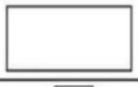
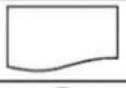
Algoritma Nedir?

Algoritma, Cebrin atası ve kurucusu olarak bilinen Harezmî tarafından 9.yüzyılda cebir alanındaki araştırmaları neticesinde ortaya çıkmıştır. Avrupalılar, Harezmî ismini telaffuz edemediklerinden dolayı algorizm (Arap sayıları kullanarak aritmetik problemler çözme kuralları) olarak kullandılar. Algorizm daha sonra "algoritma" adını aldı.

Algoritma nedir sorusunun cevabı, bir problemin çözümü için gerekli olan adımların bütününe verilen isimdir. Gelin algoritma örneklerini günlük hayatımızdan çıkarmaya çalışalım. Bir kahve hazırlarken de algoritma kullanırız, ayakkabımızı giyerken de kullanırız. Kahve örneğimizi inceleyelim. Öncelikle bir kahve bardağı bulmamız gerekiyor. Bundan sonraki adım ise kahveyi makineye koyup çalıştırmamız. Daha sonrasında kahve bardağını ilgili yere koyup tuşa basmamız gerekiyor. Bu olay örgüsü aslında bir algoritma. Adım adım sürdürmemiz gerekiyor. Öğrendiklerimizi pekiştirmek için haydi gelin resimli örneğimizi inceleyelim.

Sembol	Sembolün Adı ve Anlamı
	**Elips:** Akış diyagramının başlangıç ve bitiş yerlerini gösterir. Başlangıç simgesinden çıkış oku vardır. Bitiş simgesinde giriş oku vardır.
	**Paralel Kenar:** Programa veri girişi için kullanılır.
	**Dikdörtgen:** Aritmetik işlemler ve her türlü atama işlemlerinin temsil edilmesi için kullanılır.
	**Altıgen:** Program içinde belirli blokların art arda tekrar edileceğini gösterir.
	**Eşkenar Dörtgen:** Karar verme işlemini temsil eder.
	**Belge:** Ekrana veya yazıcıya bilgi çıkışı için kullanılır.
	**Daire:** Birleştirici veya bağlantı noktalarını temsil eder.
	**Oklar:** Diyagramın akış yönünü, yani herhangi bir adımdaki işlem tamamlandıktan sonra hangi adıma gidileceğini gösterir.

Gördüğünüz üzere ana problemimiz bilgisayarın çalışmaması. İlk adım güç kablosunun takılı olup olmadığını kontrol etmek. Bu adımın cevabı Hayır ise yapmamız gereken güç kablosunu takmaktır, cevap Evet ve hâlâ bilgisayarımız çalışmıyor ise bir sonraki adımı uygulamamız gerekiyor. İkinci adım, uzatma kablosunun durumunu inceledikten sonra eğer bilgisayarımız hâlâ çalışmıyor ise tamire götürmemiz gerektiği sonucunu veriyor.

Özetle, Algoritma belirli bir durumdan başlayıp belirli bir sonuçta biten problemlere çözüm getiren adımlardır.

Bilgi bilgisayar tarafından nasıl ifade edilir ?

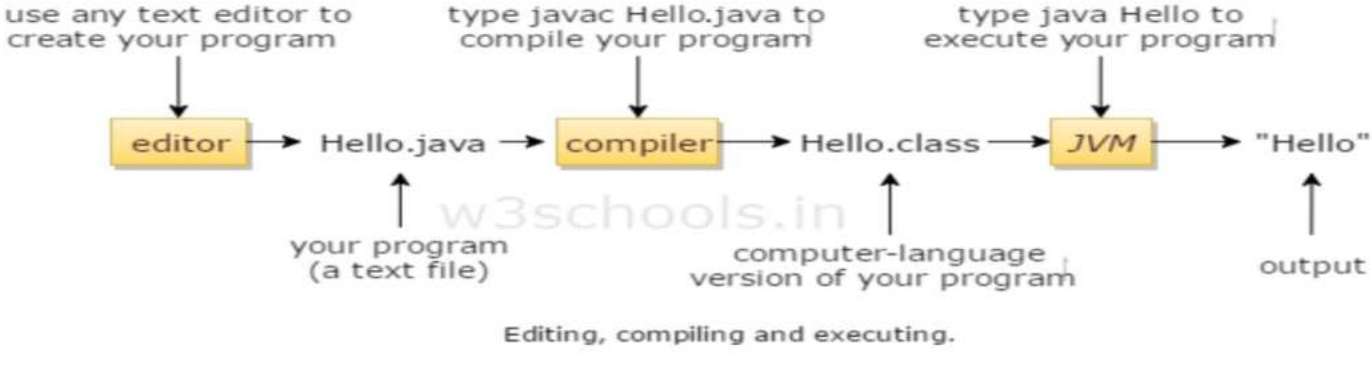
Şöyle düşünelim, bir insan kendini ifade etmek istediğinde native (ana) bir dil kullanıyor öyle değil mi? Bilgisayar da bilgiyi (Resim, ses, yazı vb) ifade etmek ve döngüyü sağlamak için bit (0 ve 1)' den oluşan ikili sayıları (Binary Numbers) kullanıyor.

İkili sayılarda bulunan 1 ve 0 rakamları (bit), bilgisayarın elektrik iletimi için kullandığı transistörlerin açık veya kapalı olma durumunu gösteriyor. Transistörlerde iki tane komut vardır, 0 (kapat) ve 1 (aç).

Konuyu pekiştirmek için bir örnekle açıklayalım. C#, Java gibi dillerde kullandığımız platformlar aracılığıyla yazdığımız kod ilk olarak derlenir ve makine koduna (0 ve 1'li sayılara) dönüştürülür. Bilgisayar okur ve çıktı verir.

Peki bu derleme nedir ve nasıl olur? Hemencecik anlatalım. Örneğin, bir dilde (örnek Java üzerinden olacak) yazdığınız kod Java'nın hali hazırda sahip olduğu Java Compiler ile derlenerek .class uzantılı dosyayı oluşturur. Haa dikkat, bu format prensesimiz olan sadece JVM (JAVA VİRTUAL MACHİNE)'ye özeldir. İşlemcimiz bu aşamada okuyamadığından işleyemez. JVM (JAVA VİRTUAL MACHİNE) .class uzantılı bytecode'u satır satır işleyerek makine koduna dönüştürür ve çalıştırır.

Özetle, Javada yazılan kod önce bytecode'a çevrilir, daha sonrasında üzerinde çalıştığı makinenin içerisindeki yalın dile (makine koduna) yorumlanarak çalıştırılır.



Editing, compiling and executing.

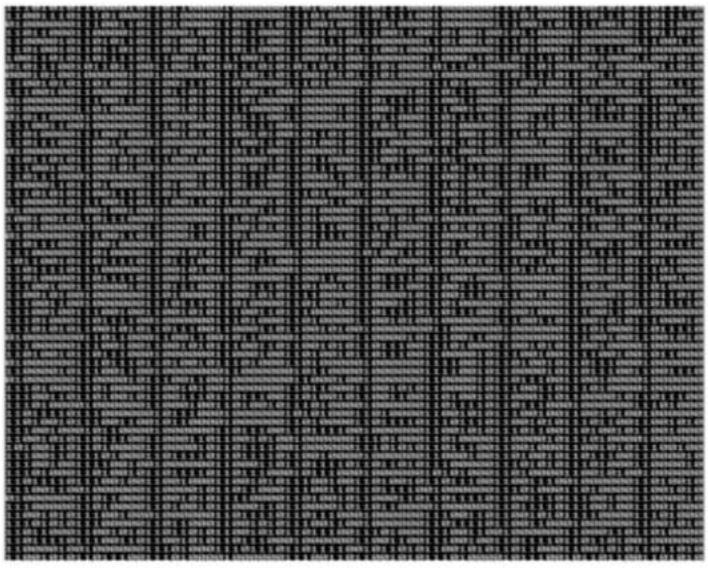
Sonuç olarak bilgisayarlar kendilerini 0 ve 1 sayıları aracılığıyla ifade eder. Bilgisayar üzerindeki her şey 1 ve 0'lardan oluşur.

Bytecode -> Java derleyicisinin Java ile yazılmış kodların makine dili yerine kendine has oluşturduğu Binary (0 ve 1) dosyasıdır.

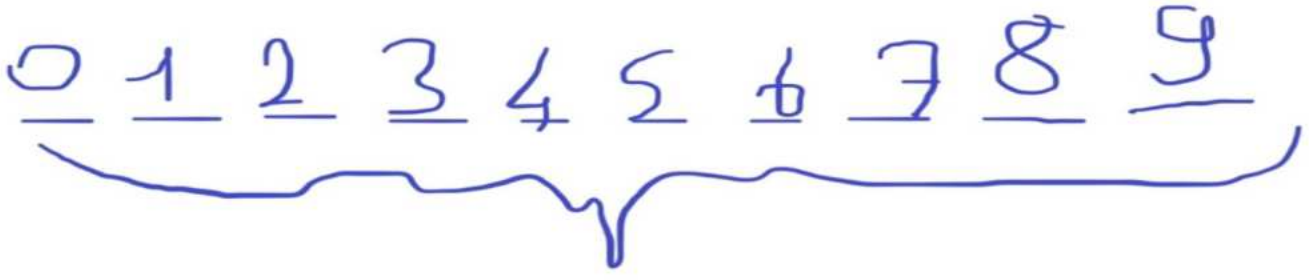
JVM -> Java Bytecode formatına derlenmiş programların çalışmasını sağlayan bir sistemdir.

Sayı Sistemleri

Evvet arkadaşlar, bugün sizinle beraber ikili (Binary) sayı sistemini göreceğiz. İlk olarak, ikili sayı sisteminde sayılar 2 tabanında yazılır. Bu yüzden, ikili sayılar 0 ve 1 kullanarak ifade edilir.



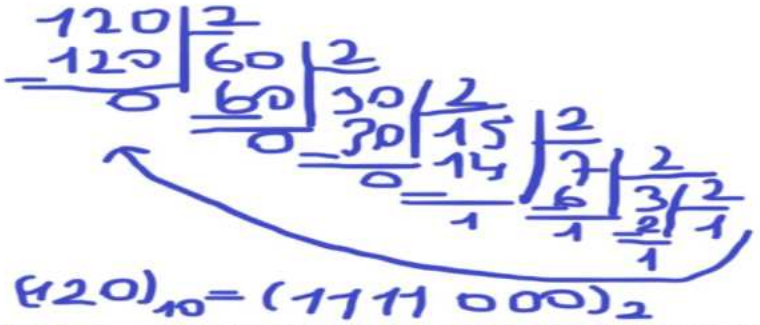
- Sayıları ifade ederken genellikle onluk sayı sistemini kullanırız. Onluk sayı sisteminde aralığımız 0 dan 9 a kadar olmasından dolayı tek basamakta 10 farklı durumu ifade edebiliriz. İkili sistemde ise tek basamakta 2 farklı durum ifade edilir; 0 ve 1. **!!!!! Tek basamakta !!!!!**



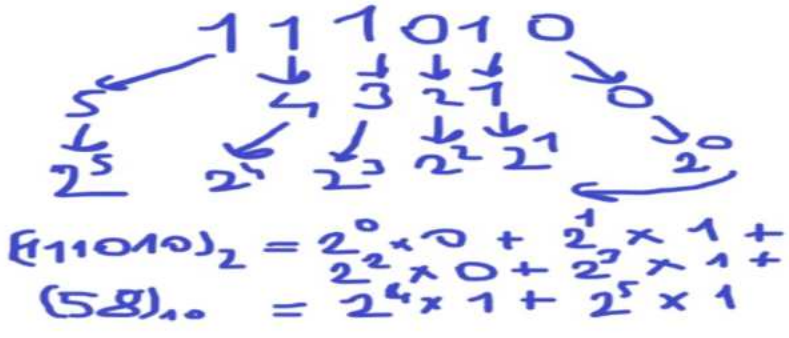
10 farklı durum

2 farklı durum

- Onluk sayı sistemindeki bir sayı ikilik sisteme, ikilik sistemdeki bir sayı ise onluk sisteme dönüştürülebilir. Nasıl mı? Hemencecik açıklayalım. Onluk tabanda 120 sayısını ikilik sayı sistemine dönüştürelim.



- Peki ya ikilik tabanındaki bir sayıyı onluk tabana nasıl dönüştüreceğiz? Nasıl mı? Beyniniz mi yandı :). Devammm. Sayımız ikilik tabanda 111010 sayısı hadi başlayalım. Az önce öğrendiğimiz bilgiyi çevirmede kullanıyoruz. Aslında 2^0 dan başlamamızın sebebi onluk sayı sistemine geçirdiğimizden kaynaklanıyor.



Bilgisayarda Sayısal Olmayan Verilerin Tutulması

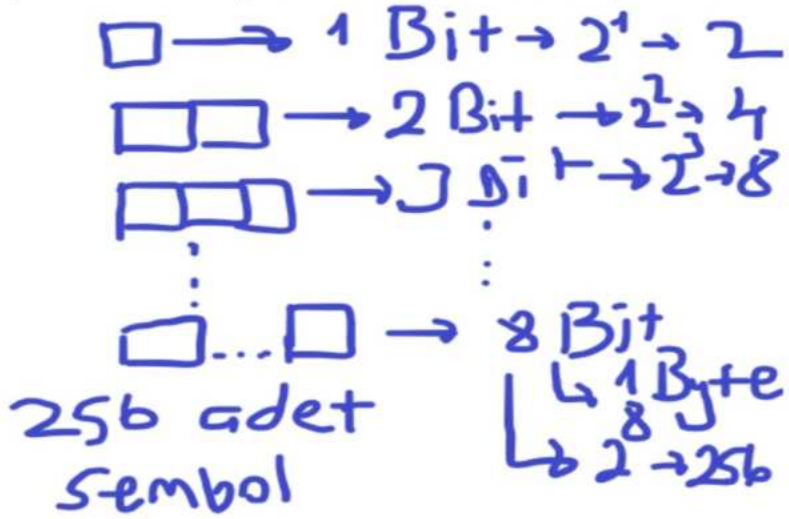
- Bilgisayarda var olan her şey 1 ve 0 dan oluştuğunu geçtiğimiz derste gördük. Binary'den onluk tabana nasıl dönüştüreceğimizi öğrendik. Peki ya bir harfi nasıl ifade edeceğiz? Sayısal olmayan bir ifadenin sembol ile gösterimi yapılabilir. Mesela, 1100010010 sayısını onluk tabana çevirdiğimizde değeri 786, fakat sayı olarak değil de bir sembol olarak incelersek karşılığı W harfi olabilir. Başka bir örnek, elimizde 100101010 sembolü var ve bu sembolün karşılık geldiği değer V olabilir.

W	i	k
01100100	01101001	01100101

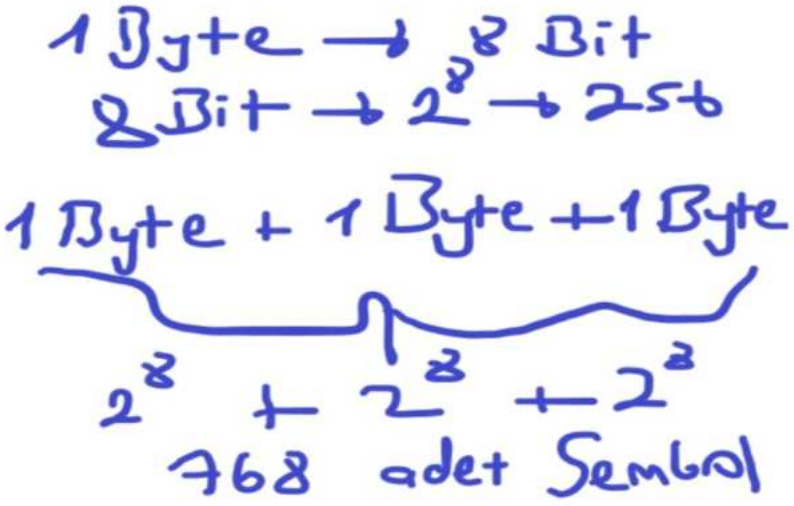
- Sayısal olmayan verilerin bir sembol olduğunu ve 1 ve 0'lardan oluştuğunu hep birlikte öğrendik. Binary olarak gördüğümüz ifade makine kodundan dolayı farklı bir nesneyi işaret edebilir.

Bilgisayarda Verilen Tutulması

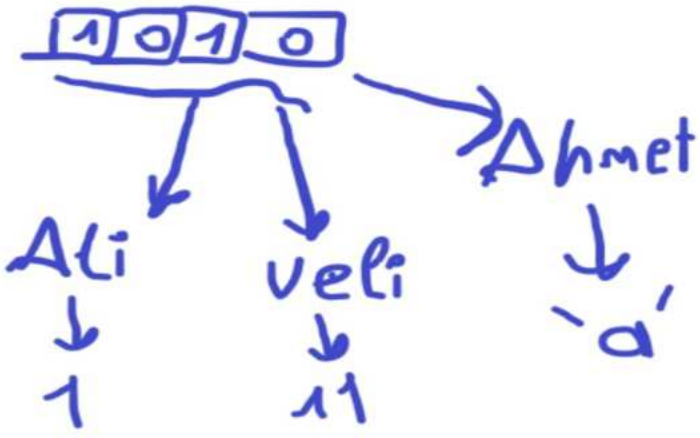
- Bilgisayar, yapısından dolayı içerisinde tutulabilecek veri miktarı sınırlıdır. Bu verilerin en küçük yapı taşları bitlerdir. Bu bitleri bir hafıza gibi düşünebiliriz. Ne kadar çok bit dolar ise o kadar az veri depolama alanımız kalmaktadır. Hadi gelin bit ve byte kavramlarını inceleyelim.



- Şimdi karşımıza çıkan problemlerin bir tanesi, 256 sembolden daha fazla bir depolama alanı isteyebiliriz. Peki bu durum da ne yapacağız? Aslında çözüm çok basit byte'ları yan yana koyarak depolama alanımızı arttıracağız. Hemmen bir örnek çözelim.

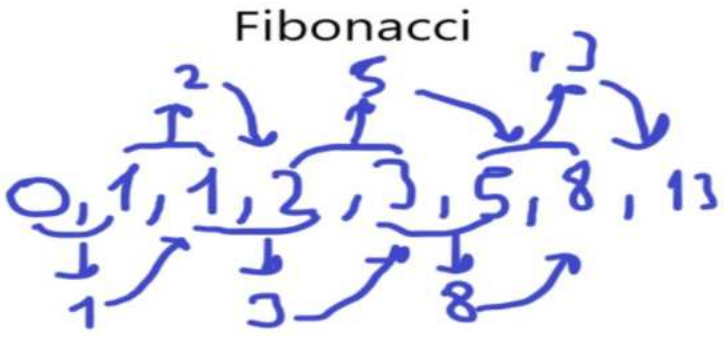


- Binary semboller, kullanan kişiye göre farklılık gösterebilir. Örneğin, 00 sembolü Ali'ye göre k karakterini sembolize ederken, Veli'ye göre 1 sayısını sembolize edebilir.



Recursion

- Bir problemin alt problemlere bölünüp hesaplanmasına, nerde son bulacağımı belirttiğimiz ifadelere recursion (Özyineleme) diyoruz. Peki bu recursion ne anlama geliyor dediğiniz duyar gibiyim. Hadi gelin fibonacci serisi ile konumuzu pekiştirelim.



Recursion için tekrarın bir öneni yoktur, önceli olan sonuçtur.

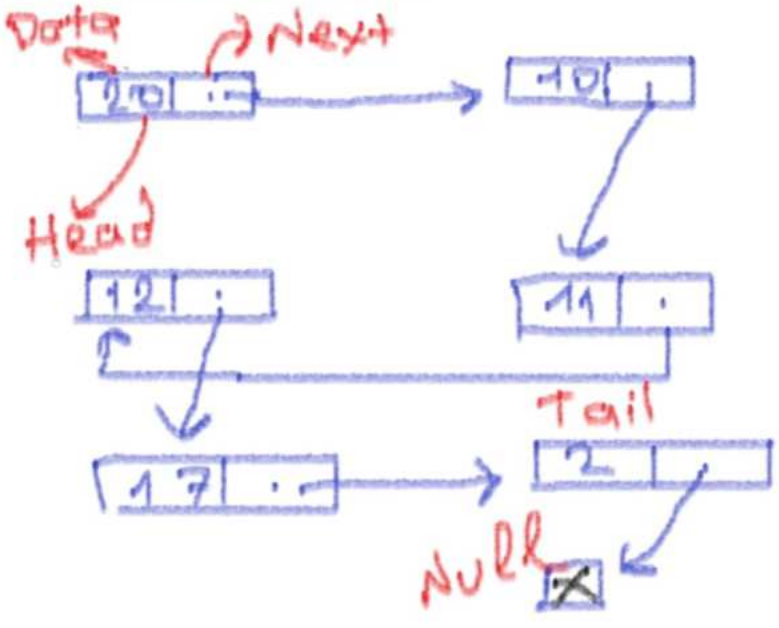
- Fibonacci serisi 0,1 den başlayarak her önceki 2 sayının toplamı şeklinde bir kurala sahiptir. 0+1 = 1, 1+1=2, 2+1=3, 3+2=5 gibi. Kendinden bir önceki eleman ile iki önceki elemanın toplamı serinin devam sayısını verir. Recursion kullanımı : Fonksiyonumuz fib(x). 3. elemana n diyelim, fib(n-1) + fib(n-2) bize 3.elemanı yani 1 sonucunu verir. fib(n-1) + fib(n-2) => Recursion

Arrays

- Arrays (Diziler), anlam ifade etmesi için birden fazla nesneye ihtiyaç duyabilir. Mesela, Şu an karşısında olduğunuz bilgisayar örneğini inceleyelim. Masaüstü bilgisayarlar, klavye-mouse-monitör üçlüsünü bir araya getirince anlam ifade eder. Herhangi biri olmadan bir işlem yapmanız olasıdır ama zorludur.
- Array (Dizi), dezavantajlarından biri olan hafıza problemini inceleyelim. Bilgisayar örneğimizden devam edelim. Hali hazırda bir klavye, bir mouse ve bir monitörümüz var. Yeni bir monitör aldığımızda daha büyük bir masaya ihtiyacımız var. Aynı şekilde yeni bir klavye veya mouse aldığımızda da aynı durum geçerli. Bir yerden bir yere taşırken zaman ve güç kaybına uğruyoruz.
- Dynamic Arrays (Dinamik diziler) ise yeni bir eleman için boşta yer tutmasından ötürü esnektir. Örneğin, bazı mutfak masaları açılan sürgülü bir yapıya sahiptir. Masanın küçük kaldığı durumlarda büyütmek için kullanılır. Dinamik dizilerde aynı mantığa sahiptir. Yeni elemanlar için yer tutarlar.
- Dynamic Array (Dinamik dizinin) dezavantajlarından biri ise hafızada fazladan yer kaplaması, gerçekleşecek olan bir diğer olayı engelleyebilir. Nasıl mı, hemen örnek ile kavrayalım. Masa örneğinden bahsetmiştik. Misafirleriniz bir işi çıkması durumunda fazladan yer kapladık ve hareket kabiliyetimizi kaybettik.
- Hep dezavantajlarını konuşuyorsun, e yahu bunun avantajı yok mu? Tabii ki var. Array'lerin birbirine bağlı olması ulaşılabilirliğini kolaylaştırır. Klavye-Mouse-Monitör örneğini vermiştik. Hepsi bir masada bulununca ulaşılması kolaydır. (Masa = Array, Klavye-Mouse-Monitör = Array Elemanı)

Linked-List

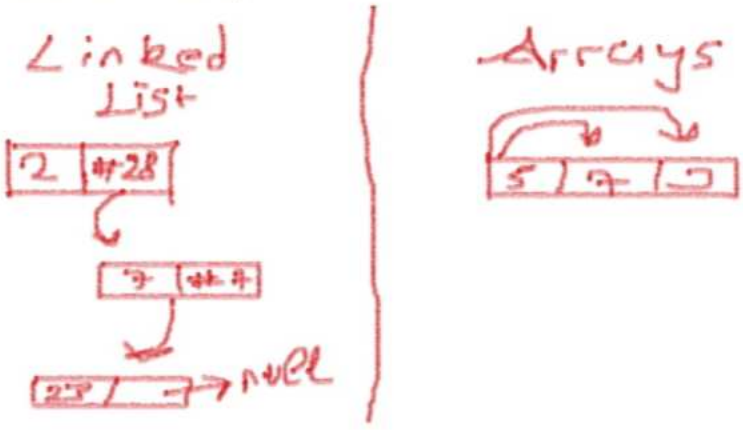
- Linked-List (Bağlı listeler), yan yana zorunluluğu olmadan veri tutmamızı sağlayan yapılardır. Yeni gelen eleman için hafıza'da yeni bir alan açmamız gerekmez. Array'dan farklı olarak evet elemanlar hafıza içerisinde dağılmış olabilir, fakat son gelen eleman kendinden bir önceki elemana adresini bildirmek zorundadır.



http://cagataykiziltan.net/veri-yapilari-data-structures/1-linked-list-bagli-listeler/

- Yukarıdaki örnekte gördüğünüz üzere, her bir düğüm bir sonrakinin adresini tutar. Her bir önceki eleman bir sonraki eleman ile bağlıdır.

Linked List vs Array

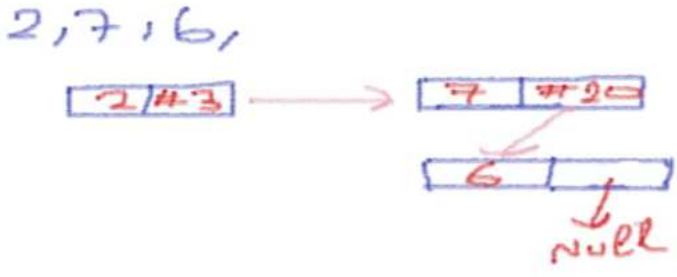


Değişken bir yapı kullanacaksan (ekleme-çıkarma olması) linked-list kullanmak daha mantıklıdır.

Array	Linked-List
Array'in herhangi bir elemanına ulaşmak aynı sürede gerçekleşir (Random Access).	Linked-List'de ulaşmak istediğimiz elemana gidebilmek için birbirine bağlı olan elemanları ziyaret etmemiz gerekiyor.
Array'ler, sadece eleman tuttuğu için hafızada daha az yer kaplarlar.	Linked-List'ler, eleman ile birlikte adres tuttuğundan dolayı hafızada daha fazla yer kaplarlar.
Daha çok statik (durağan) durumlarda daha fazla performans gösterir.	Ekleme , çıkarmanın fazla olduğu durumlarda linked-list daha fazla performans gösterir.

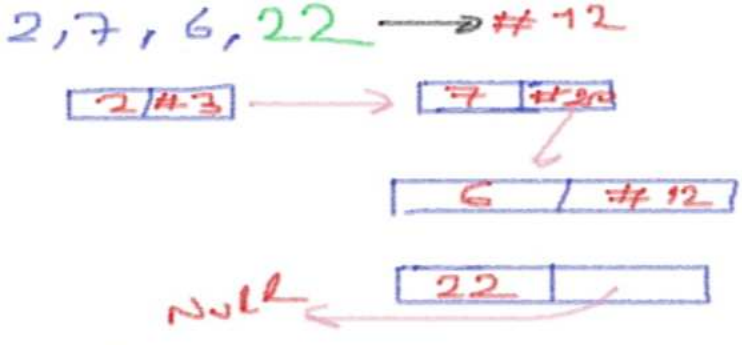
Eleman Ekleme/Çıkarma

Gelin 3 elemanlı bir hücre oluşturalım.



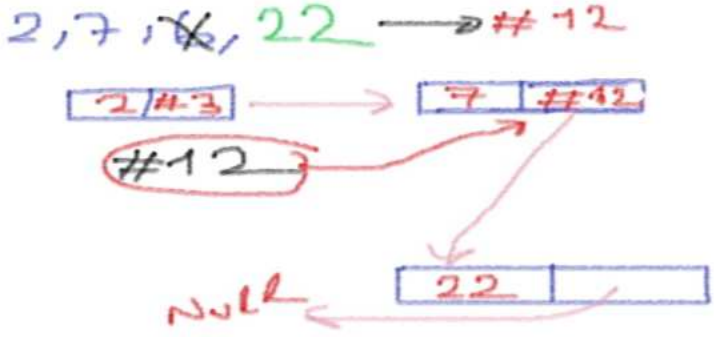
Eleman Ekleme

- Adresi #12 olan 22 sayısını listeye eklemek istiyoruz. Yapmamız gereken 6 hücresine 22 sayısının adresini yazmak.



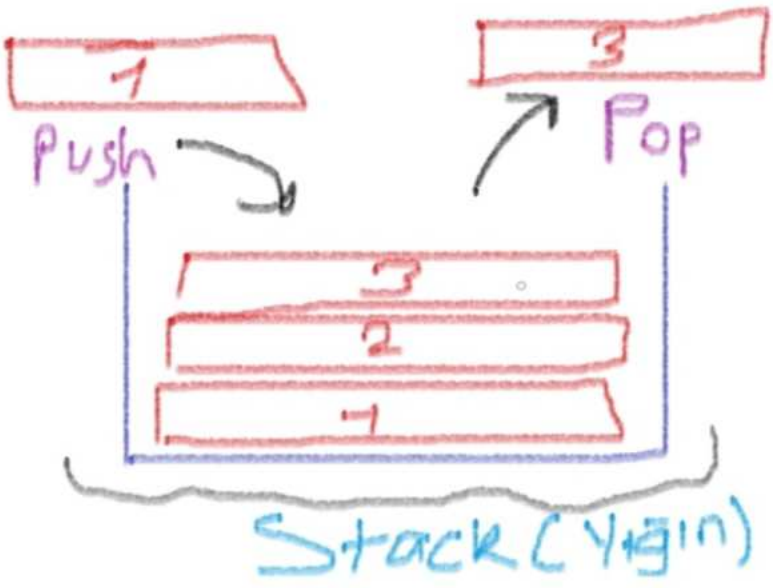
Eleman Çıkarma

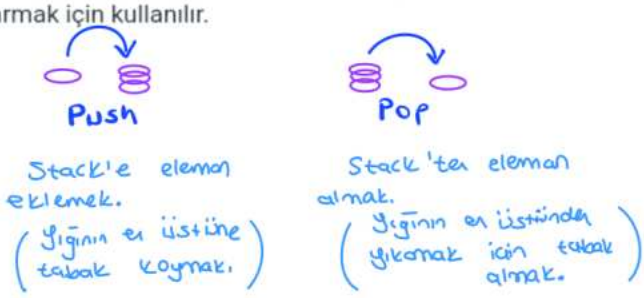
- Adresi #20 olan 6 numaralı hücresini çıkarmak istiyoruz. Linked-List'de bir önceki eleman adresini tutuyordu. Yani 7 numaralı hücrede bulunan 6'nın hücre adresini siliyoruz. Yerine 22 numaralı hücrenin adresini yazıyoruz.



Stack

- Stack, LIFO (Last in First out) (En son giren en önce çıkar) mantığına dayanan, elemanlar topluluğundan oluşan bir yapıdır. Gelin hemen örneğimize geçelim. Taşınırken topladığınız koli kutusu düşünün. İçerisinde kitaplar var ve en, boy olarak koliye tam olarak koyuluyor. Mantıken kolinin altı kapalı ve üst üste koymanız gerekmektedir. Yeni taşındığınız yerde çıkartırken en üstekinden başlarsınız. İşte stack (Yığın) da aynı mantıkta çalışıyor.
- Yığınlara eleman eklerken veya çıkartırken bazı methodlar uygulanır. Bunlardan biri push, diğeri ise pop. Push, yığının üzerine eleman eklemek için kullanılır (Koliye kitap koymak). Pop ise, yığından eleman çıkarmak için kullanılır.

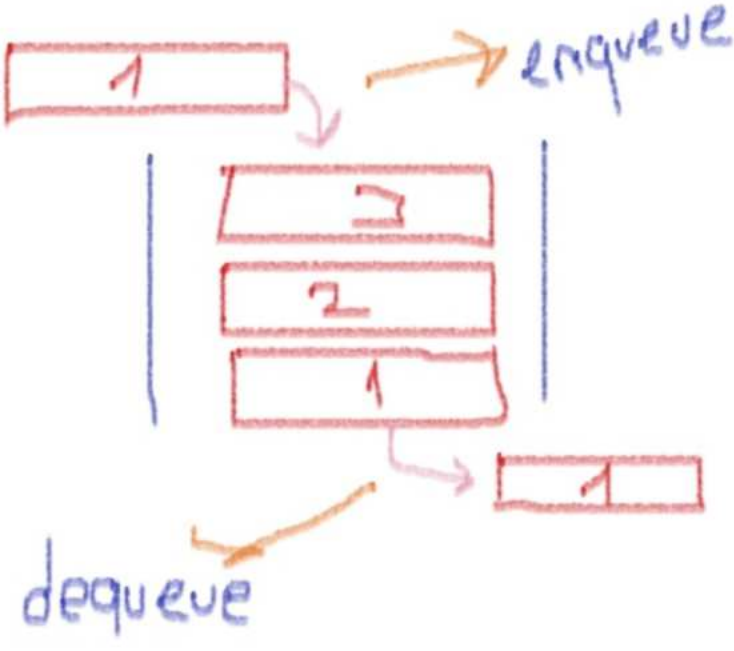




* Son giren ilk çıkar metodu
(LIFO)

QUEUE

- Queue (Kuyruk), FIFO (First in First out) (İlk giren ilk çıkar) prensibine dayanan, girişlerde ve çıkışlarda belirli bir kurala göre çalışan yapıdır. Stack de verdiğimiz örneği kuyruğa göre uyarlayalım. Biz örnekte altı kapalı bir koli kutusunu düşünmüştük. Şimdi o koli kutusunun altı yırtılmış. Sonuç olarak ne oluyor? İlk giren ilk çıkmış oluyor.
- Queue (Kuyruk)'da eleman eklemesi yaparken enqueue methodunu kullanıyoruz. Eleman silerken ise dequeue methodunu kullanıyoruz.



- Enqueue: Yeni elemanın kuyruğa eklemesi
- Dequeue: Elemanın kuyruktan çıkarılması

* İlk giren ilk çıkar metodu
(FIFO)

* Atm - banka sırası örnektir.

Hash Function

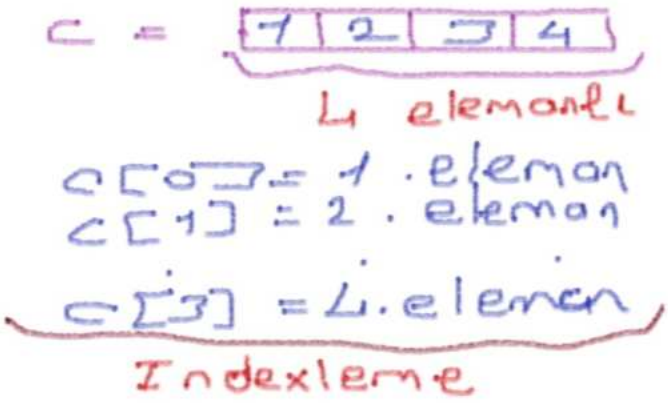
Hash Function (Karma Fonksiyonu), karma fonksiyonu olabilmesi için bazı temel şartlar vardır.Bunlar;

1. Gönderdiğimiz anahtarlar (keys) farklı olmasına rağmen bize aynı sonuçları veriyorsa bu bir hash function değildir.
2. Fonksiyona gönderilen anahtarlar aynı fakat sonuç farklı ise hash function değildir.
3. Hash Table için kullanılan dizinin boyutu verilen sonuçların sayısı kadar olmalı.

Hash Function/ Hash Table

Indexleme

Arraylerde 0 bazlı bir indexleme vardır. Bazı programlama dillerin 1 bazlı indexlemeler olsa da genel olarak 0 bazlı indexleme kullanılır.

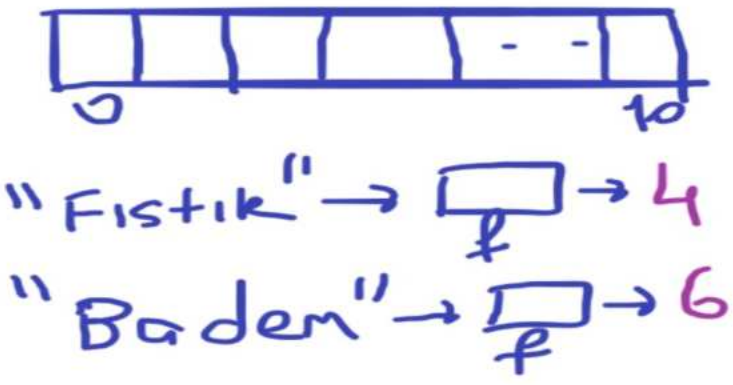


* Hash fonksiyonu her seferinde aynı girdiye aynı sonucu vermelidir.
* Hash fonksiyonunun çıktıları arrayin boyutunu aşmamalı.
* Hash fonksiyonu farklı girdilere farklı çıktılar vermelidir.
* Hash table için kullanılan dizinin boyutu verilen sonuçların sayısı kadar olmalıdır.

Hash Function/ Hash Table

Hash Table, key value prensibine dayanan bir array kümesidir. Key olarak çağırdığınız elemanın değerini (value) yansıtır.

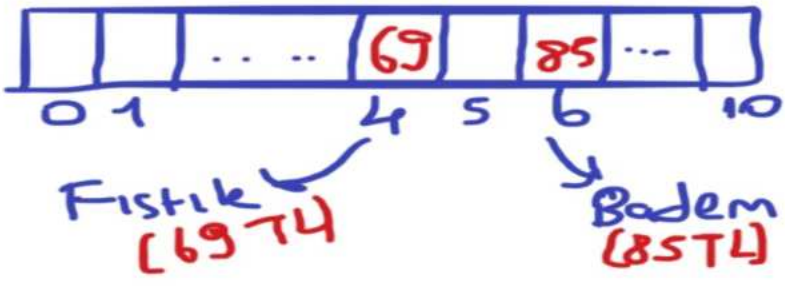
Hash Table yerine dizileri kullanabilirdik. Fakat her ürünü ve fiyatını tek tek aramak istemediğimiz için hash table kullanıyoruz. Peki bu süreç nasıl işliyor? Hemen bir örnek yapalım. Örneğimiz bir kuru yemiş dükkanından gelecek.



* Hash table key-value prensibine dayanır.
* Hash fonksiyonu ile oluşan arrayin ismine hash table denir.
* Hash table'da indexleme çok sık kullanılır.

Bu kısımda ilk olarak bulunan ürün sayımız kadar değeri olan bir Array oluşturduk.

Daha sonra hash fonksiyonundan ürünleri geçirerek index değerlerine ulaştık.



* Hash function farklı keyler alıp aynı değerler üretirse bu duruma çarpışma (Collison) denir.

Şifrelendiği için artık her badem keyi gönderildiğinde 85TL, fıstık keyi gönderildiğinde ise 69 sonucu verecektir.

Özetle, elimizde var olan verileri bir fonksiyondan geçirip indexliyoruz. Bu fonksiyona hash function, bu fonksiyon ile birleştiğimiz dizi yapısına ise Hash Table diyoruz.

Hash Collision

Hash Function farklı iki değerden aynı sayı üretilirse bu duruma Collison (çarpışma) denir. Bu olay istediğimiz bir durum değildir.

- Hash Function'lar bazen farklı durumlar için farklı sonuçlar üretemeyebilir. Örnek olarak araçları bir hash function dan geçirelim. Bu fonksiyonumuz son harflerine göre bir değer atıyor. Örneğin, motor ve tır için aynı değerleri ataması collision'a neden oluyor.
- Collision sorunuyla az karşılaşabilmek için kaliteli bir hash function olmalı. Bu sayede verimli bir Hash Table elde etmiş oluyoruz.
- Çarpışma sayısı arttıkça aradığımız şeyi bulma hızı azalır.

Algoritma Analizi

- Çalışma süresi – ifade sayısı (fonk. sayısı) ✗
- Büyüme hızı (Rate of Growth) ✓

- Algoritma analizi, var olan kaynaklara göre en uygun algoritmayı seçmek için uygulanır. Peki algoritma analizi en iyi nasıl yapılır? Kulağa karmaşık geliyor ama çok basit. Programlama dillerinden ve donanımlardan bağımsız bir şekilde Algoritma analizi yapılmalıdır. Aksi taktirde en uygun sonuç alınamayabilir.
- Donanımlar veya programlama dilleri farklı cihazlarda aynı performansı vermeyebilir. Örnek verecek olursak, cep telefonları için uygulama tasarladığımızı varsayalım. Bu uygulamanın performansı Apple telefonlar için farklı, Android telefonlar için farklı, arasında donanım farklı olanlar için ayrı olacaktır. Donanım ve diller ile algoritma analizi pek sağlıklı değildir.
- Algoritma analizi, bir algoritmanın çalışabilmesi için gerekli koşulların sağlanıp sağlanamadığını gösteren bir parametredir.

* Bir problem çözümü için birden fazla algoritma oluşturulabilir.

Ram Modeli

Bir algoritmayı farklı cihazlarda denemek bize pek fazla bir sonuç çıkarmıyordu. Çünkü kaynaklar değişebiliyordu. Bu probleme genel bir çözüm getirebilmek için hayalî bir cihaz düşünelim. Bu cihaz üzerinde bütün algoritmaları çalıştırdıktan sonra bize bir sonuç veriyor.

→ Bilgisayar RAM'i değil.

Bu hayalî cihaza RAM (Random Access Machine) diyoruz. Ram, algoritmalar arasındaki farkları belirlemek için kullanacağımız bir araç olacak.

- +, −, and, or gibi basit işlemler → 1 birim zaman
- Döngülerde, iterasyon sayısı * işlem sayısı → kadar birim zaman
- Hafızadan her okuma işlemi → 1 birim zaman alır.

Her işlemin birim zamanı var. Döngüler, kaç defa işlem yapıyorsa, (işlem sayısı * kaç kere tekrar edeceği) kadar birim zaman alır. Toplama, Çıkarma, and, or gibi aritmetik işlemler, 1 birim zaman alır.

Time Complexity

Algoritmanın verimli olması için belli kurallar vardır.

https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/

Örnekten bıktık diyenleri duyar gibi oluyorum. Hepsi sizin iyiliğiniz için :).

- Örnek: Raflara kitap yerleştirmek.

Algoritmayı en yavaş çalıştıracak durum

- Kitapları, gelişigüzel raflara dağıtırsak aradığımız kitabı daha fazla zamanda bulabiliriz. Aslında bu bir worst case'dir. Kitapları filtrelememiz gerekir. Kalın olanları bir rafa, ince olanları bir rafa, küçük boyutta olanları bir rafa koyduğumuz zaman aradığımız şeyi daha rahat bulabiliriz. Algoritma, en kötü senaryoya ne kadar hazırsa, bizi o kadar memnun edebilir.
- Algoritmalar için genellikle sık kullanılan average case'dir. Kitapların bölümüne göre kaç tane olduğunu biliyorsak average case kullanabiliriz. En büyük rafı miktarı fazla olana ayırabiliriz. Input yoksa average zordur!!!!

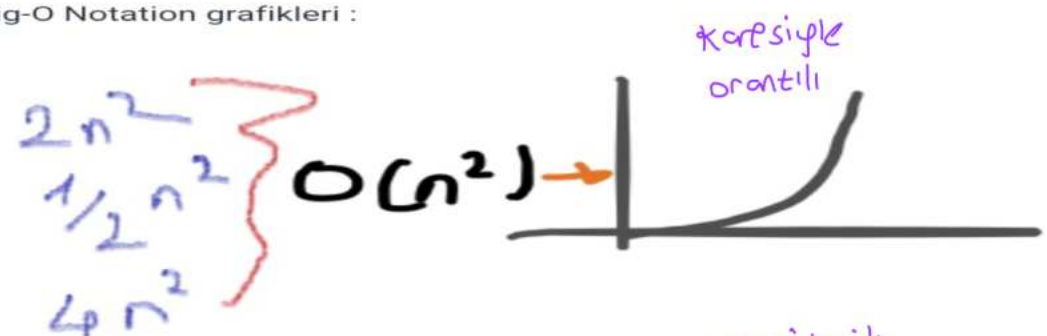
inputların dağılımı bilinirse kolaydır!!!

→ Algoritmayı en hızlı çalıştıran durum.

- Bir diğer senaryomuz ise best case'dir. Beklediğimiz en iyi durum. Kitap örneğine devam edecek olursak, bütün kitapların ayrı raflarda olması, alfabeye göre sıralanması best case olarak ifade edilebilir. Çünkü aradığımızı rahatlıkla bulabiliyoruz.

Big-O Notation

Big-O Notation grafikleri :



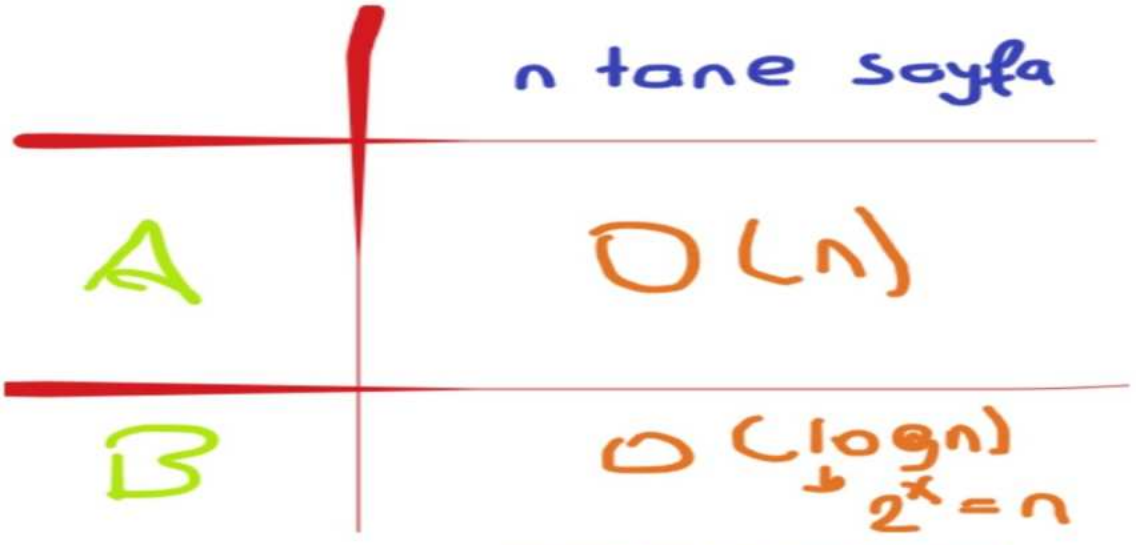
n^2, $2n^2$, $3n^2$, ... $\rightarrow O(n^2)$

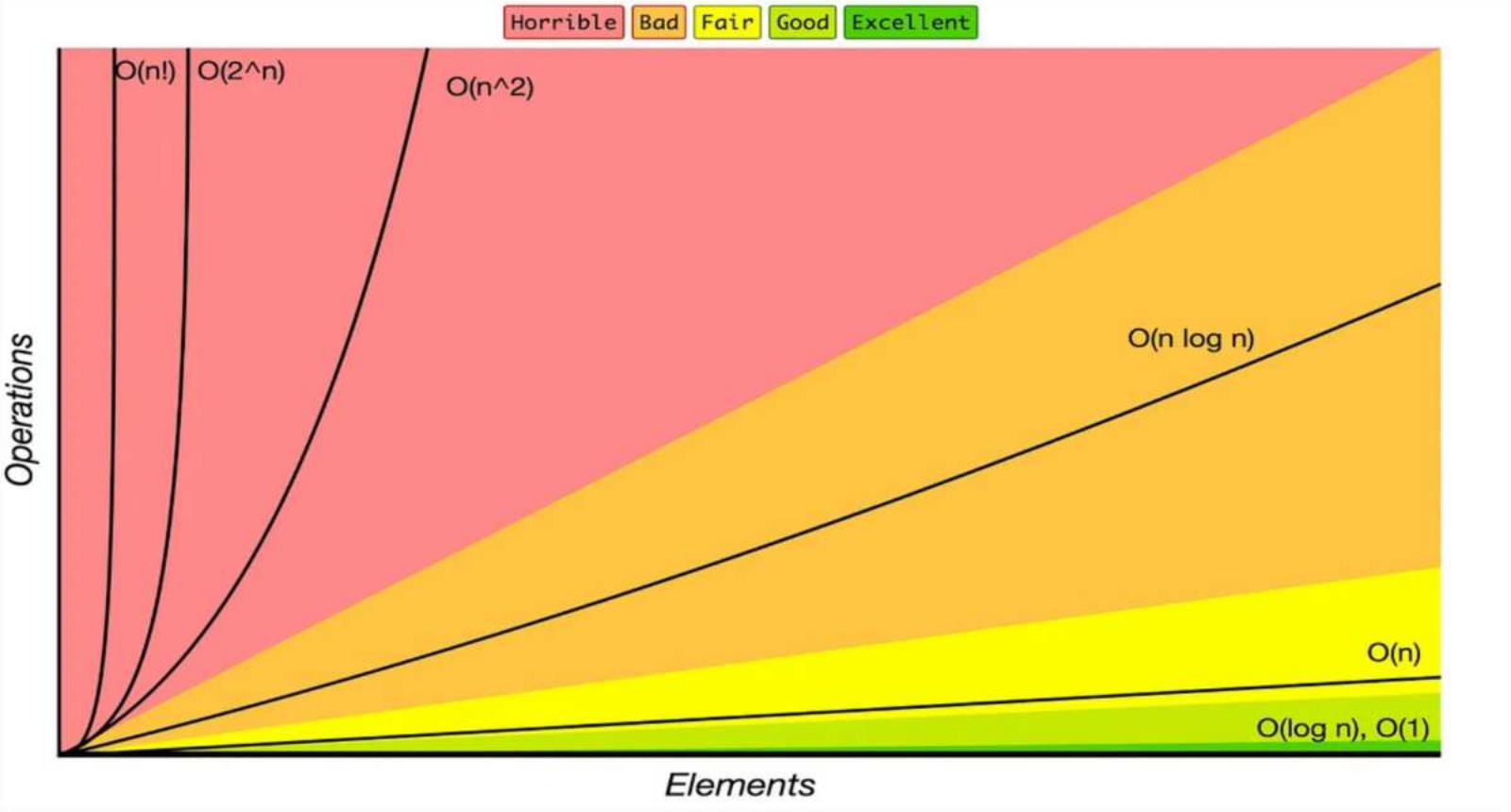
$2n$, $3n$, $1/2n$ $\rightarrow O(n)$

Yapı ne olursa olsun katsayılar yazılmaz.

* $2n^2 + 3n + 2$ şeklinde bir algoritma çıkarsa $2n^2 > 3n+2$ olduğu için bu ifadenin Big O rotasyonu $O(n^2)$ olur.

- İki farklı arama yöntemimiz var. Bunlardan A algoritması sayfa sayfa, B algoritması yarıya bölüp tarıyor. Sizce hangisi daha hızlı çalışır? Tabii ki B algoritması. Peki neden? Sürekli tarayacağı alan azalıyor. A algoritması daha işlemini bile yarılamamışken, B algoritması sonuca ulaşıyor.
- N tane işlem üzerinden big-o gösterimi yapalım. A algoritması input olarak kaç sayfa varsa o kadar işlem yapıyor. B algoritması ise sayfa sayısını azaltmak için alfabetik sıraya göre sağ ve sol olarak yarıya indiriyor.



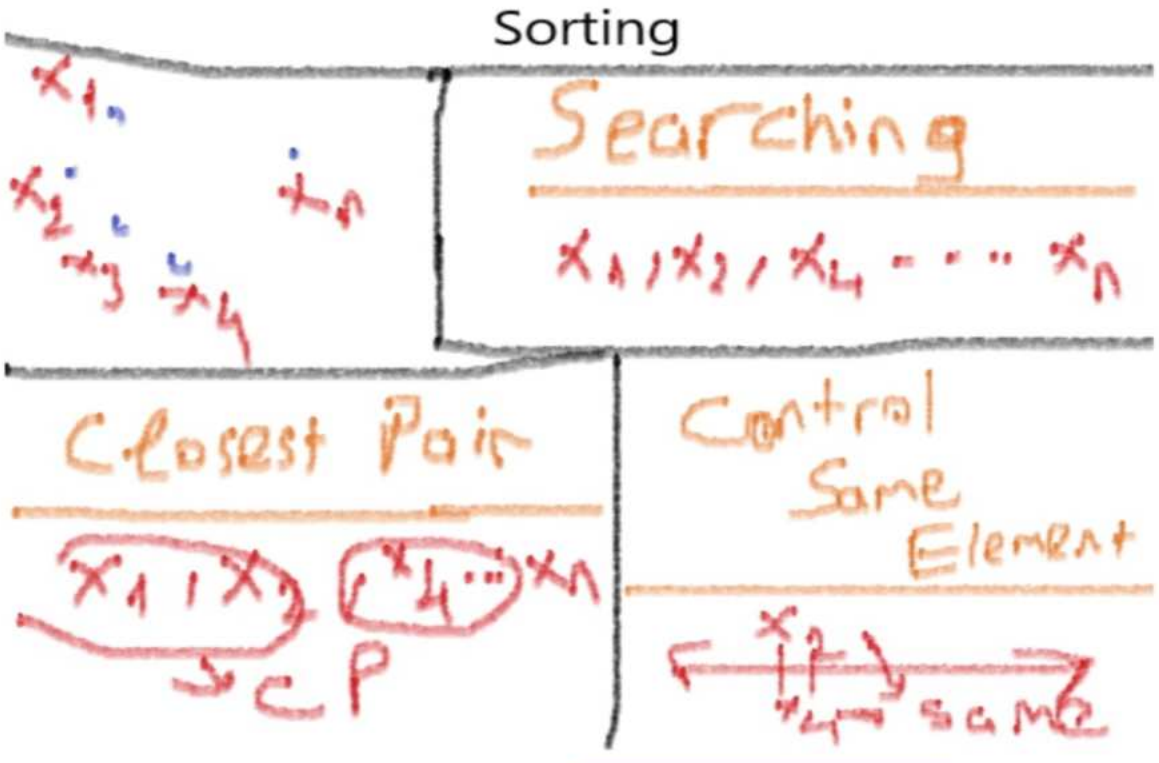


Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Sorting

Sorting, kendinden sıralama algoritmaları olarak bahsetmektedir. Sorting, bir eleman dizisini, belirli sıralama kurallarına göre sıralama yapar.



1. **Searching** yöntemini kullanarak elemanlarımızı sıraladık. Bunun sebebi, eleman ararken işimizin kolaylaşmasını istiyoruz.
2. **Closest Pair** yöntemini kullanarak birbirine yakın sayıları gruplandırdık ki arama yaparken zamanımızı efektif bir şekilde kullanalım.
3. **Aynı eleman kontrolü**: birbiriyle aynı olan sayıları örüntü içerisinde kaç tane aynı eleman varsa sayısını öğrenebilirim.
4. **Mode bulma**: eleman dizisini search ettikten sonra elemanların yan yana olanları sayarsam daha hızlı mode bulabilirim.

Selection Sort

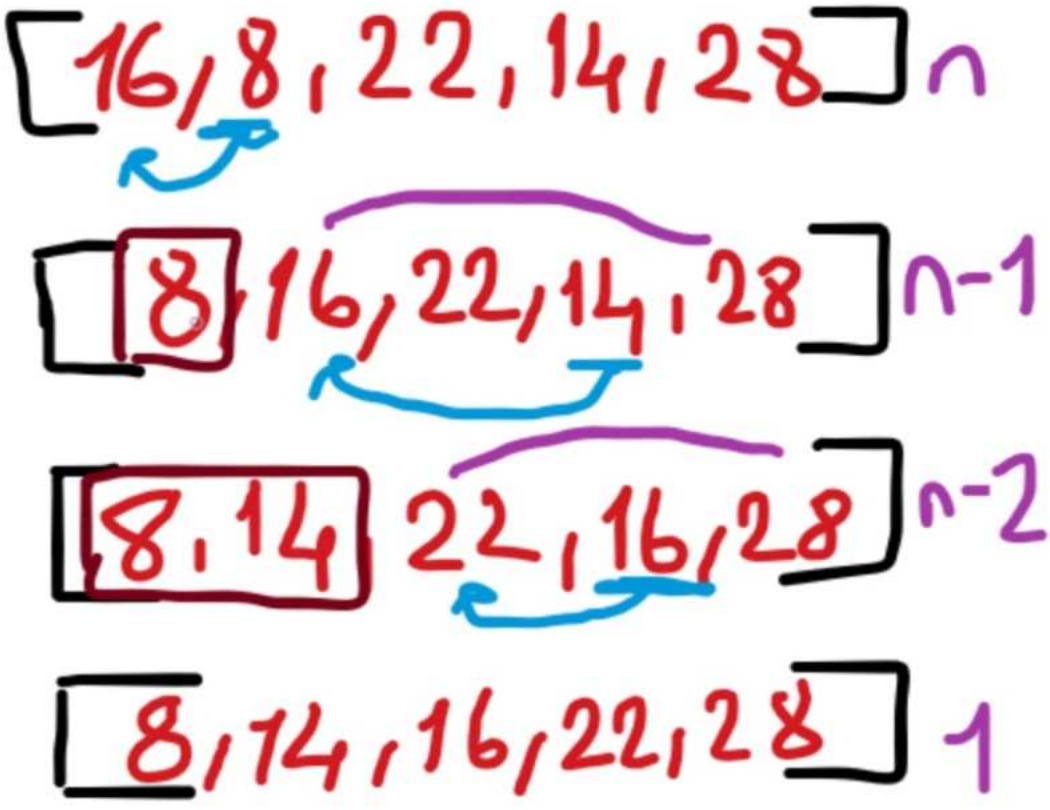
En basit sorting algoritmalarından biridir.

Tüm elemanlar arasından en küçük olanı buluyor ve en başa ekliyor.

Sonra kalan diğer elemanlara bakıyor ve 2. en küçüğü bulup 2. sıraya koyuyor.

...

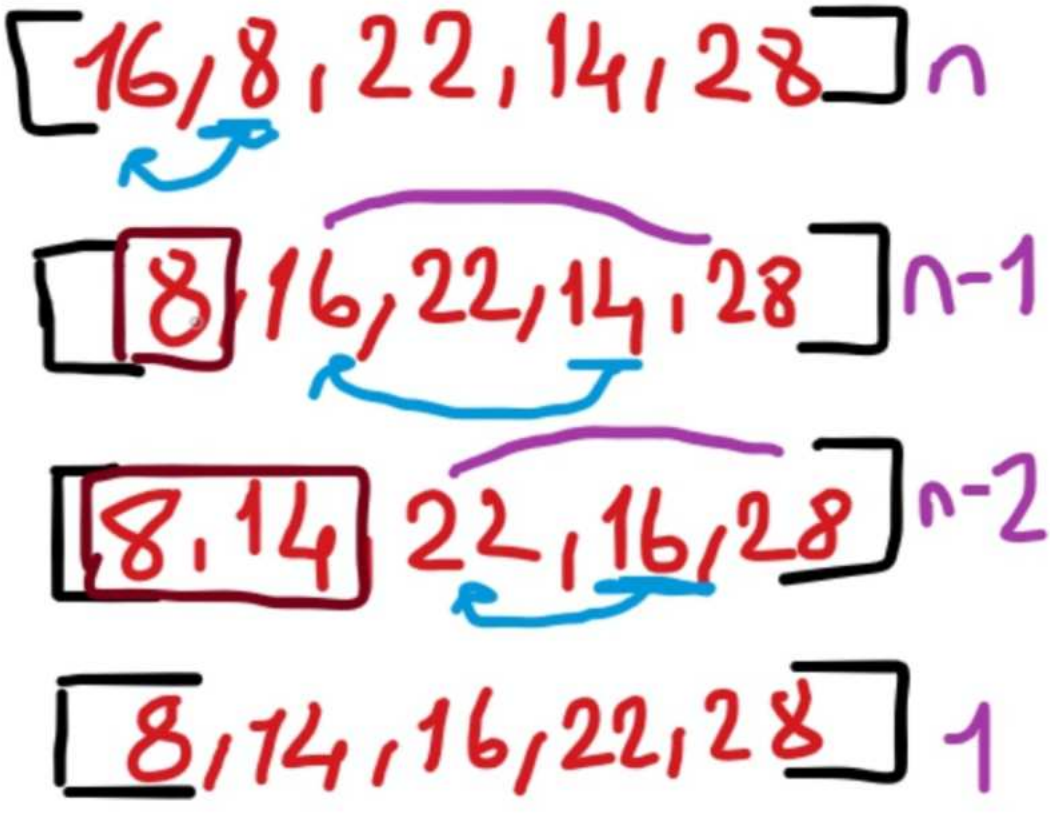
Devam ediyor ve liste küçükten büyüğe sıralanıyor.



* Selection sort hafızada ekstra olarak yer kaplamaz.

- Verilen örüntüye ait en küçük elemanı buluyor ve en baştaki sayı ile yer değiştiriyor. Peki ya devamı? İkinci en küçük elemanı buluyor ve 2. sıra ile değiştiriyor. Baktın ki 2.sıradaki eleman en küçük **hiç dokunma!!!**. Hemen 3. sıraya geç. 4, 5 derken dizi bitti. İşte insertion sort'un temel çalışma prensibini öğrendin.

Selection



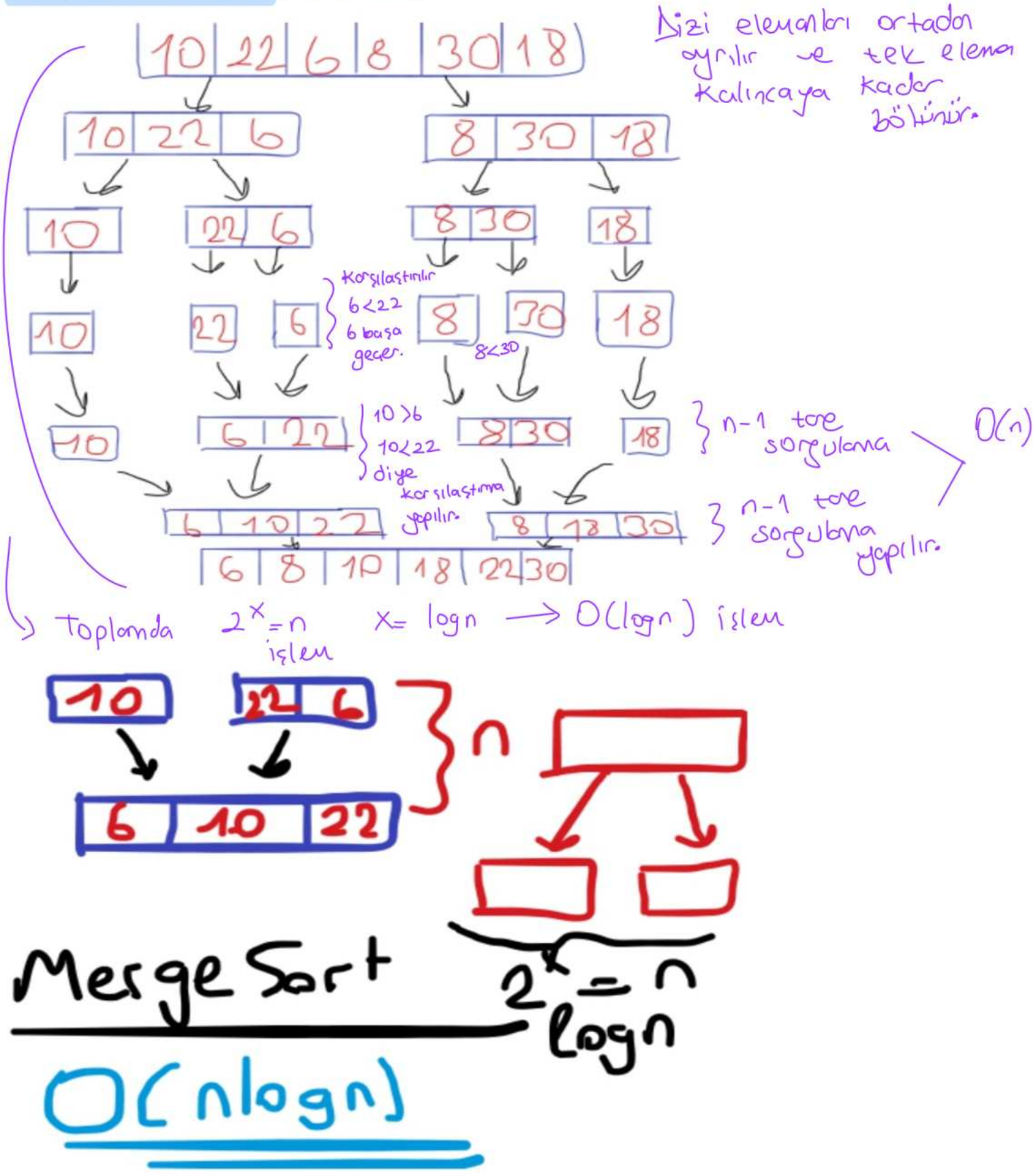
* Toplamda $1 + \dots (n-1) + (n-2) + n$ kadar işlem yapıyor.
  - Yani 1'den n'e kadar olan sayıların toplamı selection sortun algoritmasına aittir.
  - $\frac{n \cdot (n+1)}{2} = \frac{n^2+n}{2}$

Yukarıdaki ifadenin BigO notasyonu → $O(n^2)$ olur.

Merge Sort

Insertion Sort'da, Big-O gösteriminden dolayı input'um arttığında n^2 olduğunda dolayı çalışma zamanı artıyor.

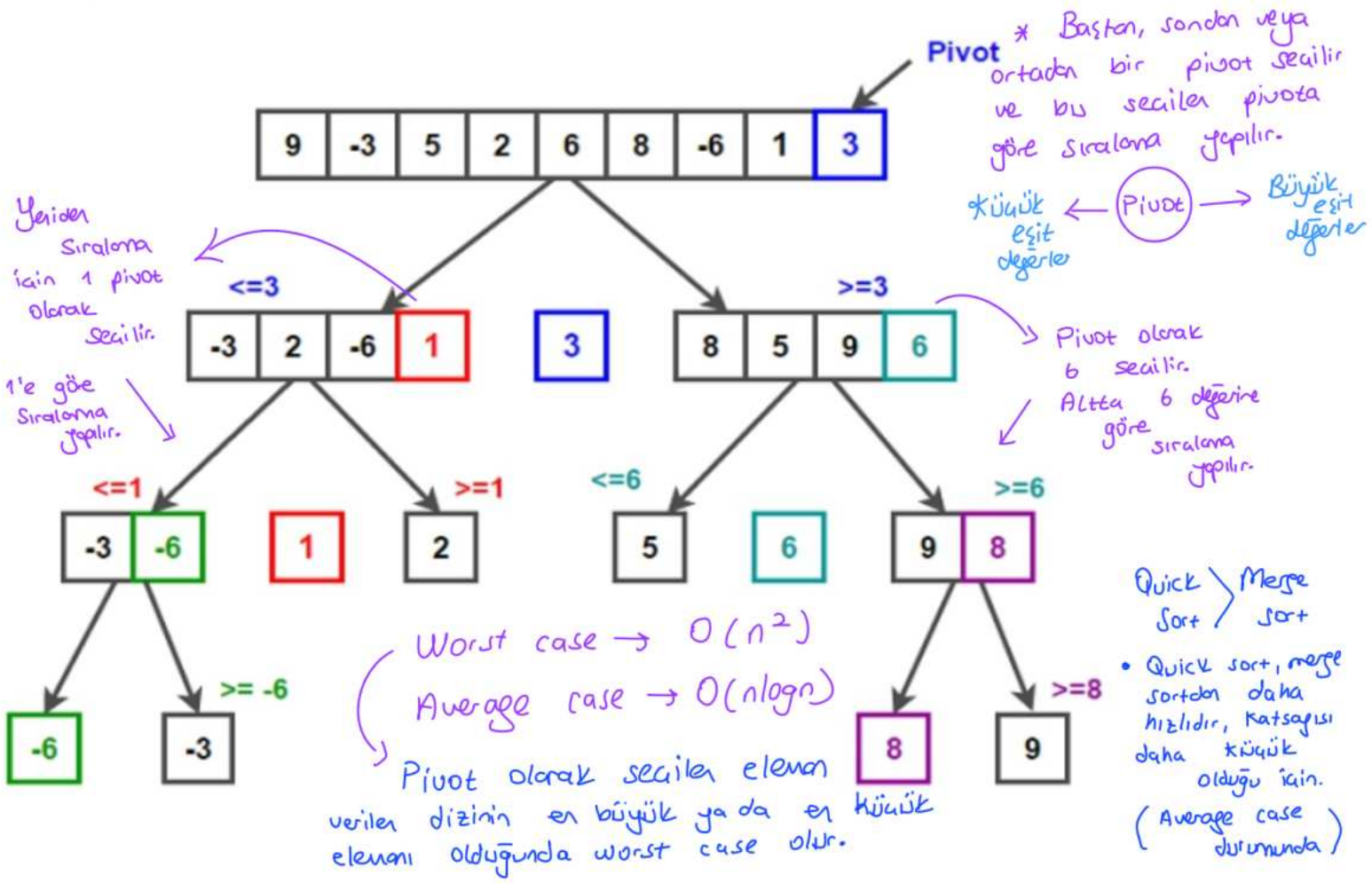
- Peki daha hızlı bir şekilde sıralama yapılabilir mi? Evet, Merge Sort burada yardımımıza koşuyor. Bir listeyi her adımda parçaya ayırıp tek eleman kalıncaya kadar bölüyor. Böldükten sonra sıralı bir şekilde bize sunuyor (Performans).



Insertion sort'da, time complexity n^2 olduğundan ötürü çalışma zamanımız artıyordu. Merge sort'da ise nlogn olduğu için açık ara performans olarak daha iyi diyebiliriz.

Quick Sort

Hızlı sıralama günümüzde çok yaygın olarak kullanılan bir sıralama algoritmasıdır. N tane sayıyı average case e göre big-o nlogn, worst case e göre big-o n^2 karmaşıklığı ile sıralanır.



- İlk olarak bir pivot belirler bu pivota göre pivottan küçük ve eşitler sol kısmına, pivottan büyük ve eşitler sağ kısmına yazılır. Parçalanmış kısımlar yeni bir pivot belirlenerek parça pinçik edilir.

Searching

- Günümüzde veriler gitgide artan bir hal alıyor. Her insanın bir bilgisayarı ve telefonu olduğunu düşünürsek, terabaytlarca veri ediyor. Arama algoritmaları ise istediğim özellikteki verinin elimdeki veri setlerinde aranıp, bulunup getirilmesi demek. Bunun hızlı olmasına önem gösterilir.

Linear Search

O(n) rotasyonuna sahiptir → Worst case'de.

Linear search, tek tek elemanları dolandıktan sonra istediğim elemanın olup olmadığına bakmaktır.

- Örneğin, [20,25,46,48] veri setini ele alalım. Benim aradığım eleman 25. İlk elemana gidiyorum ve değeri 20 sen değilsin diyorum. İkinci elemana gidiyorum ve değeri 25 evet sensin diyorum. Linear search algoritmam burada bitmiş oluyor.
- Big-o ya göre incelediğimizde bizim worst case'imiz neydi? Elemanın dizinin sonunda bulunmasıydı. Bu sebepten ötürü n elemanımız varsa big-o notasyonumuz otomatik olarak n oluyor.

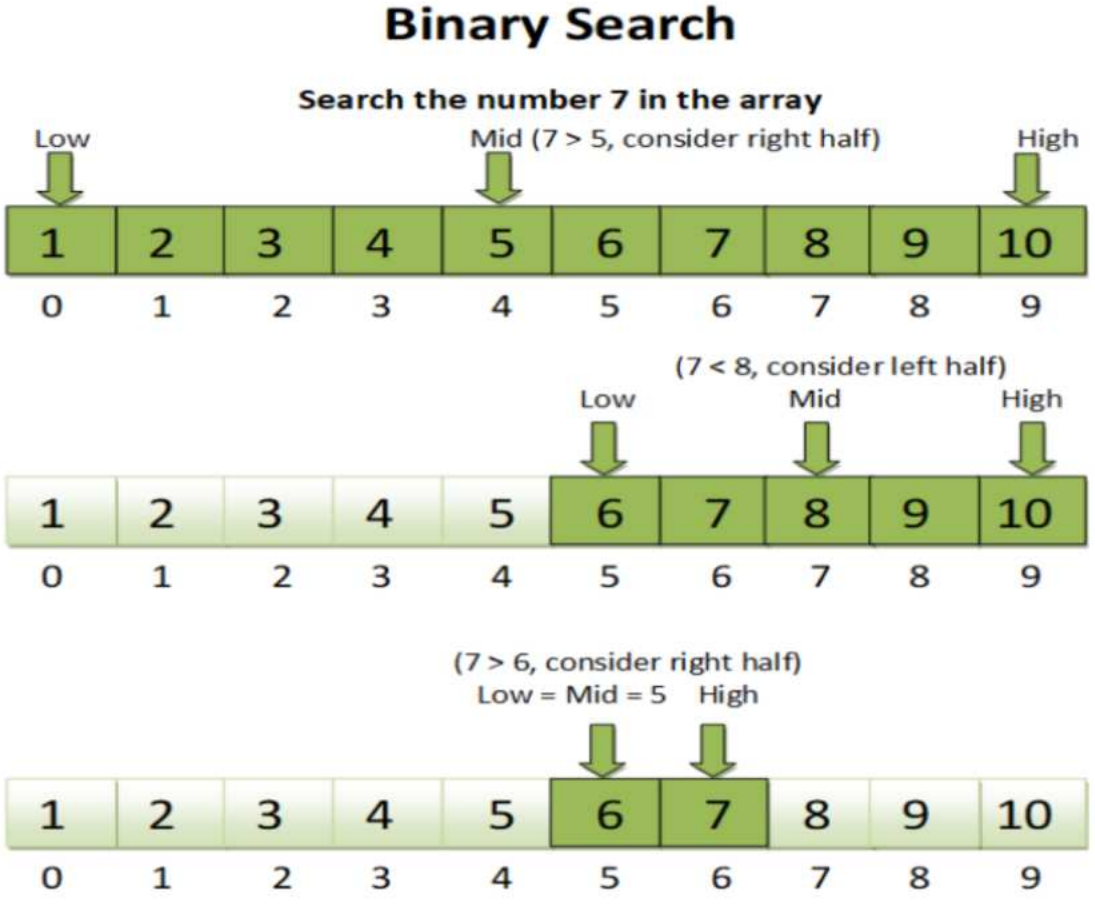
Binary Search

İkili arama algoritması, elimizde bulunan veri dizisini sıralı olduğunu varsayıyor, bu durumu değiştirerek sonuca varmak istiyor.

- İkili arama algoritması, diziyi her seferinde ikiye bölerek ikili arama yapar. Sıralı bir listem var ise benim Big-o logn olarak karşımıza çıkıyor.
- Aradığım sayı 15 ve benim değer kümem [10,15,20,16,22,36,23] diyelim. Binary Search bu diziyi manipüle ederek şu ifadeye dönüştürüyor. [10,15,16,20,22,23,36]. 36 sayısını en yüksek sayı, 10 sayısını en düşük sayı ilan ediyor. Benim aradığım sayı ile ortada kalan sayıyı kıyaslıyor eğer benim sayım büyükse kendinden küçük bütün sayıları siliyor. Ve kendine yeni bir ortanca belirliyor. Böylelikle gereksiz arama yapmaktan kurtarıyor.

- Önce liste sıralanır.
- Sonra orta nokta belirle
- Aranan sayı ile orta noktayı karşılaştır.

- Aranan sayı < Orta nokta sağ tarafı sil ve sola geç.
- Soldan orta nokta bul ve yine aranan ile karşılaştır.



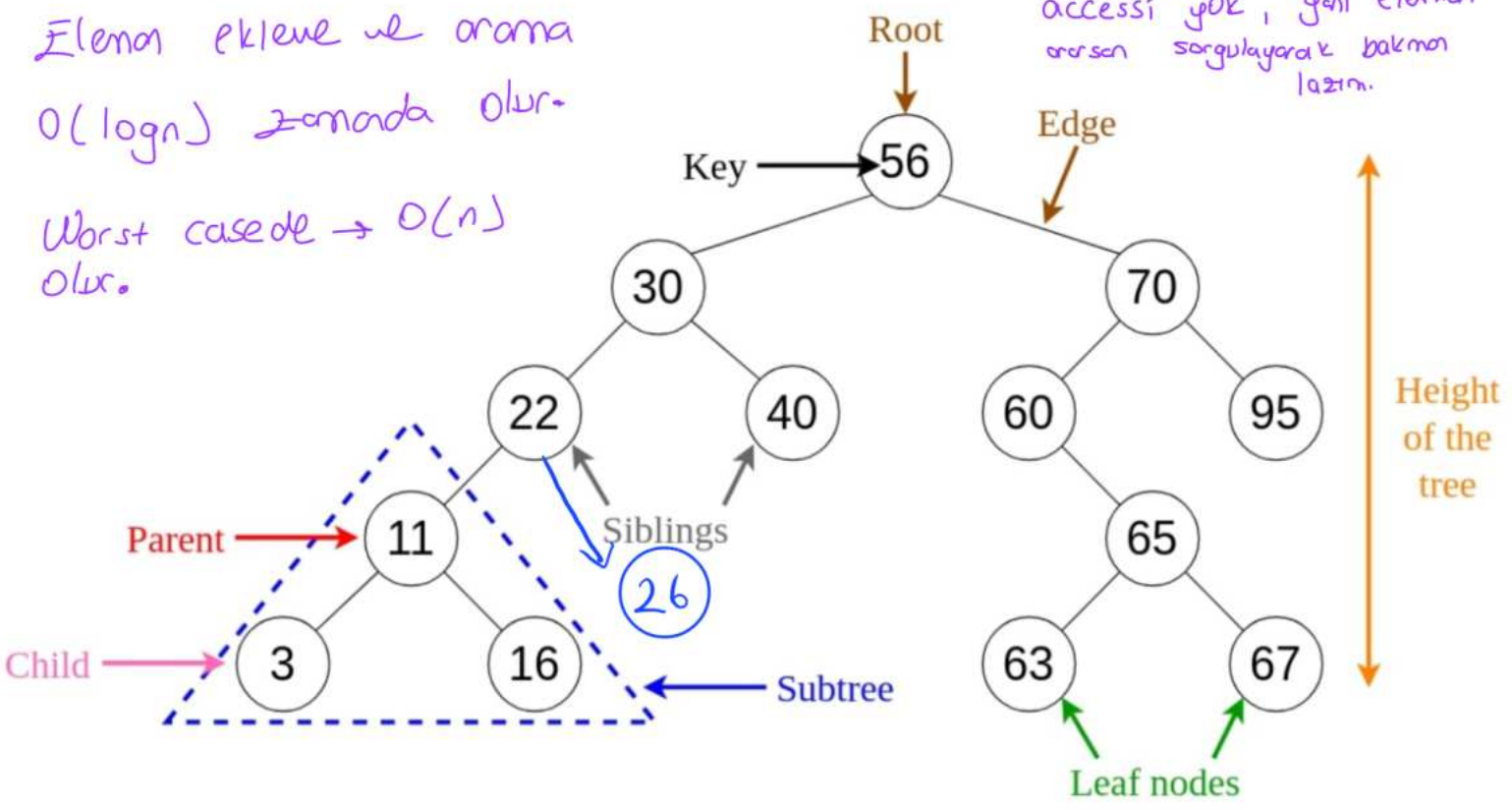
Binary Search Tree

Bir düğüm her iki tarafa da referans verebiliyor. Sağ ve sol olarak. Sağ tarafından kendinden büyük elemanlar, sol tarafında ise kendinden küçük elemanlar bulunacak.

* Arraylein aksine Random accessi yok, yani eleman ararsan sorgulayarak bakman lazım.

Eleman ekleme ve arama O(logn) zamanda olur.

Worst casede → O(n) olur.



- Tree'ye eleman eklemek istediğimde **root'dan** başlıyorum. Örnek olarak ben 26 sayısını ağaç yapısına eklemek istiyorum. Root'a soruyorum senin değerin ne 56. Baştaki açıklamamızı hatırlayalım. Sağ tarafında kendinden büyük, sol tarafında kendinden küçük elemanlar var. O yüzden sırasıyla 56 ve 30 a kadar ilerliyorum. 30 bana benim sol tarafıma geçmelisin çünkü sen benden küçüksün diyor. Karşıma 22 değerinde olan düğüm çıkıyor ve 22 den büyük olduğum için sağ tarafına bir köşe çekiyorum ve 26 sayısını bağlıyorum.

Sorting Algorithms

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	$O(nlogn)$	$O(n^2)$

3.Insertion Sort

Örneğin, masada bir deste oyun kağıdı, 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 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 kağıtlarını sıralamış oluruz.

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.

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

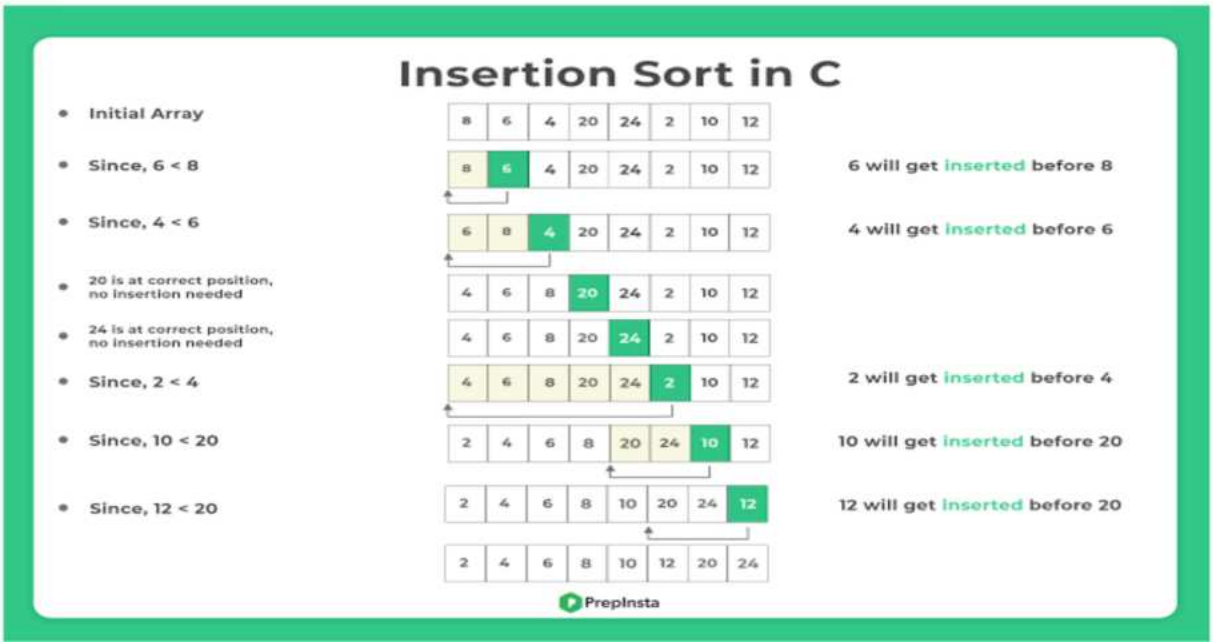
3.Insertion Sort Implementasyonu

```
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 (n2+n) veya O(n2) olur.



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. 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.

[5,1,12,-5,16]dizisinde Bubble Sort işletimi 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 geciste sıralama yapılmadıysa, bir sonraki gecişe geçme, işlemi bitir.
        if (!swapped)
            break;
    }
}
```

Quadratic işlem karmaşıklığına sahip olan Bubble Sort algoritması, büyük n değerlerinde en yavaş performansa sahip olan sıralama algoritmasıdır. Elemanların ilk dizilimi sıralama performansını önemli ölçüde etkiler. Örneğin Baştaki büyük değerli elemanların dizinin sonuna ilerlemesi hızlı iken Sondaki küçük değerli elemanların dizinin başına getirilmesi yavaştır.

Bu sorunu çözmek için "coctail sort" algoritması geliştirilmiştir.

1.Bubble Sort Karmaşıklığı

- İç döngü: (n – 1) + (n – 2) + ... + 1 = (n -1 + 1)/2
- Dış döngü: n
- Toplam İterasyon: n * (n / 2) = n2 / 2
- Big O Karmaşıklığı: O(n2)

Sözde kod, bilgisayar bilimleri alanında algoritmalar ve programlar oluşturulurken ve aktarılırken kullanılan, günlük konuşma diline benzer ve belli bir programlama dilinin detaylarından uzak anlatımlardır. Programın yapısının ve çalışma mantığının yüksek seviyeli bir biçimde, gerektiği yerde doğrudan doğal dil cümleleriyle, ama yine de bir program yapısı ve akışı içinde anlatılmasıdır. Böylelikle sözde kodu okuyan ya da yazan birisi, programlama dillerinin sözdizim detaylarına dikkat etmek zorunda kalmadan, programın ve algoritmanın çalışma mantığını düşünebilir.

Sözde kod için önceden üzerinde karar kılınmış kesin bir sözdizim yoktur. Sözde kod ile bir programı anlatan kişi, uygun gördüğü programlama dili yapılarının ve işlevlerinin sözde kod içinde bulunduğunu varsayabilir. Amaç, derleme işleminden hatasız çıkacak bir program oluşturmak değil, programın çalışma mantığını anlamak olduğu için, sözde kod yazarken uygun görülen herhangi bir soyutlama düzeyi kullanılabilir. Bazı sözde kodlar programlama dilleriyle büyük ölçüde örtüşürken, bazıları sadece program biçiminde yazılmış düz yazı olabilir.

Sözde kod örnekleri [değiştir | kaynağı değiştir]

Üçgenin alanının hesaplanması için sözde bir kod;

* Pseudo kod - kaba kod 'da denir.

```
Üçgenin taban ve yukseklik değerlerini al
alan = 1/2 * taban * yukseklik
alan değerini çıktı olarak ver
```

* Konuşma diliyle, teknik terimler bir arada kullanılır.

Kaynak kodu (İngilizce: *source code*), herhangi bir yazılımın işlenip makine diline çevrilmeden önce insanların okuyup üzerinde çalışabildiği programlama diliyle yazılmış halidir. Kaynak kod bir tümleşik geliştirme ortamında açılabilir, derlenebilir, çalışabilir kaynak kod dosyalarının tümü birleştirilip, hedef bilgisayarlarda kullanılabilir hale getirilebilir.

Ürün ve hizmet sırlarının kullandıkları yazılımda kayıtlı olduğunu düşünen kâr amaçlı kuruluşlar ekseriyetle ürünlerinin kaynak kodlarını gizlerler. Kaynak kodu gizlenmeyen yazılımlar 'açık kaynak', 'özgür yazılım' gibi isimlerle anılırlar. Özgür yazılıma örnekler GNU/Linux işletim sistemi ve "Eclipse" tümleşik geliştirme ortamıdır. Birçok kâr amaçlı kurum da güven, "good-will" kazanmak, irdeleme ve kullanım kolaylığı sağlamak veya özgür yazılım ekler üretimine olanak vermek gibi amaçlarla yazılımlarını açık kaynak hale getirmişlerdir.

Örnekler [değiştir | kaynağı değiştir]

C dilinde yazılmış örnek bir kaynak kod:

* Source code 'da denir.

```c
#include <stdio.h>
int main()
{
    printf("Merhaba Dünya");
}
```

* Herhangi bir programlama diliyle yazılmış kodlardır.