Sorting Algorithms

A **sorting algorithm** is an algorithm that rearranges elements in order. The order can be numerical or lexicographical, which can then either be ascending or descending.

Sorting is important because it allows for the efficiency of other algorithms that require data to be sorted such as a binary search. In addition, sorting is also important because it displays data in a meaningful way to humans.

There are many different sorting algorithms to help you sort data, each with strengths and weaknesses. Some of the most common sorting algorithms are: Selection sort, Bubble sort, Insertion sort, Merge sort, Quick sort, Heap sort, Counting sort, Radix sort and Bucket sort

Below we will focus on Insertion Sort, Merge Sort, Quick Sort, Bucket Sort.

However, most sorting algorithms are not efficient and should not be used in interviews. It is unlikely that you will implement any of the sorting algorithms from scratch. Instead you would need to sort the input using your language's default sorting function so that you can use binary searches on them. Python’s built-in sort function is “sorted()” and it is an implementation of TimSort

Nonetheless, we will discuss

Insertion Sort

Merge Sort

Quick Sort

Bucket Sort

HeapSort

Tim Sort

Pythons Built in Sort

# Sorting Algorithms using Python’s built-in sort functions: sorted() and sort()

Python has two built-in sort functions: sorted() and sort().

The time complexity for both functions are **O(nlog(n)))** in both worst and average case.

sorted() and sort() use Timsort, a sorting algorithm that combines merge sort and insertion sort. Timsort’s mix of insertion and merge sort makes it an efficient algorithm and is the reason you usually use Python’s built-in sorting algorithms instead of trying to code your own.

## **sorted()**

The sorted() function works on any iterable like a string, list, set, tuple and dictionary.

The sorted() function creates a new sequence type containing a sorted version of the given sequence.

The code below is the syntax for sorted()

sorted(iterable, key, reverse)

The sorted function takes in three parameters: Iterable, key and reverse.

1. **Iterable**: This is not an optional parameter. This is the iterable that needs to be sorted
2. **Key**: This is an optional parameter. This works as a basis for comparison while sorting.
3. **Reverse**: This is an optional parameter. if it = True, then it sorts the variable in descending order. Else it sorts in ascending order. If you do not include this parameter, then it will automatically be ascending order as it is the default order.

Examples

list = [43, 1, 5, 8, 2, 7]

sorted\_list = sorted(list) #Sorted() with only an iterable

print(sorted\_list)

>>> [1, 2, 5, 7, 8, 43]

sorted\_list = sorted(list, reverse=True) #Sorted with reverse

print(sorted\_list)

>>>[43, 8, 7, 5, 2, 1]

list = ['bob', 'albert', 'Cody', 'DONALD'] #Characters

sorted\_list = sorted(list)

print(sorted\_list)

>>> ['Cody', 'DONALD', 'albert', 'bob']

#Order like this because Capital letters come before lowercase

string = "davonte" #sorting a string

sorted\_string = sorted(string)

print(sorted\_string)

>>> ['a', 'd', 'e', 'n', 'o', 't', 'v']

a\_list = ["onehundred", "five", "seventy", "two"]

print(sorted(a\_list, key=len)) #Sorting using a key

>>> ['two', 'five', 'seventy', 'onehundred']

#Sorts by the order of whatever has the shortest length

## sort()

In comparison, The sort() function works only on lists & returns nothing and changes the original sequence