GAZİ ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ

ŞEVVAL ÇAKMAK
22118080010

SIRALAMA VE ARAMA ALGORİTMALARI

BM102 – BİLGİSAYAR PROGRAMLAMA II
ARAŞTIRMA ÖDEVİ

NİSAN 2023

İÇİNDEKİLER

ÖZET

Bu araştırma ödevinde birçok algoritmayı ele alıp detaylıca inceledim. İlk olarak arama (search) algoritmaları başlığı altında linear search, binary search, ternary search algoritmalarının çalışma mekanizmalarını açıklayıp en iyi durum, ortalama durum, en kötü durum analizlerini yaptım. Ayrıca arama yapılırken kullanılan yönetemlerden ve bu algoritmaların kararlılık durumlarından bahsettim. İkinci olarak da sıralama (sorting) algoritmaları başlığı altında bubble sort, insertion sort, selection sort algoritmalarının yine aynı şekilde çalışma mekanizmalarını açıklayıp en iyi durum, ortalama durum, en kötü durum analizlerini yaptım. Arama yapılırken kullanılan yönetemlerden ve bu algoritmaların kararlılık durumlarından bahsettim. Bu 2 farklı kategorideki 6 algoritmayı incelerken tablolar üzerinden algoritmaların çalışma biçimini görselleştirip örneklendirdim. Ek olarak da her bir algoritma için pseudocode yazarak bu algoritmaların programlama dillerinde nasıl kodlanabileceğine değindim. EKLER kısmında 6 adet C# kodu bulunmakta ve böylelikle bu algoritmaların programlama dilinde nasıl kodlandığını, sorunsuz bir şekilde çalışıp çalışmadığını somut bir şekilde örneklendirmiş oldum.

Üzerinde sıklıkla durduğum bir konu da herhangi bir algoritmanın tüm durumlar için en efektif çözüm olamayacağıdır. Karşılaştığımız problemlerin çözümü için birçok algoritma geliştirilmiştir ve bunların neler olduğunu benim de bu araştırma ödevinde değindiğim özellikler ve diğer etkenler çevçevesinde değerlendirmek hangi problem için hangi algoritmanın kullanılması gerektiğine karar vermek için önemlidir.

Bahsedilen algoritmaların yeterince iyi anlaşılmasının bu kadar önemli olmasındaki temel sebep, program geliştirirken optimizasyonu sağlamak ve sorunları çözebilme hızımızı maksimuma çıkarmak ve uzun ömürlü yazılımlar geliştirebilmektir.

1. GİRİŞ

Algoritmalar bir problemi çözmek ve belirli bir görevi yerine getirmek için geliştirilmektedirler. Programlama dillerinin temelini oluşturan algoritmalar girdileri işleyerek çıktılar üretirler ve problemlerimiz için çözüm oluştururlar. Algoritmaların birçok çeşidi olabilir. Arama ve sıralama algoritmaları bu algoritmalardan bazılarıdır. Arama algoritmaları veriler arasında istenilen özellikteki verinin aranması gibi birok farklı amaçla kullanılabilir. Örnek olarak linear search, binary search, hashing, three search algoritmaları, backtracking verilebilir. Sıralama algoritmaları ise verileri istenilen düzene göre sıralamak için kullanılan algoritmalardır. Alfabetik, sayısal veya kronolojik olarak sıralama, arama algoritmalarında kullanma, veri analizleri gibi amaçlarla kullanılırlar. Örnek olarak bubble sort, insertion sort, selection sort, merge sort, quick sort, heap sort verilebilir.

Bu araştırma ödevinde ise bazı yaygın kullanılan arama ve sıralama algoritmalarının çalışma mantığının anlaşılması, örneklendirilmesi ve karşılaştırma yaparak birbirleri araslındaki ilişkilerin incelenmesi hedeflenmiştir.

2. ARAMA VE SIRALAMA ALGORİTMALARI

2.1. Arama (Search) Algoritmaları

Bilgi almak, hayatımızın birçok alanında önemli bir yer kaplar ve büyük veri tabanlarından istediğimiz öğeyi aramak bilgi almanın temel yönlerinden birisidir. Aradığımız elemanı bulabilmek için birçok arama algoritması geliştirilmiştir [1]. Bu arama algoritmaların çalışma mekanizmaları, durum analizleri, arama yapılırken ne gibi yöntemler kullanıldığı ve kararlılık durumları gibi etkenleri hangi algoritmayı kullanacağımıza karar verirken göz önünde bulundurmak optimizasyon açısından önemlidir. Yaygın olarak kullanılan linear search, binary search, ternary search algoritmalarını detaylıca inceleyelim.

2.1.1. Linear Search

Linear search bir dizideki elemanın var olup olmadığını, eğer var ise bu elemanın index'ini bulmamızı sağlayan basit bir algoritmadır. Bu algoritmanın temeli aradığımız elemanı dizideki ilk elemandan başlayarak tek tek karşılaştırmak ve eşit olmadığı sürece bir sonraki elemana geçerek aramaya devam etmektir. Bu yüzden dizinin arama yapmadan önce sıralı olması gerekmez. Eğer aranan eleman dizinin sonuna gelmeden bulunursa dizinin devamını karşılaştırmaya gerek olmadığından değerin bulunduğu iletilerek arama sonlandırılır[1].

Bu algoritma kodlanırken döngüler sayesinde dizideki elemanlar aranan eleman ile sırayla kaşılaştırılır. Eğer aranan eleman bulunursa karşılaştırılan dizinin index'i döndürülür ve arama işlemi sonlandırılır, bu işlem dizinin geriye kalan elemanları için devam etmez. Aranan eleman dizinin hiçbir index'inde bulunamazsa yanlış değer döndürülerek aranan elemanın bulunamadığı bilgisi iletilir.

Tablo 1: 10 elemanlı bir dizinin index'i ve elemanları

Index:	0	1	2	3	4	**5**	6	7	8	9
	9	3	2	0	8	4	7	6	1	5

Tablo 1'de bir dizi örneği verilmiştir. Bu dizi örneğini ele alarak '4' elemanının yerini linear search ile arayalım. Linear search algoritmasına göre dizinin ilk elemanı ile aramaya başlanır. Bu eleman 9'dur ve aranan elemanla uyuşmadığı için dizinin bir sonraki elemanına geçilir. Bu arama 3, 2, 0 , 8 elemanları için sırayla yapılır ve yine hiçbiri 4 ile eşleşmez. Bir sonraki elemanın 4 olduğu görülünce dizinin aranan elemanı bulunmuş olur ve index'i döndürülerek arama işlemi sonlandırılır. Diğer elemanlar için arama işlemi devam etmez.

Yapılan işlem aşağıdaki pseudocode örneğiyle de ifade edilebilir:

(Pseudocode 1)

```
function linearSearch (array, target)
        for each element in the array
                if match element == target
                        return the element's index
                end if
        end for
        write('NOT FOUND')
        return false
end function
```

[2]

Pseudocode 1'de gösterildiği gibi bu algoritmayı istediğimiz programlama dillerinde kodlayabilir ve gereksinimlerimiz doğrultusunda kullanabiliriz.

Bu algoritma için gereken süre dizimizdeki eleman sayısıyla doğru orantılı olduğu için O(N) kamaşıklığına sahiptir. En iyi durumda yani dizinin ilk elemanı aranan eleman ise zaman karmaşıklığı O(1), en kötü durumda yani dizinin son elemanı aranan eleman ise ya da aranan eleman dizide değilse O(N), ortalama durumda ise O(N/2)'dir[1].

Linear search algoritması mevcut dizinin elemanlarının sıralaması korunduğu için kararlı bir algoritma olarak kabul edilir. Ayrıca bu algoritmayı kullanmak küçük boyuttaki veriler için kolay ve kullanışlı olacaktır ancak çok büyük boyuttaki veriler için başka arama algortimalarının kullanılması çok daha verimli sonuçlar elde edilmesinde yardımcı olur.

2.1.2. Binary Search

Binary search algoritmasını uygulamak için öncelikle elimizdeki dizinin sıralanmış olması gerekir[1, 3]. Aradığımız elemanın sıraladığımız dizinin ortanca elemandan büyük ya da küçük olduğuna bakılır. Eğer büyükse aradığımız elemanın dizinin ilk yarısında olmadığı çıkarımı yapılarak dizinin ikinci yarısı yeni bir dizi olarak ele alınır ve tekrar ortanca elemanıyla kıyaslama yapılır. Böylelikle dizinin boyutu sürekli yarıya düşülür, aranan eleman bulunana kadar bu işlem devam eder.

Tablo 2.1: 10 elemanlı sıralı bir dizinin index'i ve elemanları

Index:	0	1	2	3	4	5	**6**	7	8	9
	1	2	3	4	5	6	7	8	9	10

Tablo 2.1'i inceleyerek binary search algoritmasını '7' elemanını arayarak uygulamalı anlatalım. Elimizdeki dizi sıralı olduğu için binary search algoritmasını uygulamamız için uygundur. Dizinin boyutu 10'dur ve ortanca eleman olmadığı için '5' elemanı ortanca olarak kabul edilir. 7 sayısı 5'ten büyük olduğu için dizinin ikinci yarısı Tablo 2.2'de gösterildiği gibi ele alınır.

Tablo 2.2: Tablo 2.1'deki dizinin ikinci yarısı

Index:	5	**6**	7	8	9
	6	7	8	9	10

İşlemlere devam edilir ve elimizde kalan yarının ortancası bulunur. Ortanca 8 elemanıdır ve aradığımız elemanla ilişkisi tekrar kıyaslanır. 7 sayısı 8'ten küçük olduğu için bu sefer mevcut dizinin Tablo 2.3'te görüldüğü gibi ilk yarısıyla sonuç bulunana kadar işlem devam ettirilir.

Tablo 2.3: Tablo 2.2'deki dizinin ilk yarısı

Index:	5	**6**
	6	7

Tablo 2.3'teki dizinin ortancası bulunur, bu eleman 6'dır. 7 sayısı 6'dan büyük olduğu için dizinin tekrar ikinci yarısı alınır.

Tablo 2.4: Tablo 2.3'deki dizinin ikinci yarısı

Index:	**6**
	7

Tablo 2.4'te görüldüğü üzere tek eleman kalmıştır ve bu eleman ortanca elemandır. Bu elemanın aradığımız eleman olduğu görülür ve arama işlemlerine son verilir. Bulunan elemanın index'i dizide aradığımız elemanın yeridir.

Yapılan işlem aşağıdaki pseudocode örneğiyle de ifade edilebilir:

(Pseudocode 2)

```
function binarySearch(array, target)
        left = 0
        right = size of array – 1
        while left <= right
                middle = (left + right) / 2
                if array[middle] < target
                        left = middle + 1
                if array[middle] > target
                        left = middle -1
                if array[middle] == target
                        return middle
        end while
        write('NOT FOUND')
        return false
end function
```

[4]

Pseudocode 2'de algoritmanın ana hattları belirtilmiştir, bu sayede istediğimiz programlama dilinde bu algoritmayı kullanabiliriz.

Binary search algoritmasının karmaşıklığı $O(\log_2 N)$'dir ve büyük boyutlu verilerde linear search'e göre çok da kullanışlıdır[3]. En iyi durumda bu karmaşıklık O(1), ortalama ve en kötü durumlarda ise $O(\log_2 N)$'dir.

Bu algoritma arama yaparken elemanların sıralı olduğu diziyi kullanır, her adımda dizinin yarısını eleyerek arama algoritmasını gerçekleştirir. Bu nedenle tekrar eden verilerin olması vb. durumlardan dolayı orijinal sıralama değişebilir. Kısacası binary search kararlı bi algoritma değildir.

2.1.3. Ternary Search

Ternary search algoritması linear search'le benzerlik gösterir ancak bu algoritmanın daha verimli bi şekilde kullanılmasını hedefler. Dizideki elemanlar öncelikle sıralı olmalıdır, sıralı değilse sıralanmalıdır[3].

Tablo 3.1: 12 elemanlı sıralı bir dizinin index'i ve elemanları

Index:	0	1	**2**	3	4	5	6	7	8	9	10	11
	1	4	9	13	15	22	27	29	32	36	39	43
	left			mid1				mid2				right

Bu algoritma ile 9 elemanını arayalım;

mid1 = left + (right - left) / 3 (1)

mid2 = right - (right – left)/3 (2)

Denklem 1 ve Denklem 2 kullanılarak dizinin mid1 ve mid2 elemanları bulunur, Tablo 3.1'de bulunan elemanlar belirtilmiştir. Aradığımız eleman mid1 ve mid2 ile kıyaslanır, mid1'den küçük olduğu görülür, mid1 ve öncesi tekrar değerlendirmeye alınır.

Tablo 3.2: Tablo 3.1'deki dizinin mid1'i ve öncesi

Index:	0	1	**2**	3
	1	4	9	13
	left	mid1	mid2	right

Tablo 3.2 elde edilmiştir, bu tablo üzerinde Denklem 1 ve Denklem 2 kullanılarak yeni mid1 ve mid2'ler bulunur, aranan eleman ile kıyaslanır. Aranan eleman mid2'de olduğu için mid2'nin index'i döndürülerek arama işlemi sonlandırılır.

(Pseudocode 3)

```
function ternarySearch (array, left, right, target)
If left <= right
        mid1 = left + (right - left) / 3
        mid2 = right - (right – left)/3
        if array[mid1] == target
                return mid1
        if array[mid2] == target
                return mid2
        if target < array[mid1]
                call ternarySearch (array, left, mid1 – 1, target)
        if target > artray[mid2]
                call ternarySearch (array, mid1 + 1, right, target)
        else
                call ternarySearch (array, mid1 + 1, mid2 - 1,
target)
else
        write('NOT FOUND')
        return false
end function
```

[5]

Pseudocode 3'de bu algoritmayı nasıl kullanacağımız sözde kod ile açıklanmıştır, istediğimiz programlama dilinde bu algoritmayı kodlamamız mümkündür.

Ternary search algoritmasının karmaşıklığı $O(\log_3 N)$'dir. Binary search algoritmasıyla karşılaştırıldığında bu algoritma her adımda dizinin üçte birini alarak arama işlemini gerçekleştirir. Büyük verilerde arama yapılacağı zaman linear search ve binary search algoritmalarına göre daha verimli ve efektif bir arama algoritmasıdır[1, 3]. En iyi durumda bu karmaşıklık O(1), ortalama ve en kötü durumlarda ise $O(\log_3 N)$'dir.

Bu algoritma arama yaparken elemanların sıralı olduğu diziyi kullanır, her adımda dizinin yarısını eleyerek arama algoritmasını gerçekleştirir. Bu nedenle tekrar eden verilerin olması vb. durumlardan dolayı orijinal sıralama değişebilir. Kısacası ternary search de binary search gibi kararlı bi algoritma değildir.

2.2. Sıralama (Sorting) Algoritmaları

Sıralama algoritmaları günümüzde bilgisayar mühendisleri tarafından oldukça ihtiyaç duyulan ve sık sık kullanılan bir algoritma türüdür. Günümüze kadar bu algoritmaların daha hızlı çalışabilmeleri için birçok farklı yöntem geliştirilmiştir. Bir sıralama algoritması hiçbir zaman tüm durumlar için en iyisi olamaz, bu yüzden ihtiyacımıza göre bu algoritmaları analiz etmeli ve kullanmalıyız[6, 7]. Yaygın olarak kullanılan bubble sort, insertion sort, selection sort algoritmalarının işleyişini ve ne gibi ihtiyaçlarımızı karşılayacağını detaylıca inceleyelim.

2.2.1. Bubble Sort

Bubble sort algoritması basit tekrarlı olarak çalışan bir algoritmadır. Sıralama algoritması uygulanırken her bir çift karşılaştırılır ve eğer öğeler yanlış sıradalarsa yerleri değiştirilir. Tüm bu işlemler değiştirilmesi gereken başka öğe kalmayana kadar uygulanır[6].

Tablo 4.1: 10 elemanlı bir sırasız bir dizi

9	7	2	4	3	5	6	8	1	0

Tablo 4.1.1: Çiftlerin sırayla karşılaştırılma aşamaları

9	7	2	4	3	5	6	8	1	0
7	9	2	4	3	5	6	8	1	0
7	2	9	4	3	5	6	8	1	0

…

7	2	4	3	5	6	8	1	9	0
7	2	4	3	5	6	8	1	0	9

Tablo 4.1.1'de koyu renkle gösterilen elemanlar sırayla çiftler haline kıyaslanmış ve büyük olan elementin yeri sağdaki ile değiştirilerek olması gereken konuma ilerlemiştir. Dizinin en büyük elemanı en sağa ulaşmıştır, her döngü sonunda sonda sıralı olan elementlerın sayısı artar ve tüm elemanlar tek tek sıralandığında Tablo 4.2'deki gibi sıralı bir dizi elde edilmiş olunur.

Tablo 4.2: Bubble sort algoritması ile sıralanmış bir dizi

0	1	2	3	4	5	6	7	8	9

Yapılan işlem aşağıdaki pseudocode örneğiyle de ifade edilebilir:

(Pseudocode 4)

```
function bubbleSort (array)
size = length of array
for i = 0 to size - 1
        for j = 0 to size – 1
                if list[j] > list[j + 1]
                        swap( list[j], list[j+1])
                        swapped = true
        if(not swapped)
                break
end for
return array
```

[8]

Pseudocode 4'de bu algoritmayı nasıl kullanacağımız sözde kod ile açıklanmıştır, istediğimiz programlama dilinde bu algoritmayı kodlamamız mümkündür.

Bubble sort algoritmasının karmaşıklığı O(N^2)'dir. Büyük verilerde sıralama yapılacağı zaman bubble sort algoritması kullanışlı değildir[6, 7]. En iyi durumda bu karmaşıklık O(N), ortalama ve en kötü durumlarda ise O(N^2)'dir.

Ayrıca bu algoritma kararlı bir algoritmadır, her adımda yan yana bulunan elemanlar arasındaki karşılaştırmaları yaparak sıralama işlemini gerçekleştirir. Bu işlem sırasında, eşit elemanlar arasında yer değiştirme yapılmaz ve sıralama sonrasında dizideki elemanların sıralaması değişmez. Bu nedenle, bubble sort algoritması kararlı bir algoritmadır.

2.2.2. Insertion Sort

Insertion sort algoritması selection sort ile benzer basit bir algoritmadır, index'i 1 olan elemandan başlanarak öncesindeki elemanlarla arasındaki ilişki incelenir ve eğer küçükse swap algoritması ile yerleri değiştirilir. Öncesindeki tüm elemanlar için bu işlem yapıldığında index değeri 1 artırılarak işlemlere devam edilir[6].

Tablo 5.1: 10 elemanlı bir sırasız bir dizi

9	7	2	4	3	5	6	8	1	0

Tablo 5.1'de index'i 1 olan eleman 7'dir. Solundaki elemandan küçük olduğu için swap algoritması uygulanır ve kıyaslanacak başka bir eleman kalmadığı için 2. index'e geçilir.

Tablo 5.1.1: Insertion sort ilk aşama

7	9	2	4	3	5	6	8	1	0

2 numaralı index'te 2 elemanı vardır, bu önce 9 ile kıyaslanır ve yerleri değiştirilir tekrardan 7 ile kıyaslanır ve yine yerleri değiştirilerek 2 numaralı index için de işlemler tamamlanmış olur.

Tablo 5.1.1: Insertion sort aşamaları

2	7	9	4	3	5	6	8	1	0
2	4	7	9	3	5	6	8	1	0
2	3	4	7	9	5	6	8	1	0
2	3	4	5	7	9	6	8	1	0
2	3	4	5	6	7	9	8	1	0
2	3	4	5	6	7	8	9	1	0
1	2	3	4	5	6	7	8	9	0

Tablo 5.1.1'deki koyu renkli kutular sollarındaki elementlerle kıyaslanarak algoritma aşamalarına devam edilmiştir ve sonuncu index de kıyaslandıktan sonra tablo 5.2'deki sıralama insertion sort kullanılarak elde edilmiştir.

Tablo 5.2: Insertion sort algoritması ile sıralanmış bir dizi

0	1	2	3	4	5	6	7	8	9

Yapılan işlem aşağıdaki pseudocode örneğiyle de ifade edilebilir:

(Pseudocode 5)

```
function insertionSort(array)

    size = length of array
```

```
        for i = 1 to size - 1 do
                j = i
                while j > 0 and A[j-1] > A[j] do
                        swap(A[j], A[j-1])
                        j = j – 1
                end while
        end for
end function
```

[9]

Pseudocode 5'de bu algoritmayı nasıl kullanacağımız sözde kod ile açıklanmıştır, istediğimiz programlama dilinde bu algoritmayı kodlamamız mümkündür.

Insertion sort algoritmasının karmaşıklığı O(N^2)'dir. Büyük verilerde sıralama yapılacağı zaman insertion sort algoritması kullanışlı değildir ancak küçük boyutlu ve neredeyse sıralı listeler için etkilidir[6, 7]. En iyi durumda bu karmaşıklık O(N), ortalama ve en kötü durumlarda ise O(N^2)'dir.

Ayrıca insertion sort algoritması kararlı bir algoritmadır, eşit elemanlar arasında yer değiştirme yapılmaz.

2.2.3. Selection Sort

Selection sort algoritması Bölüm 2.2.2.'de de bahsedildiği gibi insertion sorta benzer ve en basit sıralama algoritmalarından birisidir. Küçük boyutlu veriler için uygundur ve her seferinde yalnızca tek bir elementin yer değiştiriyor olması oldukça önemlidir. Avantajları basit ve küçük veriler için kullanışlı olmasıdır[6].

Tablo 6.1: 10 elemanlı bir sırasız bir dizi

9	7	2	4	3	5	6	8	1	0

İlk elemandan başlanarak tüm dizi boyunca en küçük olan veri bir minimum değere atanır ve elde edilen minimum değer ilk eleman ile yer değiştirilir. Daha sonra ikinci elemandan başlanarak dizideki geri kalan elemanlar arasındaki en küçük veri tekrar minimum değere atanır ve ikinci elemanla yer değiştirilir. Tüm elemanlar için bu döngü tekrar edilir. Algoritma sonlandığında ise Tablo 6.2'deki gibi sıralanmış bir dizi elde edilmiş olunur.

Tablo 6.2: Selection sort algoritması ile sıralanmış bir dizi

0	1	2	3	4	5	6	7	8	9

Yapılan işlem aşağıdaki pseudocode örneğiyle de ifade edilebilir:

(Pseudocode 6)

```
function selectionSort (array)
size = size of array
        for i = 1 to size – 1
                min = i
                for j = i+1 to size
                        if list[min] > list[j]
                                min = j;
                        end if
                end for
                if indexMin != i  then
                        swap list[min] and list[i]
                end if
```

```
        end for
end function
```

[10]

Pseudocode 6'de bu algoritmayı nasıl kullanacağımız sözde kod ile açıklanmıştır, istediğimiz programlama dilinde bu algoritmayı kodlamamız mümkündür.

Selection sort algoritmasının karmaşıklığı O(N^2)'dir. Küçük veriler için basit ve uygun bir sıralama algoritmasıdır[6, 7]. En iyi durumda bile her elemanın en az bir kere karşılaştırılması gerektir bu yüzden en iyi durumda, ortalama durumda ve en kötü durumda O(N^2) zaman karmaşıklığına sahiptir.

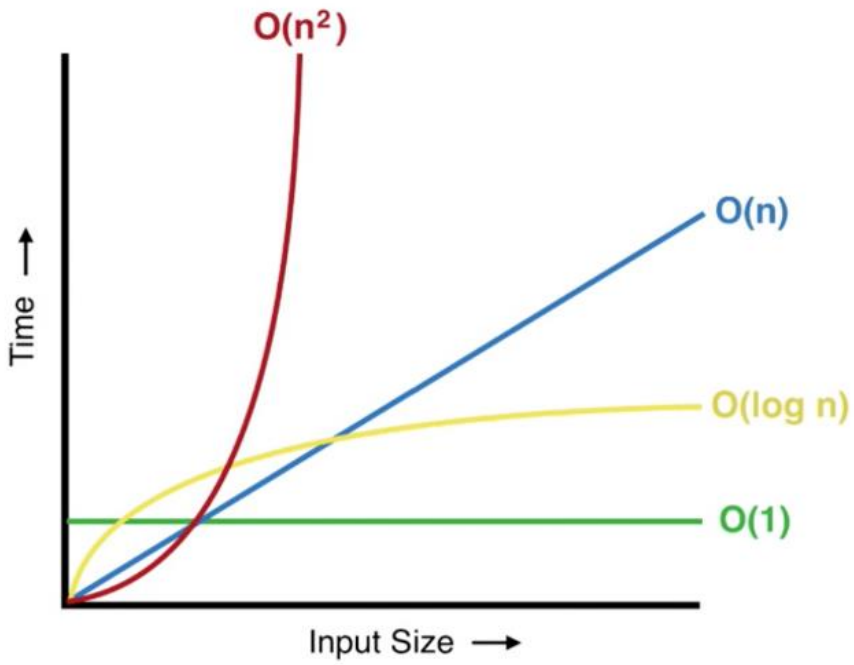
Selection sort algoritması kararlı bir algoritma değildir çünkü bu algoritma sıralama yaparken eşit elemanların yerlerini değiştirebilir.

3. SONUÇ VE DEĞERLENDİRME

Algoritma	Algoritma Türü	Ortalama Durum	En Kötü durum
Linear search	Arama	O(N/2)	O(N)
Binary search	Arama	$O(\log_2 N)$	$O(\log_2 N)$
Ternary search	Arama	$O(\log_3 N)$	$O(\log_3 N)$
Bubble sort	Sıralama	O(N^2)	O(N^2)
Insertion sort	Sıralama	O(N^2)	O(N^2)
Selection sort	Sıralama	O(N^2)	O(N^2)

Tablo 7.1: Algoritmalar ve Zaman Karmaşıklıkları

Tablo 7.2: Big O Notasyonu



[11]

Geçmişten günümüze kadar geliştirilen ve geliştirilmeye devam edilen bu algoritmaların temel gayesi sorunlarımıza en uygun çözümü bulmaktır. Birçok farklı sorun birçok farklı algoritmayla çözülebilir, bu yüzden her şey için en iyi algoritma hiçbir zaman var olmamıştır. Örneğin daha küçük veri setleri için O(logn) zaman karmaşıklığı yerine O(N) zaman karmaşıklığını tercih etmek çok daha uygun ve yerinde olacaktır.

KAYNAKLAR

[1] Parmar, V. P., & Kumbharana, C. K. (2015). Comparing linear search and binary search algorithms to search an element from a linear list implemented through static array, dynamic array and linked list. *International Journal of Computer Applications*, *121*(3).

[2] Erişim tarihi: [30 Nisan 2023]

https://www.tutorialspoint.com/data_structures_algorithms/linear_search_algorithm.htm

[3] Bajwa, M. S., Agarwal, A. P., & Manchanda, S. (2015, March). Ternary search algorithm: Improvement of binary search. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)* (pp. 1723-1725). IEEE

[4] Erişim tarihi: [30 Nisan 2023]

https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm

[5] Erişim tarihi: [30 Nisan 2023]

https://www.tutorialspoint.com/Ternary-Search

[6] Al-Kharabsheh, K. S., AlTurani, I. M., AlTurani, A. M. I., & Zanoon, N. I. (2013). Review on sorting algorithms a comparative study. *International Journal of Computer Science and Security (IJCSS)*, *7*(3), 120-126.

[7] Mishra, A. D., & Garg, D. (2008). Selection of best sorting algorithm. *International Journal of intelligent information Processing*, *2*(2), 363-368.

[8] Erişim tarihi: [30 Nisan 2023]

https://www.geeksforgeeks.org/bubble-sort/

[9] Erişim tarihi: [30 Nisan 2023]

https://www.geeksforgeeks.org/insertion-sort/

[10] Erişim tarihi: [30 Nisan 2023]

https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm

[11] Erişim tarihi: [30 Nisan 2023]

https://medium.com/dataseries/how-to-calculate-time-complexity-with-big-o-notation-9afe33aa4c46

EKLER

EK-1 Linear Search Algoritmasının C# kodu

```
using System;

class LinearSearch
{
    static void Main(string[] args)
    {
        int[] array = {9, 7, 2, 4, 3, 5, 6, 8, 1, 0};
        int element = 1;
        int index = linearSearch(array, element);

        if (index != -1)
        {
            Console.WriteLine("Element found at index: {0}", index);
        }
        else
        {
            Console.WriteLine("Element not found");
        }
    }

    private static int linearSearch(int[] array, int element)
    {
        for(int i = 0; i < array.Length; i++)
        {
            if (array[i] == element)
                return i;
        }
        return -1;
    }
}
```

EK-2 Binary Search Algoritmasının C# kodu

```
using System;

class BinarySearch
{
    static void Main(string[] args)
    {
        int[] array = new int[100000];

        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
        }
        int element = 2346;

        int index = binarySearch(array, element);
        if (index != -1)
        {
            Console.WriteLine("Element found at index: {0}", index);
        }
        else
        {
            Console.WriteLine("Element not found");
        }
    }

    private static int binarySearch(int[] array, int element)
    {
        int low = 0;
        int high = array.Length - 1;

        while (low <= high)
        {
            int middle = (low + high) / 2;
            int value = array[middle];

            Console.WriteLine("Middle: {0}", value);

            if (value < element)
            {
                low = middle + 1;
            }

            else if (value > element)
            {
                high = middle - 1;
            }
            else
            {
                return middle;
            }
        }
        return -1;
    }
}
```

EK-3 Ternary Search Algoritmasının C# kodu

```
using System;
class TernarySearch
{
    static void Main(string[] args)
    {
        int left, right, index, element;
        int[] array = new int[100000];
        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
        }
        left = 0;
        right = array.Length;
        element = 2346;
        index = ternarySearch(array, left, right, element);

        if (index != -1)
        {
            Console.WriteLine("Element found at index: {0}", index);
        }
        else
        {
            Console.WriteLine("Element not found");
        }
    }

    private static int ternarySearch(int[] array, int left, int right, int element)
    {
        if (left <= right)
        {
            int mid1 = left + (right - left) / 3;
            int mid2 = right - (right - left) / 3;

            if (array[mid1] == element){
                return mid1;
            }
            if (array[mid2] == element){
                return mid2;
            }
            if (element < array[mid1]){
                return ternarySearch(array, left, mid1 - 1, element);
            }
            else if (element > array[mid2]){
                return ternarySearch(array, mid2 + 1, right, element);
            }
            else{
                return ternarySearch(array, mid1 + 1, mid2 - 1, element);
            }
        }
        return -1;
    }
}
```

EK- 4 Bubble Sort Algoritmasının C# kodu

```
using System;
class BubbleSort
{
    static void Main(string[] args)
    {
        int[] array = { 9, 7, 2, 4, 3, 5, 6, 8, 1, 0 };
        bubbleSort(array);
        Console.WriteLine("Sorted array:");
        foreach (int element in array)
        {
            Console.Write(element + " ");
        }
    }
    private static void bubbleSort(int[] arr)
    {
        int size = arr.Length;
        for (int i = 0; i < size - 1; i++)
        {
            for (int j = 0; j < size - i - 1; j++)
            {
                if (arr[j] > arr[j + 1])
                {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
}
```

EK- 5 Insertion Sort Algoritmasının C# kodu

```
using System;
class InsertionSort
{
    static void Main(string[] args)
    {
        int[] array = { 9, 7, 2, 4, 3, 5, 6, 8, 1, 0 };
        insertionSort(array);
        Console.WriteLine("Sorted array:");
        foreach (int element in array)
        {
            Console.Write(element + " ");
        }
    }
    private static void insertionSort(int[] array)
    {
        for (int i = 0; i < array.Length; i++)
        {
            int temp = array[i];
            int j = i - 1;
            while (j >= 0 && array[j] > temp)
            {
                array[j + 1] = array[j];
                j--;
            }
            array[j + 1] = temp;
        }
    }
}
```

EK- 6 Selection Sort Algoritmasının C# kodu

```
using System;

class SelectionSort
{
    static void Main(string[] args)
    {
        int[] array = { 9, 7, 2, 4, 3, 5, 6, 8, 1, 0 };

        selectionSort(array);

        Console.WriteLine("Sorted array:");
        foreach (int element in array)
        {
            Console.Write(element + " ");
        }
    }

    private static void selectionSort(int[] array)
    {
        for (int i = 0; i < array.Length - 1; i++)
        {
            int min = i;
            for (int j = i + 1; j < array.Length; j++)
            {
                if (array[min] > array[j])
                {
                    min = j;
                }

            }
            int temp = array[i];
            array[i] = array[min];
            array[min] = temp;
        }
    }
}
```