**Nedir Bu “Big O Notation”?**

Günümüzde sahip olduğumuz en değerli şeylerden birinin zaman olduğunu biliyoruz. Bu yüzden bir kullanıcın uygulamamız üzerinden yapacağı bir iş’e gerekli olduğu kadar zaman ayırmasını, tarayıcıyı veya uygulamayı sıkılıp/sinirlenip kapatmamasını sağlamamız gerekir.

Bir uygulamanın kullanıcı dostu bir yapıya sahip olması için, uygulama içerisinde yazdığımız algoritmaların performansının iyi olması gerekmektedir. Yazılan bir algoritmanın performansını ölçebilmemiz için kullanacağımız en önemli araçlardan biri ise***Big-O notation***’dır.

Big-O notation bir algoritmanın performansını veya time complexity’sini hesaplamak için kullanılır.

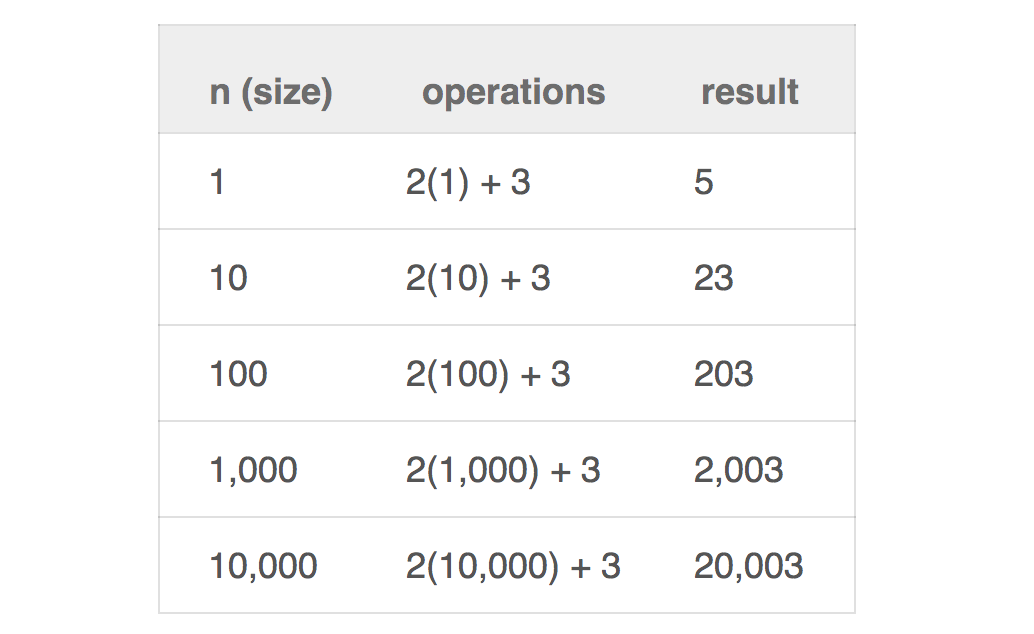
**Time Complexity Nedir?**

Time complexity bir algoritmanın çalışması için gerekli olan süredir. Ancak buradaki süre, saniyeleri hesaplayarak değil, kaç tane işlem gerçekleştirdiğine göre hesaplanmaktadır. Uygulama tarafından gerçekleştirilen işlem sayısı, veri setinin büyüklüğüne ve o veri setindeki elemanlarına sırasına göre belirlenir.

Örneğin bir dizinin en küçük elemanını bulma işlemine bakalım:

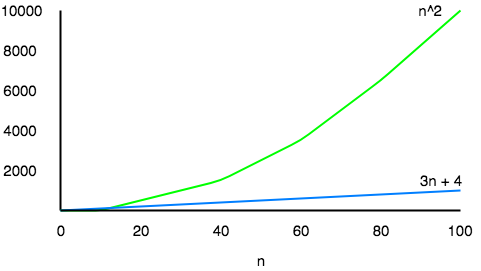
Satır 2: 1 işlem (Eşitlik toplama çıkarma çarpma gibi basit assignment gibi işlemlerin karmaşıklığını 1 alıyoruz)Satır 3: 1 işlemSatır 5–9 : Bir döngü işlemi gerçekleşir ve Array’in boyutu yani n değeri kadar döngü devam eder -> O(n)Satır 6: 1 işlem (if gibi karşılaştırma operatörlerin karmaşıklığını 1 alıyoruz)Satır 7: 1 işlemSatır 10: 1 işlem

Sonuç olarak, 3 işlem döngünün dışarısında ve 2 işlem döngü içerisinde gerçekleşir. Bu algoritmanın Time complexity’si 2(n) + 3'tür. Genelde bir algoritmanın complexity formülünde düşük önem taşıyan değerler bulunur. Örneğin bulmuş olduğumuz Time complexity’de n değeri arttıkça formülde bulunan sabit değerlerin önemi azalır.



Bu durumda bu değerleri yok sayabiliriz. Sonuç olarak yukarıda ki örneğimizin Time Complexity’si**O(N)** olur ve bu da ***Linear Complexity’e***eşittir. (Yazının devamında bulunan ***Big-O Terimleri ve Senaryoları***başlığı altında bu ve diğer terimlerin detaylarına erişebilirsiniz)

Bir başka örnek olarak; Elimizde bir algoritmanın Time complexity değerinin *T(n) =(n^2 + 3n + 4)* olduğunu düşünelim. *n* değerinin çok büyük olması durumunda, 3n+4 kısmı, n^2 kısmı ile karşılaştırıldığında önemsiz hale gelecektir.



studytonight — Asymptotic Notations

n = 1000 için n^2, 1000000 değerine sahip olurken 3n + 4, 3004 değerine sahip olur. Bu yüzden bu sabit kısımları yok sayabiliriz.

Time complexity bize bir algoritma için 3 durum sunar. **Best-case**, **average-case** ve **worst-case.**

Bu üç durumu örnekler ile açıklayacak olursak eğer:

Elimizde sıralı olmayan bir dizi olduğunu düşünelim. Örneğin [5,1,4,2,6,3,7]

Bu dizi de aradığınız bir değeri bulabilmek için dizideki tüm index değerleri ile aradığınız değerin eşleşmesi gerekmektedir (Linear Search). Bu örneğin case’lerine bir bakalım:

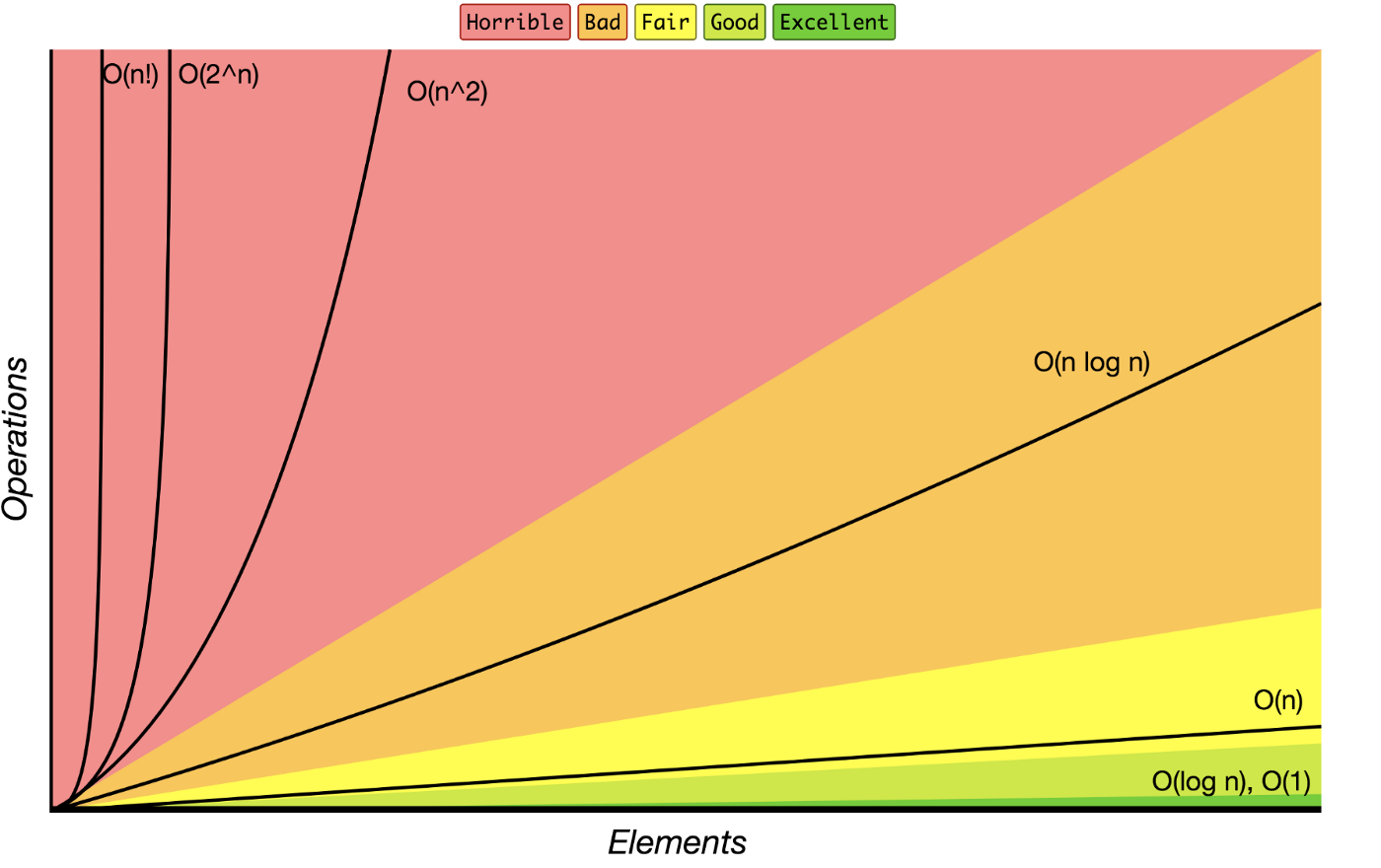
**Best-case**: Aradığınız değerin 5 olması bu algoritmanın best-case’dir. Çünkü daha ilk iterasyonda aradığımız değere ulaşmış oluruz.**Average-case**: Bu case, verilen input setindeki değerlerin dizi içerisindeki dağılımına göre belirlenir. Bu örnek için average-case aradığımız değerin, dizi’nin tam ortasındaki, yani 2 değerinin olması olabilir.**Worst-case**: Bu örneğimiz için worst-case, yani en kötü durum senaryosu, aradığımız değerin dizinin en sondaki değer olması durumudur. Bu örnek için bu değer 7'dir.**Worst-case:** Aradığımız değerin dizi içerisinde bulunmaması durumu da worst-case'e örnektir.

**Big-O Terimleri ve Senaryoları**

Big-O terimlerinden bazıları şu şekildedir:

* O(1) -> **Constant**
* O(N) -> **Linear**
* O(N^ 2) → **Quadratic**
* O(log N) → **Logarithmic**
* O(N log N) → **Linearithmic**
* O(c^N)→ **Exponential**
* O(N!) → **Factorial**

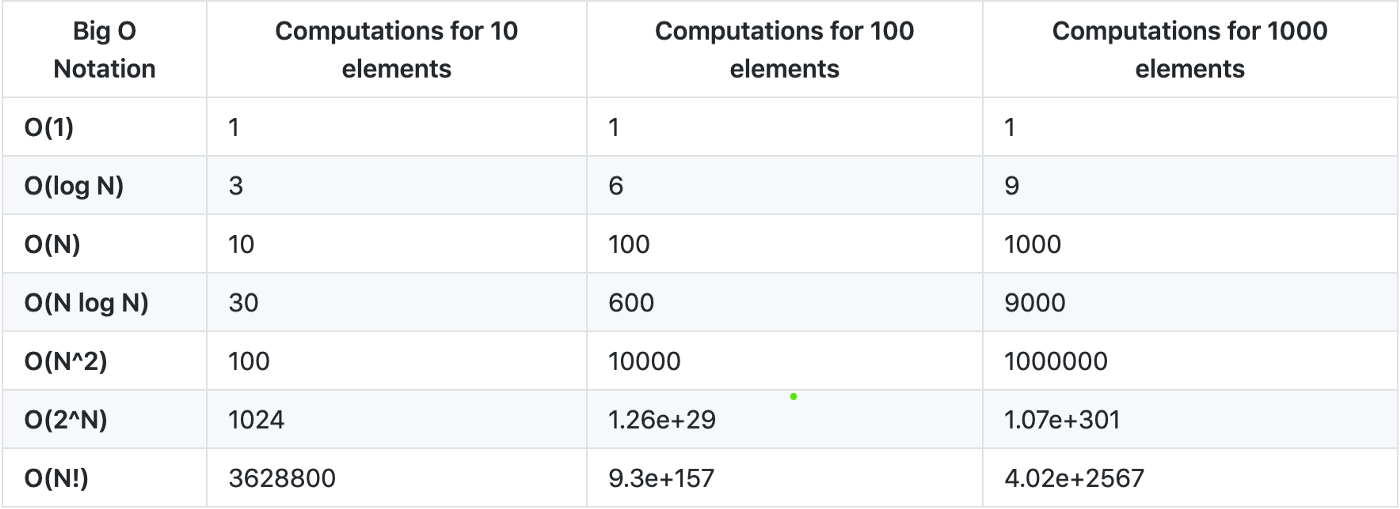
*(N = veri setinin büyüklüğü || eleman sayısı, c — sabit bir değer)*



trekhleb — javascript-algorithms

Yukarıdaki chart’ta, veri setine ve yapılan işleme göre algoritmanın Big-O sınıflandırması yer almaktadır.

Aşağıdaki tabloda ise en çok kullanılan Big-O terimlerinin farklı büyüklüklerdeki input setlerine göre karşılaştırmaları yer almaktadır.

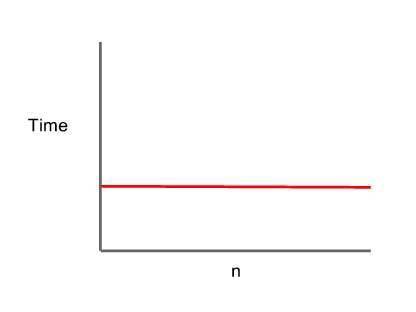


trekhleb — javascript-algorithms

Bu terimleri, örnekler ile beraber daha net anlamaya çalışalım.

**O(1) — Constant Complexity**

**Constant complexity’de** elinizde bulunan veri seti ne kadar büyük olursa olsun, çalıştırılma süresi ve kullanılan kaynak her zaman **sabittir**.



Jennifer Bland — Time Complexity Analysis in JavaScript

Örneğin:

Fonksiyona argüman olarak yazılan array ne kadar büyük olursa olsun, çalıştırılma süresi hep **sabit** olacaktır çünkü bizim istediğimiz array’in son elemanıdır.

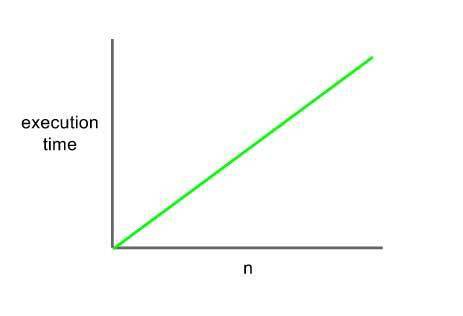
Fonksiyona argüman olarak yazılan bir değerin tek sayı mı çift sayı mı olduğunun kontrolü de constant complexity’e örnek olarak verilebilir.

Veya Time Complexity’i anlatırken kullanmış olduğum örneğin best case’i Constant Complexity’e örnektir.

**O(N) — Linear Complexity**

**Linear complexity’de**, elimizdeki veri seti arttıkça, çalıştırılma süreside doğru orantılı şekilde artış gösterir.

Günlük yaşantımızdan bir örnek verecek olursak eğer, gün içerisinde okuduğunuz bir kitab’ı veya izlediğiniz bir filmi ele alalım. Sizin ne kadar süre kitap okuduğunuz veya filmi izlediğiniz, kitabın sayfa sayısına veya filmin süresine bağlı olarak değişir. Eğer bir film 2 saat ise, siz 2 saatinizi filmi izlemek için harcamış olursunuz. Eğer kitap 100 sayfa ise ve siz 50 sayfasını 1 saatte okuduysanız, tüm kitabı okumak için 2 saat harcamanız gerekecektir.



Jennifer Bland — Time Complexity Analysis in JavaScript

Örneğin bir dizinin tüm elemanlarını yazdırma işlemi Linear Complexity’e örnektir çünkü array’in eleman sayısı arttıkça, fonksiyonun çalışma süresi de artacaktır.

Bu fonksiyonun time complexity’sine bakalım:

* Satır 2: n boyutunda bir döngü gerçekleşir
* Satır 3: For döngüsü içerisinde 1 işlem gerçekleşir.

Böylelikte Complexity olarak (N) + 1 değerini elde ederiz. 1 değerini, n arttıkça önemi azalacağı için yok sayabiliriz.

Bir başka örnek:

Array’deki bir değeri bulma işleminin best case scenario (en iyi senaryo) ve worst case senario (en kötü senaryo) durumlarını düşünecek olursak eğer:

let numberArray = [321, 12, 312, 3, 123, 4, 12, 43, 34, 64, 6453, 64, 412, 31, 2132, 13, 21, 421, 5, 5325, 142];findIndex(numberArray, 321); //0 (1 iterasyon gerçekleşir - Best case scenario - O(1))findIndex(numberArray, 142); //20 (21 iterasyon gerçekleşir - worst case scenario - O(N))findIndex(numberArray, 8); //-1 (21 iterasyon gerçekleşir - worst case scenario - O(N))

* **Best case scenario** — Aranan değerin index’i, ilk iterasyonda bulunur.
* **Worst case scenario**— Aranan değerin index’i, son iterasyonda bulunur.
* **Worst case scenario**— Aranan değer array içerisinde bulunmamaktadır.

Sıralanmamış bir array’de en büyük elemanı bulma örneğine bakalım;

findMax fonksiyonunun kaç işlem gerçekleştireceğine bakacak olursak eğer;

Array’daki her eleman kontrol edilir. Eğer o anki tanımlı maksimum değerden daha büyük bir değer ile karşılaşılmış ise, o değer maksimum değer olarak atanır.

Counter değişkeni ile kaç kez for döngüsünün gerçekleştiğini görebiliriz.

Bu fonksiyonun time complexity’sine bakalım:

* Satır 2–3: 2 işlem gerçekleşir.
* Satır 4: n boyutunda bir döngü gerçekleşir. — [O(N)]
* Satır 6–8: For döngüsü içerisinde 3 işlem gerçekleşir.

Sonuç olarak 3(N) + 2 time complexity değerini elde ederiz. Sabit değerleri yoksayarsak bu fonksiyonun complexity’si O(N) olmuş olur.

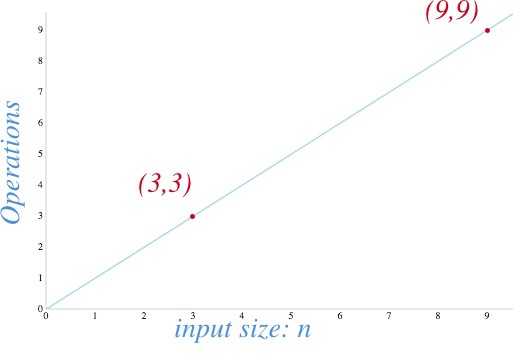
Eğer verisetinde 3 elamanı varsa, counter’ımız 3 olur.

findMax([3, 1, 2]);  
// n: 3, counter: 3

Verisetinde 9 eleman varsa, counter’ımız 9 olur.

findMax([4,5,6,1,9,2,8,3,7])  
// n: 9, counter: 9

Sonuç olarak, Linear complexity’de n değeri arttıkça, yani verisetimiz büyüdükçe, fonksiyonun çalıştırılma süresi de buna bağlı olarak artacaktır.

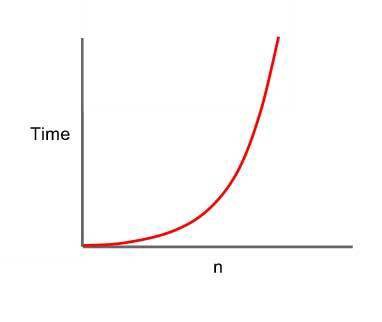


Adrian Meja — 8 time complexities that every programmer should know

**O(N²) — Quadratic Complexity Time Algorithm**

**Quadratic Complexity**input büyüklüğünün karesi ile doğru orantılıdır.

Eğer input büyüklüğü 2 ise, 4 işlem gerçekleşir. Eğer input büyüklüğü 8 ise, 64 işlem gerçekleşir ve bu şekilde devam eder. Genellikle sıralama (sorting) algoritmalarında bu complexity görülür.



Örneğin bir array içerisindeki her elemanı, array’in diğer elemanları ile yazdırma işleminde, iç içe for döngüleri kullanırız. Tek for kullanımında O(N) bir işlem gerçekleştiriz. Bu işlemi iç içe yaptığımızdan, N \* N sonucu ile O(N²) Time complexity değerini elde etmiş oluruz.

**O(log N) — Logarithmic Complexity**

Logarithmic time complexity, genelde her seferinde problemi ikiye bölen algoritmalarda kullanılır. Örneğin sözlükten bir kelime baktığımızı düşünelim. Sözlüklerde, her kelimenin alfabetik olarak sıralı olduğunu biliyoruz. Kelimeyi bulabilmek için yapılabilecek olanlar:

A algoritması:

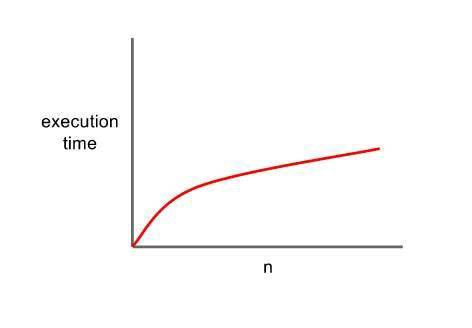
Kitabın en başından başlayarak sırasıyla kelimeyi bulana kadar gitmek.

B algoritması:

Kitabın ortasından açıp ilk kelimeye bakmak.

Eğer aranan kelime alfabetik olarak kitabın ortasında bulunan ilk kelimeden büyükse kitabın sağ tarafına, değilse sol tarafına bakmaya devam etmek.

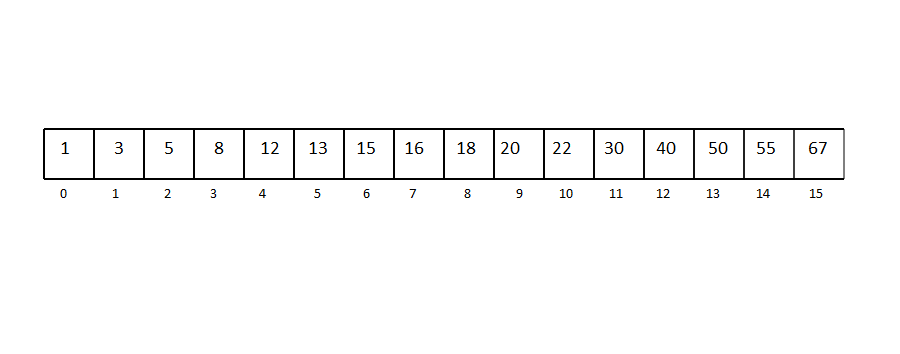
A algoritmasında kelime kelime gideceğimizden dolayı Time complexity değeri O(N)’dir. B algoritmasının Time Complexity’si O(logN)’dir. Her iterasyonda problemi ikiye böler. B algoritması aynı zamanda Binary search’e örnektir.



**Peki Neden Logaritmik?**

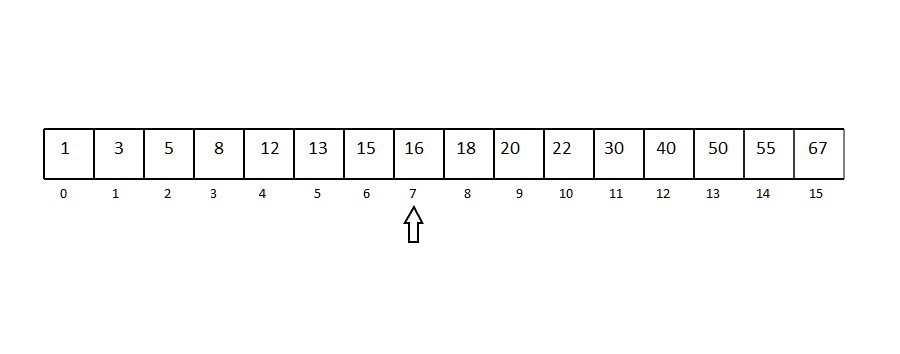
Bu kısmı açıklamak için [*What does the time complexity O(log n) actually mean?*](https://hackernoon.com/what-does-the-time-complexity-o-log-n-actually-mean-45f94bb5bfbf)yazısını referans alıyor olacağım.

16 elemanlı sıralanmış bir array’i düşünelim ve bu array içerisinde 13 sayısını arayalım.

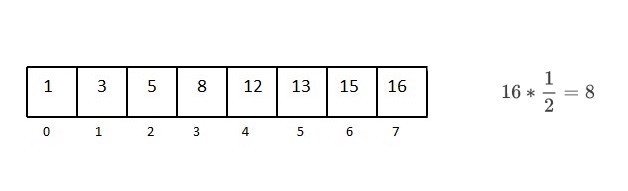


16 Elemanlı sıralanmış Array

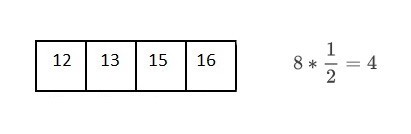
13 sayısını arama işlemini gerçekleştirirken, öncelikle dizinin ortasındaki değeri buluruz. Böylelikle 13 sayısının konumu, dizinin ortasından itibaren sağında yer elemanlar arasında mı yoksa sol tarafında yer alan elemanlar arasında mı olduğunu belirleriz.



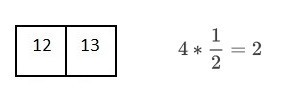
Array’in ortasındaki eleman seçilir



13, 16'dan düşük olduğu için dizinin ortasından itibaren sağ tarafında bulunan kısmı atarız.

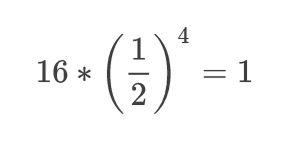


Aynı işlemi kalan kısım içinde tekrar gerçekleştiririz.

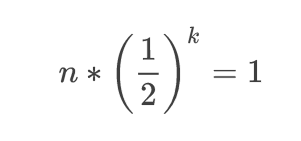


Gördüğünüz gibi, her arama işleminde diziyi ikiye bölerek aradığımız sonuca ulaşmaya çalıştık.

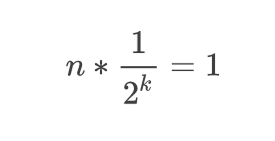
16 elemanlı dizi içerisinde 13 sayısına ulaşabilmek için array’i 4 kez böldük. Formülize edecek olursak eğer;



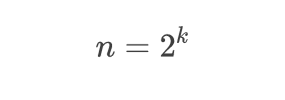
Genel olarak n eleman için;



k değerini paydaya alalım.



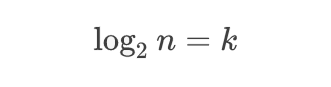
Sonuç olarak:



formülünü elde etmiş oluruz. Logaritmanın tanımına bakacak olursak eğer;

*Logaritma, üstel işlevlerin tersi olan bir matematiksel işlevdir. Örneğin 1000'in 10 tabanına göre logaritması 3'tür çünkü 1000, 10'un 3. kuvvetidir, 1000 = 10 × 10 × 10 = 10³.*

Bu da bize logaritmik olarak şu form’u verir:



Binary search’ün Flamenco versiyonunu da inceleyebilirsiniz :)

**O(c^N) — Exponential Time Complexity**

Base değeri (c) 2 olan Exponential Time Complexity’de, input N büyüklüğünde ise output 2^N olur.

Bu örneğimizde, verilen bir string’in tüm subsetlerini yani tüm alt kümeleri hesaplamaktadır (Örneği daha iyi anlayabilmek için [***Array.prototype.reduce()***](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce)konusunu incelemenizi tavsiye ederim*)*

Herhangi bir değer girilmemişse default olarak boş eleman döndürülür.

N = 0 --> f(N) = 2⁰= 1

getSubsets metoduna argüman olarak ‘a’ değerini verdiğimizde çıktı olarak bize [*‘ ’, ‘a’]*sonucunu verir.

N = 1 --> f(N) = 2¹ = 2

getSubsets metoduna argüman olarak ‘ab’ stringini verdiğimizde çıktı olarak bize [*‘ ’, ‘a’,’b’,’ab’]*sonucunu verir.

N = 2 --> f(N) = 2²= 4

* Farkedebileceğiniz üzere, fonksiyonu ilk çağırdığımızda bize boş elemanı döndürdü.
* İkinci durumda boş eleman + ilk eleman döndürüldü.
* Üçüncü durumda 2.durumun sonucu + ve yeni eklenen ‘b’ elemanın bir önceki durumdaki array’ın elemanları ile olabilecek subset sonuçları döndürüldü.

**Faydalı Kaynaklar**

* Diğer complexity’ler için [***Big-O Cheat Sheet***](https://www.bigocheatsheet.com/)***’ini***inceleyebilirsiniz.
* Sıralama algoritmalarının animasyonlara aktarılmış hallerini şuradan inceleyebilirsiniz → [***Sorting Algorithms Animations***](https://www.toptal.com/developers/sorting-algorithms)
* Yaygın olarak kullanılan algoritmaların ve veri yapılarının listesini bulabileceğiniz şu repository’i inceleyebilirsiniz → [***Javascript Algorithms and Data Structures***](https://github.com/trekhleb/javascript-algorithms)

Umarım faydalı bir yazı olmuştur. Eksik veya yanlış olduğunu düşündüğünüz kısımları bana iletirseniz çok sevinirim.

İletişim kanalları: [Twitter](https://twitter.com/SafaElmali) — [LinkedIn](https://www.linkedin.com/in/tsafaelmali/) — [Mail](mailto:tsafelmali@gmail.com)

Bir sonraki yazıda görüşmek üzere!

*Bu yazının referansları :*

[*https://www.jenniferbland.com/time-complexity-analysis-in-javascript/*](https://www.jenniferbland.com/time-complexity-analysis-in-javascript/)

[*https://dev.to/charlie117/big-o-notation-beginners-guide-1h38*](https://dev.to/charlie117/big-o-notation-beginners-guide-1h38)

[*https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/*](https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/)

[*https://adrianmejia.com/most-popular-algorithms-time-complexity-every-programmer-should-know-free-online-tutorial-course/*](https://adrianmejia.com/most-popular-algorithms-time-complexity-every-programmer-should-know-free-online-tutorial-course/)

[*https://www.studytonight.com/data-structures/aysmptotic-notations*](https://www.studytonight.com/data-structures/aysmptotic-notations)

[*https://hackernoon.com/what-does-the-time-complexity-o-log-n-actually-mean-45f94bb5bfbf*](https://hackernoon.com/what-does-the-time-complexity-o-log-n-actually-mean-45f94bb5bfbf)

6