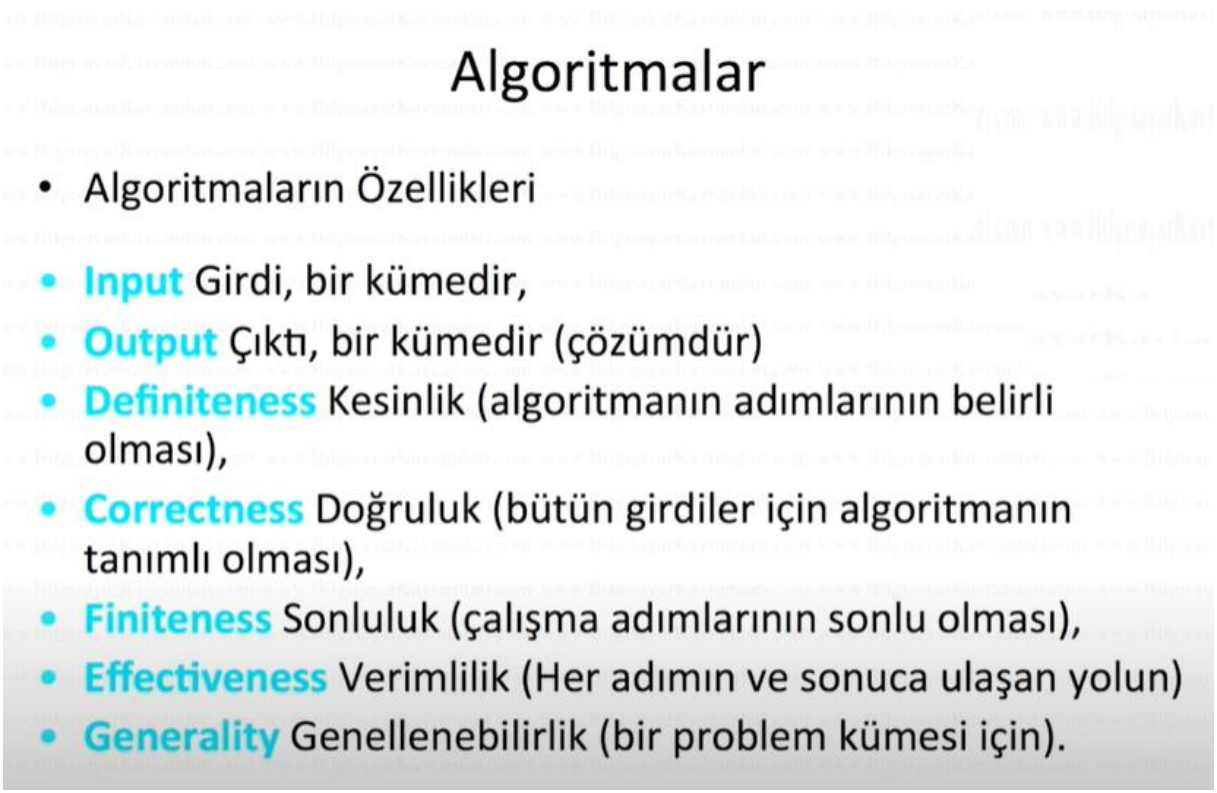
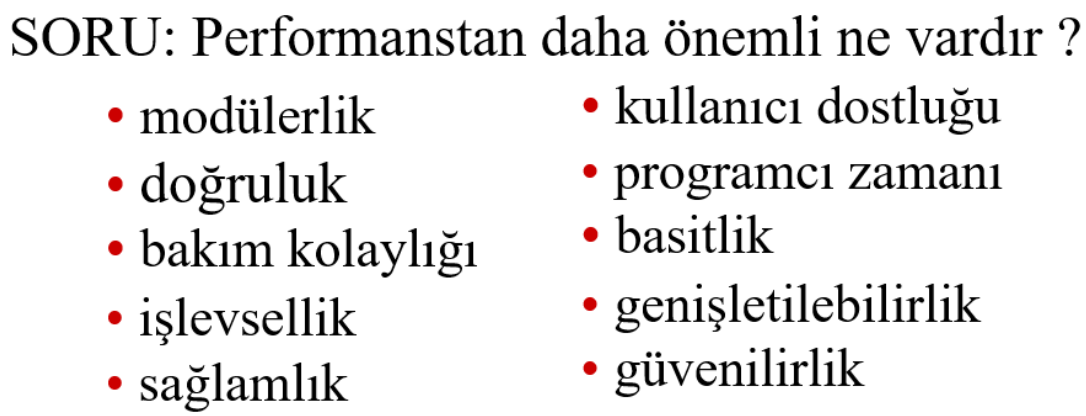


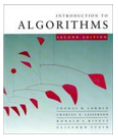


Insertion sort, bir sıralama algoritmasıdır ve temelde bir liste veya diziyi sıralamak için kullanılır. Bu algoritma, listeyi sıralı ve sırasız olarak iki bölüme ayırır. Sırasız bölümden elemanlar sıralı bölüme doğru tek tek yerleştirilirken, her adımda bir elemanın yerine yerleştirme işlemi yapılır.

- İlk adımda, sıralanmış bölümde sadece bir eleman (genellikle ilk eleman) bulunur ve sırasız bölümde diğer elemanlar vardır.

- Sırasız bölümdeki bir elemanı alın ve bu elemanı sıralanmış bölümdeki uygun konumuna eklemeye çalışın.

- Sırasız bölümdeki elemanı, sıralanmış bölümdeki elemanlarla karşılaştırın ve uygun konumunu bulana kadar yer değiştirin.
- Sırasız bölümdeki diğer elemanları sırasıyla alın ve her birini uygun konumlarına eklemeye devam edin.
- Tüm elemanlar sırasız bölümden sıralanmış bölüme eklenene kadar bu işlemi tekrarlayın.



Araya yerleştirme sıralaması çözümlemesi

En kötü durum: Giriş tersten sıralıysa.

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2)$$ [aritmetik seri]

Ortalama durum: Tüm permutasyonlar eşit olasılıklı.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

Araya yerleştirme sıralaması hızlı bir algoritma mıdır ?

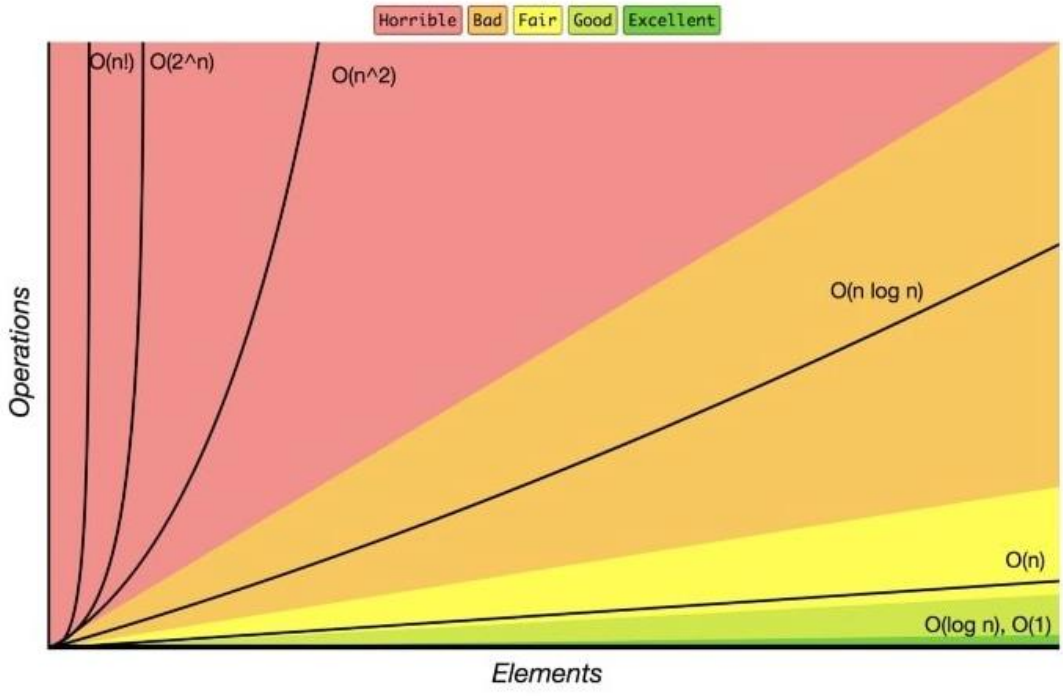
- Küçük n değerleri için olabilir.
- Büyük n değerleri için asla!

Örnek 1

- Aşağıdaki algoritma ne işe yarar?
- **Procedure** gizemli(a_1, a_2, …, a_n: integers)
- m := 0
- **for** i := 1 to n-1
- **for** j := i + 1 to n
- **if** $|a_i - a_j| > m$ **then** m := $|a_i - a_j|$
- {m verilen girdideki herhangi iki sayı arasındaki en uzak mesafeyi verir}
- Karşılaştırma: n-1 + n-2 + n-3 + … + 1
- = $(n-1)n/2 = 0.5n^2 - 0.5n$
- Zaman karmaşıklığı $O(n^2)$.

Örnek 2

- Aynı problemi çözen farklı bir algoritma
- **procedure** max_diff(a_1, a_2, …, a_n: integers)
- min := a1
- max := a1
- **for** i := 2 to n
- **if** a_i < min **then** min := a_i
- **else if** a_i > max **then** max := a_i
- m := max - min
- Karşılaştırma Sayıları: 2n - 2
- Zaman Karmaşıklığı O(n).



Big-O Karmaşıklık Grafiği

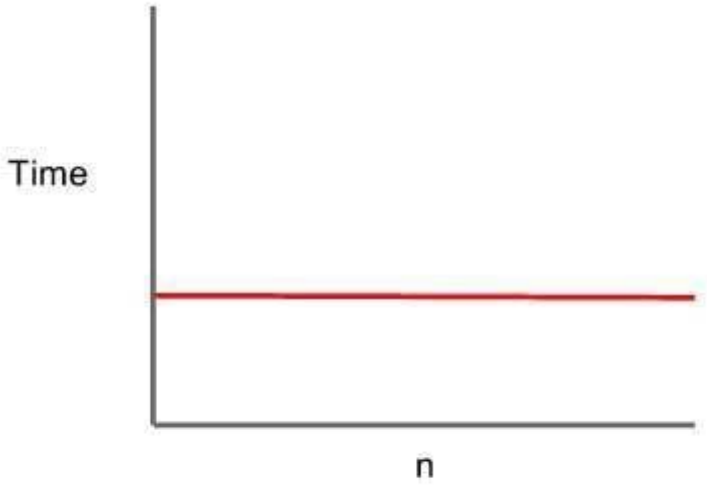
Big O Notation	Computations for 10 elements	Computations for 100 elements	Computations for 1000 elements
O(1)	1	1	1
O(log N)	3	6	9
O(N)	10	100	1000
O(N log N)	30	600	9000
O(N^2)	100	10000	1000000
O(2^N)	1024	1.26e+29	1.07e+301
O(N!)	3628800	9.3e+157	4.02e+2567

Big-O Karmaşıklıklarının Sıralanması

Sabit Karmaşıklık (Constant Complexity) - O(1)

Sabit karmaşıklıkta algoritmaya girilen veri seti ne kadar büyük olursa olsun çalışma zamanı ve kullanılan kaynak miktarı sabittir.

- O(1) zamanda çalışır.



Sabit Karmaşıklık(Constant Complexity)

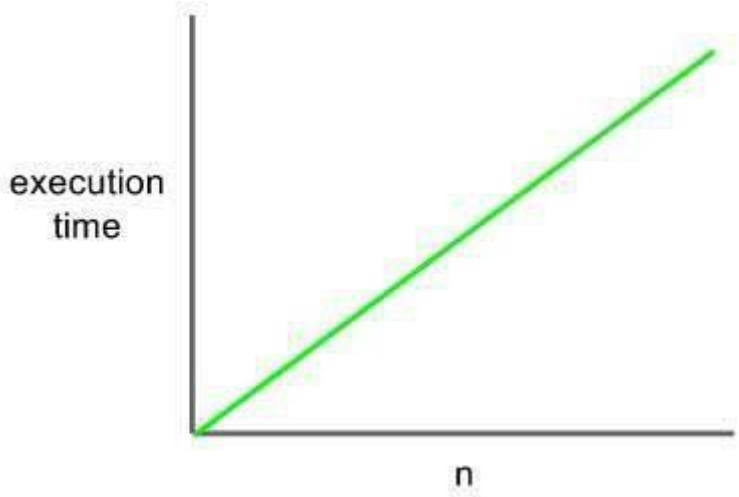
```
var arr = [ 1,2,3,4,5];

arr[1]; // => 2
```

Doğrusal Karmaşıklık (Linear Complexity) - O(N)

Doğrusal karmaşıklıkta algoritmaya girdiğimiz veri setinin büyüklüğü arttıkça çalışma zamanı da doğrusal olarak artar. Yani girilen veri setinin büyüklüğü ile çalışma zamanı arasında doğru orantı vardır.

- O(N) zamanda çalışır.

Örnek olarak elimizde bulunan bir CD yığını düşünelim. Biz bu CD yığınının en altındaki CD ye ulaşmak için CD sayısı kadar zaman harcamalıyız. Eğer CD sayısı artarsa harcadığımız zamanda orantılı olarak artacaktır.



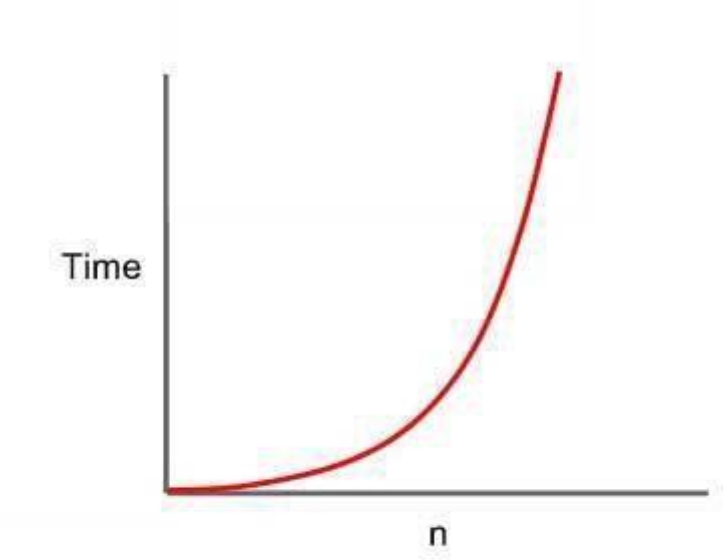
Doğrusal Karmaşıklık(Linear Complexity)

Quadratic Complexity - O(N²)

Basitçe, çalışma zamanı girdi büyüklüğünün karesiyle doğru orantılıdır. Yani eğer girdi büyüklüğü 2 ise 4 işlem, 8 ise 16 işlem gerçekleşir.

- $O(N^2)$ zamanda çalışır.

Genellikle sıralama(sorting) algoritmalarında bu karmaşıklık görülür.



Quadratic Karmaşık(Quadratic Complexity)

İç içe iki tane for döngüsü buna çok iyi bir örnektir.

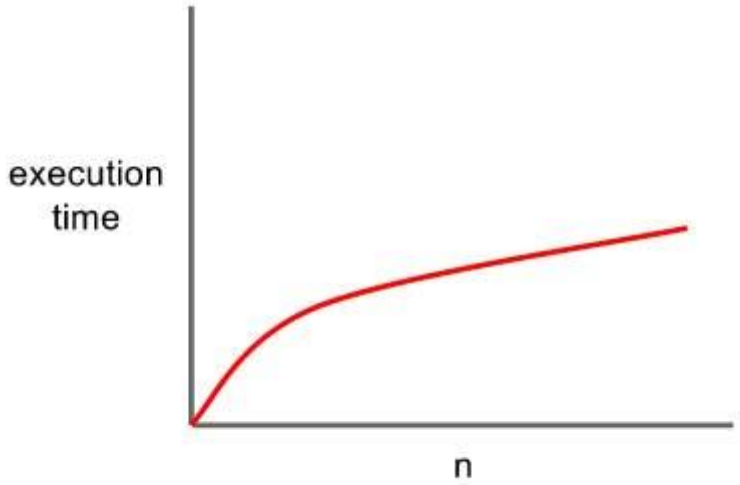
```
for(int i=0; i<arr.length; i++){
    for(int j=0; j<arr.length; j++){
        print("Hello World");
    }
}
```

Tek bir for döngüsünde çalışma zamanı O(N) olur. Ancak iç içe 2 tane for döngüsü kullanıldığı zaman çalışma zamanı $O(N^2)$'dir.

Logaritmik Karmaşıklık (Logarithmic Complexity) - O(log N)

Logaritmik karmaşıklık genel olarak her seferinde problemi ikiye bölen algoritmalarda karşımıza çıkar. Diğer bir deyişle, algoritmayı çalıştırmak için geçen süre N girdi boyutunun logaritması ile orantılıysa bu algoritma logaritmik zaman karmaşıklığına sahiptir.

- O(logN) zamanda çalışır.



Logaritmik Karmaşıklık(Logarithmic Complexity)

```
for(let i=n; i>1; i/=2){
    console.log(i);
}
```

Linearitmik Complexity - O(NlogN)

Linearitmik zaman karmaşıklığı, doğrusal bir algoritmadan biraz daha yavaştır, ancak yine de ikinci dereceden bir algoritmadan çok daha iyidir.

- O(NlogN) zamanda çalışır.

[Understanding Time Complexity with Simple Examples - GeeksforGeeks]
[Zaman Karmaşıklığı Analizi Üzerine Alıştırma Soruları - GeeksforGeeks]

Linkleri inceleyebilirsiniz.