Quick Sort – Kilitler ve Anahtarlar

Öğrenci Adı: Mert Güler

Öğrenci Numarası:

Dersin Eğitmeni:

Video Linki:

1- Problemin Çözümü:

Kullanıcıdan ilk başta veriyi içeren dosyanın ismi alınmıştır. Eğer dosya bulunduysa, dosyanın ilk satırından dizilerin boyutu alınıp anahtar ve kilit dizileri için dinamik olarak yer açılmıştır. İkinci ve üçüncü satırdan dizilerin elemanları okunmuştur.

Oluşturulan diziler Quick Sort algoritması ile Lomuto algoritması kullanılarak sıralanacaktır. Quick Sort algoritması, bir referans eleman seçilerek diğer elemanlar ile karşılaştırma sonrasında ana diziyi referanstan küçükler ve büyükler olarak iki parçaya ayıran ve bu işlemleri dizi sıralanana dek özyinelemeli şekilde tekrar eden bir algoritmadır. Her parçalamada referans değerin dizi içerisindeki konumunu döndürür.

Quick Sort fonksiyonun adımları şu şekildedir:

1 – Anahtar dizisinden rastgele bir eleman seç

2 – Bu elemanı referans değeri olarak kabul edip kilitler dizisini Lomuto algoritması kullanarak sırala ve parçala.

3 – Kilitler dizisinden referans indeksinde bulunan kilidi seç.

4 – Seçilen kilidi kullanarak anahtarlar dizisini sırala.

5 - Referans değerin solunda ve sağında kalan elemanlar için fonksiyonu tekrardan çağır.

Bu fonksiyon kullanılarak anahtarlar ve kilitler kendi içlerinde karşılaştırma yapılmadan karşılıklı ve özyinelemeli olarak sıralanmıştır. Elemanlar matematiksel olarak okunduğu ve işlendiği için sanki aynı değerler karşılaştırılıyor algısı oluşabilmektedir fakat bu kural çiğnenmemiştir.

Lomuto algoritmasında varsayılan olarak dizinin en büyük elemanı referans olarak seçildiği için algoritma konumu belli olmayan bir referans değere göre sıralama yapmayı sağlayacak şekilde düzenlenmiştir. Çözüm karşılaşılan sorunlar bölmesindedir.

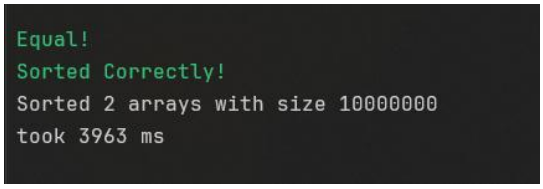
Verilerin tamamen doğru giriş yapıldığı (eksik, fazla olmayan eleman ve doğru dizi boyutu) kabul edilmiştir.

Pseudo Kod:

```
FUNCTION reflectionSort(keys, locks, low, high)
  IF (low >= high) THEN
    RETURN
  END IF

  key = selectRandom(keys, low, high)

  pivotIndex = divide(locks, low, high, key)

  lock = locks[pivotIndex]

  divide(keys, low, high, lock)

  reflectionSort(keys, locks, low, pivotIndex - 1)
  reflectionSort(keys, locks, pivotIndex + 1, high)
END FUNCTION
```

```
FUNCTION divide(array, low, high, pivot)
  i = low - 1

  FOR j = low TO high - 1 DO
    IF array[j] = pivot THEN
      swap(array[j], array[high])
    END IF

    IF array[j] < pivot THEN
      i = i + 1
      IF i != j THEN
        swap(array[i], array[j])
      END IF
    END IF
  END FOR

  IF i + 1 != high THEN
    swap(array[i + 1], array[high])
  END IF

  RETURN i + 1
END FUNCTION
```

Algoritmanın performansı 10 milyon boyutundaki diziler ile denenmiştir. Toplam 20 milyonluk veri seti için sıralama süresi orta düzey bir işlemcide ortalama 4 saniyedir. Bellek kullanımı 10 milyon veri başına 38 MB olarak ölçülmüştür.



QuickSort algoritmasına "if (i != j)" kontrolünü eklemenin performansa katkısı 0.7 saniye yani yaklaşık %17'dir.

Veri büyüklüğü iki katına çıkarıldığında ortalama O(N logN) büyüklüğünde büyüme gözlenmiştir.

Problemin çözüm adımları aşağıdaki görsellerden daha iyi anlaşılabilir:

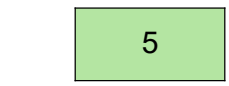
Step 1

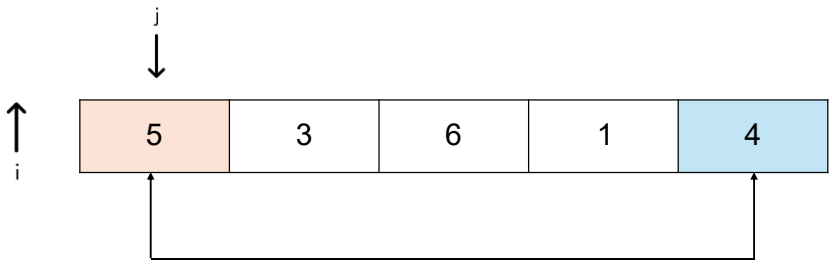
Initial pointers: i = -1, j = 0, low = 0, high = 4

Since, arr[0] == pivot

Swap arr[j] with arr[high]

Pivot





Step 1

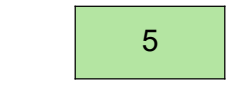
Pointers: i = -1, j = 0, low = 0, high = 4

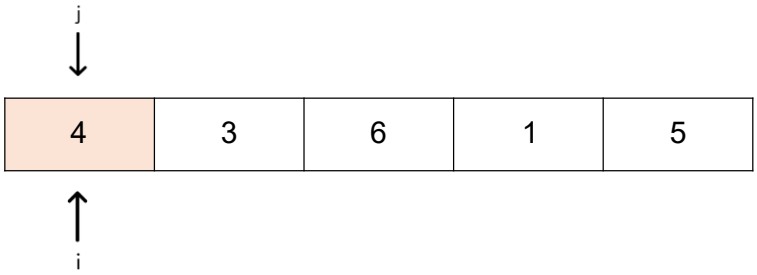
Since arr[0] < pivot

Increase i, i = 0

Since i == j, Skip swapping

Pivot





Step 2

Pointers: i = 0, j = 1, low = 0, high = 4

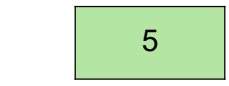
Since arr[1] < pivot

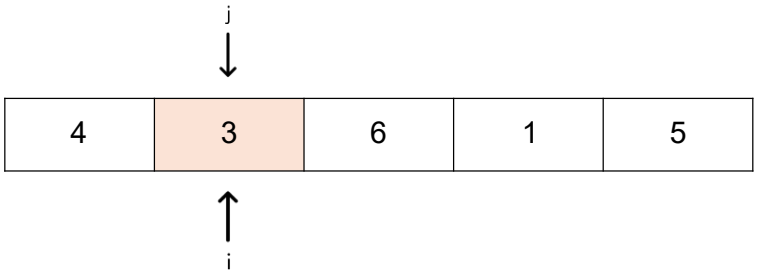
Increase i, i = 1

Since i == j, Skip swapping

Increase j

Pivot



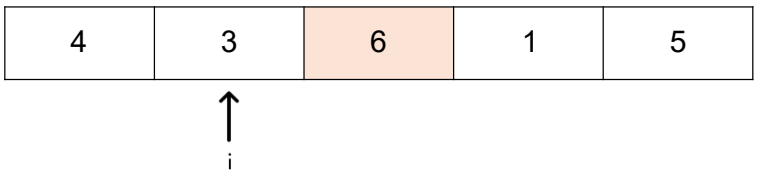


Step 3

Pointers: i = 1, j = 2, low = 0, high = 4

Since arr[2] > pivot
Skip swapping
Increase j

Pivot

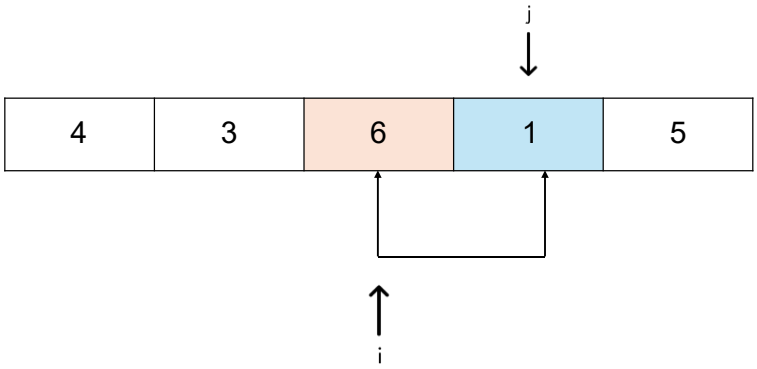


Step 4

Pointers: i = 1, j = 3, low = 0, high = 4

Since arr[3] < pivot
Increase i, i = 2
Since i != j,
Swap arr[2] and arr[3]
Increase j

Pivot

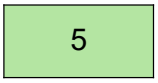


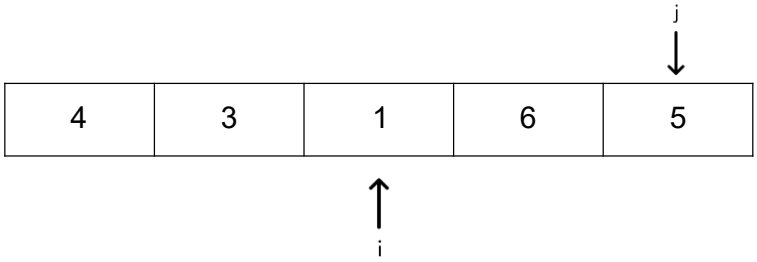
Step 5

Pointers: i = 2, j = 4, low = 0, high = 4

Since j < high is false,
Exit loop

Pivot



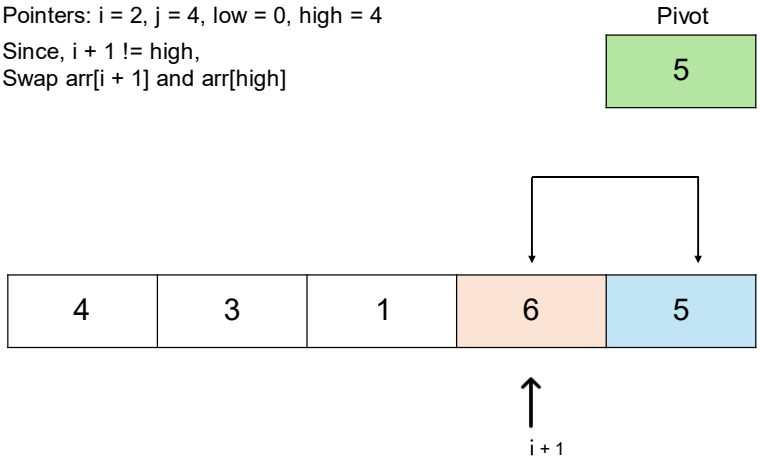


Step 5

Pointers: i = 2, j = 4, low = 0, high = 4

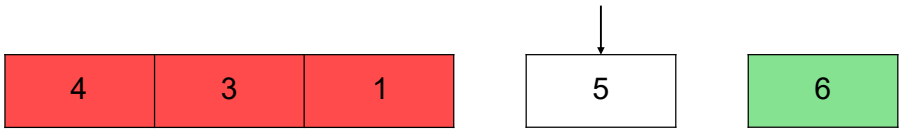
Since, i + 1 != high,

Swap arr[i + 1] and arr[high]



Continue

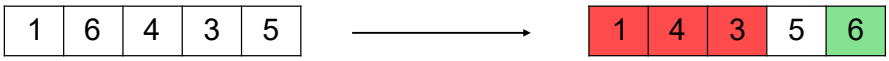
Now array is divided to two parts



1 - Return index of the pivot, which is 3

2 – Get array[3] as pivot, which is 5 again.

3 – Divide reflection arraywith pivot



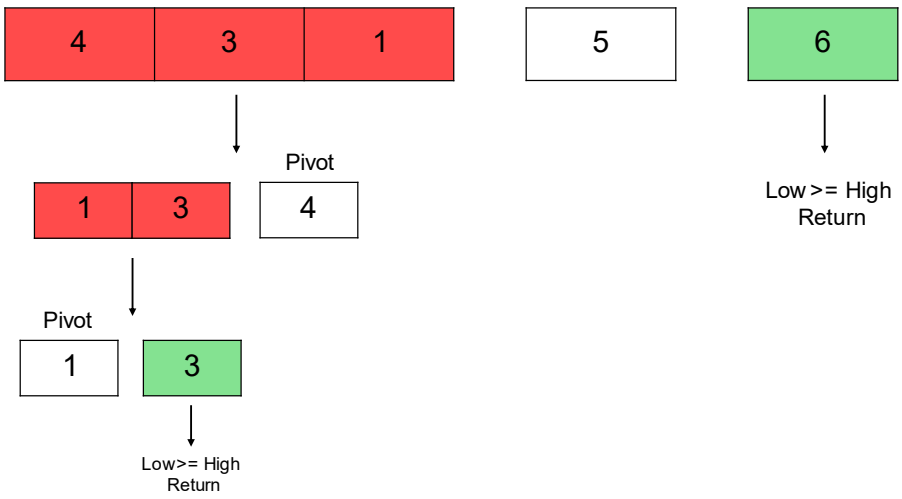
Do same steps with calling function again

For left side use range[low, pivotIndex– 1]

For right side use range[pivotIndex + 1, high]

Recursion

Steps of recursion



2- Karşılaşılan Sorunlar:

Anahtarlar ve kilitler kendi içlerinde birbirleri ile karşılaştırılamayacağı için Quick Sort algoritmasının düzenlenmesi gerekmekteydi.

Bu sorunu çözmek için ilk olarak aşağıdaki yöntem değerlendirmeye alındı:

1 – Rastgele seçilen elemanı kilitler dizisi içerisinde bul.

2 – Bulunan konumu ve dizinin en büyük elemanının konumunu değiştir.

3 – Quick Sort algoritmasını varsayılan hali ile çağır.

Bu çözüm her seferinde diziler içerisinde arama yapmayı gerektiriyordu. Bulma işlemini uygulamak fazladan O(N) maliyete sahipti.

Bu sorunu çözmek için Lomuto algoritmasını düzenlenmiştir. Bu düzenlemede eğer Lomuto algoritması içerisinde karşılaştırma yapılan eleman, referans değere eşit ise elemanı en yüksek indeksteki eleman ile yer değiştirmesi sağlandı.

Bu şekilde sorun herhangi bir performans kaybı olmadan çözülmüş oldu.

İkinci bir sorun ise çok nadir bir şekilde Lomuto algoritmasında geriye iki boyutlu dizi kaldığında ve seçilen referans dizinin en küçük ve ilk elamanı olduğunda gereksiz yer değiştirme işlemi yapılmasaydı, örneğin referans 10 için:

10 12 – İlk yer değiştirme: 10 & 12

12 10 – İkinci yer değiştirme: 12 & 10

10 12 – Son durum

Durumu oluşmakta. Bu sorunu çözmek Lomuto algoritması içerisinde bazı kontroller eklenebilir fakat durumun sadece son adımda karşılaşıldığı düşünüldüğünde maliyet faydadan yüksek olmaktadır. Bu sebeple durum göz ardı edildi.

3- Ekran Çıktıları:

```
Green : Higher than pivot
Red : Lower than pivot
Blue : Locks
Yellow : Keys
Orange : Swaps

Sorting Locks - Key: 3

12 3 8 10 1 4 - Swap Pivot with Highest: 3 & 4
12 4 8 10 1 3 - Swap: 12 & 1
1 4 8 10 12 3 - Pivot Swap: 4 & 3
1 3 8 10 12 4

Sorting Keys - Lock: 3

8 4 1 12 10 3 - Swap: 8 & 1
1 4 8 12 10 3 - Pivot Swap: 4 & 3
1 3 8 12 10 4
```

```
Sorting Locks - Key: 12

8 10 12 4 - Swap Pivot with Highest: 12 & 4
8 10 4 12

Sorting Keys - Lock: 12

8 12 10 4 - Swap Pivot with Highest: 12 & 4
8 4 10 12

Sorting Locks - Key: 10

8 10 4 - Swap Pivot with Highest: 10 & 4
8 4 10

Sorting Keys - Lock: 10

8 4 10
```

```
Sorting Locks - Key: 4

8 4 - Pivot Swap: 8 & 4
4 8

Sorting Keys - Lock: 4

8 4 - Pivot Swap: 8 & 4
4 8

Lock & Keys!

Locks:
1 3 4 8 10 12

Keys:
1 3 4 8 10 12

Equal!
Sorted Correctly!
```

Ekstrem Durumlar:

İki dizinin de sıralı olduğu durum:
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

Algoritma minimum yer değiştirme işlemi gerçekleştirmiştir.

```
Sorting Locks - Key: 4

1 2 3 4 5 6 7 8 - Swap Pivot with Highest: 4 & 8
1 2 3 8 5 6 7 4 - Pivot Swap: 8 & 4
1 2 3 4 5 6 7 8

Sorting Keys - Lock: 4

1 2 3 4 5 6 7 8 - Swap Pivot with Highest: 4 & 8
1 2 3 8 5 6 7 4 - Pivot Swap: 8 & 4
1 2 3 4 5 6 7 8

Sorting Locks - Key: 3

1 2 3

Sorting Keys - Lock: 3

1 2 3

Sorting Locks - Key: 2

1 2

Sorting Keys - Lock: 2

1 2

Sorting Locks - Key: 8

5 6 7 8
```

```
Sorting Keys - Lock: 8

5 6 7 8

Sorting Locks - Key: 7

5 6 7

Sorting Keys - Lock: 7

5 6 7

Sorting Locks - Key: 6

5 6

Sorting Keys - Lock: 6

5 6

Lock & Keys!

Locks:
1 2 3 4 5 6 7 8

Keys:
1 2 3 4 5 6 7 8

Equal!
Sorted Correctly!
```

İki dizinin birbirinin ters sıralanmış hali olduğu durum:

1 2 3 4 5 6
6 5 4 3 2 1

```
Sorting Locks - Key: 6

1 2 3 4 5 6

Sorting Keys - Lock: 6

6 5 4 3 2 1 - Swap Pivot with Highest: 6 & 1
1 5 4 3 2 6

Sorting Locks - Key: 5

1 2 3 4 5

Sorting Keys - Lock: 5

1 5 4 3 2 - Swap Pivot with Highest: 5 & 2
1 2 4 3 5

Sorting Locks - Key: 1

1 2 3 4 - Swap Pivot with Highest: 1 & 4
4 2 3 1 - Pivot Swap: 4 & 1
1 2 3 4

Sorting Keys - Lock: 1

1 2 4 3 - Swap Pivot with Highest: 1 & 3
3 2 4 1 - Pivot Swap: 3 & 1
1 2 4 3
```

```
Sorting Locks - Key: 2

2 3 4 - Swap Pivot with Highest: 2 & 4
4 3 2 - Pivot Swap: 4 & 2
2 3 4

Sorting Keys - Lock: 2

2 4 3 - Swap Pivot with Highest: 2 & 3
3 4 2 - Pivot Swap: 3 & 2
2 4 3

Sorting Locks - Key: 4

3 4

Sorting Keys - Lock: 4

4 3 - Swap Pivot with Highest: 4 & 3
3 4

Lock & Keys!

Locks:
1 2 3 4 5 6

Keys:
1 2 3 4 5 6

Equal!
Sorted Correctly!
```

Aynı boyuttaki kilit ve anahtardan iki adet bulunması durumu:

1 2 5 4 5 6
6 5 4 5 2 1

```
Sorting Locks - Key: 6

1 2 5 4 5 6

Sorting Keys - Lock: 6

6 5 4 5 2 1 - Swap Pivot with Highest: 6 & 1
1 5 4 5 2 6

Sorting Locks - Key: 4

1 2 5 4 5 - Swap Pivot with Highest: 4 & 5
1 2 5 5 4 - Pivot Swap: 5 & 4
1 2 4 5 5

Sorting Keys - Lock: 4

1 5 4 5 2 - Swap Pivot with Highest: 4 & 2
1 5 2 5 4 - Swap: 5 & 2
1 2 5 5 4 - Pivot Swap: 5 & 4
1 2 4 5 5

Sorting Locks - Key: 1

1 2 - Swap Pivot with Highest: 1 & 2
2 1 - Pivot Swap: 2 & 1
1 2

Sorting Keys - Lock: 1

1 2 - Swap Pivot with Highest: 1 & 2
2 1 - Pivot Swap: 2 & 1
1 2
```

```
Sorting Locks - Key: 5

5 5 - Swap Pivot with Highest: 5 & 5
5 5 - Pivot Swap: 5 & 5
5 5

Sorting Keys - Lock: 5

5 5 - Swap Pivot with Highest: 5 & 5
5 5 - Pivot Swap: 5 & 5
5 5

Lock & Keys!

Locks:
1 2 4 5 5 6

Keys:
1 2 4 5 5 6

Equal!
Sorted Correctly!
```