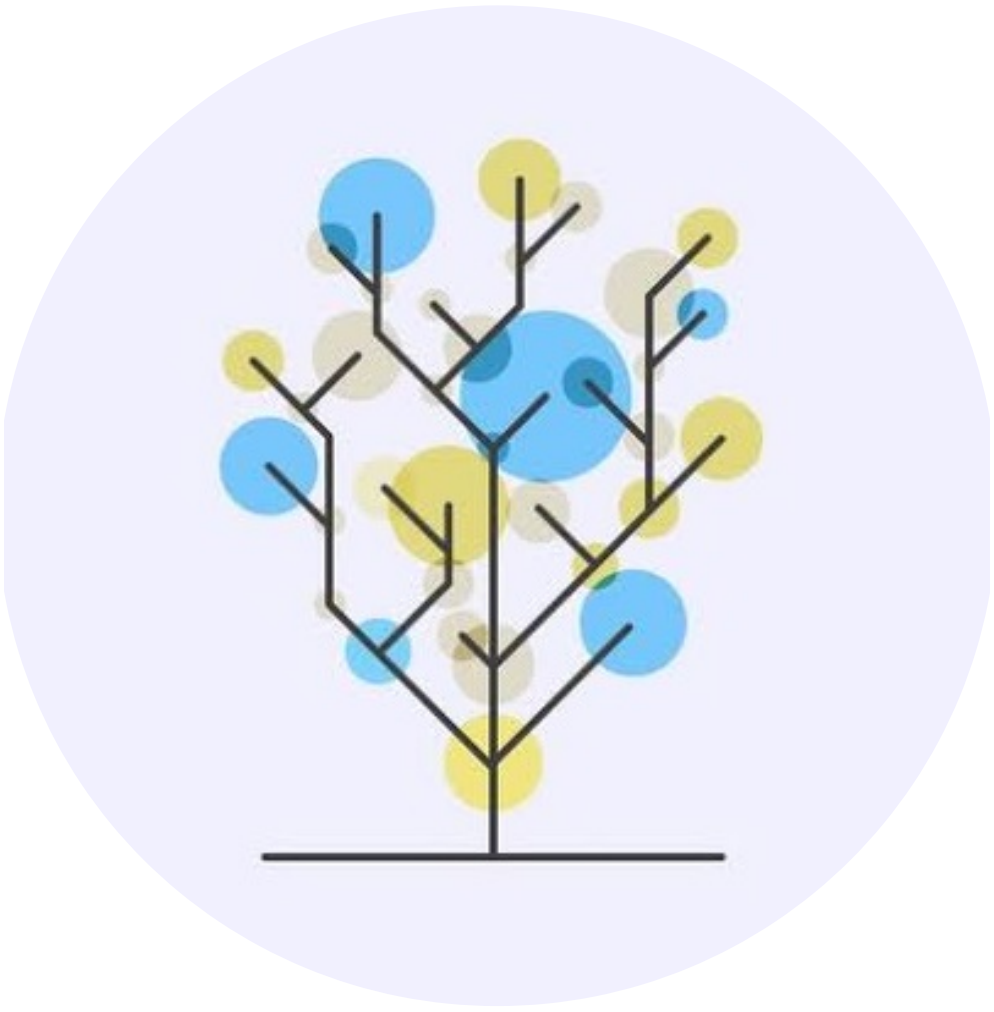


Sorting

06.01.2020

Data Structures & Algorithms

Intro

- Farklı sorting yöntemlerinin önemi de problemin boyutu büyüdükçe ortaya çıkar.
- Here is a funny youtube channel that might be helpful to understand different algorithms.
- BigO konusunda gördüğümüz gibi input büyüdükçe Time Complexity önem kazanır. Bu yüzden belki 10 elemanlık bir arrayi istediğimiz şekilde sort edebiliriz ama iş Facebook, Netflix, Youtube gibi büyük firmaların büyük uygulamalarına geldiği zaman hangi tip veri için hangi sorting metodunu kullandığımız çok büyük farklar yaratabiliriz.
- There are so many different sorting algorithms, and it is most likely that we won't need to write a sorting algorithm from scratch.

- Ayrıca built-in sort metodları her zaman istediğimiz gibi çalışmayabilir:
  - Örneğin Javascript içindeki sort metodu beklenmedik sonuçlar verebilir.
  - Bir array yarattık diyelim
    - `const letters = ['a', 'c'', 'k', 'f']`
    - `const numbers = [2, 65, 34, 2, 1, 7 ,8]`
    - `letters.sort()` beklendiği gibi çalışırken `numbers.sort()` hatalı çalışır çünkü built-in sort metodu önce sayıları alır, string'e çevirir ve charCodeAt(0) ile ilk index'in character code'una bakar ve ona göre sıralar.
  - Yani burada anlatılan şu ki built-in metodların da dökümantasyonuna bakarak nasıl çalıştığını anlamak önemli olabilir.

- Sorting algoritmalarının nasıl yazıldığını bilmek şart değil, ama nasıl çalıştığını anlamak yardımcı olabilir.

- BigOCheatSheet linkinden sort algoritmalarının complexity'lerine bakabilirsin.

- Ayrıca Bu Link'e göz atarak hangi sorting algoritmasının hangi durumlarda daha etkili veya verimsiz olduğunu görebilirsin.

Bubble Sort

- First of the Elementary Sorting Algorithms (**Bubble Sort**, **Insertion Sort**, **Selection Sort**)
- Bunun yanında **Merge Sort** ve **Quick Sort**'dan bahsedeceğiz bunlar biraz daha kompleks.
- Buble sort'un çalışma mantığı gayet basit elemanlara ikişer ikişer baştan başlayarak bakıyor, karşılaştırma yapıp iki elemanın yerini değiştiriyor veya olduğu gibi bırakıyor.
- Bu işlemi hiçbir elemanın yerini değiştirmeye gerek kalmayıncaya kadar tekrarlıyor.

- 6 5 3 1 8 7 2 4

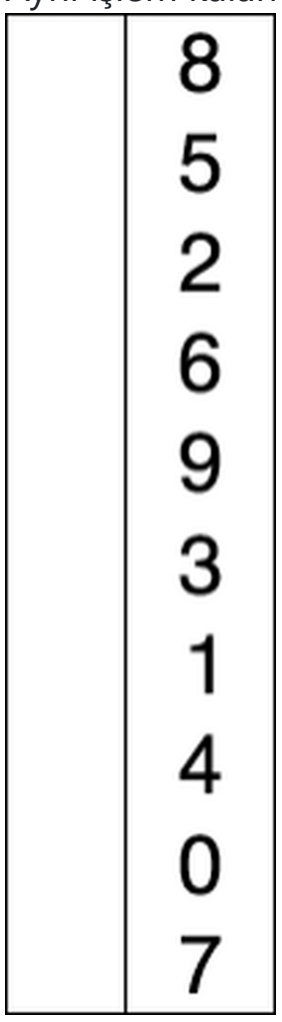
- Obviously it is not the most efficient sorting algorithm. Look at how many comparison it is doing.
- It is one of the simplest and also least efficient algorithms. Average ve Worst case için O(n^'2) time complexity'e sahip.
- Here is another helpful link about bubble sort.

Implement Bubble Sort

```
def bubble_sort(arr):
    # For every element (arranged backwards)
    for n in range(len(arr)-1,0,-1):
        #
        for k in range(n):
            # If we come to a point to switch
            if arr[k]>arr[k+1]:
                temp = arr[k]
                arr[k] = arr[k+1]
                arr[k+1] = temp

arr = [3,2,13,4,6,5,7,8,1,20]
bubble_sort(arr)
```

Selection Sort

- Yine basit bit sorting algorithm.

- Önce tüm listeye bakılır ve en küçük eleman bulunur, daha sonra bu eleman en başa taşınır.
- Aynı işlem kalan elemanlar içinde tekrar edilir.
- 

- Blue box: current element, red box: current minimum, yellow box: sorted element.
- Time complexity yine O(n^2).
- Here is another helpful link about selection sort.

Implement Selection Sort

```
def selection_sort(arr):

    # For every slot in array
    for fillslot in range(len(arr)-1,0,-1):
        positionOfMax=0

        # For every set of 0 to fillslot+1
        for location in range(1,fillslot+1):
            # Set maximum's location
            if arr[location]>arr[positionOfMax]:
                positionOfMax = location

        temp = arr[fillslot]
        arr[fillslot] = arr[positionOfMax]
        arr[positionOfMax] = temp

arr = [3,5,2,7,6,8,12,40,21]
selection_sort(arr)
```

Insertion Sort

- Üçüncü basit ve çok verimli olmayan algoritmamız.

- Ancak burada önemli bir nokta var, Insertion sort is useful when the list is already almost sorted. (or completely sorted)
- Instead of O(n^2) we can get **O(n) when the list is almost sorted.**

- 6 5 3 1 8 7 2 4

- İkinci elemana bakarız ilkinden küçük olduğu için soluna alırız, sıradaki elemana geçeriz ve sorted elemanlar arasında yerine yerleştiririz, bu şekilde son elemana kadar gideriz.
- This type of sorting performs really well when it comes to a small dataset.
- Here is another helpful link about insertion sort.

Implement Insertion Sort

```
def insertion_sort(arr):

    # For every index in array
    for i in range(1,len(arr)):

        # Set current values and position
        currentvalue = arr[i]
        position = i

        # Sorted Sublist
        while position>0 and arr[position-1]>currentvalue:

            arr[position]=arr[position-1]
            position = position-1

        arr[position]=currentvalue

arr =[3,5,4,6,8,1,2,12,41,25]
insertion_sort(arr)
```

Merge Sort and O(N logN)

- Şimdiye kadar ki algoritmalar O(n^2) idi, yalnızca insertion sort almost sorted datasets için O(n) olabiliyordu.
- **O(N logN)** Time Complexity elde etmek için daha önce bahsettiğimiz **Divide and Conquer** yöntemini kullanırlar.
- Divide and Conquer genelde O(N logN) avantajı sağlar. Bu yüzden iyi bir sort algoritmasıdır.
- Space complexity diğerlerinde O(1) iken burada O(n) olur.
- Ayrıca algoritma da daha komplekstir.
- Avantaj şuradan gelir, yine her elemana en az bir kez bakılıp karşılaştırılıyor bu yüzden O(n) var ama ikinci bir loopla her elemanı her elemanla karşılaştırmıyoruz.

- 6 5 3 1 8 7 2 4

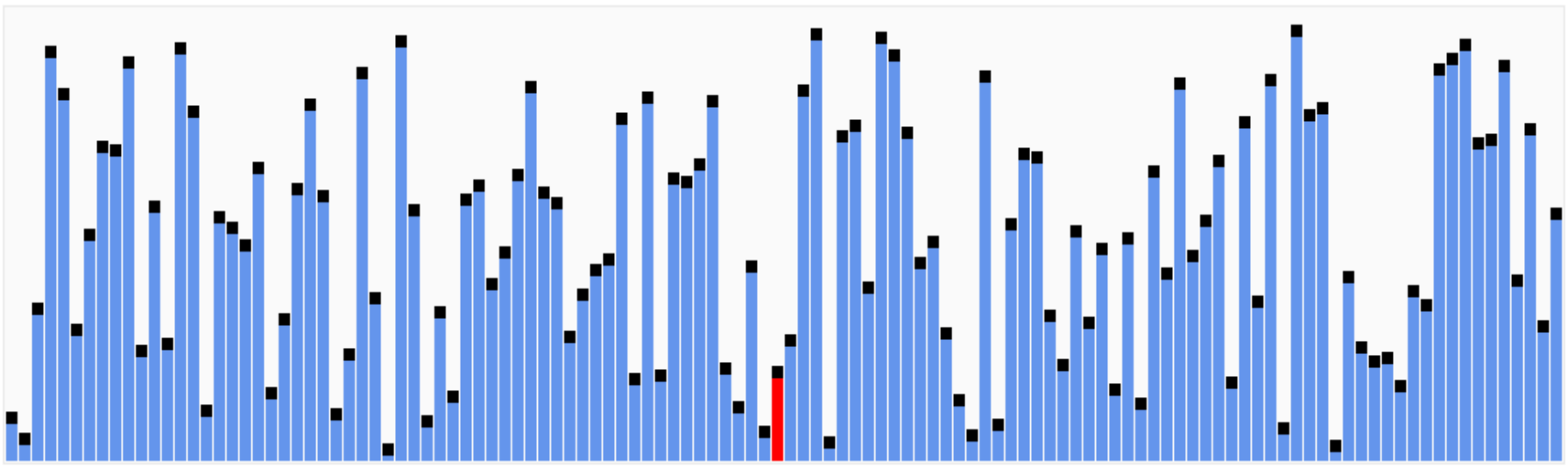
- Önce listeyi yarıya böleriz, daha sonra elde edilenleri de ikiye böleriz, bu şekilde tek tek n tane liste elde edene kadar devam ederiz.
- Daha sonra 1. ve 2. elemanı karşılaştırırz, hangisi küçükse onu başa alarak 2 elemanı birleştirim, daha sonra sıradaki ikisi için aynısını yaparım, ikişerli sıralı listeler elde ederim.
- Daha sonra 1. listenin ilk elemanı ile 2. listenin ilk elemanını karşılaştırım küçük olanı başa yazarım, sonra kalanlar arasında önce soldakiler karşılaştırılarak devam edilir ve 4'lü listeler elde edilir.
- Bu şekilde tersten büyük listeye geri dönülür.
- Ayrıca Merge Sort **stable** bir algoritmadır. Sanıyorum bu şu anlama geliyor, mesela listede üç adet 6 elemanı var yanyana bunlar sıralandığında ilk sırası bozulmuyor. Bu stabilitenin de yararlı olduğu örnekler olabilir.

Implement Merge Sort

```
def merge_sort(arr):

    if len(arr)>1:
        mid = len(arr)/2
        lefthalf = arr[:mid]
        righthalf = arr[mid:]

        merge_sort(lefthalf)
        merge_sort(righthalf)

        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                arr[k]=lefthalf[i]
                i=i+1
            else:
                arr[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            arr[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            arr[k]=righthalf[j]
            j=j+1
            k=k+1

arr = [11,2,5,4,7,6,8,1,23]
merge_sort(arr)
```

Quick Sort

- Just like Merge Sort it uses Divide & Conquer which allows O(N logN) Time Complexity and O(logN) Space Complexity which is better than O(n).
- It uses something called Pivoting Technique to break the main list into smaller lists and these small lists are using pivoting technique until they are sorted.
- It is pretty complex to understand
- Genelde Quick Sort en verimli çalışan olur ancak pivot point seçimi önemli, eğer yanlış bir pivot point seçilirse worst case scenario'su O(n^2) olur ki bu MergeSort'un worst case'inden kötü.
-



- Önce rastgele bir pivot item seçilir, herhangi bir eleman.
- Daha sonra bu seçilen pivottan küçük olanları sola yazacağım, büyük olanları pivot'ın sağına.
- Bunu yapmak için 1. elemanla pivot'ı karşılaştırırım küçükse kalsın büyükse sağa taşı.
- İkinci elemana bak, büyük diyelim pivot'ın sağına.
- Sonuç olarak pivot'ın solu ve sağı doldurulunca artık pivot olması gerektiği yerdedir.
- Şimdi divide and conquer uygulamak için liste pivot'tan bölünür ve yeni pivotlar seçilir her biri için işlem tekrarlanır.

When to Use Which Sorting Algorithm?

- **Insertion Sort:** If our input is small or items are mostly sorted it is really fast and uses little space and easy to implement.
- **Bubble Sort, Selection Sort:** Almost never used, only for educational purposes.
- **Merge Sort:** Fast, it is always O(N logN) for best, average and worst cases. Problematic space complexity.
- **Quick Sort:** Except for the worst case, it is the best both for time and space complexity.
- Big O Cheat Sheet'e bakılırsa daha bir çok sorting algoritması var hatta bunlardan daha iyi ancak sadece özel veri tipleri ile veya özel durumlarda kullanılabiliyorlar yani yine en iyisi Quick Sort olarak düşünülebilir.

More blogs