YZM 2116- VERİ YAPILARI

DERS#5: SIRALAMA VE ARAMA ALGORİTMALARI

İÇERİK

Bu bölümde,

- Sıralama(Sort) Algoritmaları
  1. Bubble Sort
  2. Selection Sort
  3. Insert Sort
  4. Quick Sort
- Arama(Search) Algoritmaları
  1. Linear Search
  2. Binary Search

konusuna değinilecektir.

Sıralama Algoritmaları

- Sıralama, sayısal ortamdaki bilgilerin veya verilerin **belirli bir anahtar sözcüğe** göre belirli bir anlamda sıralı erişilmesini sağlayan bir düzenlemedir.
- **Örneğin:** Telefon rehberindeki bir kişinin telefon numarasının bulunması bir **arama (search)** işlemidir.
- Genel olarak eleman toplulukları
  - **Daha etkin erişim amacıyla** (aramak ve bilgi getirmek) **sıralanır** ve/veya **sıralı şekilde** saklı tutulurlar.

Sıralama Algoritmaları (devam…)

- Eleman (kayıt, yapı ...) toplulukları **genelde** (her zaman değil) **bir anahtara** göre *sıralı tutulurlar*.
- Bu anahtar genelde *elemanların* **bir alt alanı** yani **üyesidir**.
- **Örneğin, Öğrenci**
  - Soyada göre sıralı olursa soyadı anahtardır,
  - Numaraya göre sıralı olursa numara anahtardır,
  - Nota göre sıralı olursa not alanı anahtardır.
- Elemanlar içindeki her elemanın anahtar değeri, **kendinden önceki** elemanın anahtar değerinden **büyükse** artan sırada (**ascent** – AZ), **küçükse** azalan (**descent** – ZA) sırada sıralıdır.

Sıralama Algoritmaları (devam…)

- Sıralama algoritmalarının hesaplama verimliliği açısından; VY olarak **array, stack, queue** veya **tree** kullanılabilir.
- Literatürde ***çok farklı*** sıralama algoritmaları mevcuttur:
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
  - Quick Sort
  - Heap Sort
  - Merge Sort
  - Shell Sort
  - …

1. Bubble (Kabarcık) Sort

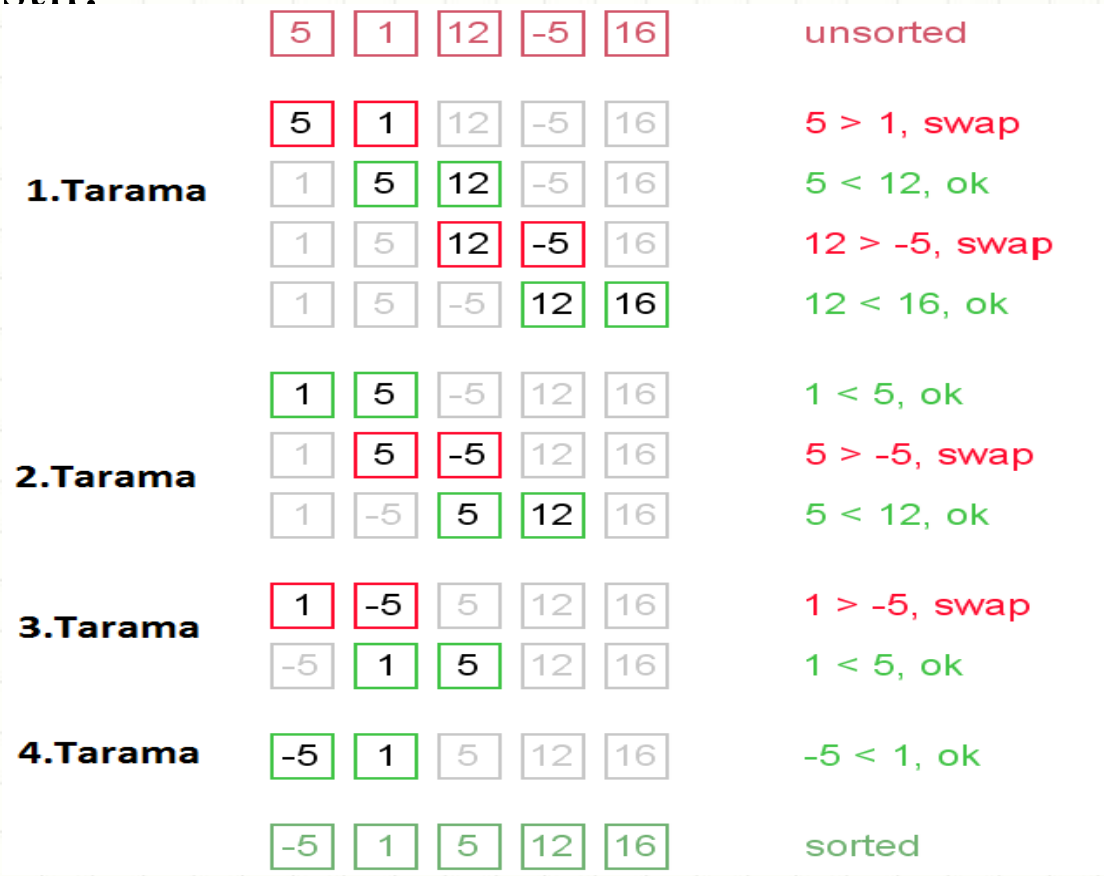
- Verimliliği düşük ancak **mantığı basit** bir sıralama algoritmasıdır.
- **"n" boyutlu** bir **a[] dizisi** için artan sıralamayı düşünelim.
- Bubble Sort (BS) **en fazla (n-1) taramada (pass)** *sıralamayı tamamlar*.

1. Bubble (Kabarcık) Sort (devam…)

- **İlk taramada** a_0 ve a_1 kıyaslanır eğer $a_0 > a_1$ ise iki değer takas (**swap**) edilir.
- Değilse bu defa a_1 ve a_2 kıyaslanarak **gerekirse yer değişikliği** yapılır.
- Bu işlem her seferinde **takas edilme durumu oluşmayana** kadar **sürer**.
- *Sadece* **1. taramada tüm dizi elemanları dolaşılmaktadır**.
- Her taramada
  - ***(dizinin eleman sayısı) – (tarama sayısı)*** kadar eleman dolaşılır.

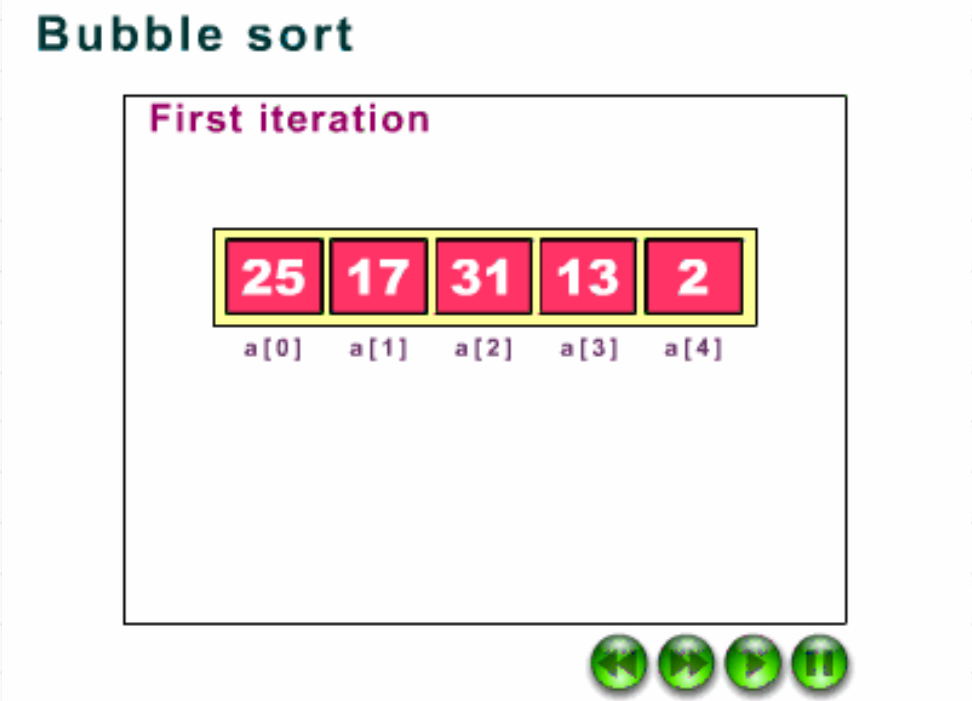
1. Bubble (Kabarcık) Sort (devam…)

- **[5, 1, 12, -5, 16]** dizisinde **Bubble Sort** işletimi gösterilmistir.



1. Bubble (Kabarcık) Sort (devam…)

- Aşağıdaki animasyonda **[25, 17, 31, 13, 2]** dizisinin **Bubble Sort** algoritmasının işletilmesi sonucu oluşan sıralamalar gösterilmiştir.



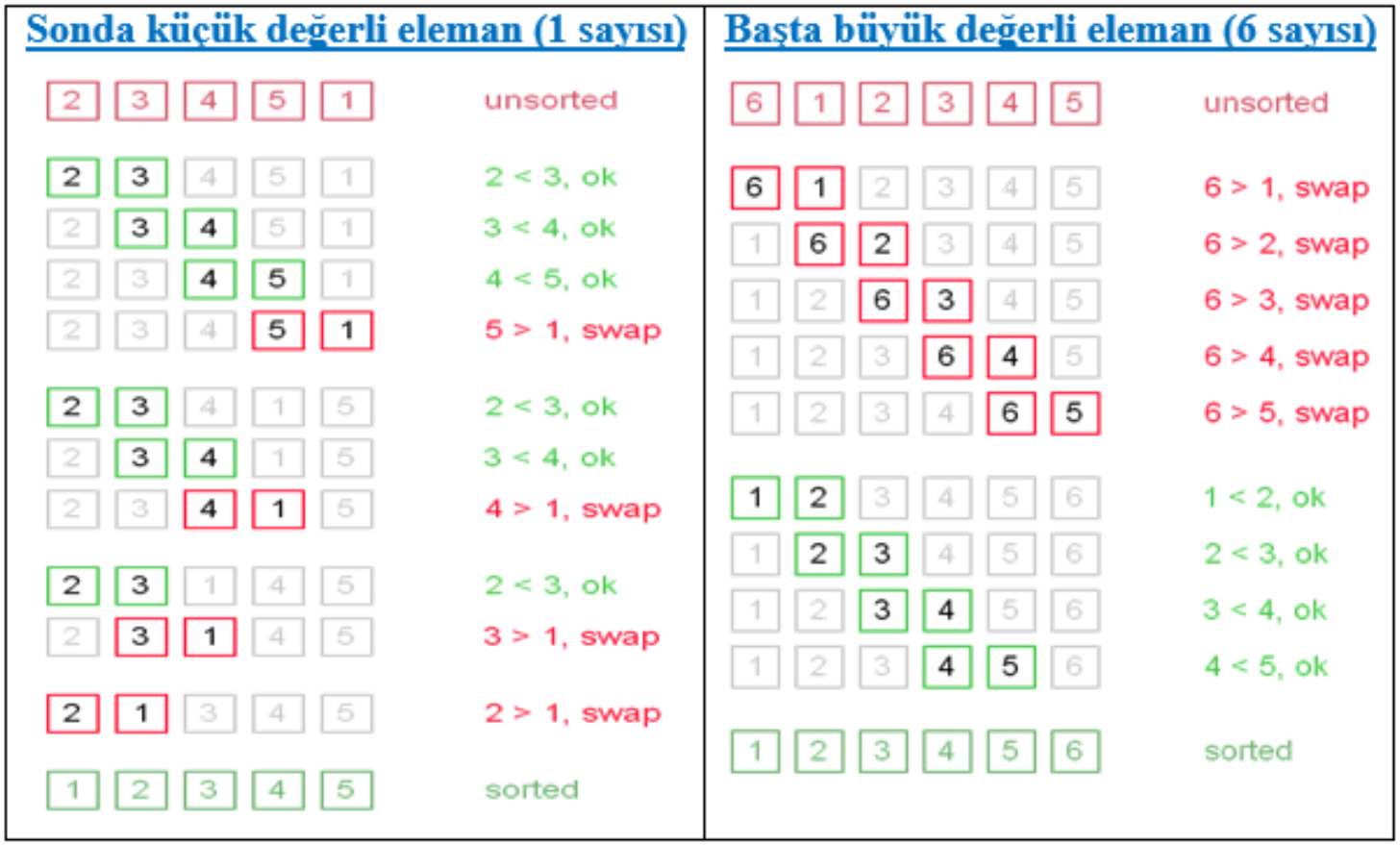
1. Bubble (Kabarcık) Sort Implementasyon

```
public void Sort(int[] items)
{
    int tarama;
    bool swapped = false;
    for (tarama = 0; tarama < items.Length; tarama++)
    {
        swapped = false;
        //Her tarama sonrası sondaki elemanları zaten sıralı olacağından
        onları karşılaştırmamak için tarama sayısı çıkart
        for (int i = 0; i < (items.Length - tarama - 1); i++)
        {
           if (items[i] > items[i + 1])
           {
              int temp;
              temp = items[i];
              items[i] = items[i + 1];
              items[i + 1] = temp;
              swapped = true;
           }
        }
       //Eğer geçişte sıralama yapılmadıysa, bir sonraki geçişe geçme, işlemi bitir.
        if (!swapped)
           break;
   }
}
```

1. Bubble (Kabarcık) Sort (devam…)

- Quadratic **işlem karmaşıklığı sahip** olan Bubble Sort algoritması, **büyük n değerlerinde** **en yavaş** ***performansa sahip*** olan sıralama algoritmasıdır.
- **Elemanların ilk dizilimi** sıralama performansını **önemli ölçüde** etkiler.
- Örneğin *Neredeyse sıralanmış bir dizide,*
  - ***Baştaki*** *Büyük değerli elemanların dizinin sonuna ilerlemesi hızlı iken*
  - ***Sondaki*** *Küçük değerli elemanların dizinin başına getirilmesi yavaştır*

1. Bubble (Kabarcık) Sort (devam…)



- Bu sorunu çözmek için **“coctail sort”** algoritması geliştirilmiştir.

1. Bubble Sort Karmaşıklığı

- **İç döngü:** $(n - 1) + (n - 2) + \ldots + 1 = (n - 1 + 1)/2$
- **Dış döngü:** n
- **Toplam İterasyon:** $n * (n / 2) = n^2 / 2$
- **Big O Karmaşıklığı:** $O(n^2)$

2. Selection Sort

- Bubble Sort algoritması gibi verimliliği düşük ancak mantığı **yine** basit bir ***sıralama algoritmasıdır*****.**
- Takas sayısı **çok daha az olduğu** için Bubble Sort algoritmasının **iyileştirilmiş** hali gibi düşünülebilir.

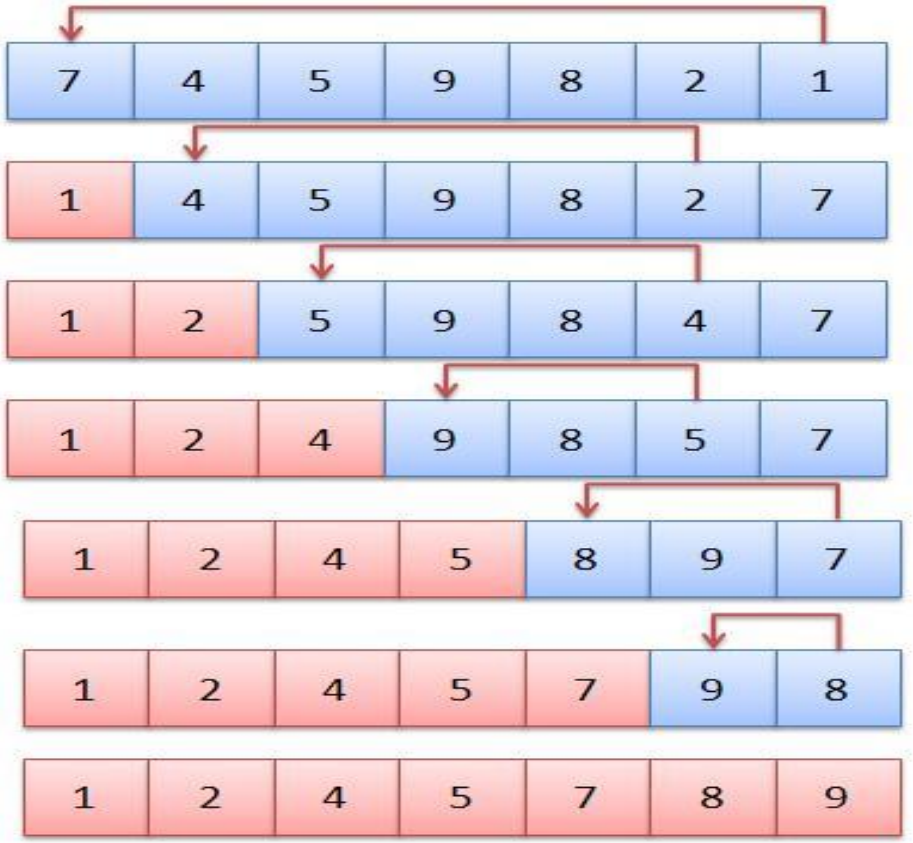
2. Selection Sort (devam…)

Algoritmanın temel çalışma mantığı aşağıdaki gibidir:

- Dizinin **ilk elemanından** başla.
- Dizideki **en küçük elemanı bul**, **A[0]** indisindeki elemanla karşılıklı takas yap (**swap**).
- **İkinci en küçük elemanı A[1] ile A[n] arasında** ara,
- İkinci en küçük elemanı bul,
  - Bu eleman ile **A[1]'i** takas yap.
- *Takas işlemine* **dizi bitene kadar** devam et.
- Her adımda
  - **Sol** tarafta **sıralı** bir alt dizi,
  - **Sağ** tarafta ise **sırasız** bir alt dizi elde edilir.

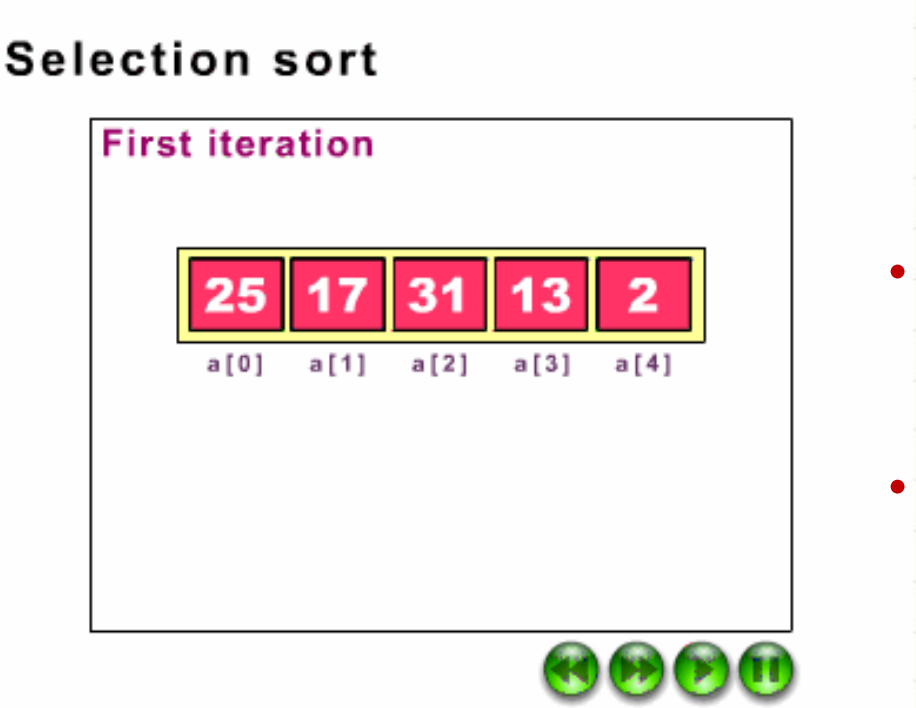
2. Selection Sort (devam…)

- Aşağıda **[7, 4, 5, 9, 8, 2, 1]** dizisinde Selection Sort algoritmasının işletilme adımları gösterilmiştir.



2. Selection Sort (devam…)

- Aşağıdaki animasyonda **[25, 17, 31, 13, 2]** dizisinin **Selection Sort** algoritmasının işletilmesi sonucu oluşan sıralamalar gösterilmiştir.



DİKKAT

- Animasyondaki gibi **her iterasyonda swap etmek gereksizdir**.
- **Kalanlar arasında en küçük bulunduğunda swap edilmesi** yeterlidir.

2. Selection Sort Implementasyon

```
public void Sort(int[] items)
{
      int n = items.Length;
      int minIndis = 0;

      for (int i = 0; i < n; i++)
      {
            //minimumum i olarak ayarla
            minIndis = i;
            //i'den sonraki tüm elemanları tara
            for (int j = i + 1; j < n; j++)
            {
                //daha küçük eleman bulursan indisini sakla
                if (items[j] < items[minIndis])
                    minIndis = j;
            }

            //en küçük indis değiştiyse, yani i'den sonraki elemanlardan
            birisi i'den küçükse
            //takas işlemi gerçekleştir
            if (minIndis != i)
            {
                int temp = items[i];
                items[i] = items[minIndis];
                items[minIndis] = temp;
            }
      }
}
```

2. Selection Sort Karmaşıklığı

- Algoritma ilk geçiş için (n - 1) karşılaştırma, ikinci geçiş için ise (n - 2) karşılaştırma yapar.
- A(n) elemanlı dizi için toplam karşılaştırma adedi ve karmaşıklık
- $A(n) = (n - 1) + (n - 2) + \ldots + 2 + 1$ veya
- $(n * n) / 2 = 1 / 2 (n^2)$
- veya $O(n^2)$ olur.

3. Insertion Sort

Örneğin, masada bir deste oyun kâğıdı, sırasız ve sırtları yukarıya doğru duruyor olsun.

- Desteden **en üstteki kartı** alalım. Onu masaya yüzü görünür biçimde koyalım. Tek kart olduğu için sıralı bir kümedir.
- Sırasız destenin üstünden bir **kart daha çekelim**. *Masadaki ilk çektiğimiz kart ile **karşılaştıralım***.
- Gerekirse yerlerini değiştirerek, çektiğimiz **iki kartı küçükten büyüğe doğru sıralayalım**.
- Sırasız destenin üstünden bir kart daha çekelim. Masaya sıralı dizilen iki kart ile karşılaştıralım. Gerekirse yerlerini değiştirerek çekilen **üç kartı küçükten büyüğe doğru sıralayalım**.
- Bu işleme sırasız deste bitene kadar **devam edelim**. Sonunda, oyun kâğıtlarını sıralamış oluruz.

3. Insertion Sort (devam…)

- **Örnek:** A[n]= **[65, 50, 30, 35, 25, 45]** dizisini Insertion Sort kullanarak sıralayalım.:

Pass 0	**İlk adımda sıralanmış dizide hiç eleman yokken tüm elemanlar sıralanmamış dizide yer alır.**
Pass 1	Sıralamaya başlarken A[0] (ya da 65) sıralanmış dizinin ilk elemanı kabul edilir ve sıralanmamış dizinin ilk elemanı ile A[1] (ya da 50) karşılaştırılır. 50, 65'ten küçük olduğu için sıralı dizide 65 bir sağa kayarken 50 ilk sıraya yerleşir.
Pass 2	İşlem A[2] (30) için tekrarlanır. 30 sıralı dizide önce 65'ile kıyaslanır ve 65 bir sağa kayar. 30 bu defa 50 ile kıyaslanır ve 50 bir sağa kayar.
Pass 3	İşlem A[3] (35) için tekrarlanır. 35 sıralı dizide önce 65'ile kıyaslanır ve 65 bir sağa kayar. 35 bu defa 50 ile kıyaslanır ve 50 bir sağa kayar. 35, son olarak 30 ile karşılaştırılır, 30'dan büyük olduğu için yer değiştirme olmaz ve 35 A[1]'e insert edilir.
Pass 4	İşlem A[4] (25) için tekrarlanır. 25 sıralı dizide önce 65'ile kıyaslanır ve 65 bir sağa kayar. 25 bu defa 50 ile kıyaslanır ve 50 bir sağa kayar. 25 daha sonra 35 ile kıyaslanır ve 35 bir sağa kayar. 25, son olarak 30 ile karşılaştırılır, 30'dan küçük olduğu için yer değiştirme olur 30 sağa kayar ve 25 A[0]'a insert edilir.
Pass 5	İşlem A[5] (45) için tekrarlanır. 45 sıralı dizide önce 65'ile kıyaslanır ve 65 bir sağa kayar. 45 bu defa 50 ile kıyaslanır ve 50 bir sağa kayar. 45 son olarak 35 ile karşılaştırılır, 35'den büyük olduğu için yer değiştirme olmaz ve 45 A[3]'e insert edilir.

3. Insertion Sort (devam…)

pass	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	*Process*
	65	50	30	35	25	45	Original array
1	**50**	65	30	35	25	45	50 is inserted
2	**30**	50	65	35	25	45	30 is inserted
3	30	**35**	50	65	25	45	35 is inserted
4	**25**	30	35	50	65	45	25 is inserted
5	25	30	35	**45**	50	65	45 is inserted

3. Insertion Sort Implementasyonu

```
public override void Sort(int[] items)
{
    int i, j, moved;
    for (i = 1; i < items.Length; i++)
    {
        moved = items[i];
        j = i;
        while (j > 0 && items[j - 1] > moved)
        {
            items[j] = items[j - 1];
            j--;
        }
        items[j] = moved;
    }
}
```

3. Insertion Sort Karmaşıklığı

- A(n) elemanlı dizi için toplam karşılaştırma adedi ve karmaşıklık

- **A(n) = (n – 1) + (n – 2) + ... + 3 + 2 + 1** veya
- **(n * (n + 1) / 2) = 1 / 2 (n^2+n)**
- veya
- **O(n^2)** olur.

4. Quick Sort

- Quick Sort (QS) algoritması **"böl-fethet"** mantığına uygun olarak sıralama yapan **recursive (özyinelemeli)** bir algoritmadır.
- Ticari sıralama algoritmalarında QS sıklıkla kullanılır.
- QS algoritması temel olarak diziyi bir **pivot (mihenk)** etrafında **iki alt-diziye bölerek:**
  - *Pivota **küçük eşit** elemanları* pivotun sol tarafına
  - *Pivottan **büyük elemanları*** pivotun sağ tarafına yerleştirir.

4. Quick Sort (devam…)

- Aynı işlem, **her alt dizi için bir pivot seçerek** tekrarlar ve *bu işlem tek eleman kalana kadar devam eder.*
- QS için ***farklı pivot seçim yöntemleri*** bulunmaktadır.
- ***Pivot olarak*** dizinin
  - İlk elemanı,
  - Son elemanı,
  - Orta elemanı,
  - Rastgele bir dizi elemanı,
- seçilerek sıralama işlemi gerçekleştirilebilmektedir.

4. Quick Sort **Örnek** (devam…)

- **A[i] = {2, 17, -4, 42, 9, 26, 11, 3, 5, 28}** dizisi verilsin.
- Dizinin terimleri arasından birisini *mihenk (pivot)* olarak seçelim. Yapacağımız işlemde eş uygulamayı sağlamak için mihengi şöyle seçelim: Ayrıştırılacak (alt) dizinin ilk teriminin indisi ile son teriminin indislerini **toplayıp** 2 ye bölelim. Bölümün tamsayı parçasını mihengin indisi olarak seçelim.
- (0 + 9) / 2 = 4 (pivot, tam kısım)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	17	-4	42	9	26	11	3	5	28

4. Quick Sort **Örnek** (devam…)

- Şimdi (A[4]=9) mihenginin solunda
  - **9'dan büyük olan terimler varsa onları sağ tarafa taşıyacağız**.
  - Benzer şekilde, **9'un sağında 9'dan küçük terimler varsa**, onları sol tarafa taşıyacağız.
  - Özel olarak mihenge eşit olan başka terimler varsa, onları sağa ya da sola taşıyabiliriz.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	17	-4	42	9	26	11	3	5	28

4. Quick Sort **Örnek** (devam…)

- **Pratik yöntem:** Sol elimizle sol kısmı sağ elimizle de sağ kısmı tarayacağız.
- Solda mihenkten büyük ilk eleman:
  - **A[1] = 17** solda tut
- Sağda mihenkten küçük ilk eleman:
  - **A[8] = 5** sağda tut

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	17	-4	42	9	26	11	3	5	28

- İki elemanı **yer değiştir**.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	5	-4	42	9	26	11	3	17	28

4. Quick Sort **Örnek** (devam…)

- Aynı işlemi solda 9'dan büyük, sağda 9'dan küçük olan sayılar çifti için tekrarlayacağız.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	5	-4	42	9	26	11	3	17	28

- Görüldüğü gibi, **42 ve 3** sayıları takas edilecektir.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	5	-4	3	9	26	11	42	17	28

4. Quick Sort **Örnek** (devam…)

- Bu eylem sonunda, diziyi **{2, 5, -4, 3}**, **{9}** ve **{26, 11, 42, 17, 28}** olmak üzere **üç altdiziye** ayırmış oluruz.

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	5	-4	3	9	26	11	42	17	28

- Soldaki alt dizideki her terim, sağdaki alt dizideki her teriminden küçüktür.
- Şimdi soldaki ve sağdaki altdizileri **yeni dizilermiş gibi düşünüp**, aynı yöntemle onları ayrıştırabiliriz.
- Sonra onların da sol ve sağ alt dizilerini ayrıştırabiliriz. Bu süreç, ***alt diziler tek terimli birer diziye indirgenene kadar*** devam edecektir.

4. Quick Sort **Örnek** (devam…)

Sol	Sağ	Pivot	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	Açıklama
			2	17	-4	42	9	26	11	3	5	28	
1	8	9	2	5	-4	42	9	26	11	3	17	28	1 ↔ 8
3	7	9	2	5	-4	3	9	26	11	42	17	28	3 ↔ 7
4	4	9	2	**5**	-4	3	9	26	11	42	17	28	*Ayrışma*
1	3	5	2	3	-4	5	9	26	11	42	17	28	1 ↔ 3
1	2		2	-4	3	5	9	26	11	42	17	28	1 ↔ 2
		-4	2	-4	3	5	9	26	11	42	17	28	*Ayrışma*
0	1		-4	2	3	5	9	26	11	42	17	28	0 ↔ 1
		42	-4	2	3	5	9	26	11	42	17	28	*Ayrışma*
7	9		-4	2	3	5	9	26	11	28	17	42	7 ↔ 9
5	8	11	-4	2	3	5	9	26	11	28	17	42	*Ayrışma*
5	6		-4	2	3	5	9	11	26	28	17	42	5 ↔ 6
6	8	28	-4	2	3	5	9	11	26	28	17	42	*Ayrışma*
7	8	28	-4	2	3	5	9	11	26	17	28	42	7 ↔ 8
6	7	26	-4	2	3	5	9	11	26	17	28	42	*Ayrışma*
6	7	26	-4	2	3	5	9	11	17	26	28	42	6 ↔ 7
			-4	2	3	5	9	11	17	26	28	42	sıralandı

4. Quick Sort Implementasyon

```
public void quickSort(int[] items, int altindis, int ustindis)
{
    // altindis o adımda sıralanan altdizinin ek küçük indisidir
    // üstindis o adımda sıralanan altdizinin ek büyük indisidir
    int yeni_altindis = altindis, yeni_ustindis = ustindis, h;
    // pivot
    int pivot = items[(altindis + ustindis) / 2];
    // Takas ile diziyi ayrıştırma
    do
    {
        while (items[yeni_altindis] < pivot)
            yeni_altindis++;
        while (items[yeni_ustindis] > pivot)
            yeni_ustindis--;
        if (yeni_altindis <= yeni_ustindis)
        {
            h = items[yeni_altindis];
            items[yeni_altindis] = items[yeni_ustindis];
            items[yeni_ustindis] = h;
            yeni_altindis++;
            yeni_ustindis--;
        }
    } while (yeni_altindis <= yeni_ustindis);
    // recursion
    if (altindis < yeni_ustindis)
        quickSort(items, altindis, yeni_ustindis);
    if (yeni_altindis < ustindis)
        quickSort(items, yeni_altindis, ustindis);
}
```

İLK ÇAĞRIM

```
public override void Sort(int[] items)
{
    quickSort(items, 0, items.Length - 1);
}
```

DİZİ İLK HALİ

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	17	-4	42	9	26	11	3	5	28

Watch 1

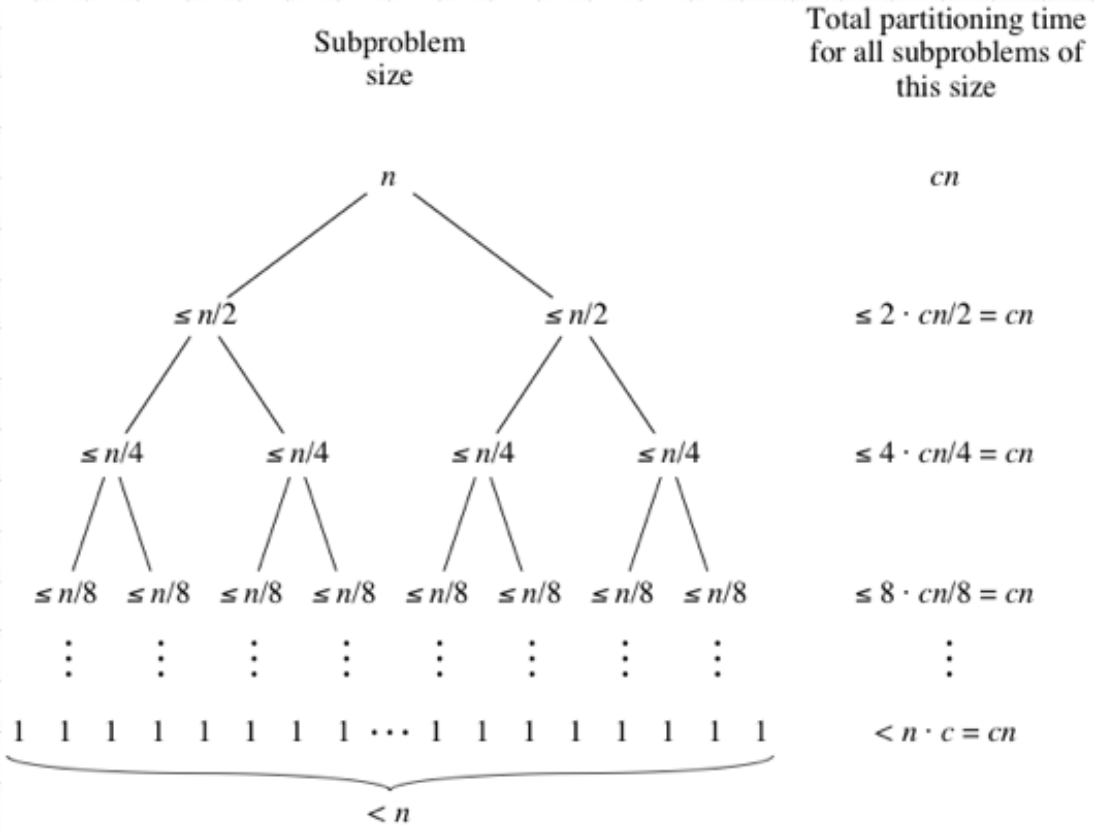
Name	Value
pivot	9
yeni_altindis	1
yeni_ustindis	8
items[yeni_altindis]	17
items[yeni_ustindis]	5

SWAP

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
2	5	-4	42	9	26	11	3	17	28

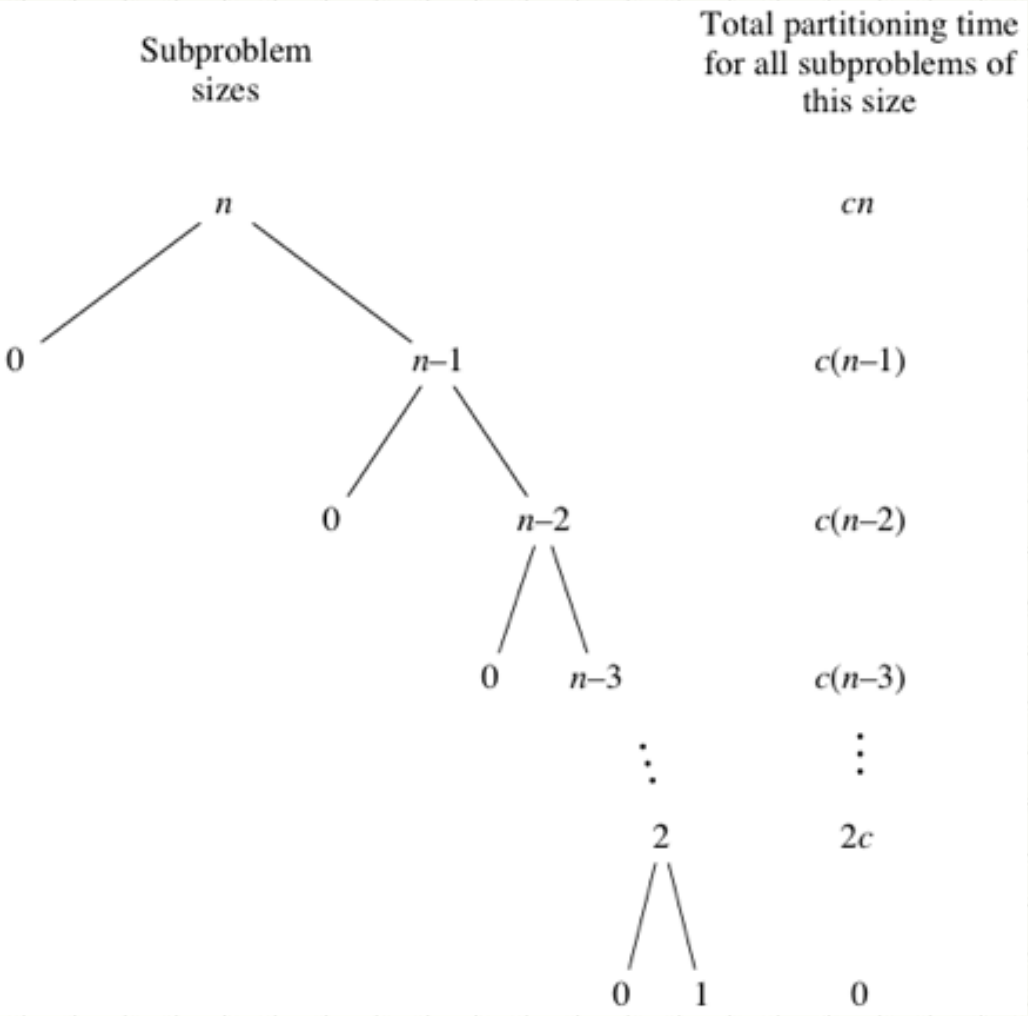
4. Quick Sort Karmaşıklığı

- Eğer şanslı isek, seçilen her eleman ortanca değere yakınsa (**Average Case**)
  - **O(nlogn)** karmaşıklığa sahipken



4. Quick Sort Karmaşıklığı

- En kötü durumda ise parçalama dengesiz olacak ve n iterasyonla sonuçlanacaktır (**Worst Case**)
  - **$O(n^2)$**



Sıralama Algoritmaları Karmaşıklık Kıyaslama

Name	Average Case	Worst Case
Bubble	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$
Insertion	$O(n^2)$	$O(n^2)$
Shell	-	$O(nlog^2n)$
Merge	$O(nlogn)$	$O(nlogn)$
Heap	$O(nlogn)$	$\underline{O}(nlogn)$
Quick Sort	$O(nlogn)$	$O(n^2)$
Tree sort	$O(nlogn)$	$O(n^2)$

Arama Algoritmaları

- **Arama**, bir bilgi kümesi içerisinde belirli bir anahtar sözcüğe dayanılarak **onunla ilgili diğer bilgilere erişme/elde etme** işlemidir.
- Bilgisayar ve benzeri sayısal ortamlarda tutulan bilgilerin anlamlı olarak kullanılması ve onlar üzerinde işlemler yapılması için her şeyden önce **bilginin olup olmadığı** ve **varsa belleğin neresinde olduğunun belirlenmesi gerekir.**
- Bu işlem, bir **arama algoritması** ile gerçekleştirilir.

Arama Algoritmaları (devam…)

- Arama işleminin yapılış şekli bilgiye ait verilerin **düzenlenme şekline** (**sıralı** / **sırasız olması**) ve **bellekte tutulmasına** göre farklılıklar gösterir.

- **İki tür** *arama işlemi* gerçekleştirilebilmektedir:

  1. **Dahili (internal) arama**
  2. **Harici (external) arama**

Arama Algoritmaları (devam…)

1. **Dahili (internal) arama:** Arama işlemi bellek üzerinde tutulan veriler üzerinde yapılır. Veriye erişim **hızlı** olduğu için verilerin *yer değiştirmesi*, *araya ekleme*, aradan çıkarma gibi işlemler **daha hızlı** gerçekleşir.
2. **Harici (external) arama:** Arama işlemi **disk**, ***yedekleme birimi*** gibi belleğe göre **daha yavaş** olan saklama birimleri üzerinde yapılır. Dolayısı ile verilerin yer değiştirmesi, bir veriden diğerine atlanması, arkada kalanların öne doğru kaydırılarak bir verinin silinmesi veya arkaya doğru kaydırılarak yeni bir veriye yer açılması gibi işlemler **zaman alır.**

Arama Algoritmaları (devam…)

- Bu bölümde üzerinde duracağımız arama algoritmaları aşağıdaki gibidir:

1. **Linear Search (Doğrusal Arama)**
2. **Binary Search (İkili Arama)**

1. Linear Search (Doğrusal Arama)

- Herhangi bir liste veya dizi içerisindeki **ilk** elemandan başlanarak **son** elemana kadar, *dizi terimleri ile aranan sözcük* ***karşılaştırılır***, sözcük bulunduğunda geriye dizi teriminin **indisi**, bulunamadığında ise **-1 (false)** değeri döndürülür.

Not: Dizi sırasız ve az elemana sahipse tercih edilen bir algoritmadır.

1. Linear Search (Doğrusal Arama) (devam…)

```
public override int Search(int[] items, int searchKey)
{
     for (int i = 0; i < items.Length; i++)
     {
        if (items[i] == searchKey)
           return i;
     }
     return -1;
}
```

Linear Search Big O Karmaşıklığı: O(n)

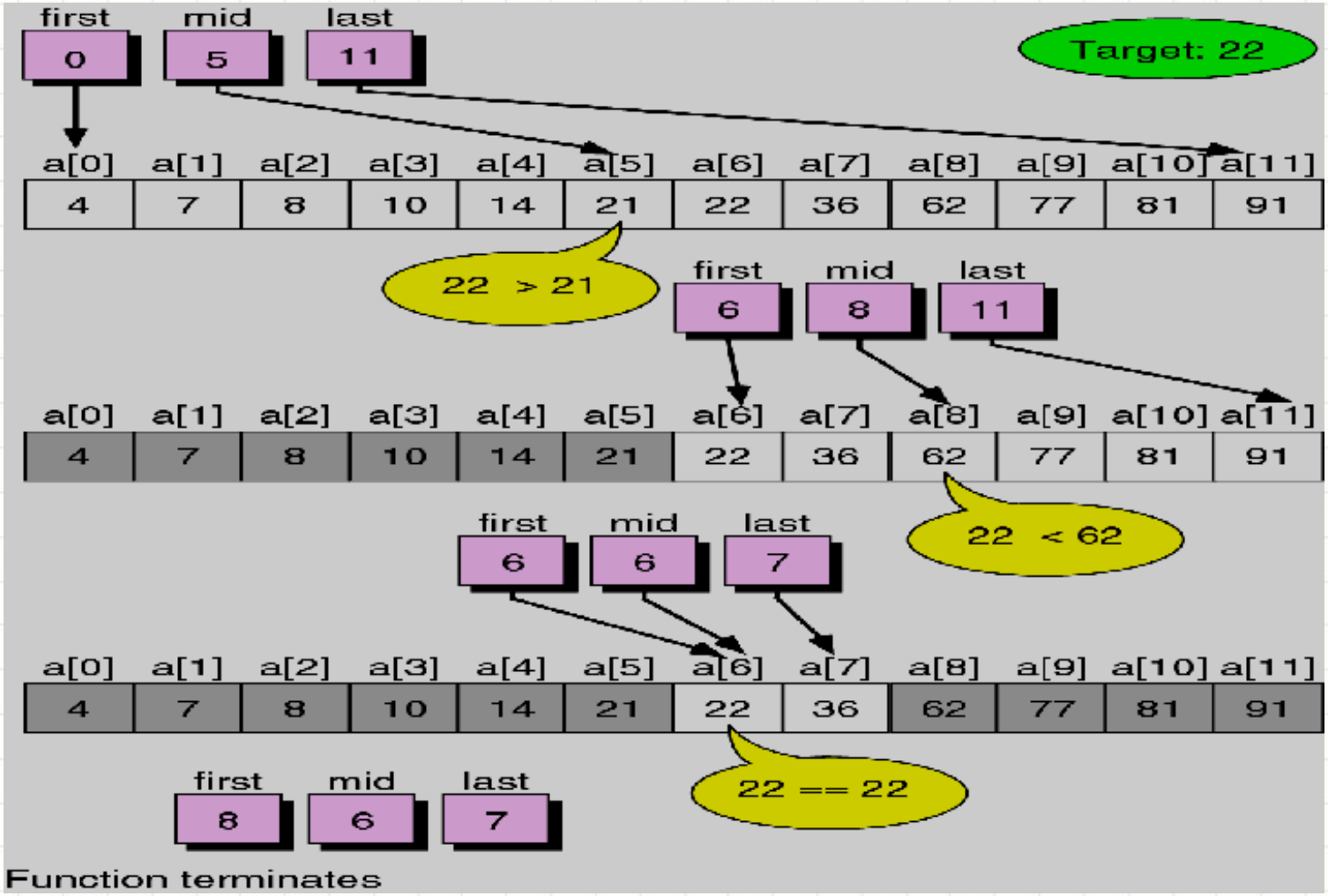
2. Binary Search (İkili Arama)

- **İkili arama** **sıralı veriler** üzerinde çalışan bir algoritmadır.
- Verilerin ***önceden*** belirli bir **anahtar** sözcüğe göre *sıralanması gerekmektedir*.
- **Örneğin,** veriler bir dizi üzerinde ardışık olarak sıralı şekilde tutuluyorsa ikili arama **yapılabilir**.
- Sıralı veriler her adımda **iki parçaya bölünerek** *arama işlemi* gerçekleştirilir.
- İkili arama algoritması, tasarım olarak **parçala fethet (divide and conquere)** türündedir.

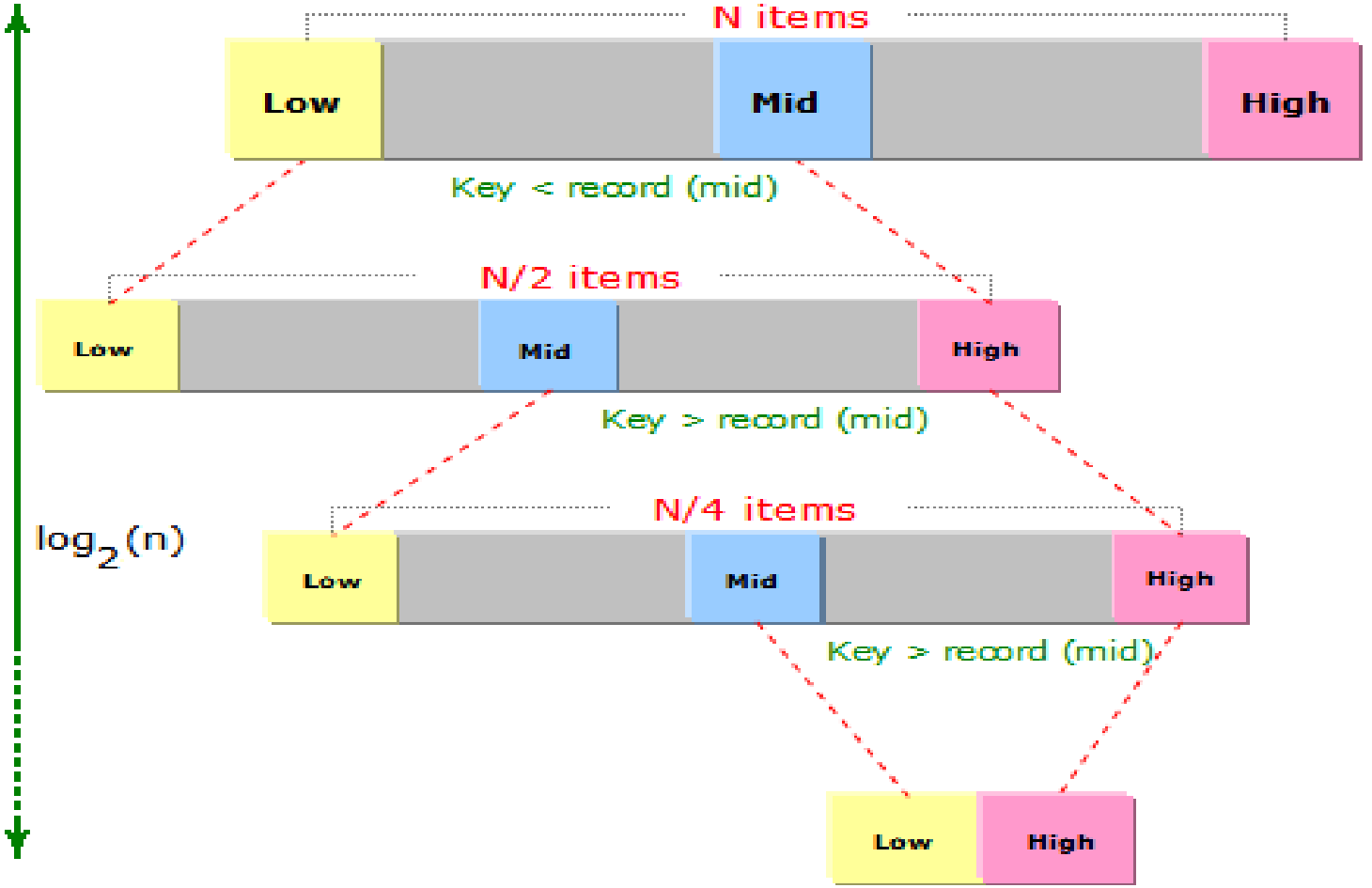
2. Binary Search (İkili Arama) (devam…)

- İkili arama algoritmasının sözde kodu aşağıdaki gibidir:
  - Aranacak uzayın **tam orta noktasına** bak
  - Eğer aranan değer bulunduysa **bitir**
  - Eğer **bakılan değer** *aranan değerden* **büyükse**
    - Arama işlemini problem uzayının küçük elemanlarında devam ettir.
  - Eğer bakılan değer aranan değerden **küçükse**
    - Arama işlemini problem uzayının büyük elemanlarında devam ettir.
  - Şayet **bakılan aralık 1** veya daha küçükse aranan değer bulunamadı olarak bitir.

2. Binary Search (İkili Arama) (devam…)



2. Binary Search (İkili Arama) (devam…)



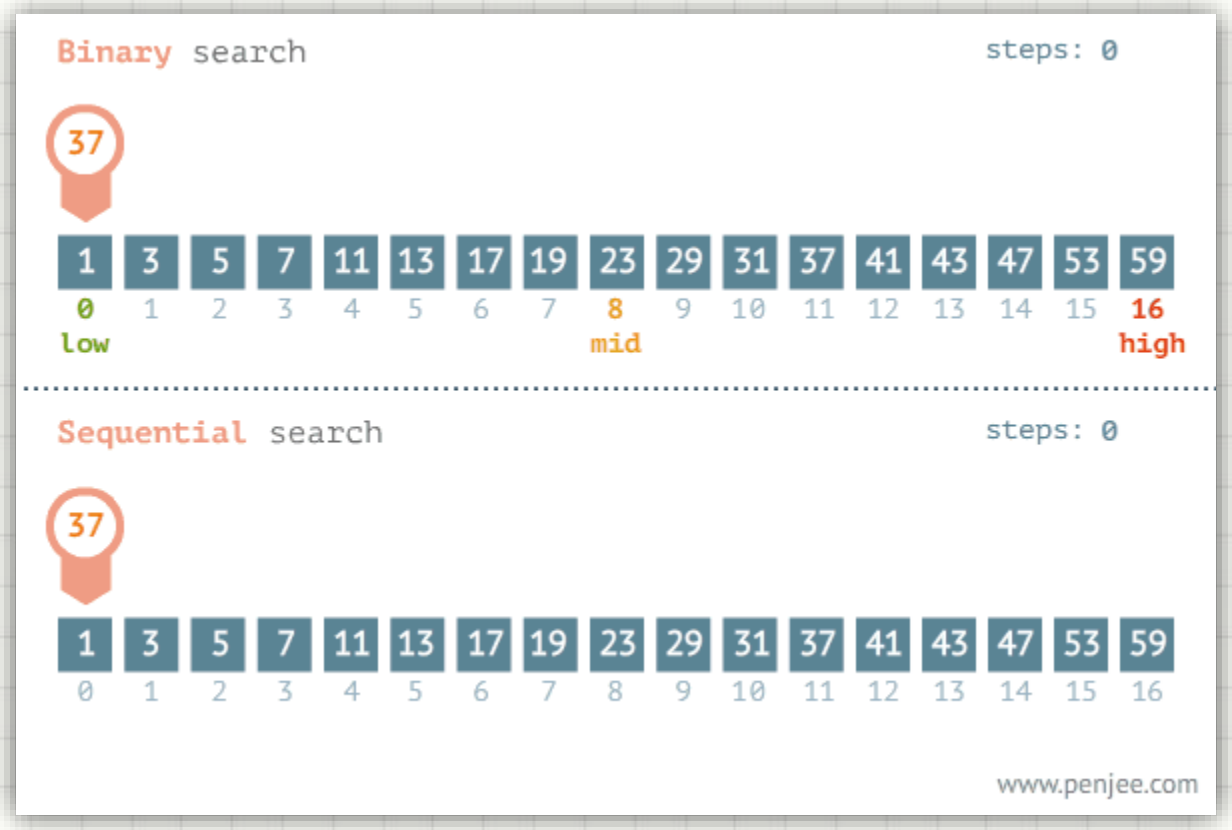
Binary Search Big O Karmaşıklığı: O(logn)

2. Binary Search Implementasyonu

```
public override int Search(int[] items, int searchKey)
{
    int baslangic = 0, bitis = items.GetUpperBound(0), orta = baslangic + bitis / 2;
    while (baslangic <= bitis)
    {
        orta = (baslangic + bitis) / 2;
        if (items[orta] > searchKey)
        {
            bitis = orta - 1;
        }
        else if (items[orta] < searchKey)
        {
            baslangic = orta + 1;
        }
        else
        {
            return orta;
        }
    }
    return -1;
}
```

Binary Search vs. Linear Search

17 elemanlı bir dizide 37 sayısını arıyoruz…



Binary Search vs. Linear Search