

ALGORİTMALAR

- Assoc. Prof. Mehmet Akif Cifci is an accomplished Associate Professor in computer science with over 15 years of experience in the field. He is currently affiliated with TU Wien in Austria and has held previous positions at Bandırma Onyedi Eylül University in Turkey.





Algoritma Nedir?

Algoritmalar bir problemi çözmek veya belirli bir görevi yerine getirmek için kullanılan adım adım talimatların bir koleksiyonudur. Algoritmaların belirli özellikleri vardır ve geniş bir kullanım alanı bulunmaktadır. Bu nedenle, algoritmaların anlaşılması ve etkili bir şekilde kullanılması bilgisayar biliminin ve diğer alanların temelini oluşturur.

Algoritma Özellikleri

- Açıklık:
Bir algoritmanın anlaşılabilir ve açık bir şekilde tanımlanması gerekmektedir.
 - Sonluluk:
Her adımdan sonra algoritmanın bir sonraki adıma geçmesi ve sonunda durması gerekmektedir.
-

Algoritma Özellikleri

- Etkinlik:

Algoritmanın her adımının belirli bir süre içinde tamamlanması gerekmektedir.

- Çıktı Üretme:

Herhangi bir belirli girdi için algoritmanın doğru bir çıktı üretmesi gerekir.

Algoritma Kullanım Alanları

- Algoritmaların geniş bir kullanım alanı vardır. Örneğin, **matematikte**, bir problemi çözmek için kullanılan bir algoritma olabilirler. Bilgisayar biliminde, bir programın nasıl yazılacağını belirlemek için algoritmalar kullanılır. **Finansal analizde** veya **veri bilimi alanında** da algoritmalar yaygın olarak kullanılır. Dolayısıyla, algoritmaların sadece bir problemi çözmekle kalmayıp aynı zamanda birçok alanda kullanılan önemli araçlar olduğunu söyleyebiliriz.
-

Algoritma Tasarımı Temelleri

Algoritma tasarımı, bir problemi çözmek için algoritma oluşturma sürecidir. Bu süreç;

- Problemi anlama
- Algoritma tasarlama
- Algoritmanın analizi
- Algoritmayı uygulama

adımlarını içerir.



Algoritma Tasarımında Dikkat Edilmesi Gerekenler

- **Verimlilik:**

Algoritmanın gereksiz yere kaynak tüketmeden verimli bir şekilde çalışmasını sağlar.

- **Doğruluk:**

Algoritmanın problemin gereksinimlerini tam olarak karşıladığından emin olmayı sağlar.

- **Anlaşılabilirlik:**

Algoritmanın kolayca okunabilir ve anlaşılabilir olmasını sağlar, bu da algoritmanın diğer geliştiriciler tarafından kullanılabilirliğini artırır.

Akış Diyagramları

Sürecin akış hattı



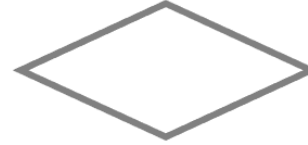
Başlangıç / Bitiş



Süreç adımı



Karar



Giriş çıkış



Belge



Önden
tanımlanmış işlem



Hazırlık



Sayfa dışı
bağlantı/konektör





Akış Diyagramları

Akış diyagramları, bir algoritmanın adımlarını görsel olarak temsil etmek için kullanılan bir tekniktir. Bu diyagramlar, çizgiler, kareler, elipsler ve diğer semboller kullanılarak algoritmanın akışını gösterir.



Pseudo-code

Pseudo-code, bir algoritmanın programlama diline bağlı olmayan, insanların anlayabileceği basit bir dilde ifade edilmesini sağlayan bir tekniktir. Pseudo-code, algoritmanın mantığını anlamak ve geliştirmek için kullanılır, ancak doğrudan bir programlama dili olarak uygulanamaz. Bu nedenle, pseudo-code'u gerçek bir programlama diline dönüştürmek gerekir.

Pseudo-code ve Akış Diyagramları

Akış diyagramları ve pseudo-code, algoritmaların görsel ve metinsel temsillerini sağlayan önemli tekniklerdir. Akış diyagramları, algoritmanın adımlarını görsel olarak temsil ederken, pseudo-code daha anlaşılabilir bir dilde algoritmayı ifade eder. Bu teknikler, algoritmaların **geliştirilmesi**, **anlaşılması** ve **optimize edilmesi** süreçlerinde önemli bir rol oynar.

Pseudo-code Nasıl Yazılır?

Pseudo-code yazarken, algoritmanın adımlarını basit bir dilde ifade etmek önemlidir. Her bir adım, algoritmanın bir parçasını veya belirli bir işlemi temsil eder. Bu adımlar genellikle doğal dil kullanılarak yazılır ve algoritmanın mantığını net bir şekilde ortaya koyar. Pseudo-code'un anlaşılabilirliği ve okunabilirliği için, açıklayıcı değişken ve ifadeler kullanılmalıdır.

Pseudo-code Nasıl Yazılır?

Pseudo-code yazarken, algoritmanın akışını ve kontrol yapılarını göz önünde bulundurmak önemlidir. Bu, döngüler, karar yapıları ve işlem sırasını içerir. Bu yapılar, algoritmanın belirli koşullara veya durumlara nasıl tepki vereceğini ve nasıl işleyeceğini belirler. Pseudo-code'un bu yapıları kullanarak algoritmanın akışını açıklıkla ifade etmesi gerekir.

Pseudo-code

Örnek-1

Bu pseudo-code, bir karenin alanını hesaplayan basit bir algoritmayı temsil eder. Algoritmanın adı KareAlanıHesapla olarak belirlenmiştir. Algoritma, kullanıcıdan karenin bir kenar uzunluğunu girmesini ister. Ardından, girilen kenar uzunluğunu kullanarak karenin alanını hesaplar. Son olarak, hesaplanan alanı ekrana yazdırır.

java

Copy code

ALGORİTMA KareAlanıHesapla

// Girdi: Bir kenar uzunluğu

// Çıktı: Kare alanı

BAŞLA

kenarUzunluğu = kullanıcıdan_al("Karenin kenar uzunluğunu giriniz")

kareAlanı = kenarUzunluğu * kenarUzunluğu

ekrana_yaz("Karenin alanı:", kareAlanı)


SON

Pseudo-code

Örnek-1

Pseudo-code, gerçek bir programlama diline bağlı olmadığı için, algoritmanın mantığını anlamak için basit bir dilde ifade edilmiştir. Bu pseudo-code'u gerçek bir programlama diline dönüştürmek mümkündür.

python

 Copy code

```
def kare_alanı_hesapla():  
    kenar_uzunluğu = float(input("Karenin kenar uzunluğunu giriniz: "))  
    kare_alanı = kenar_uzunluğu * kenar_uzunluğu  
    print("Karenin alanı:", kare_alanı)  
  
kare_alanı_hesapla()
```



ALGORİTMA ANALİZİ & KARMAŞIKLIK



Algoritma Analizi ve Karmaşıklık

- **Algoritma Analizi:**

Bir algoritmanın performansını ve verimliliğini inceleyen bir süreçtir. Bu analiz, algoritmanın işlemci zamanı, bellek kullanımı ve diğer kaynaklar üzerindeki etkisini değerlendirir.

- **Karmaşıklık:**

Bir algoritmanın çalışma süresinin veya kullanılan kaynakların miktarının, problemin boyutu ile nasıl değiştiğini ifade eder. Karmaşıklığı belirlemek için zaman karmaşıklığı ve alan karmaşıklığı gibi kriterler kullanılır. İyi bir algoritma, en kötü durumda bile kabul edilebilir bir performans gösterir.



İyi Bir Algoritma

- en kötü durumda bile kabul edilebilir bir performans gösterir.
 - problem boyutu arttıkça performansının kötüleşmediği veya en azından kabul edilebilir bir şekilde kötüleştiği bir özellik gösterir.
 - tasarımı, karmaşıklık analizi ile başlar ve algoritmanın etkili bir şekilde performans göstermesini sağlar.
-

Karmaşıklık Örneği-1

Bir karmaşıklık örneği olarak, bir algoritmanın **bir dizideki en büyük veya en küçük öğeyi bulma** işlemini ele alalım. Bu algoritma, bir dizi üzerinde dolaşarak her öğeyi diğerleriyle karşılaştırarak en büyük veya en küçük öğeyi bulabilir. Bu tür bir algoritmanın zaman karmaşıklığı $O(n)$ olacaktır çünkü en kötü durumda, dizinin tüm öğeleriyle karşılaştırma yapılması gerekecektir.

Karmaşıklık Örneği-1

```
ALGORİTMA EnBuyukBul
```

```
BAŞLA
```

```
    enBuyuk = dizinin_ilk_öğesi
```

```
    DÖNGÜ i = 1'den başlayarak dizinin_son_öğesine_kadar
```

```
        EĞER dizinin_i.öğesi > enBuyuk
```

```
            enBuyuk = dizinin_i.öğesi
```

```
    SONSUZ
```

```
    ekrana_yaz("Dizinin en büyük öğesi:", enBuyuk)
```

```
SON
```

Karmaşıklık Örneği-1

Ancak, daha etkili bir algoritma olan sıralama algoritmaları kullanarak bu işlemi gerçekleştirebiliriz. Örneğin, hızlı sıralama algoritması $O(n \log n)$ karmaşıklığına sahiptir. Bu algoritma, diziyi bölerek ve her alt dizide sıralama işlemini gerçekleştirerek çalışır. Bu şekilde, dizideki en büyük veya en küçük öğeyi bulmak için bir dizi sıralandığında, sıralı dizinin ilk veya son öğesine bakmak yeterli olacaktır. Bu durumda, sıralama algoritmasının zaman karmaşıklığı daha düşüktür ve algoritmanın performansı daha iyidir.



Bir algoritma aşağıdaki davranışları sergileyebilir;

- Hızlı olup ve daha az bellek alanına ihtiyaç duyabilir.
 - Hızlı olup ve daha çok bellek alanına ihtiyaç duyabilir.
 - Yavaş olup ve daha az bellek alanına ihtiyaç duyabilir.
 - Yavaş olup ve daha çok bellek alanına ihtiyaç duyabilir.
-

Bir algoritmanın analizini yaparken aşağıdaki sorulara cevap aranır:

- Çıktının üretilmesi veya hesaplanması ne kadar zaman alır?
 - Sorunu çözmek için ne kadar hafıza gerekir?
 - Veri girişi arttığında nasıl davranır?
 - Daha hızlı ya da daha yavaşlıyor mu? Daha yavaşsa, ne kadar yavaş? Giriş boyutunun iki katı için dört kat yavaşlıyor mu?
 - Veri girişi büyüklüğü ile zamanın bir ilişkisi var mı?
-

Yürütme Zamanı ve Zaman Karmaşıklığı (Running Time)

- Algoritmanın belirli bir işleme veya eyleme kaç kez gereksinim duyulduğunu gösteren bağıntıdır ve $T(n)$ ile gösterilir.
 - Temel hesap birimi olarak, programlama dilindeki deyimler seçilebildiği gibi döngü sayısı, toplama işlemi sayısı, atama sayısı, dosyaya erişme sayısı gibi işler de temel hesap birimi olarak seçilebilir.
-

Yürütme Zamanı ve Zaman Karmaşıklığı (Running Time)

- Yürütme zamanı bağıntısı fiziksel gerçeğe yakın bir sonuç verir, ancak sapmalar da olabilir.
 - Kabul edilen temel hesap birimi tüm hesaplar için aynı olmayabilir.
 - Örneğin: Bir tam sayı sayacın bir artırılmasıyla iki gerçel sayının çarpımı maliyeti farklı olabilir veya iki tam sayıyı karşılaştırmak ile iki diziyi karşılaştırma maliyetleri farklı olur.
-

Karmaşıklık Analizi Nedir?

- Algoritmaların/fonksiyonların kodlamadan önce davranışlarının ortaya konulması işlemidir.
 - Algoritma analizi
 - Programları karşılaştırmak için yazmak mantıklı değildir.
 - Zaman alıcı bir işlemdir.
 - Programcının tecrübesine göre programlarken etkili veri yapılar kullanılabilir.
 - Büyük verilerde algoritma çalışma zamanı önemlidir.
-

Karmaşıklık Analizi Nedir?

- Bir problemi çözerken olası tüm algoritmaları ele almamız, analiz etmemiz ve uygun olanı seçmemiz gerekiyor. Analiz işlemi iki temel kriter baz alınarak yapılır. Bunlar;
 - Time Complexity (Zaman Karmaşıklığı)
 - Space Complexity (Alan Karmaşıklığı)
-

Zaman Karmaşıklığı

- Bir algoritmanın verilen bir girdi için çıktı üretirken harcadığı veya ihtiyaç duyduğu süredir. Bu süre ne kadar kısa olursa algoritmanın performansı o kadar iyi olur. Zaman karmaşıklığını bir algoritmanın performansı olarak düşünebiliriz.
 - Bir bilgisayarın donanımı ne kadar iyi olursa olsun, her zaman için en performanslı algoritmayı seçmek mantıklı bir yaklaşım olacaktır. Donanımda yapılan iyileştirmeler ile beraber algoritmaların da iyileştirilmesi sağlanarak daha performanslı sistemler oluşturulabilir.
-

Alan Maliyeti ve Alan Karmaşıklığı (Space Cost)

- Alan maliyeti (space cost) : Bir programın veya algoritmanın işlevini yerin getirebilmesi için gerekli bellek alanını veren bir bağıntıdır.
 - Alan karmaşıklığı (space complexity) : Alan karmaşıklığı, eleman sayısı n 'nin çok büyük değerleri için bellek alanı gereksinimin artış mertebesini gösteren asimptotik ifadedir. Zaman karmaşıklığındaki ifadeler kullanılır.
-

Asimptotik Karmaşıklık

- Algoritmalarda t (süre) ve n (giriş boyutu) arasındaki ilişki çoğu zaman çok karmaşıktır.
 - Fonksiyon içerisindeki önemsiz kısımlar ve katsayılar atılarak basitleştirilir ve gerçek fonksiyona göre yaklaşık bir değer bulunur.
-

Asimptotik Karmaşıklık

- Elde edilen bu yeni etkinlik ölçümüne “Asymptotic Complexity” denir.
 - Genellikle girişin büyümesine bağlı olarak fonksiyonun büyümesinde en büyük etkiye sahip olan parametre alınır.
-

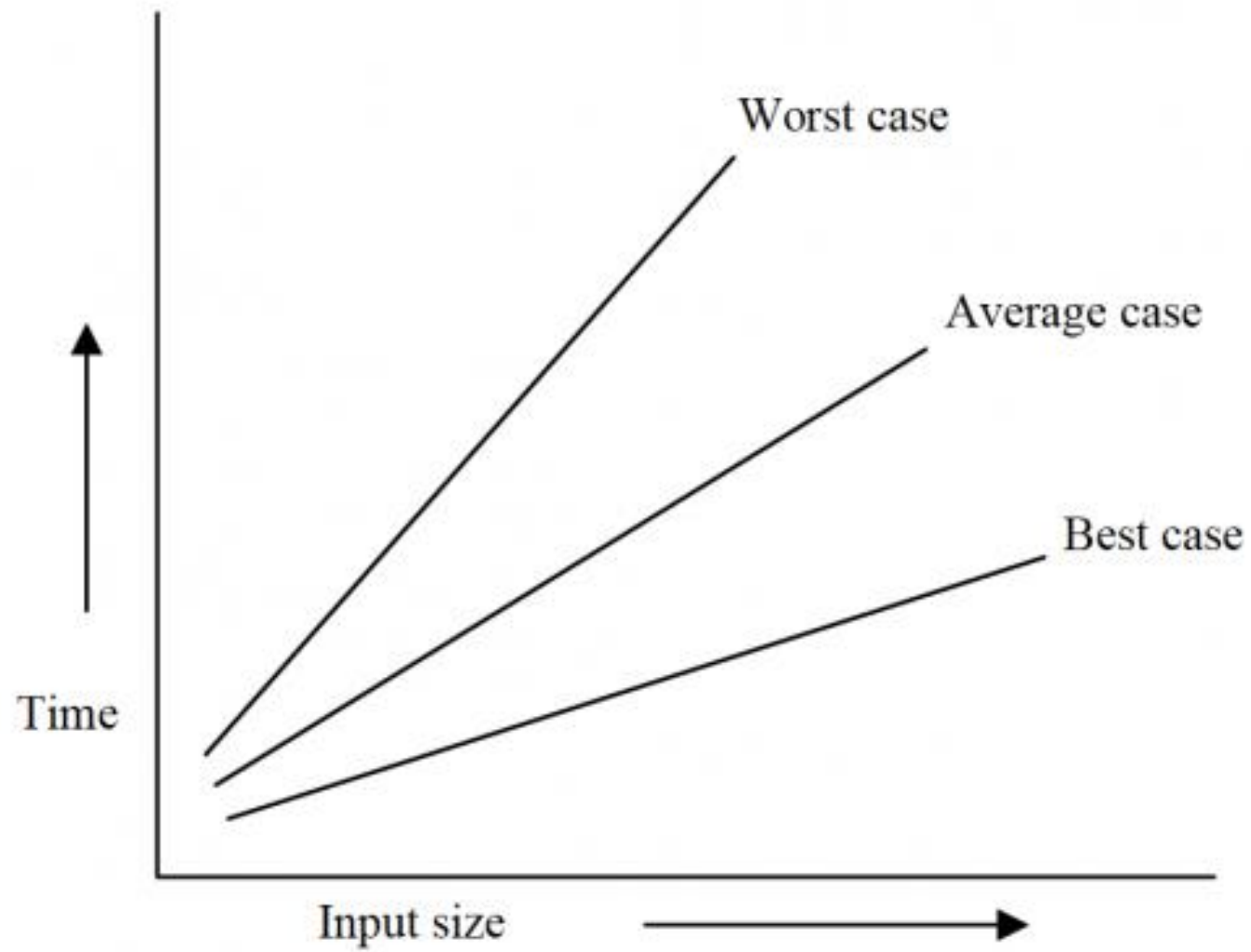
Asimptotik Karmaşıklık

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		



Asimptotik Analiz

- Amaç: detaylardan kurtularak çalışma süresi analizini basitleştirmek
 - Sayılar için “rounding” işlemi: $1,000,001 \sim 1,000,000$
 - Fonksiyonlar için “rounding” işlemi: $3n^2 \sim n^2$
 - Niteliğini belirlemek (Capturing the essence): belirlenen limit içerisinde girişin boyutuna göre algoritmanın çalışma süresinin nasıl arttığının bulunması.
 - Algoritmaların best, worst ve average case çalışma süreleri bir fonksiyonla ifade edilebilir.
-



Worst Case (En Kötü)

- Algoritma çalışmasının en fazla sürede gerçekleştiği analiz türüdür. En kötü durum, çalışma zamanında bir üst sınırdır ve o algoritma için verilen durumdan “daha uzun sürmeyeceği” garantisi verir.
 - Çıktının üretilmesi için bir algoritmanın ihtiyaç duyduğu maksimum süreyi tanımlar.
-

Best Case (En İyi)

- Algoritmanın en kısa sürede ve en az adımda çalıştığı giriş durumu olan analiz türüdür. Bir alt sınırdır.
 - Çıktının üretilmesi için bir algoritmanın gerektirdiği minimum süreyi tanımlar.
-

Average Case (Ortalama)

- Algoritmanın ortalama sürede ve ortalama adımda çalıştığı giriş durumu olan analiz türüdür.
 - Farklı boyutlardaki girdiler için çıktı üretmekte bir algoritmanın ihtiyaç duyduğu ortalama süreyi tanımlar.
-

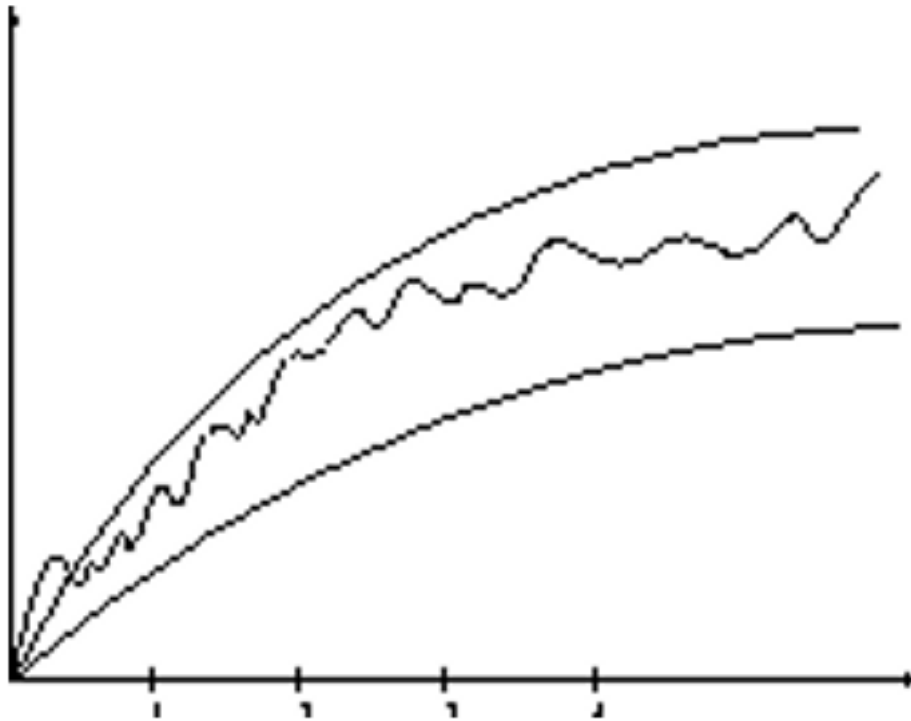
Asimptotik Notasyonlar

- Asimptotik Notasyon, eleman sayısı n 'nin sonsuza gitmesi durumunda algoritmanın, benzer işi yapan algoritmalarla karşılaştırmak için kullanılır.
 - Algoritmamıza gelen girdi sayısı çok büyüdüğünde ve sonsuza giderken algoritmamızın nasıl davranacağını gösterir. Eğer girdi boyutu ile Big-o sonucu da çok fazla büyüyorsa ise algoritma verimsizdir diyebiliriz.
 - Algoritmanın girdi boyutu büyüdükçe küçük kalan Big-o notasyonuna sahip olan algoritma daha iyidir.
-



Asimptotik Notasyonlar

- Big O: Asimptotik üst sınır
 - Omega: Asimptotik alt sınır
 - Teta: Asimptotik alt ve üst sınır
-



Üst limit – $O(n)$

Algoritmanın gerçek fonksiyonu

Alt limit- $\Omega(n)$

Büyük O Notasyonu (Big O Notation)

- Big O notasyonu ilk olarak 1894 yılında Alman matematikçi Bachmann tarafından kullanılmış ve Landau tarafından da yaygınlaştırılmıştır. Bundan dolayı adına Landau notasyonu da denir.
 - Sabit zamanlı ifadeler $O(1)$ ile gösterilir. Örnek olarak atama işlemleri.
-

Büyük O Notasyonu (Big O Notation)

- $T(n) = O(f(n))$
 - c ve n_0 şeklinde pozitif sabitlerimiz olduğunu düşünelim. $n \geq n_0$ ifadesini sağlayan tüm değerler için $T(n) \leq c \cdot f(n)$ dir.
-

Büyük O Notasyonu (Big O Notation)

- If-else durumunda, ifadenin if veya else bloğundaki hangi ifade karmaşıklık olarak daha büyükse O fonksiyonu o değeri döndürür.
 - Örneğin: $f(n) = n^4 + 100n^2 + 10n + 50$ algoritma fonksiyonunda $g(n) = n^4$ olur
 - $f(n)$ 'nin maksimum büyüme oranı $g(n) = n^4$ $O(n^4)$
-

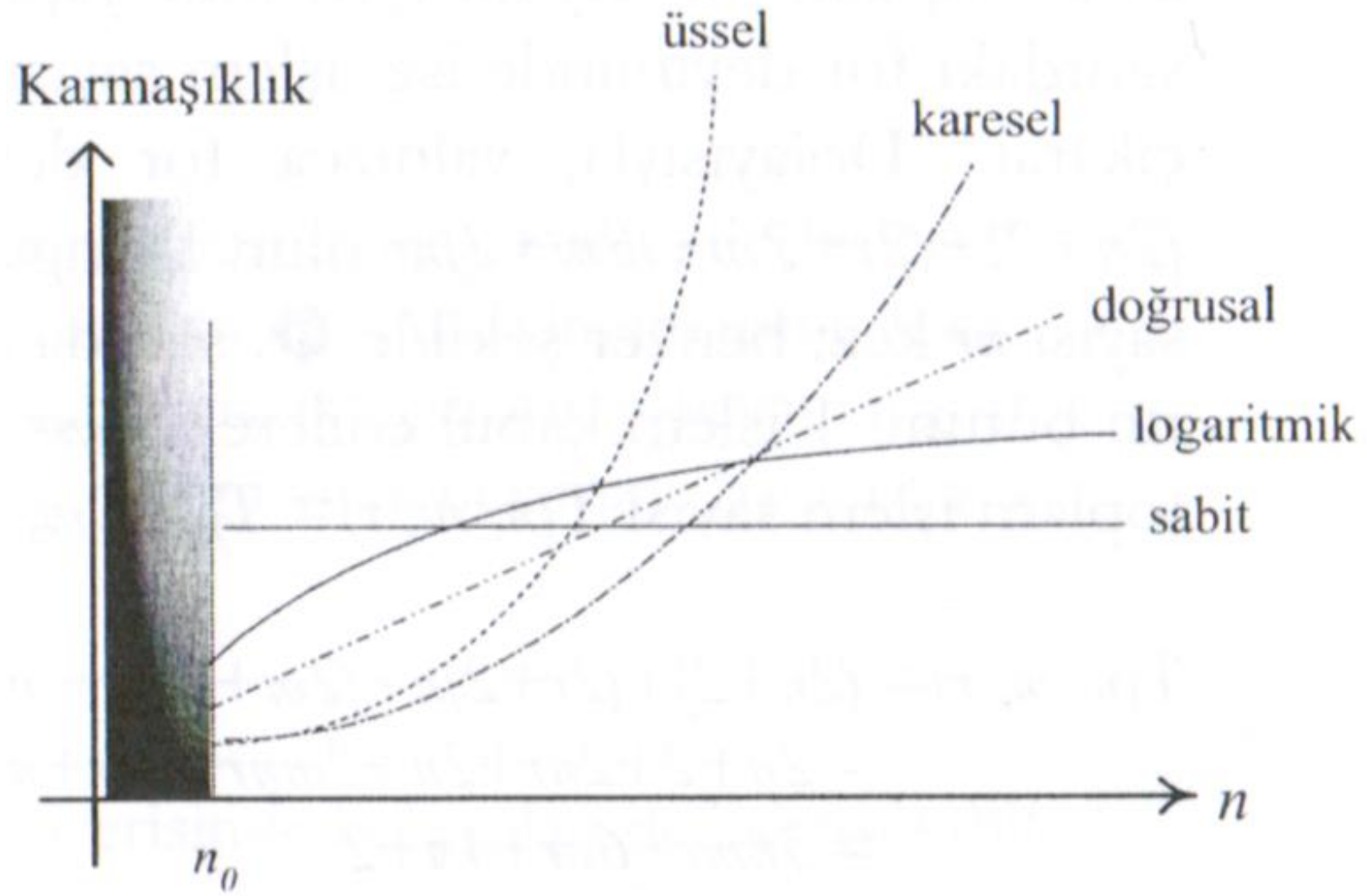


Büyük O Notasyonu (Big O Notation)

- Donanım, işletim sistemi, derleyici ve algoritma detaylarından bağımsız, sadece büyük n değerlerine odaklanıp, sabitleri göz ardı ederek daha basit bir şekilde algoritmaları analiz etmemize ve karşılaştırmamızı sağlar.
 - Big O-notasyonu gösteriminde bir fonksiyonun düşük n değerlerindeki performansı önemsiz kabul edilir.
-

BÜYÜME ORANI TABLOSU

ZAMAN KARMAŞIKLIĞI	AÇIKLAMA	ÖRNEK
$O(1)$	<u>Sabit</u> : Yenilmez!	Bağlı listeye ilk eleman olarak ekleme yapma
$O(\log N)$	<u>Logaritmik</u> : Problemi küçük veri parçalarına bölen algoritmalarda görülür.	Binary search, tree veri yapısı üzerinde arama
$O(N)$	<u>Lineer – doğrusal</u> : Hızlı bir algoritmadır. N tane veriyi girmek için gereken zaman	Sıralı olmayan bir dizide bir eleman arama
$O(N \log N)$	<u>Doğrusal çarpanlı logaritmik</u> : Problemi küçük veri parçalarına bölen ve daha sonra bu parçalar üzerinde işlem yapan.	N elemanı böl-parçala-yönet yöntemiyle sıralama. Quick Sort. Çoğu sıralama algoritması bu gruptadır.
$O(N^2)$	<u>Karesel</u> : Veri miktarı azsa uygun ($n < 1000$)	Bir grafikte iki düğüm arasındaki en kısa yolu bulma veya Bubble Sort.
$O(N^3)$	<u>Kübik</u> : veri miktarı azsa uygun ($n < 1000$)	Ardarda gerçekleştirilen lineer denklemler
$O(2^n)$	<u>İki tabanında üssel</u> . Veri çok azsa uygun ($N \leq 20$)	Hanoi'nin Kuleleri problemi





Çalışma Zamanı

$O(N)$

```
MaxSubsequenceSum(const int A[], int n)
ThisSum=MaxSum=0;
for (j=0; j<N; j++)
    ThisSum+=A[j];
    if (ThisSum<MaxSum)
        MaxSum=ThisSum;
    else if (ThisSum<0)
        ThisSum=0;
Return MaxSum;
```

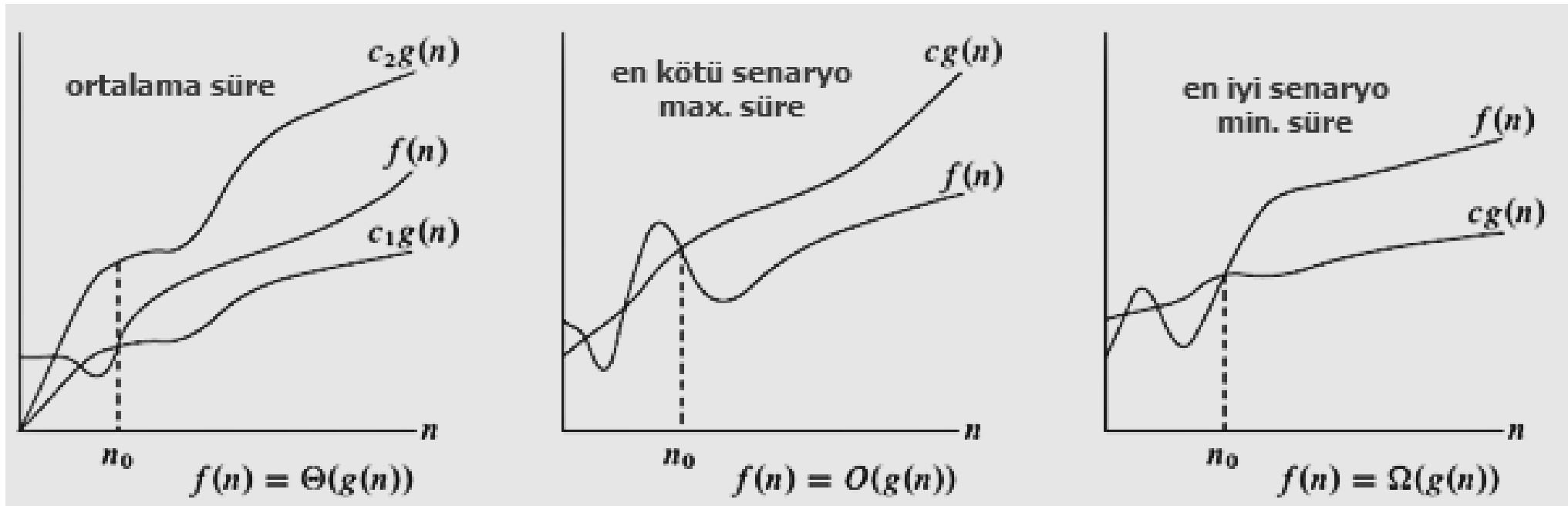

Ω Notasyonu (Ω Notation)

- $T(n) = \Omega(f(n))$
 - c ve n_0 şeklinde pozitif sabitlerimiz olduğunu düşünelim. $n \geq n_0$ ifadesini sağlayan tüm değerler için $T(n) \geq c \cdot f(n)$ dir.
-

Θ Notasyonu (Θ Notation)

- $T(n) = \Theta(f(n))$
 - c_1, c_2 ve n_0 şeklinde pozitif sabitlerimiz olduğunu düşünelim $n \geq n_0$ ifadesini sağlayan tüm değerler için $c_1 * f(n) \leq T(n) \leq c_2 * f(n)$ dir.
-

$f(n) = O(g(n))$	\equiv	$f \leq g$
$f(n) = \Omega(g(n))$	\equiv	$f \geq g$
$f(n) = \Theta(g(n))$	\equiv	$f = g$



ALGORİTMA TÜRLERİ



Algoritma Türleri

- Bilgisayar bilimleri ve matematikte, algoritmalar bir problemi çözmek veya belirli bir görevi yerine getirmek için adımların mantıksal bir dizisini ifade eder. Farklı algoritma türleri, çeşitli problemleri ele almak ve etkili bir şekilde çözmek için farklı yaklaşımlar sunar.
-



Brute Force (Kaba Kuvvet) Algoritmalar

- Bu algoritma türü, her olası çözümü deneyerek problemin çözümünü bulma yöntemidir. Problemin boyutu arttıkça hesaplama süresi ve kaynak kullanımı da artar. Ancak, basit problemlerde etkili bir çözüm sunabilir.
 - Kullanım Örnekleri: Bir şifrenin tüm olası kombinasyonlarını deneyerek şifreyi kırmak veya tüm alt kümeleri listelemek.
-



Recursive (Özyinelemeli) Algoritmalar

- Bu tür algoritmalar, çözümü arama sürecinde kendini tekrar çağırarak çalışır. Karmaşık problemleri daha küçük ve daha anlaşılır alt problemlere indirgeyerek çözümü elde eder. Fibonacci sayıları veya ağaç yapısı dolaşma gibi problemler için kullanışlıdır.
 - Kullanım Örnekleri: Fibonacci sayılarını hesaplamak, bir dizinin elemanlarını tersine çevirmek veya bir ağaç yapısını dolaşmak.
-



Divide and Conquer (Böl ve Fethet) Algoritmalar

- Bu algoritma türü, problemleri daha küçük parçalara bölerek çözer ve ardından bu parçaların çözümlerini birleştirerek orijinal problemin çözümüne ulaşır. Sıralama ve birleştirme işlemlerinde yaygın olarak kullanılır.
-



Divide and Conquer (Böl ve Fethet) Algoritmalar

- **Merge Sort (Birleştirme Sıralaması):** Merge Sort, bir diziyi sıralamak için Divide and Conquer algoritması kullanır. Diziyi ikiye böler, her iki parçayı ayrı ayrı sıralar ve ardından sıralanmış alt dizileri birleştirir. Bu sayede daha büyük bir problemi daha küçük ve sıralanmış alt problemlere indirger.
-



Divide and Conquer (Böl ve Fethet) Algoritmalar

- **Quick Sort (Hızlı Sıralama):** Quick Sort da bir sıralama algoritmasıdır ve Divide and Conquer yaklaşımını kullanır. Bir "pivot" elemanı seçer, pivotun solunda pivot'tan küçük, sağında pivot'tan büyük elemanları gruplar. Bu adım diziyi ikiye böler. Sonra her iki gruba da aynı işlemi uygular. Bu işlem adım adım dizinin sıralanmış hâlini oluşturur.
-

Divide and Conquer (Böl ve Fethet) Algoritmalar

- **Binary Search (İkili Arama):** Sıralı bir dizide belirli bir elemanı bulmak için kullanılan Binary Search algoritması da Divide and Conquer yaklaşımını benimser. Diziyi ortadan böler, aranan elemanı ortadaki elemanla karşılaştırır. Eğer ortadaki elemandan büyükse, sağ tarafı incelemeye devam eder; küçükse sol tarafı incelemeye devam eder. Bu şekilde aranan elemanı bulana kadar devam eder.
-



Divide and Conquer (Böl ve Fethet) Algoritmalar

- **Closest Pair of Points (En Yakın Çift Nokta):** 2D uzayda verilen noktalar arasındaki en yakın çift noktayı bulmak için kullanılan bir algoritmadır. Divide and Conquer kullanarak problemi daha küçük alt problemlere böler ve alt problemlerin sonuçlarını birleştirerek en yakın çift noktayı bulur.
-



Backtracking (Geri İzlemeli) Algoritmalar

- Backtracking algoritmaları, tüm olası çözümleri deneyerek ilerler. Ancak, bir adımda yanlış bir sonuca ulaşılması durumunda geri dönüp doğru yolu arar. N-Queen problemleri veya Sudoku çözümü için kullanılabilir.
 - Kullanım Örnekleri: N-Queen probleminde, satranç tahtasına N adet vezir yerleştirme sorununu çözmek veya Sudoku çözümü.
-



Dynamic Programming (Dinamik Programlama)

- Bu algoritma türü, büyük problemleri daha küçük alt problemlere böler ve alt problemlerin çözümlerini hafızasında saklayarak tekrar eden hesaplamaları önler. En kısa yol ve çanta problemleri gibi optimizasyon problemlerinde kullanışlıdır.
 - Kullanım Örnekleri: En kısa yol problemi, çanta problemleri gibi tekrar eden alt problemlerin çözümünden faydalanan problemler.
-

Greedy (Aç Gözlü) Algoritmalar

- Greedy algoritmaları, her adımda en iyi seçeneği yaparak ilerler. Anlık en iyi çözümü seçer, ancak bu, global en iyi çözümü garanti etmez. Minimum para sayısı hesaplama veya minimum ağırlıklı ağaç bulma gibi problemlerde kullanılır.
 - Kullanım Örnekleri: Minimum para sayısı ile verilen bir tutarı vermek için en büyük bozuk parayı seçmek veya Kruskal algoritmasıyla minimum ağırlıklı ağaç bulma.
-



Graf Algoritmaları (Graph Algorithms)

- Graf bir olay veya ifadenin düğüm ve çizgiler kullanılarak gösterilmesine olanak sağlayan bir veri modelidir. Eğer bir problem graf veri modeli şekline benzetilebiliyorsa, probleme algoritmik bir bakış açısı sağlanır ve bu veri modelinde tanımlanmış olan teoriler, algoritmalar ve fonksiyonlar çözüm de kullanılabilir. Bu algoritma gezgin satıcı ve en kısa yolun bulunması gibi problemlere hızlı çözümler sunmaktadır.
-



Randomized (Rastgele) Algoritmalar

- Rastgele algoritmalar, çözümü rastgele seçerek ulaşır. Sorunları belirli bir olasılıkla doğru şekilde çözme avantajı sağlar. QuickSort veya Monte Carlo simülasyonları gibi rastgele örneklemelerde kullanılır.
 - Kullanım Örnekleri: QuickSort algoritması veya Monte Carlo algoritmaları gibi rastgele örneklemelerle doğru sonuca yakınsama.
-

Azalt ve Yönet Algortiması

- Problemden verilen boyutu azaltarak yapılan algoritma tasarımı yöntemidir.
 - Azaltma sabit sayı kadar (genellikle bir) veya sabit çarpan kadar (genelde iki) olabilir.
 - Bir sıralama algoritması olan Eklemeli Sıralama (Insertion Sort) algoritması azalt ve yönet (bir azaltma ile) yöntemi ile tasarlanmıştır.
 - Sıralı dizide bir arama algoritması olan İkili Arama (Binary Search) algoritması azalt ve yönet (yarıya kadar azaltma) yöntemi ile tasarlanmıştır.
-

Dönüştür ve Yönet Algoritması

- Problemi daha kolay çözülebilen başka bir probleme dönüştürerek yapabilen algoritma tasarlama yöntemidir.
 - Örneğin, bir sıralama algoritması olan Yığın Sıralama (Heapsort) algoritması dönüştür ve yönet yöntemi ile tasarlanmıştır.
-

Genetic (Genetik) Algoritmalar

- Bu algoritma türü, biyolojik evrim prensiplerini taklit ederek çalışır. Popülasyondaki bireylerin çaprazlama ve mutasyonlarıyla en iyi çözüme ulaşmaya çalışır. Evrimsel optimizasyon veya yapay zeka problemleri için idealdir.
 - Kullanım Örnekleri: Evrimsel optimizasyon, yapay zeka problemleri veya karmaşık matematiksel optimizasyon problemleri.
-

Heuristic (Heuristik) Algoritmalar

- Heuristic algoritmaları, karmaşık problemleri yaklaşık çözümlerle ele alır ve çözümü hızlı bir şekilde bulma eğilimindedir. Optimal çözüm garantisi vermez, ancak genellikle pratik ve etkili sonuçlar üretir. Yapay öğrenme ve genetik algoritmalar bu kategoridedir.
 - Kullanım Örnekleri: Yapay öğrenme algoritmaları, genetik algoritmalar, simüle edilen tavlama (simulated annealing) gibi optimizasyon problemleri için.
-



Approximation (Yaklaşık) Algoritmalar

- Yaklaşık algoritmaları, NP-zor problemlere yakın çözümler sağlamak için tasarlanmıştır. Optimal çözüm garantisi vermeden, hızlı ve pratik sonuçlar sunarlar. Traveling Salesman Problem gibi zor problemlerde kullanışlıdır.
 - Kullanım Örnekleri: Traveling Salesman Problem gibi NP-zor problemler için, yaklaşık çözümler elde etmek için kullanılabilirler.
-



Network Flow (Ağ Akışı) Algoritmalar

- Bu algoritmalar, ağ üzerindeki akışları belirleyerek en verimli yol veya kaynakları bulmayı amaçlar. Maksimum akış veya minimum kesim problemlerini çözmek için kullanılır.
 - Kullanım Örnekleri: Maksimum akış problemi, minimum kesim problemi gibi ağ optimizasyon problemleri için.
-



Probabilistic (Olabilirlik Tabanlı) Algoritmalar

- Probabilistic algoritmaları, rastgele değişkenler içeren durumlarda kullanılır. Monte Carlo simülasyonları veya rastgele grafik oluşturma gibi uygulamalarda kullanılabilir.
 - Kullanım Örnekleri: Monte Carlo simülasyonları, rastgele grafik oluşturma, simüle edilen tavlama gibi uygulamalar için.
-

String Matching Algoritmaları

- Bu tür algoritmalar, metin içinde belirli bir deseni veya alt-diziyi bulmayı sağlar. Knuth-Morris-Pratt, Boyer-Moore veya Rabin-Karp gibi algoritmalar bu amaçla kullanılır.
 - Kullanım Örnekleri: Knuth-Morris-Pratt, Boyer-Moore, Rabin-Karp gibi string eşleştirme algoritmaları.
-



SIRALAMA ALGORTİMALARI



1. Bubble Sort Algoritması

- Bubble sort, en basit sıralama algoritmalarından biridir. Karşılaştırma temelli olan bu algoritmada, listedeki her bir eleman yanındaki eleman ile karşılaştırılır. Eğer ilk elemanın değeri, ikinci elemanın değerinden büyükse, iki eleman yer değiştirir. Daha sonra ikinci ve üçüncü elemanların değerleri karşılaştırılır. İkinci elemanın değeri üçüncü elemanın değerinden büyükse bu iki eleman yer değiştirir ve bu işlem, tüm liste sıralanana kadar bu şekilde devam eder.
-

Bubble Sort Algoritması (Elemeli Sıralama)

- Örneğin [3,10,5,12,9,20] listesini ele alalım. Bubble sort kullanarak bu listeyi sıralayacak olursak eğer; Karşılaştırmaya öncelikle 3 ve 10 değerlerinin karşılaştırılması ile başlanır. 3, 10'dan küçük olduğu için yer değiştirmezler. Daha sonrasında 10 ve 5 değerleri karşılaştırılır. 10 5'ten büyük olduğu için yer değiştirirler ve iterasyon işlemleri bu şekilde liste elemanları sıralanana kadar devam eder.
-

Bubble Sort Algoritması Nasıl Çalışır?

- Iterasyon : $[3, 5, 10, 9, 12, 20] \rightarrow [3, 5, 10, 9, 12, 20] \rightarrow [3, 5, 10, 9, 12, 20] \rightarrow [3, 5, 9, 10, 12, 20] \rightarrow [3, 5, 9, 10, 12, 20]$
 - Iterasyon : $[3, 5, 9, 10, 12, 20] \rightarrow [3, 5, 9, 10, 12, 20] \rightarrow [3, 5, 9, 10, 12, 20] \rightarrow [3, 5, 9, 10, 12, 20] \rightarrow [3, 5, 9, 10, 12, 20] \rightarrow [3, 5, 9, 10, 12, 20]$
 - Bu örneğimizde, 2 iterasyon gerçekleştirdikten sonra array elemanları sıralanmış olur.
-

First pass

7	6	4	3
---	---	---	---



6	7	4	3
---	---	---	---



6	4	7	3
---	---	---	---



6	4	3	7
---	---	---	---

Second pass

6	4	3	7
---	---	---	---



4	6	3	7
---	---	---	---



4	3	6	7
---	---	---	---

Third pass

4	3	6	7
---	---	---	---

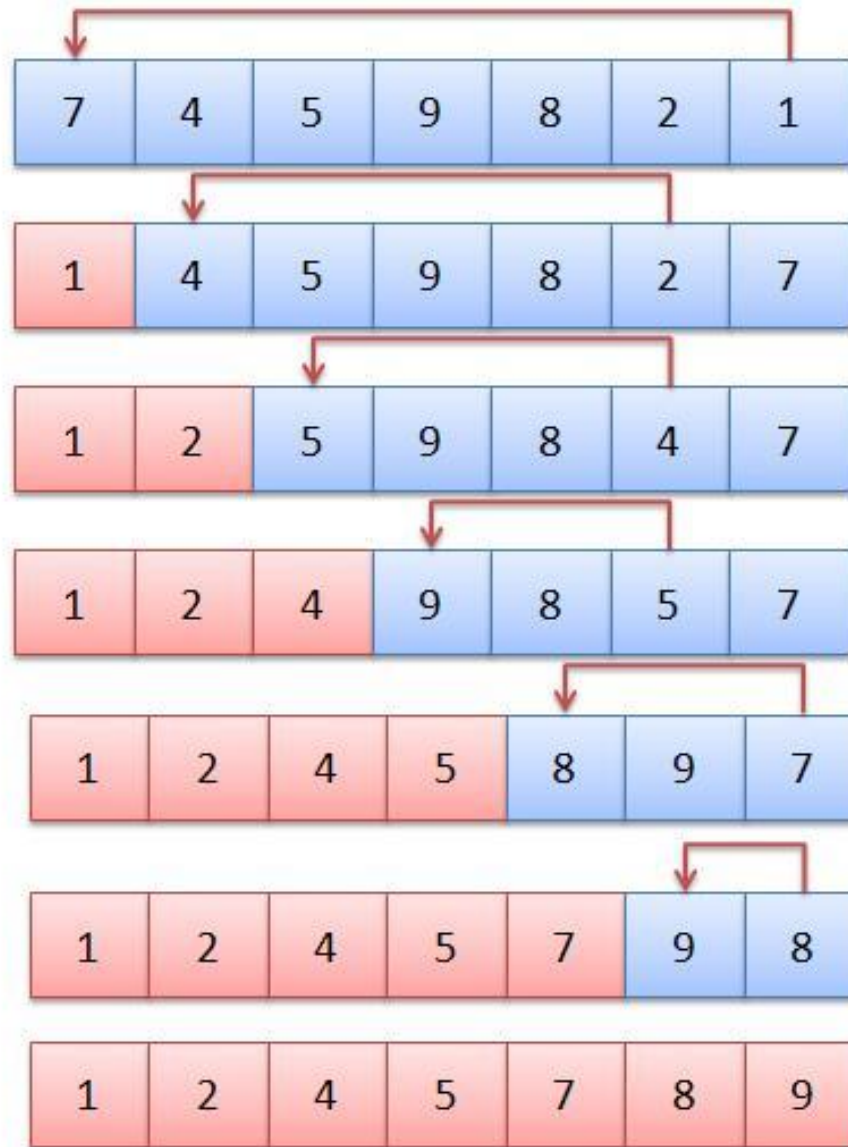


3	4	6	7
---	---	---	---



2. Seçmeli Sıralama (Selection Sort)

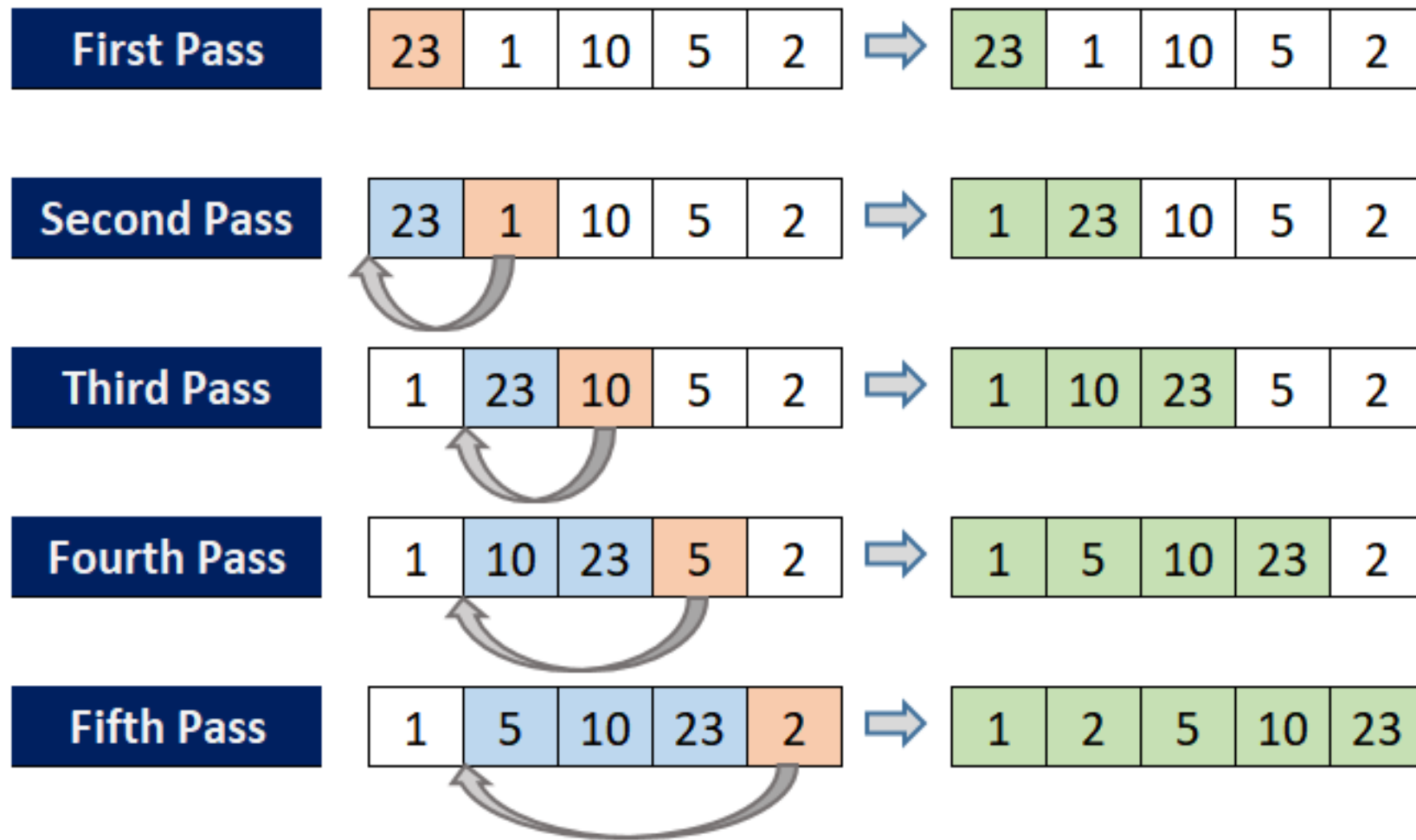
- Listenin ilk elemanını alır ve diğer tüm elemanlarla kıyaslar. Aralarındaki en küçük eleman ile ilk elemanın yerini değiştirir.
 - Ardından ikinci elemana geçer, listede kalan tüm elemanlarla kıyaslar ve bulunan en küçük eleman ikinci ile yer değiştirir. Bu şekilde kıyaslamalar sürer.
-





3. Insertion Sort Algoritması (Eklemeli Sıralama)

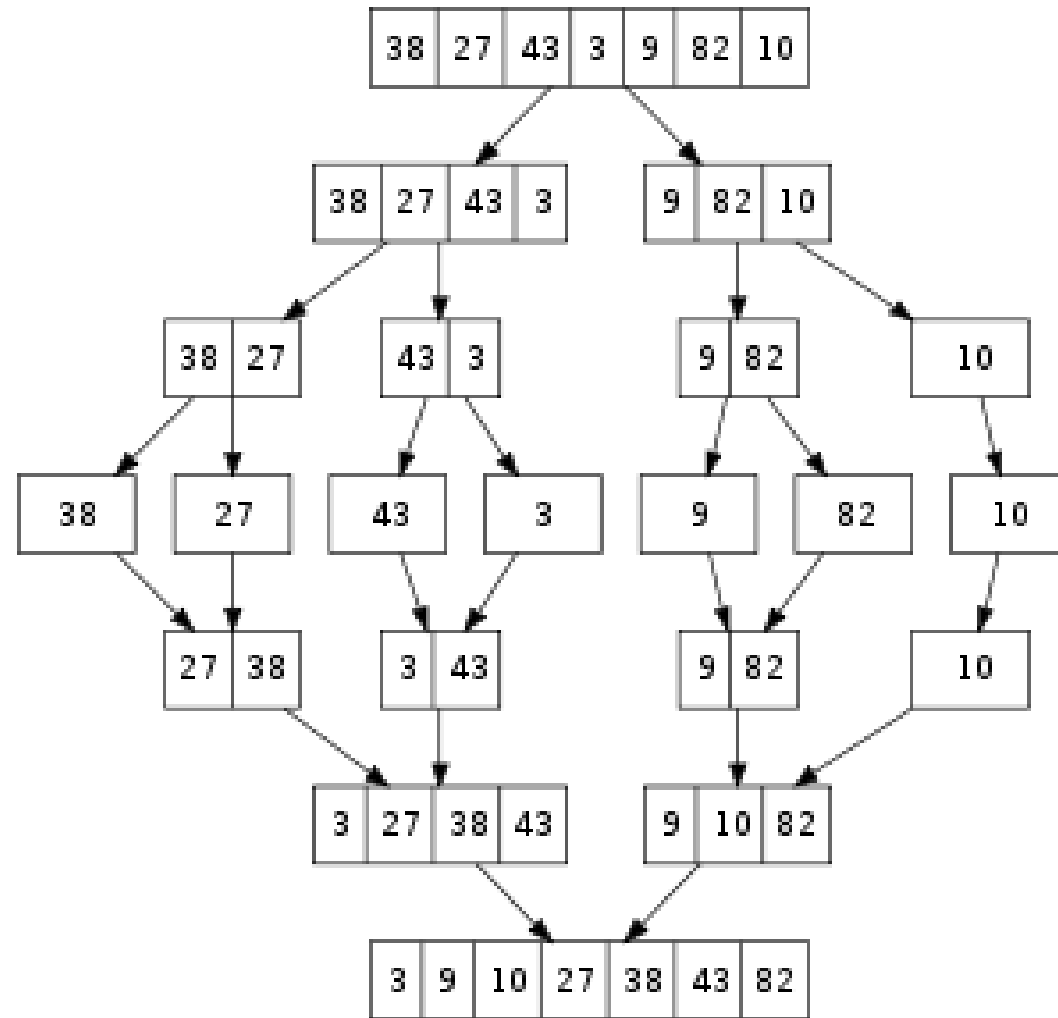
- Insertion Sort, bir sıralama algoritmasıdır. İsmi, sıralanacak listeyi sanki elinizdeki bir deste kartı sıralıyormuş gibi ele almasından alır. Her adımda, sıralanacak bir sonraki elemanı alır ve bu elemanı, sıralanmış listeye doğru konumuna “yerleştirir”. Bu işlem, tüm liste sıralanana kadar devam eder.
-





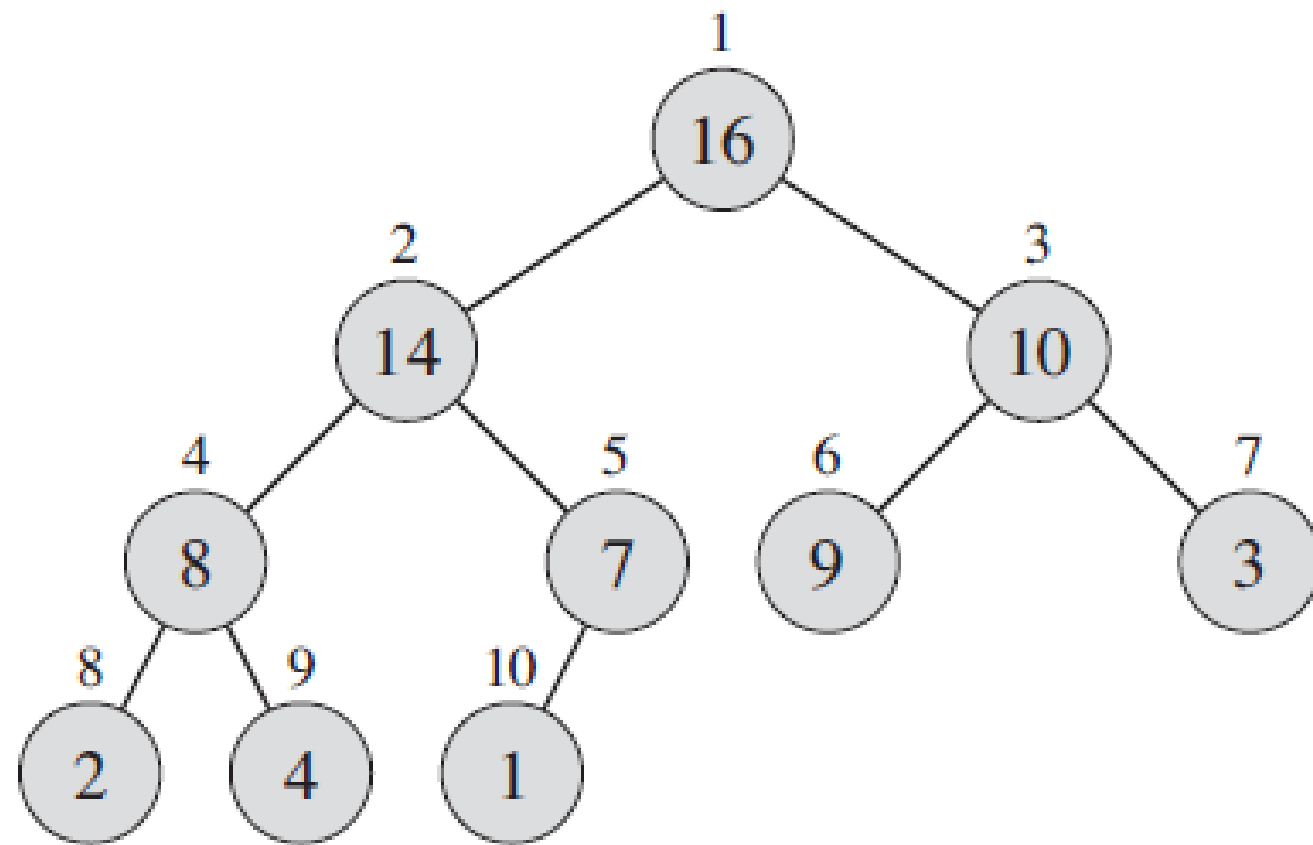
4. Merge Sort Algoritması (Sıralama Birleştirme)

- Merge Sort algoritması parçala ve fethet (divide and conquer), birleştir prensibiyle çalışır. Dizi sürekli olarak parçalanır ve en sonunda her eleman bir dizi haline gelir. Ardından bu diziler sıralanarak birleştirilir. Alttaki şemada daha somut halini görebilirsiniz.
-



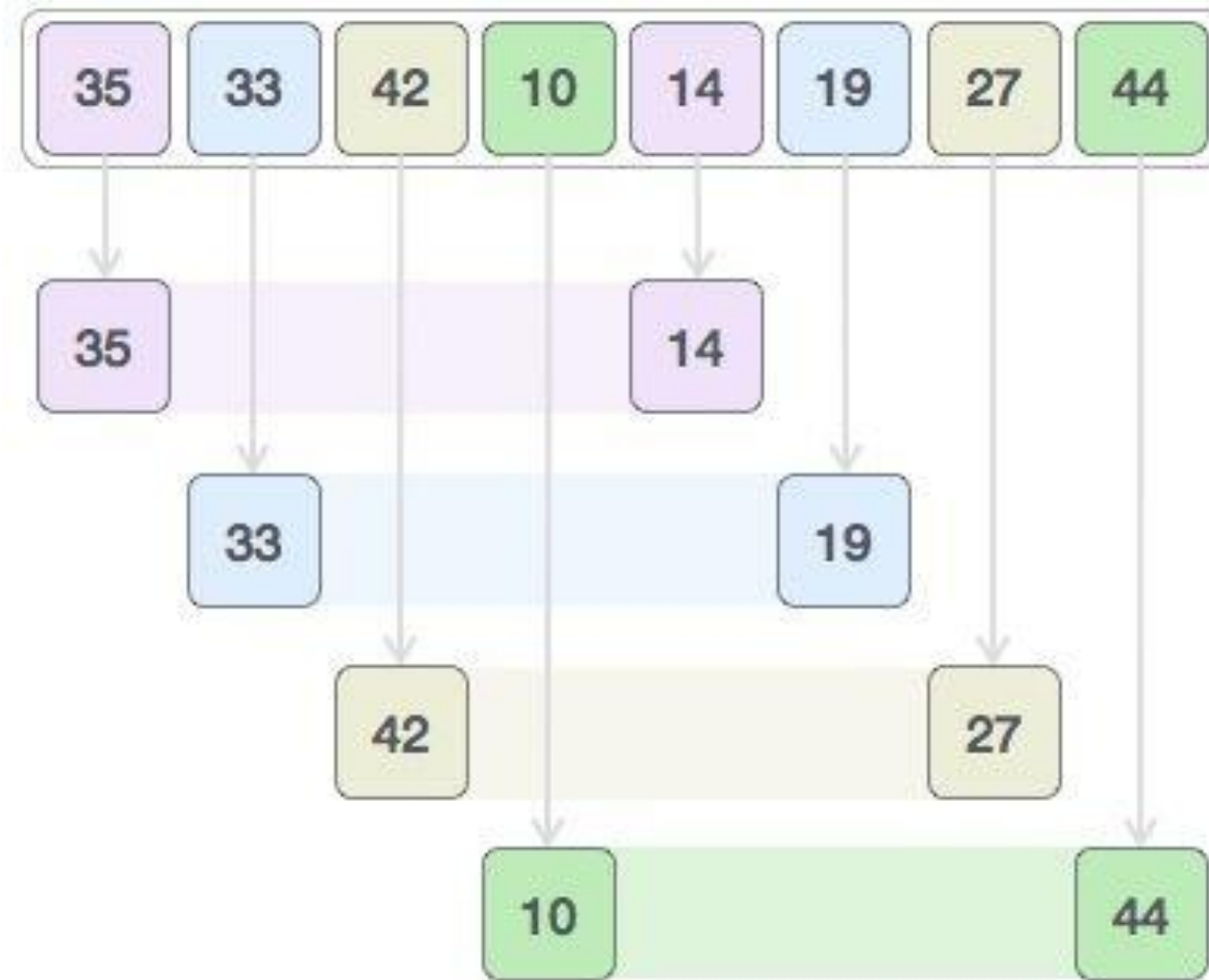
5. Heap Sort Algoritması (Yığın Sıralama)

- Üstteki bireylere parent (ata), alttaki bireylere child(çocuk) denir. Min heap algoritmasında ata birey en küçük sayıdır. Ata bireyden çocuklar türer ve ağaç şeklinde aşağıya doğru devam eder. Bireyler aşağıya doğru soldan sağa numaralandırılır.
 - Bir üyenin solundaki çocuğun numarası için $(2i)$
 - Bir üyenin sağındaki çocuğun numarası için $(2i+1)$
 - Bir üyenin ata bireyinin numarası için $(i/2)$ formülleri kullanılır.
-



6. Shell Sort Algoritması (Kabuk Sıralama)

- Shell'in yöntemi veya Veri yapısındaki Shell sıralaması, etkili bir yerinde karşılaştırma sıralama algoritmasıdır. Adını 1959'da ilk fikri ortaya atan Donald Shell'den almıştır. Kabuk sıralaması, ekleme sıralama algoritmasının genelleştirilmiş bir uzantısıdır.
 - Bu sıralama algoritmasının temel fikri birbirinden uzak olan elemanları gruplandırıp buna göre sıralamaktır. Daha sonra yavaş yavaş aralarındaki mesafeyi azaltın. Kabuk sıralaması ortalama vaka süresinin üstesinden gelir.
-





7. Quick Sort Algoritması (Hızlı Sıralama)

- Quick Sort algoritması tıpkı Merge Sort gibi parçala ve fethet (divide and conquer) prensibiyle çalışır.
 - Dizide bir pivot eleman seçilir, bu pivot bir nevi referans noktası olarak kabul edilir. Bu referans noktasının soluna kendinden küçük değerler, sağına ise kendinden büyük değerler getirilir.
 - Ardından solda ve sağda oluşan dizilerde de pivot eleman seçilip aynı işlem uygulanır. Bu işlemlerin sonucunda dizimiz sıralanmış olur.
-

A

0	1	2	3	4
76	6	4	19	50



A

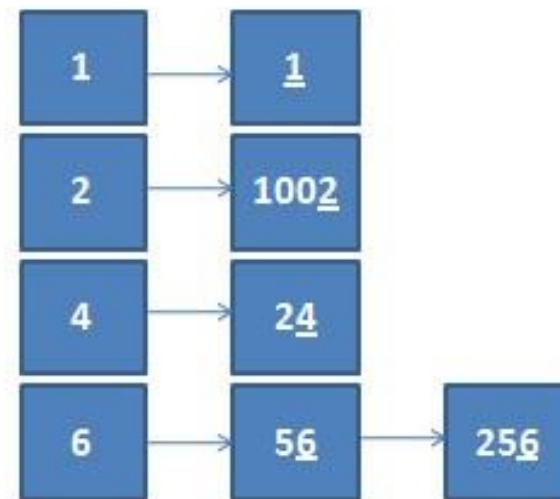
0	1	2	3	4
6	4	19	50	76

$6, 4, 19 < 50 < 76$

8. Radix Sort Algoritması

- Radix Sort, sayıları basamaklarının üzerinde işlem yaparak sıralayan doğrusal sıralama algoritmalarından biridir. Radix Sort algoritması, 1887 yılında Hollerith'in patentini aldığı “tabulating machine” için kullandığı yöntemeye dayalıdır. Esasta, 2 tabanına göre yazılmış sayıları sıralayan hızlı bir algoritmadır.
-

	0	1	2	3	4
A	5 <u>6</u>	100 <u>2</u>	2 <u>4</u>	<u>1</u>	25 <u>6</u>



	0	1	2	3	4
A	1	1002	24	56	256

9. Bucket Sort Algoritması

- Kova Sıralaması (ya da sepet sıralaması), sıralanacak bir diziyi parçalara ayırarak sınırlı sayıdaki kovalara (ya da sepetlere) atan bir sıralama algoritmasıdır. Ayırışma işleminin ardından her kova kendi içinde ya farklı bir algoritma kullanılarak ya da kova sıralamasını özyinelemeli olarak çağırarak sıralanır.
-

12	9	24	4	19	21	14	6	2	16
0	1	2	3	4	5	6	7	8	9



2	4	6	9	12	14	16	19	21	24
0	1	2	3	4	5	6	7	8	9



10. Coupting Sort Algoritması (Sayarak Sıralama)

- Sayarak sıralama algoritması, dizideki her sayının kaç tane olduğunu farklı bir dizide tutar daha sonra bu sayıların bulunduğu dizi içinde sıralı bir şekilde değerler adet sayısına dikkat edilerek yazılır.
-

inputArray

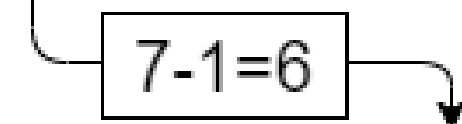
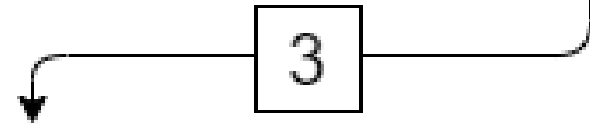
0	1	2	3	4	5	6	7
2	5	3	0	2	3	0	3

countArray

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

outputArray

0	1	2	3	4	5	6	7
						3	





11.Shaker Sort Algoritması (Sallama Sıralama)

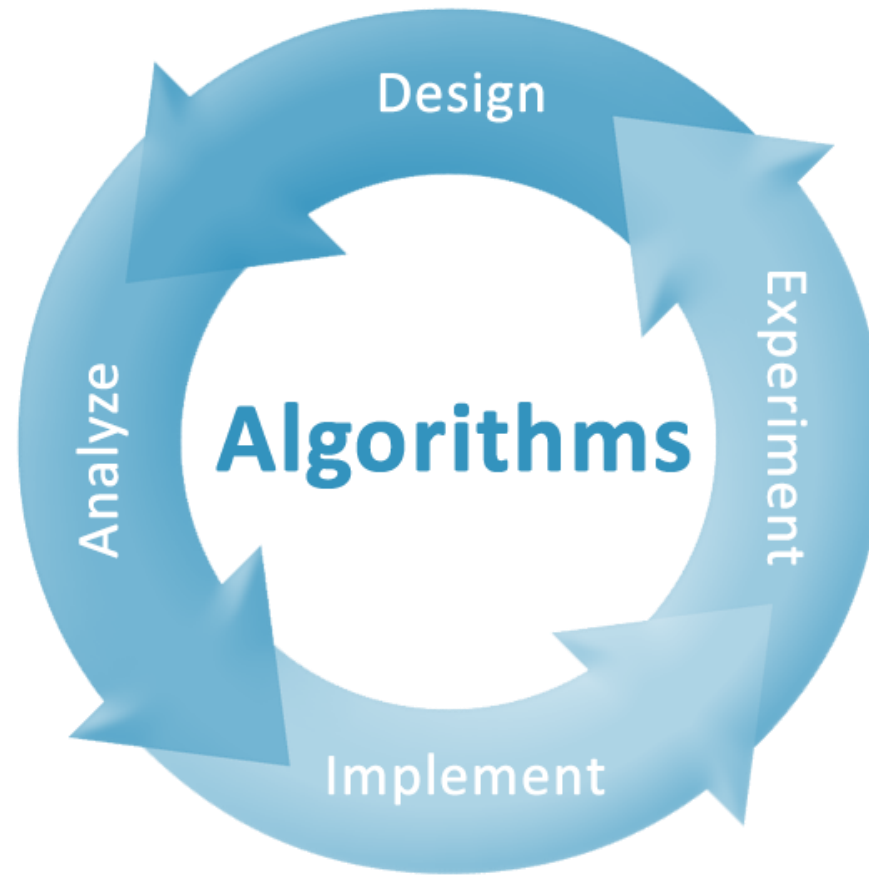
- Veri sıralama için kullanılan ve kabarcık sıralamasının (bubble sort) neredeyse aynısı olan sıralama algoritmasıdır (sort algorithm). Kabarcık sıralamasından tek farkı, kabarcık sıralaması tek yönlü olarak kabarcığı hareket ettirirken, sallayıcı sıralaması bir sağdan bir soldan iki yönden de sıralamaktadır.
-

12.Bogo Sort Algoritması (Rastgele Sıralama)

- Bogosort, verilen bir diziyi sıralamak için rast gele bir dizilim üretir ve sıralı olup olmadığına bakar, şayet sıralıysa algoritma sona erer, şayet sıralı değilse rastgele olarak yeni bir dizilim elde eder, ta ki sayılar sıralanana kadar sayıları rastgele dizmeye devam eder.
 - Bu algoritma basitçe bir dizinin sıralı olana kadar rastgele dizilmesi olarak ifade edilebilir. Algoritmaya rastgele sıralama (random sort) veya maymun sıralaması (monkey sort) veya çifteli tüfek sıralaması (shotgun sort) anlamında isimlerde verilmektedir.
-

13.Lucky Sort Algoritması (Şanslı Sıralama)

- Sadece teorik olarak literatürde geçen bir sıralama algoritmasıdır (sorting algorithm). Buna göre sıralanacak olan dizi şanslı bir şekilde zaten sıralı verilmiştir. Dolayısıyla dizinin sıralanmasına gerek yoktur. Hatta bu kabulü yaptığımız için dizinin sıralı olup olmadığını kontrol etmemize de gerek yoktur.
-



Doc. Dr .Mehmet Akif Cifci

- Viyana Teknik Üniversitesi (Avusturya)
- Klaipeda Üniversitesi (Litvanya)
- Bandırma Onyedi Eylül Üniversitesi

To Follow and Connect

<https://github.com/themanoftalent>

<https://www.linkedin.com/in/themanoftalent/>

<https://www.researchgate.net/profile/Mehmet-Akif-Cifci>