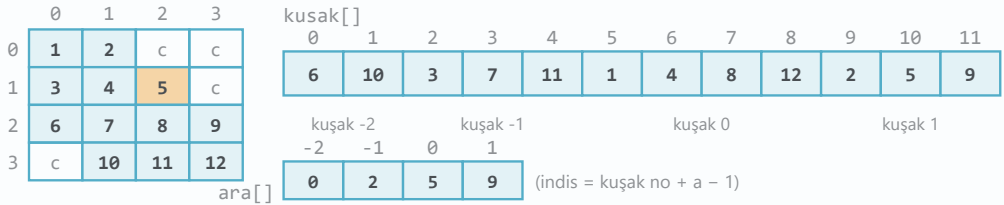
DEĞİŞKENLER

`i = 3` `j = 1` `j−i = −2` `ret = 1`

ÇAĞRI YIĞINI

`KusakMatrisEris(3, 1, 4, 3, 2, ara)`

Kuşak matriste O(1) erişim



d_{12} isteniyor; indisler geçerli. Kuşak no **j − i = 1**.


```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
▶    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
5    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                /* geçerli kuşak */
7        return ara[j - i + a - 1] + i;
8      else return -1;               /* üst sabit bölge */
9    } else if (i - j < a)           /* köşegen / alt */
10     return ara[j - i + a - 1] + j;
11   else
12     return -2;                    /* alt sabit bölge */
13 }
```

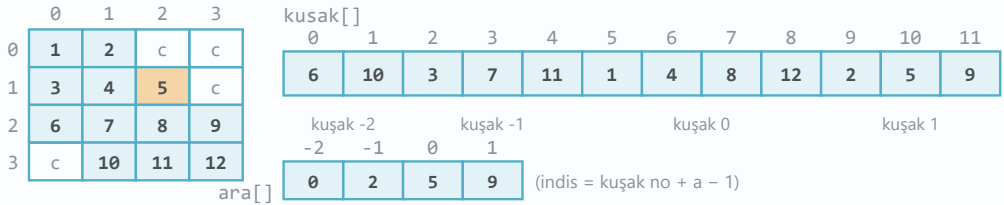
DEĞİŞKENLER

`i = 1` `j = 2` `j−i = 1`

ÇAĞRI YIĞINI

`KusakMatrisEris(1, 2, 4, 3, 2, ara)`

Kuşak matriste O(1) erişim



1 ≥ 1 → üst üçgen.

Adım 5 / 12

```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
▶    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                  /* geçerli kuşak */
7        return ara[j - i + a - 1] + i;
8      else return -1;                 /* üst sabit bölge */
9    } else if (i - j < a)             /* köşegen / alt */
10     return ara[j - i + a - 1] + j;
11   else
12     return -2;                      /* alt sabit bölge */
13 }
```

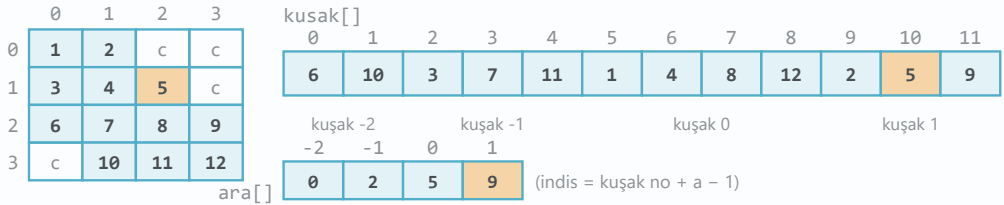
DEĞİŞKENLER

`i = 1` `j = 2` `j−i = 1`

ÇAĞRI YIĞINI

`KusakMatrisEris(1, 2, 4, 3, 2, ara)`

Kuşak matriste O(1) erişim



1 < b = 2 → geçerli üst kuşak: ara[1+2] + i = 9 + 1 = **10**.


```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
5    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                 /* geçerli kuşak */
▶        return ara[j - i + a - 1] + i;
8      else return -1;                /* üst sabit bölge */
9    } else if (i - j < a)            /* köşegen / alt */
10     return ara[j - i + a - 1] + j;
11   else
12     return -2;                     /* alt sabit bölge */
13 }
```

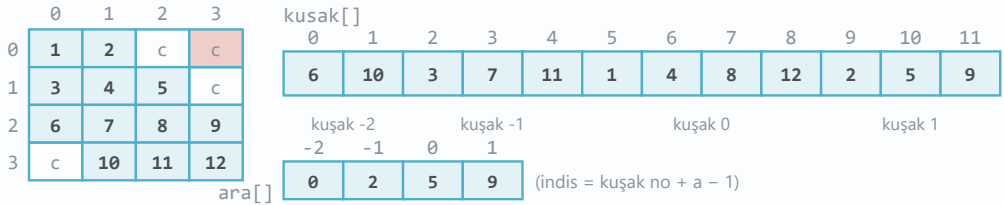
DEĞİŞKENLER

`i = 1` `j = 2` `j−i = 1` `ret = 10`

ÇAĞRI YIĞINI

`KusakMatrisEris(1, 2, 4, 3, 2, ara)`

Kuşak matriste O(1) erişim



d_{03} isteniyor; indisler geçerli. Kuşak no **j − i = 3**.

Adım 7 / 12

```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
▶    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
5    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                 /* geçerli kuşak */
7        return ara[j - i + a - 1] + i;
8      else return -1;                /* üst sabit bölge */
9    } else if (i - j < a)            /* köşegen / alt */
10     return ara[j - i + a - 1] + j;
11   else
12     return -2;                     /* alt sabit bölge */
13 }
```

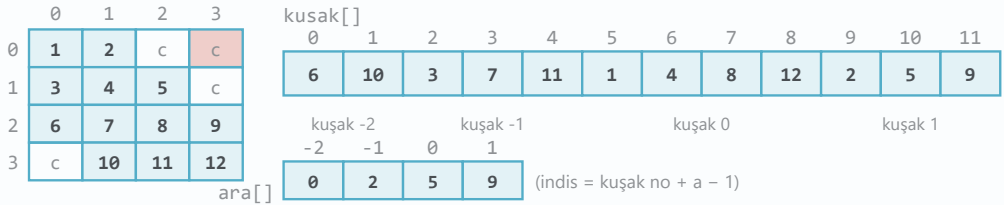
DEĞİŞKENLER

`i = 0` `j = 3` `j−i = 3`

ÇAĞRI YIĞINI

`KusakMatrisEris(0, 3, 4, 3, 2, ara)`

Kuşak matriste O(1) erişim



3 ≥ 1 → üst üçgen.


```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
▶    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                  /* geçerli kuşak */
7        return ara[j - i + a - 1] + i;
8      else return -1;                 /* üst sabit bölge */
9    } else if (i - j < a)             /* köşegen / alt */
10     return ara[j - i + a - 1] + j;
11   else
12     return -2;                      /* alt sabit bölge */
13 }
```

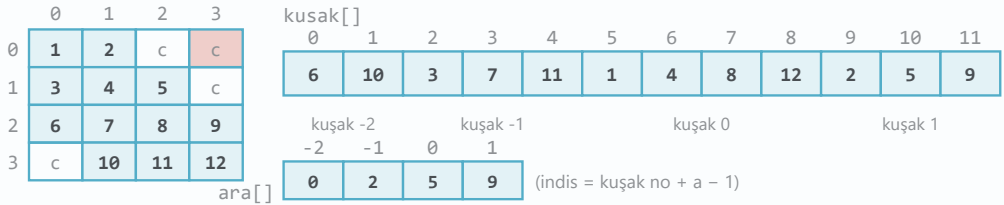
DEĞİŞKENLER

`i = 0` `j = 3` `j−i = 3`

ÇAĞRI YIĞINI

`KusakMatrisEris(0, 3, 4, 3, 2, ara)`

Kuşak matriste O(1) erişim



3 ≥ b = 2 → kuşak dışı, üst sabit bölge: **−1** (değer c).


```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
5    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                 /* geçerli kuşak */
7        return ara[j - i + a - 1] + i;
▶      else return -1;                /* üst sabit bölge */
9    } else if (i - j < a)            /* köşegen / alt */
10     return ara[j - i + a - 1] + j;
11   else
12     return -2;                     /* alt sabit bölge */
13 }
```

DEĞİŞKENLER

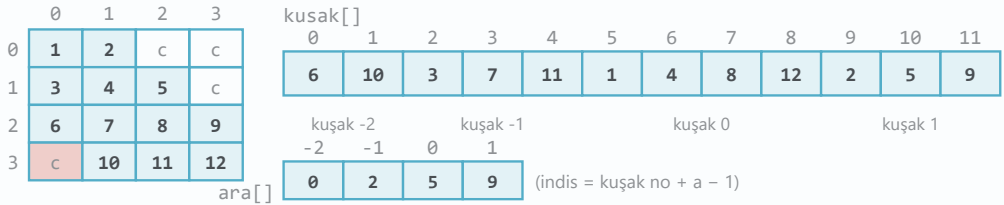
`i = 0` `j = 3` `j−i = 3`

`ret = −1`

ÇAĞRI YIĞINI

`KusakMatrisEris(0, 3, 4, 3, 2, ara)`

Kuşak matriste O(1) erişim



d_{30} isteniyor; indisler geçerli. Kuşak no **j − i = -3**.


```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
▶    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
5    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                /* geçerli kuşak */
7        return ara[j - i + a - 1] + i;
8      else return -1;               /* üst sabit bölge */
9    } else if (i - j < a)           /* köşegen / alt */
10     return ara[j - i + a - 1] + j;
11   else
12     return -2;                    /* alt sabit bölge */
13 }
```

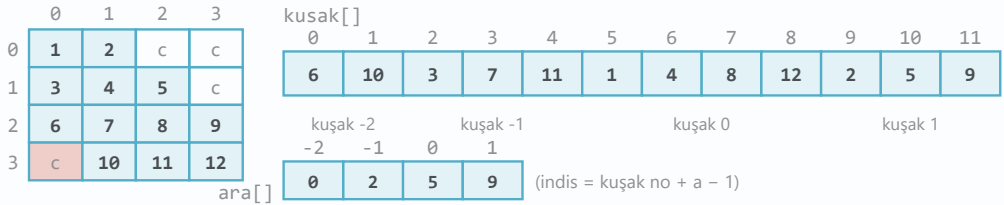
DEĞİŞKENLER

`i = 3` `j = 0` `j−i = -3`

ÇAĞRI YIĞINI

`KusakMatrisEris(3, 0, 4, 3, 2, ara)`

Kuşak matriste O(1) erişim



i − j = 3 < a = 3? Hayır → kuşak dışı.


```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
5    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                 /* geçerli kuşak */
7        return ara[j - i + a - 1] + i;
8      else return -1;                /* üst sabit bölge */
▶    } else if (i - j < a)            /* köşegen / alt */
10     return ara[j - i + a - 1] + j;
11   else
12     return -2;                     /* alt sabit bölge */
13 }
```

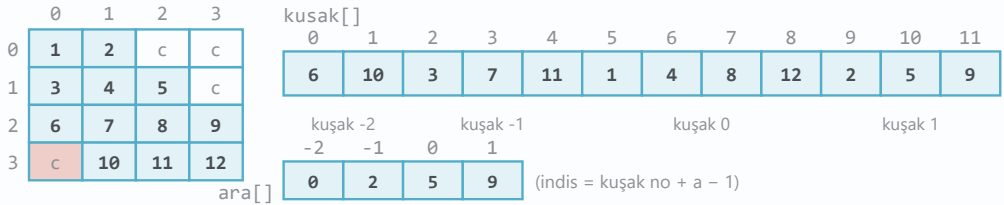
DEĞİŞKENLER

`i = 3` `j = 0` `j−i = −3`

ÇAĞRI YIĞINI

`KusakMatrisEris(3, 0, 4, 3, 2, ara)`

Kuşak matriste O(1) erişim



Alt sabit bölge: **−2**. Her durumda sabit sayıda işlem → **O(1)**.

Adım 12 / 12

```
1  int KusakMatrisEris(int i, int j, int n, int a,
2                      int b, int ara[]) {
3    if (i < 0 || j < 0 || i >= n || j >= n) {
4      printf("Indis tasmasi\n"); return -3;
5    } else if (j - i >= 1) {          /* üst üçgen */
6      if (j - i < b)                 /* geçerli kuşak */
7        return ara[j - i + a - 1] + i;
8      else return -1;                /* üst sabit bölge */
9    } else if (i - j < a)            /* köşegen / alt */
10     return ara[j - i + a - 1] + j;
11   else
▶      return -2;                     /* alt sabit bölge */
13 }
```

DEĞİŞKENLER

`i = 3` `j = 0` `j−i = −3`

`ret = −2`

ÇAĞRI YIĞINI

`KusakMatrisEris(3, 0, 4, 3, 2, ara)`

ARA dizisi mi, formül mü? (ÖDEV 1)

`KusakMatrisEris` girdiye bağlı hiçbir döngü içermez → **O(1)**. Kuşak başlarını döngüyle saymak karmaşıklığı artırırdı; **ARA** bunu önler ama bellek harcar ve programı "dinamik olmaktan uzaklaştırır".

> **Hocadan** *"Eğer performansı artıracaksa makul ölçüde olmak kaydıyla fazladan bellek kullanmakta sakınca yok."*

ÖDEV 1: ARA dizisinden yardım almadan kuşak başını hesaplayın. (Örnek çözüm yanda.)

```
/* t: kuşak no (1-a..b-1); ara[t+a-1] ile aynı */
int kusak_basi(int t, int n, int a) {
  if (t <= 0)                       /* alt kuşaklar */
    return (t + a - 1) * n
           - (a * (a - 1) + t * (1 - t)) / 2;
  return (a - 1) * n - a * (a - 1) / 2  /* t=0 başı */
         + t * n - t * (t - 1) / 2;
}
```

n = 4, a = 3: t = −2, −1, 0, 1 → **0, 2, 5, 9** (ARA ile aynı). Betikle n ≤ 11 için tüm a, b denendi. Erişim: `kusak_basi(j-i, n, a) + (j > i ? i : j)`.

`if (ifade1 && ifade2)` güvenli mi?

Sık duyulan bir kanı: "Derleyicinin sağdan sola mı, soldan sağa mı taradığını bilemeyiz; bu yüzden şu iki eşdeğer ifadeden ikincisi daha güvenlidir."

```
/* 1 */ if (i < n && a[i] != 0) ...

/* 2 */ if (i < n)
          if (a[i] != 0) ...
```

Doğrusu: C standardı `&&` ve `||` için **soldan sağa** değerlendirmeyi ve **kısa devreyi** garanti eder (C99 §6.5.13–14): sol taraf yanlışsa sağ taraf hiç hesaplanmaz. Yani iki biçim **aynı derecede güvenlidir**; 1. biçim yaygın ve doğru bir kalıptır.

Asıl tuzak

`&` (bit düzeyi VE) kısa devre yapmaz; `f(x) + g(x)` gibi işlenenlerin sırası ise **belirtilmemiştir**.

struct: farklı türde alanları tek kayıtta toplamak

```c
typedef struct {
  char ad[10];
  int yas;
  float maas;
} insan;

insan k1, k2;
strcpy(k1.ad, "Ali");
k1.yas = 21;
k1.maas = 3500.0f;
k2 = k1;          /* alan alan kopyalanır */
/* if (k1 == k2) -> DERLENMEZ */
```

Nokta ve ok

`k1.yas` değişkenin alanı; `p→yas` ≡ `(*p).yas` (p: `insan *`).

Atama var, == yok

Yapılar bütün olarak atanabilir ama karşılaştırılamaz; alan alan karşılaştıran bir fonksiyon yazılır.

typedef

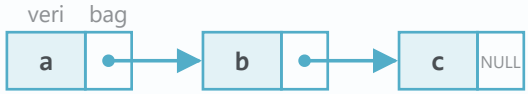
`typedef struct {…} insan;` ile her seferinde `struct` yazmaktan kurtuluruz.

Yapı içinde yapı · kendine başvuran yapı

```c
typedef struct {
  int gun, ay, yil;
} tarih;

typedef struct {
  char ad[10];
  tarih dogum;          /* iç içe yapı */
} kisi;
/* k.dogum.yil = 2004; */

typedef struct liste {
  char veri;
  struct liste *bag;    /* kendine başvuru */
} liste;
```



Kendine başvuran yapılar bağlı listelerin temelidir.

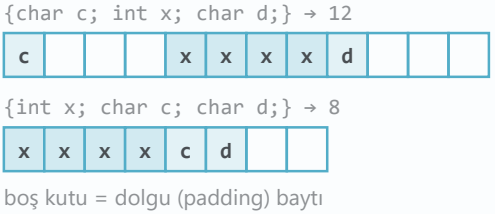
Bir yapı kendi türünden alan **içeremez** (sonsuz boyut), ama kendi türüne **işaretçi** içerebilir.

union: alanlar aynı belleği paylaşır

```c
typedef struct {
  enum { KADIN, ERKEK } tur;    /* etiket */
  union {
    int cocuk_sayisi;           /* KADIN ise */
    int sakal;                  /* ERKEK ise */
  } u;
} cinsiyet;

cinsiyet c;
c.tur = ERKEK;
c.u.sakal = 0;    /* yalnız etiketle uyumlu alanı oku */
```

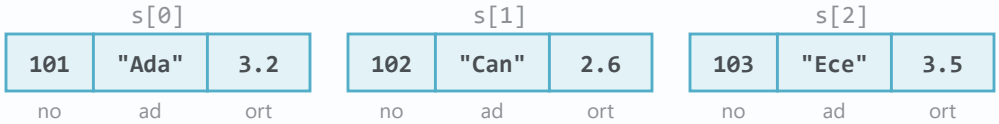
union boyutu en büyük alan kadardır (+ hizalama). Hangi alanın geçerli olduğunu **etiket** alanıyla izlemek programcının işidir.



Alan sırası bellek düzenini değiştirir (derleyiciye bağlı; tipik 32/64 bit değerler)

YAPILAR · KODU ADIM ADIM ÇALIŞTIR

Yapı dizisi üzerinde gezinme



Yapı dizisi bellekte kayıt kayıt ardışık durur; her kayıt `sizeof(Ogrenci)` bayt.

Adım 1 / 9

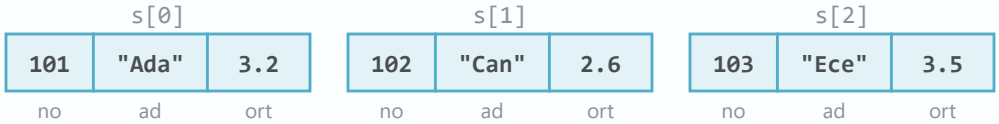
```
1  typedef struct {
2    int no;
3    char ad[16];
4    double ort;
5  } Ogrenci;
6
7  double sinif_ort(const Ogrenci s[], int n) {
▶    double top = 0.0;
9    int i;
10   for (i = 0; i < n; i++)
11     top += s[i].ort;               /* . ile alan */
12   return n > 0 ? top / n : 0.0;    /* boş sınıf */
13 }
```

DEĞİŞKENLER

`n = 3` `top = 0.0`

`const Ogrenci s[]` : dizi kopyalanmaz; `const` fonksiyonun kayıtları değiştirmeyeceğini belgeler.

Yapı dizisi üzerinde gezinme



0 < 3 → 0. kayıt.

Adım 2 / 9

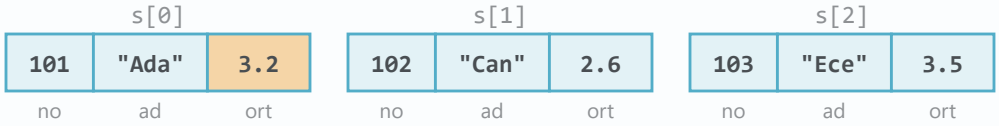
```
 1 typedef struct {
 2   int no;
 3   char ad[16];
 4   double ort;
 5 } Ogrenci;
 6
 7 double sinif_ort(const Ogrenci s[], int n) {
 8   double top = 0.0;
 9   int i;
▶   for (i = 0; i < n; i++)
11     top += s[i].ort;              /* . ile alan */
12   return n > 0 ? top / n : 0.0;   /* boş sınıf */
13 }
```

DEĞİŞKENLER

`n = 3` `top = 0.0` `i = 0`

`const Ogrenci s[]` : dizi kopyalanmaz; `const` fonksiyonun kayıtları değiştirmeyeceğini belgeler.

Yapı dizisi üzerinde gezinme



`s[0].ort` = 3.2 eklenir → top = **3.2**.

Adım 3 / 9

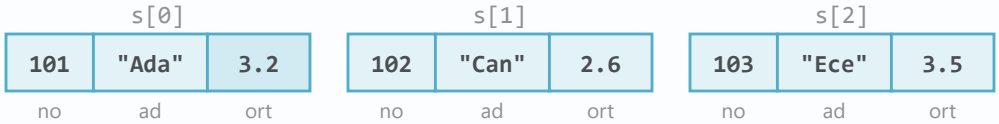
```
 1 typedef struct {
 2   int no;
 3   char ad[16];
 4   double ort;
 5 } Ogrenci;
 6
 7 double sinif_ort(const Ogrenci s[], int n) {
 8   double top = 0.0;
 9   int i;
10   for (i = 0; i < n; i++)
▶      top += s[i].ort;              /* . ile alan */
12   return n > 0 ? top / n : 0.0;    /* boş sınıf */
13 }
```

DEĞİŞKENLER

`n = 3` `top = 3.2` `i = 0`

`const Ogrenci s[]` : dizi kopyalanmaz; `const` fonksiyonun kayıtları değiştirmeyeceğini belgeler.

Yapı dizisi üzerinde gezinme



1 < 3 → 1. kayıt.

Adım 4 / 9

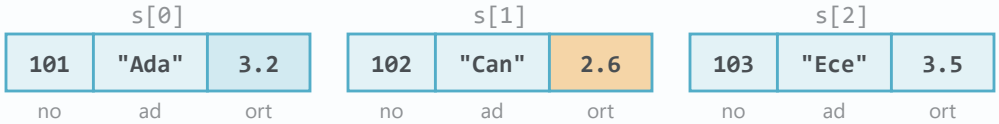
```
 1 typedef struct {
 2   int no;
 3   char ad[16];
 4   double ort;
 5 } Ogrenci;
 6
 7 double sinif_ort(const Ogrenci s[], int n) {
 8   double top = 0.0;
 9   int i;
▶   for (i = 0; i < n; i++)
11     top += s[i].ort;               /* . ile alan */
12   return n > 0 ? top / n : 0.0;    /* boş sınıf */
13 }
```

DEĞİŞKENLER

`n = 3` `top = 3.2` `i = 1`

`const Ogrenci s[]` : dizi kopyalanmaz; `const` fonksiyonun kayıtları değiştirmeyeceğini belgeler.

Yapı dizisi üzerinde gezinme



`s[1].ort` = 2.6 eklenir → top = **5.8**.

Adım 5 / 9

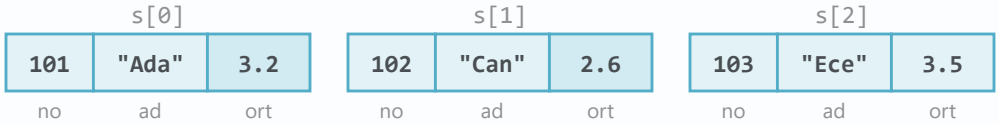
```
 1 typedef struct {
 2   int no;
 3   char ad[16];
 4   double ort;
 5 } Ogrenci;
 6
 7 double sinif_ort(const Ogrenci s[], int n) {
 8   double top = 0.0;
 9   int i;
10   for (i = 0; i < n; i++)
▶      top += s[i].ort;              /* . ile alan */
12   return n > 0 ? top / n : 0.0;    /* boş sınıf */
13 }
```

DEĞİŞKENLER

`n = 3` `top = 5.8` `i = 1`

`const Ogrenci s[]` : dizi kopyalanmaz; `const` fonksiyonun kayıtları değiştirmeyeceğini belgeler.

Yapı dizisi üzerinde gezinme



2 < 3 → 2. kayıt.

Adım 6 / 9

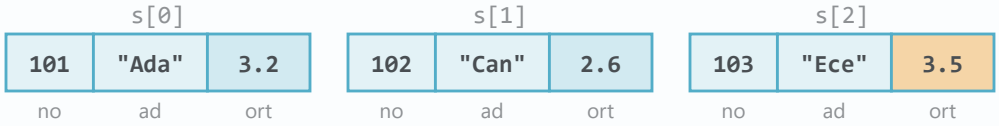
```
 1 typedef struct {
 2   int no;
 3   char ad[16];
 4   double ort;
 5 } Ogrenci;
 6
 7 double sinif_ort(const Ogrenci s[], int n) {
 8   double top = 0.0;
 9   int i;
▶   for (i = 0; i < n; i++)
11     top += s[i].ort;               /* . ile alan */
12   return n > 0 ? top / n : 0.0;    /* boş sınıf */
13 }
```

DEĞİŞKENLER

`n = 3` `top = 5.8` `i = 2`

`const Ogrenci s[]` : dizi kopyalanmaz; `const` fonksiyonun kayıtları değiştirmeyeceğini belgeler.

Yapı dizisi üzerinde gezinme



`s[2].ort` = 3.5 eklenir → top = **9.3**.

Adım 7 / 9

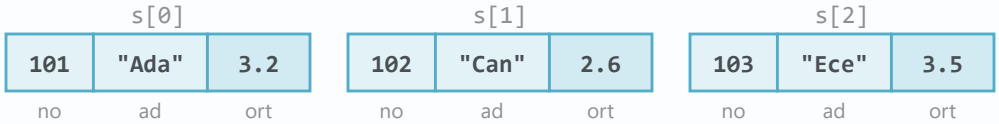
```
 1 typedef struct {
 2   int no;
 3   char ad[16];
 4   double ort;
 5 } Ogrenci;
 6
 7 double sinif_ort(const Ogrenci s[], int n) {
 8   double top = 0.0;
 9   int i;
10   for (i = 0; i < n; i++)
▶      top += s[i].ort;              /* . ile alan */
12   return n > 0 ? top / n : 0.0;    /* boş sınıf */
13 }
```

DEĞİŞKENLER

`n = 3` `top = 9.3` `i = 2`

`const Ogrenci s[]` : dizi kopyalanmaz; `const` fonksiyonun kayıtları değiştirmeyeceğini belgeler.

Yapı dizisi üzerinde gezinme



3 < 3 yanlış → döngü biter.

Adım 8 / 9

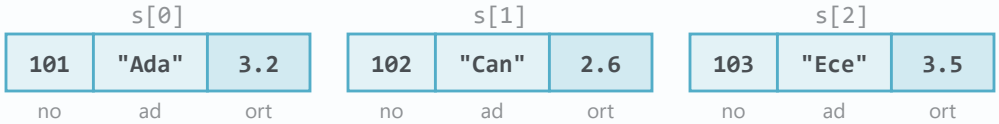
```
 1 typedef struct {
 2   int no;
 3   char ad[16];
 4   double ort;
 5 } Ogrenci;
 6
 7 double sinif_ort(const Ogrenci s[], int n) {
 8   double top = 0.0;
 9   int i;
▶   for (i = 0; i < n; i++)
11     top += s[i].ort;               /* . ile alan */
12   return n > 0 ? top / n : 0.0;    /* boş sınıf */
13 }
```

DEĞİŞKENLER

`n = 3` `top = 9.3` `i = 3`

`const Ogrenci s[]` : dizi kopyalanmaz; `const` fonksiyonun kayıtları değiştirmeyeceğini belgeler.

Yapı dizisi üzerinde gezinme



n > 0 → **9.3 / 3 = 3.10** döner. n = 0 olsaydı sıfıra bölme yerine 0.0.

Adım 9 / 9

```
1  typedef struct {
2    int no;
3    char ad[16];
4    double ort;
5  } Ogrenci;
6
7  double sinif_ort(const Ogrenci s[], int n) {
8    double top = 0.0;
9    int i;
10   for (i = 0; i < n; i++)
11     top += s[i].ort;              /* . ile alan */
▶    return n > 0 ? top / n : 0.0;    /* boş sınıf */
13 }
```

DEĞİŞKENLER

`n = 3` `top = 9.3`

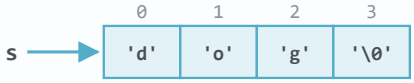
`const Ogrenci s[]` : dizi kopyalanmaz; `const` fonksiyonun kayıtları değiştirmeyeceğini belgeler.

C'de dizgi: `'\0'` ile biten char dizisi

```c
char s[] = "dog";       /* 4 bayt: d o g \0 */
char t[10];
char *p = "dog";        /* salt okunur sabit */

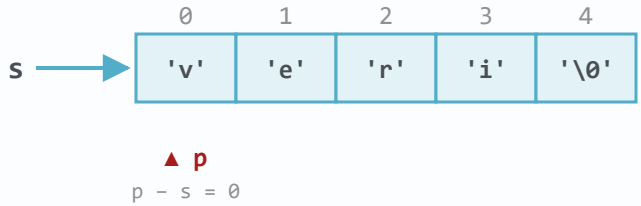
strlen(s)   /* 3 : \0'a kadar */
sizeof s    /* 4 : dizinin boyutu */

strcpy(t, s);                 /* t yeterince büyük mü? */
snprintf(t, sizeof t, "%s!", s);   /* güvenli */
if (strcmp(s, "dog") == 0) ...     /* == değil! */
```



fonksiyon	iş
`strlen`	uzunluk (\0 hariç)
`strcpy` / `strncpy`	kopyala
`strcat`	sona ekle
`strcmp`	sözlük sırası karşılaştır
`strchr` / `strstr`	karakter / alt dizgi ara

strlen nasıl çalışır?



`p`, `s` ile aynı yeri (ilk karakteri) gösterir.

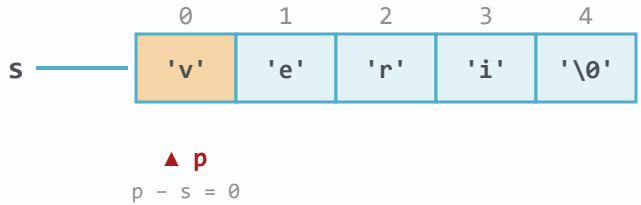
Adım 1 / 11

```
1 size_t uzunluk(const char *s) {
▶   const char *p = s;
3   while (*p != '\0')     /* sonlandırıcıya dek */
4     p++;
5   return (size_t)(p - s);  /* işaretçi farkı */
6 }
7
8 /* uzunluk("veri") */
```

DEĞİŞKENLER

`s = &"veri"[0]` `p = s`

strlen nasıl çalışır?



`*p = 'v'` ≠ '\0' → devam.

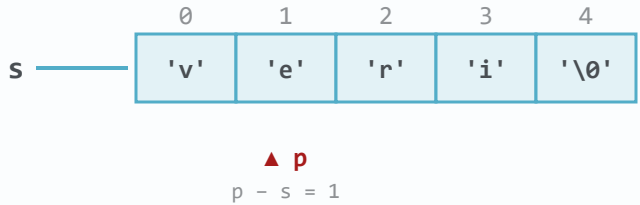
Adım 2 / 11

```
1 size_t uzunluk(const char *s) {
2   const char *p = s;
▶   while (*p != '\0')     /* sonlandırıcıya dek */
4     p++;
5   return (size_t)(p - s);  /* işaretçi farkı */
6 }
7
8 /* uzunluk("veri") */
```

DEĞİŞKENLER

`s = &"veri"[0]` `p = s+0` `*p = 'v'`

strlen nasıl çalışır?



p++: bir karakter (1 bayt) ileri.

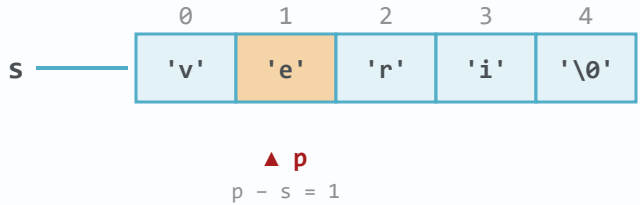
Adım 3 / 11

```
1 size_t uzunluk(const char *s) {
2   const char *p = s;
3   while (*p != '\0')      /* sonlandırıcıya dek */
▶     p++;
5   return (size_t)(p - s);  /* işaretçi farkı */
6 }
7
8 /* uzunluk("veri") */
```

DEĞİŞKENLER

`s = &"veri"[0]` `p = s+1`

strlen nasıl çalışır?



`*p = 'e'` ≠ '\0' → devam.

Adım 4 / 11

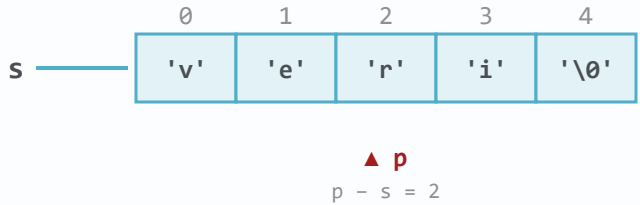
```
1 size_t uzunluk(const char *s) {
2   const char *p = s;
▶   while (*p != '\0')     /* sonlandırıcıya dek */
4     p++;
5   return (size_t)(p - s);  /* işaretçi farkı */
6 }
7
8 /* uzunluk("veri") */
```

DEĞİŞKENLER

`s = &"veri"[0]` `p = s+1`

`*p = 'e'`

strlen nasıl çalışır?



p++: bir karakter (1 bayt) ileri.

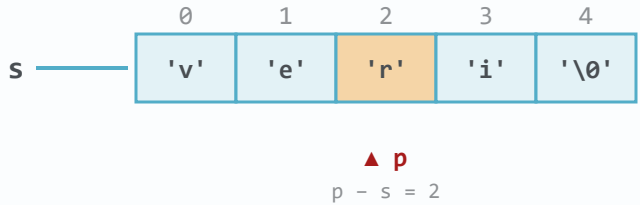
Adım 5 / 11

```
1 size_t uzunluk(const char *s) {
2   const char *p = s;
3   while (*p != '\0')      /* sonlandırıcıya dek */
▶     p++;
5   return (size_t)(p - s);  /* işaretçi farkı */
6 }
7
8 /* uzunluk("veri") */
```

DEĞİŞKENLER

`s = &"veri"[0]` `p = s+2`

strlen nasıl çalışır?



`*p = 'r'` ≠ '\0' → devam.

Adım 6 / 11

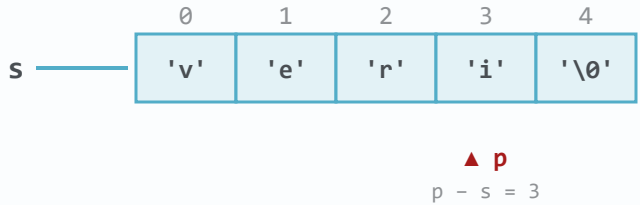
```
1 size_t uzunluk(const char *s) {
2   const char *p = s;
▶   while (*p != '\0')     /* sonlandırıcıya dek */
4     p++;
5   return (size_t)(p - s);  /* işaretçi farkı */
6 }
7
8 /* uzunluk("veri") */
```

DEĞİŞKENLER

`s = &"veri"[0]` `p = s+2`

`*p = 'r'`

strlen nasıl çalışır?



p++: bir karakter (1 bayt) ileri.

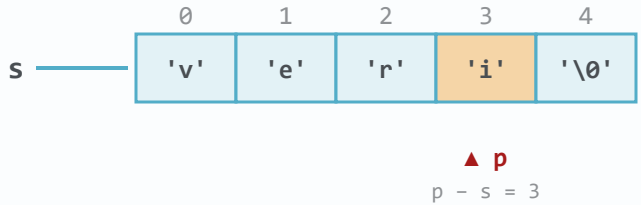
Adım 7 / 11

```
1 size_t uzunluk(const char *s) {
2   const char *p = s;
3   while (*p != '\0')      /* sonlandırıcıya dek */
▶     p++;
5   return (size_t)(p - s);  /* işaretçi farkı */
6 }
7
8 /* uzunluk("veri") */
```

DEĞİŞKENLER

`s = &"veri"[0]` `p = s+3`

strlen nasıl çalışır?



`*p = 'i'` ≠ '\0' → devam.

Adım 8 / 11

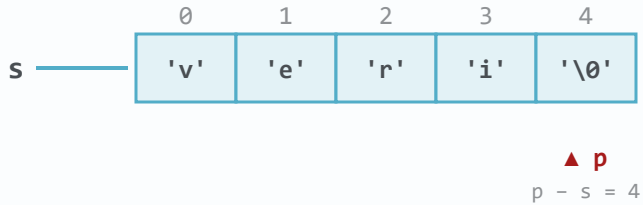
```
1 size_t uzunluk(const char *s) {
2   const char *p = s;
▶   while (*p != '\0')      /* sonlandırıcıya dek */
4     p++;
5   return (size_t)(p - s);  /* işaretçi farkı */
6 }
7
8 /* uzunluk("veri") */
```

DEĞİŞKENLER

`s = &"veri"[0]` `p = s+3`

`*p = 'i'`

strlen nasıl çalışır?



p++: bir karakter (1 bayt) ileri.

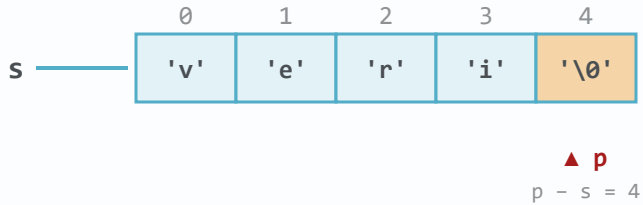
Adım 9 / 11

```
1 size_t uzunluk(const char *s) {
2   const char *p = s;
3   while (*p != '\0')      /* sonlandırıcıya dek */
▶     p++;
5   return (size_t)(p - s);  /* işaretçi farkı */
6 }
7
8 /* uzunluk("veri") */
```

DEĞİŞKENLER

`s = &"veri"[0]` `p = s+4`

strlen nasıl çalışır?



`*p = '\0'` = '\0' → döngü biter.

Adım 10 / 11

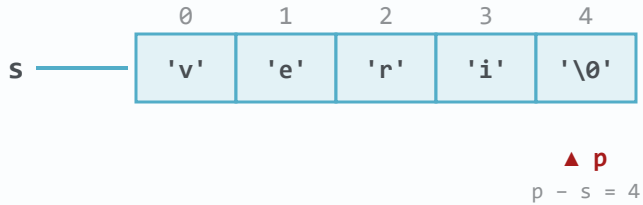
```
1 size_t uzunluk(const char *s) {
2   const char *p = s;
▶   while (*p != '\0')     /* sonlandırıcıya dek */
4     p++;
5   return (size_t)(p - s);  /* işaretçi farkı */
6 }
7
8 /* uzunluk("veri") */
```

DEĞİŞKENLER

`s = &"veri"[0]` `p = s+4`

`*p = '\0'`

strlen nasıl çalışır?



p − s = 4 döner. Uzunluk için dizgi baştan sona taranır: **O(n)** — döngü koşulunda `strlen` çağırmak O(n^2) yapar!

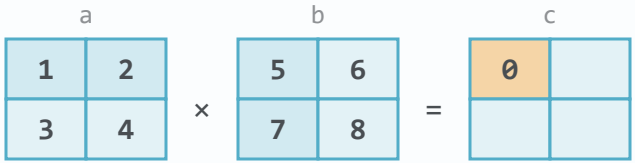
Adım 11 / 11

```
1 size_t uzunluk(const char *s) {
2   const char *p = s;
3   while (*p != '\0')      /* sonlandırıcıya dek */
4     p++;
▶   return (size_t)(p - s);  /* işaretçi farkı */
6 }
7
8 /* uzunluk("veri") */
```

DEĞİŞKENLER

`s = &"veri"[0]` `p = s+4`

Matris çarpımı: üç iç içe döngü



c[0][0] için a'nın **0. satırı** ile b'nin **0. sütunu** kullanılacak; önce 0.

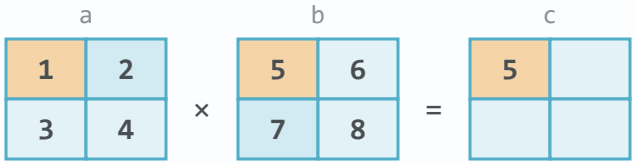
Adım 1 / 13

```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
▶        c[i][j] = 0;
7        for (k = 0; k < N; k++)
8          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

`i = 0` `j = 0` `c[0][0] = 0`

Matris çarpımı: üç iç içe döngü



k = 0: += a[0][0]·b[0][0] = 1·5 → **5**.

Adım 2 / 13

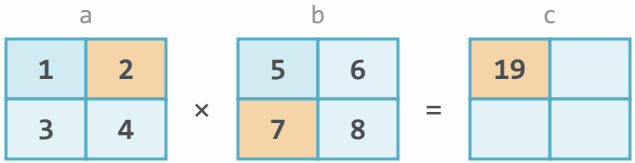
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

`i = 0` `j = 0` `k = 0`

`c[0][0] = 5`

Matris çarpımı: üç iç içe döngü



k = 1: += a[0][1]·b[1][0] = 2·7 → **19**.

Adım 3 / 13

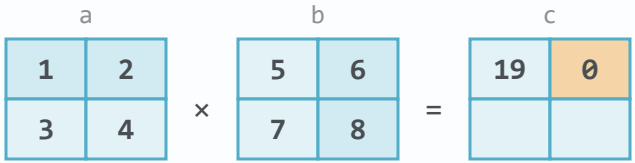
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

`i = 0` `j = 0` `k = 1`

`c[0][0] = 19`

Matris çarpımı: üç iç içe döngü



c[0][1] için a'nın **0. satırı** ile b'nin **1. sütunu** kullanılacak; önce 0.

Adım 4 / 13

```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
▶        c[i][j] = 0;
7        for (k = 0; k < N; k++)
8          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

`i = 0` `j = 1` `c[0][1] = 0`

Matris çarpımı: üç iç içe döngü



k = 0: += a[0][0]·b[0][1] = 1·6 → **6**.


```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

`i = 0` `j = 1` `k = 0`

`c[0][1] = 6`

Matris çarpımı: üç iç içe döngü



k = 1: += a[0][1]·b[1][1] = 2·8 → **22**.

Adım 6 / 13

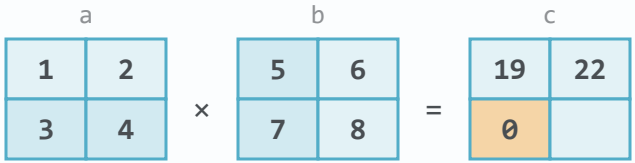
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

`i = 0` `j = 1` `k = 1`

`c[0][1] = 22`

Matris çarpımı: üç iç içe döngü



c[1][0] için a'nın **1. satırı** ile b'nin **0. sütunu** kullanılacak; önce 0.

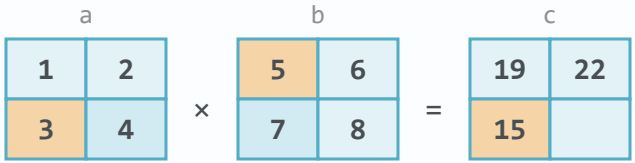
Adım 7 / 13

```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
▶        c[i][j] = 0;
7        for (k = 0; k < N; k++)
8          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

`i = 1` `j = 0` `c[1][0] = 0`

Matris çarpımı: üç iç içe döngü



k = 0: += a[1][0]·b[0][0] = 3·5 → **15**.

Adım 8 / 13

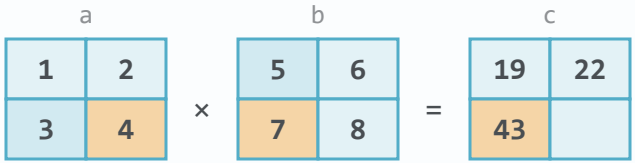
```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

i = 1 | j = 0 | k = 0

c[1][0] = 15

Matris çarpımı: üç iç içe döngü



k = 1: += a[1][1]·b[1][0] = 4·7 → **43**.

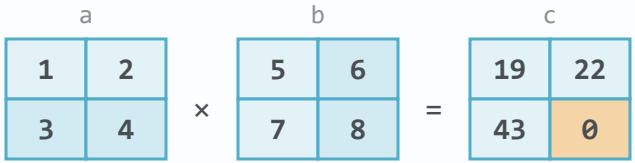

```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

`i = 1` `j = 0` `k = 1`

`c[1][0] = 43`

Matris çarpımı: üç iç içe döngü



c[1][1] için a'nın **1. satırı** ile b'nin **1. sütunu** kullanılacak; önce 0.

Adım 10 / 13

```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
8          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

`i = 1` `j = 1` `c[1][1] = 0`

Matris çarpımı: üç iç içe döngü



k = 0: += a[1][0]·b[0][1] = 3·6 → **18**.


```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

`i = 1` `j = 1` `k = 0`

`c[1][1] = 18`

Matris çarpımı: üç iç içe döngü



k = 1: += a[1][1]·b[1][1] = 4·8 → **50**.


```
1  #define N 2
2  void mult(int a[][N], int b[][N], int c[][N]) {
3    int i, j, k;
4    for (i = 0; i < N; i++)
5      for (j = 0; j < N; j++) {
6        c[i][j] = 0;
7        for (k = 0; k < N; k++)
▶          c[i][j] += a[i][k] * b[k][j];
9      }
10 }
```

DEĞİŞKENLER

`i = 1` `j = 1` `k = 1`

`c[1][1] = 50`

Matris çarpımı: üç iç içe döngü



Bitti: [[19, 22], [43, 50]]. En içteki satır **$N^3 = 8$** kez çalıştı → **$O(n^3)$**.

Adım 13 / 13

```
1 #define N 2
2 void mult(int a[][N], int b[][N], int c[][N]) {
3   int i, j, k;
4   for (i = 0; i < N; i++)
5     for (j = 0; j < N; j++) {
6       c[i][j] = 0;
7       for (k = 0; k < N; k++)
8         c[i][j] += a[i][k] * b[k][j];
9     }
▶ }
```

DEĞİŞKENLER

—

Ödevler: matris çarpımı ve transpoz

1. `mult` (n×n) için adım sayısı tablosunu çıkarın; kaç çarpma yapılır?
2. m×n ile n×p matrisin çarpımının karmaşıklığı nedir? Sonuç matrisi kaç hücre?
3. n×n bir matrisin transpozunu **ek dizi kullanmadan** (yerinde) alan yordamı yazın.

▶ **Çözümler**

Laboratuvar: Dinamik Dizi

7, 3, 9, 4, 6 eklenirken

Girdi (virgülle ayrılmış sayılar) `7, 3, 9, 4, 6` Uygula

KOD

```
 1 typedef struct {
 2   int *veri;  int boyut, kap;
 3 } DinDizi;
 4
 5 int ekle(DinDizi *d, int x) {
 6   if (d->boyut == d->kap) {      /* dolu */
 7     int yeni = d->kap ? 2 * d->kap : 1;
 8     int *t = realloc(d->veri,
 9                      yeni * sizeof *t);
10     if (t == NULL) return 0;
11     d->veri = t;
12     d->kap = yeni;
13   }
14   d->veri[d->boyut++] = x;
```

veri

x – boyut 0 kap 0 toplam kopya 0

ÇIKTI (henüz yok)

Boş dinamik dizi: `{veri = NULL, boyut = 0, kap = 0}`. Dolunca kapasite ikiye katlanır; her büyütmede eski öğeler yeni bloğa taşınır.

KAPANIŞ

Karmaşıklık Özeti

İşlem	Zaman	Ek bellek
A[i] erişimi; n boyutlu adres (n−1 çarpma)	O(1)	O(1)
Dinamik diziye sona ekleme (katlama)	amortize O(1), en kötü O(n)	≤ 2·boyut
Araya ekleme / silme	O(n)	O(1)
Alt üçgen / kuşak matris erişimi	O(1)	n(n+1)/2 · kuşak öğeleri
KusakMatrisKur, AltUcgenOku	O(saklanan öğe)	ARA: a+b−1
strlen / uzunluk	O(n)	O(1)
Matris toplama (m×n)	O(mn)	O(1)
Matris çarpımı (n×n)	$O(n^3)$	O(1)

KAPANIŞ

Sık Yapılan Hatalar

1. Dizi sınırı dışına yazmak (`a[n]`); C bunu denetlemez.
2. `a = realloc(a, …)`: başarısızlıkta eski bloğu kaybetmek.
3. `sizeof(int *)` ile `sizeof(int)`'i karıştırmak; parametrede `sizeof a` kullanmak.
4. 2B dinamik dizide satırları `free` etmeden yalnız işaretçi dizisini bırakmak.
5. Satır/sütun öncelikli formülleri karıştırmak; öğe boyutunu (s) unutmak.
6. Dizgiler için `==` kullanmak ya da `'\0'` için yer ayırmamak.

Alıştırmalar

1. `double d[5][8]` satır öncelikli, α = 2000: `&d[3][5]` kaçtır? Sütun öncelikli olsaydı?
2. `int A[4][5][6]` için A[2][3][4]'ün bağıl konumunu Horner ile hesaplayın.
3. 5×5 alt üçgen matriste a_{43} ALT dizisinin kaçıncı konumunda?
4. n = 6, a = 2, b = 3 kuşak matrisi kaç öğe tutar? ARA dizisi nedir?
5. Kapasitesi 1'den başlayıp ikiye katlanan dinamik diziye 100 öğe eklenirse toplam kaç kopya yapılır?
6. `struct { char a; double b; char c; }` için alanlar yeniden sıralanarak boyut nasıl küçültülür?

▶ **Cevaplar**

KAPANIŞ

Bölüm Özeti

Dizi = eşleme

ADT: ⟨indis, değer⟩; C gerçeklemesi ardışık bellek ve $\alpha + i \cdot s$ adresi.

Adres hesabı

Satır öncelikli: $\alpha + (i \cdot u_1 + j) \cdot s$; n boyutta Horner ile $n-1$ çarpma.

Yer kazanmak

Üçgen: $n(n+1)/2$ öğe, $k = i(i+1)/2 + j$. Kuşak: $j - i$ numaralı kuşaklar + ARA.

Dinamik bellek

malloc/realloc/free; katlayarak büyüme amortize O(1); 2B dizi = işaretçi dizisi.

Kayıtlar

struct alanları toplar, union paylaştırır; dizgi '\0' ile biten char dizisidir.

Matrisler

Toplama $O(mn)$, çarpma $O(n^3)$.

KAPANIŞ

Kaynaklar ve Devam

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. bs., Bölüm 2 (diziler, yapılar, çok boyutlu diziler, dizgiler) ve 1.4.
- B. Kernighan, D. Ritchie, *The C Programming Language*, 2. bs., Bölüm 5–6 (işaretçiler, diziler, yapılar).
- T. Cormen ve ark., *Introduction to Algorithms*, 17.4 (dinamik tablolar ve amortize analiz).

← Bölüm 1: Giriş ve Algoritma Analizi

Ders sayfası

Bölüm 3: Özyineleme, İkili Arama ve Geri İzleme

Son

Bölüm 2: Diziler ve Yapılar

Sonraki bölüm: Bölüm 3: Özyineleme, İkili Arama ve Geri İzleme · Ders sayfası