CS 131 Semester Project (Professor Yuen)

By Anikait Roy

Sorting - What? Why? When? Where?

What exactly is Sorting?

Sorting is among the most useful algorithms that is primarily taught in Data/Discrete Structures and Algorithms. Sorting is used for arranging data in a particular format or logically arranging data, i.e. either in ascending, descending, or any custom order.

Why Sorting Algorithms?

Sorting is important for optimizing the efficiency of a program or other algorithms which require input data to be in sorted lists. It is easier and faster to locate items or elements in a sorted list than an unsorted one.

When is Sorting used?

Sorting is used whenever an array or list needs to be rearranged or reordered in a program.

Where is Sorting used?

The majority, if not all, softwares and web applications use Sorting. Also, sorting is among the most commonly asked questions during technical interviews.

Different types of Sorting Algorithms

Popular Sorting Algorithms

Simple Sorts:

Insertion Sort

Selection Sort

Efficient Sorts:

Merge Sort

Heap Sort

Quick Sort

... and MANY MORE!

Analysis of different Sorting techniques

Comparison Sort

A comparison sort algorithm sorts items by comparing values between each other. It can be applied to any sorting cases. The best complexity is O(n*log(n)) which can be proved mathematically.

Ex: Quick Sort, Merge Sort, Insertion Sort, Selection Sort, Bubble Sort.

Non-Comparison Sort

A non-comparison sort algorithm uses the internal character of the values to be sorted and sort the data without comparing the elements. The best complexity is case dependent, such as O(n). **All sorting problems that can be sorted with a non-comparison sort algorithm can be sorted with a comparison sort algorithm, but not vice versa.**

Ex: Bucket Sort, Counting Sort, Radix Sort.

Time Complexity vs Space Complexity Analysis

Time Complexity

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Time complexity lets us know how the performance of the algorithm changes as the size of the data set increases.

Space Complexity

Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input. Space complexity describes how much space an algorithm needs to run.

Note: The main aspect time and space complexity depend on is the execution time of an algorithm.

Time and Space Complexity Continued

Order of growth is how the time of execution depends on the length of the input. Order of growth helps us to compute the running time with ease. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input. We use different notation to describe limiting behavior of a function.

O-notation: To denote asymptotic upper bound, we use O-notation. For a given function g(n), we denote by O(g(n)) the set of functions:

$O(g(n)) = \{f(n)$: $\exists$ +ve const. c and n_0, such that $0 \le f(n) \le c * g(n)\ \forall\ n \ge n_0\}$

Ω-notation: To denote asymptotic lower bound, we use Ω-notation. For a given function g(n), we denote by Ω(g(n)) the set of functions:

$\Omega(g(n)) = \{f(n)$: $\exists$ +ve const. c and n_0, such that $0 \le c * g(n) \le f(n)\ \forall\ n \ge n_0\}$

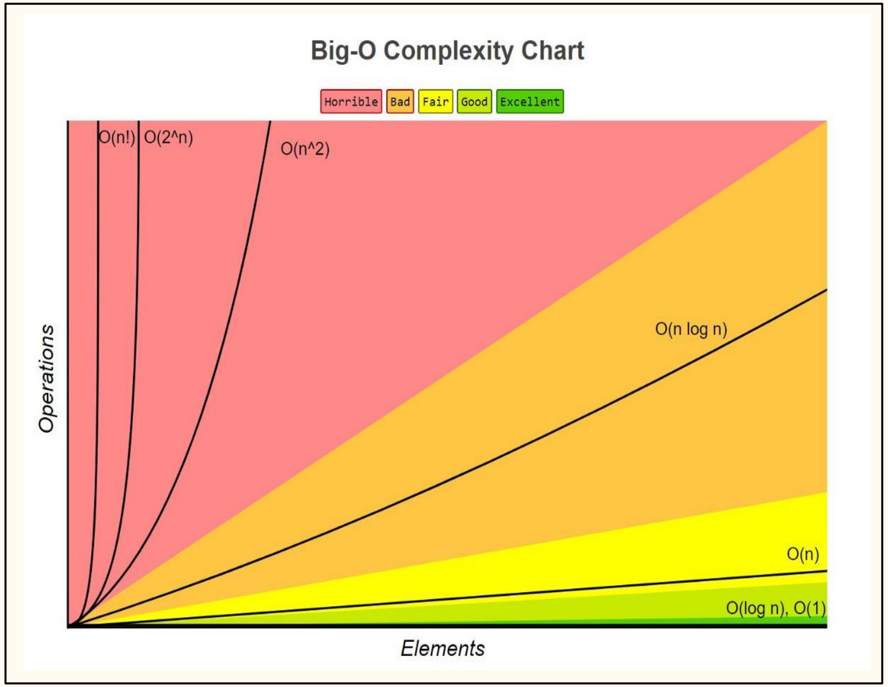
Θ-notation: To denote asymptotic tight bound, we use Θ-notation. For a given function g(n), we denote by Θ(g(n)) the set of functions:

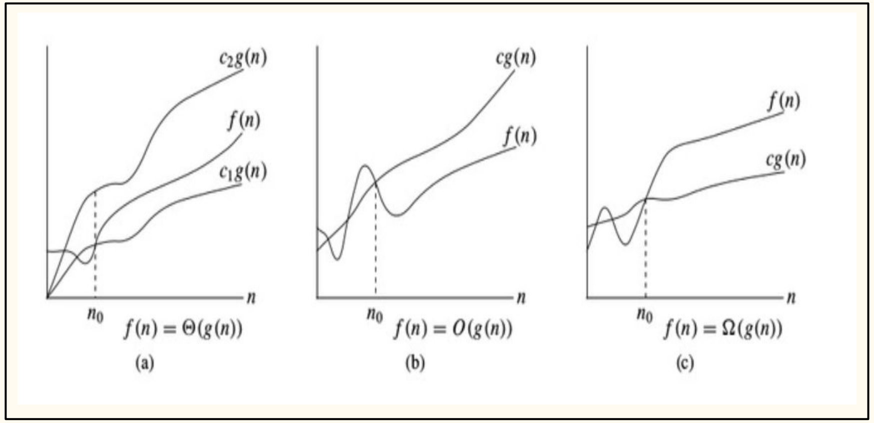
$\Theta(g(n)) = \{f(n)$: $\exists$ +ve const. $c1$, $c2$, n_0, such that $0 \le c1 * g(n) \le f(n) \le c2 * g(n)\ \forall\ n \ge n_0\}$

Time Complexity Analysis

Algorithm	Best	Average	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(nlog(n))	θ(nlog(n))	O(nlog(n))
Quick Sort	Ω(nlog(n))	θ(nlog(n))	O(n^2)
Merge Sort	Ω(nlog(n))	θ(nlog(n))	O(nlog(n))
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	θ(nk)	O(nk)
Count Sort	Ω(n+k)	θ(n+k)	O(n+k)

Complexity Chart





Some other Classifications and Performance Criteria of Sorting Algorithms

Memory Usage

The amount of extra memory required by a sorting algorithm is also an important consideration. *In place* sorting algorithms are the most memory efficient, since they require practically no additional memory. In particular, some sorting algorithms are "**in place**". This means that they need only $O(1)$ or $O(\log n)$ memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporarily stored, as in other sorting algorithms.

Recursion

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Merge Sort is a **recursive** function which follows the divide-and-conquer algorithm that divides an array of length n into n subarrays, and then recombines them using merge.

Stable and Unstable Sorting Algorithms

An algorithm is **stable** when, upon finding two elements that are equal for sorting, the algorithm preserves the original order of the items. An algorithm is **unstable** when there is no guarantee that the equal elements will end up in the original order.

Bubble Sort Example

1st Pass:

5	**2**	3	1	4
2	**5**	**3**	1	4
2	3	**5**	**1**	4
2	3	1	**5**	**4**
2	3	1	4	5

2nd Pass:

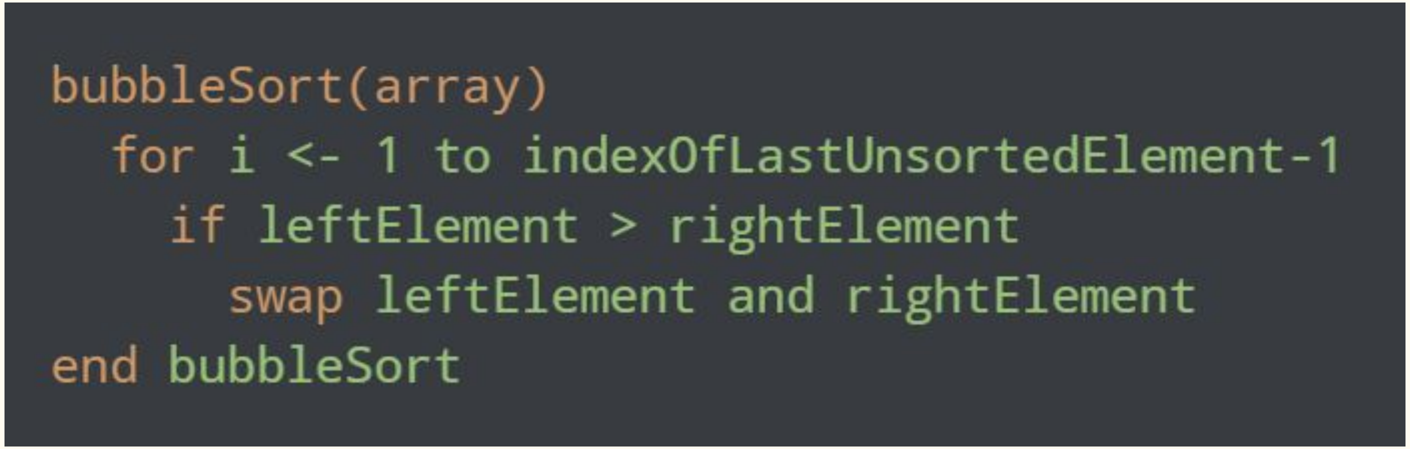
2	**3**	**1**	4	5
2	1	3	4	5

3rd Pass:

2	**1**	3	4	5
1	2	3	4	5

Sorting Algorithm Example

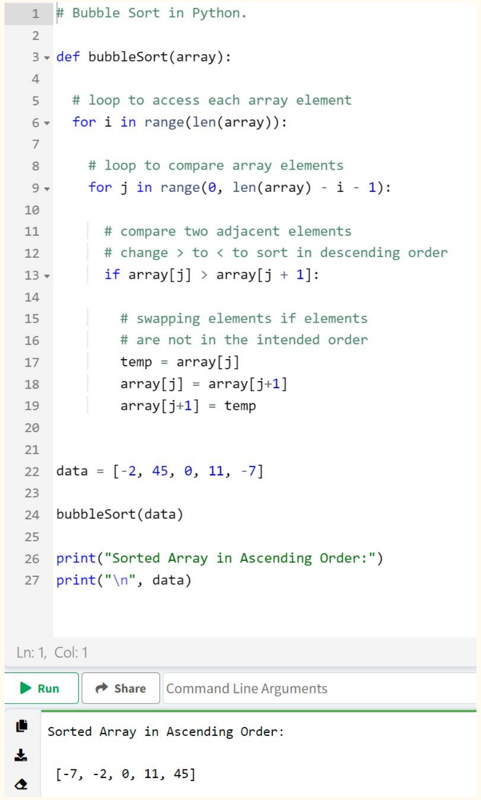
This is a Bubble Sort algorithm example.



```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

Sorting Algorithm Code

This is an example for a Bubble Sort code in Python Programming Language.



```python
# Bubble Sort in Python.

def bubbleSort(array):

  # loop to access each array element
  for i in range(len(array)):

    # loop to compare array elements
    for j in range(0, len(array) - i - 1):

      # compare two adjacent elements
      # change > to < to sort in descending order
      if array[j] > array[j + 1]:

        # swapping elements if elements
        # are not in the intended order
        temp = array[j]
        array[j] = array[j+1]
        array[j+1] = temp


data = [-2, 45, 0, 11, -7]

bubbleSort(data)

print("Sorted Array in Ascending Order:")
print("\n", data)
```

```
Sorted Array in Ascending Order:

 [-7, -2, 0, 11, 45]
```

Thank You!

I hope you understood the important concepts of sorting algorithms :)