Data Structure & Algorithms

Introduction

1.1 Some Definitions

- Algorithm
- Data Structure
- Time Complexity

Algorithm

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

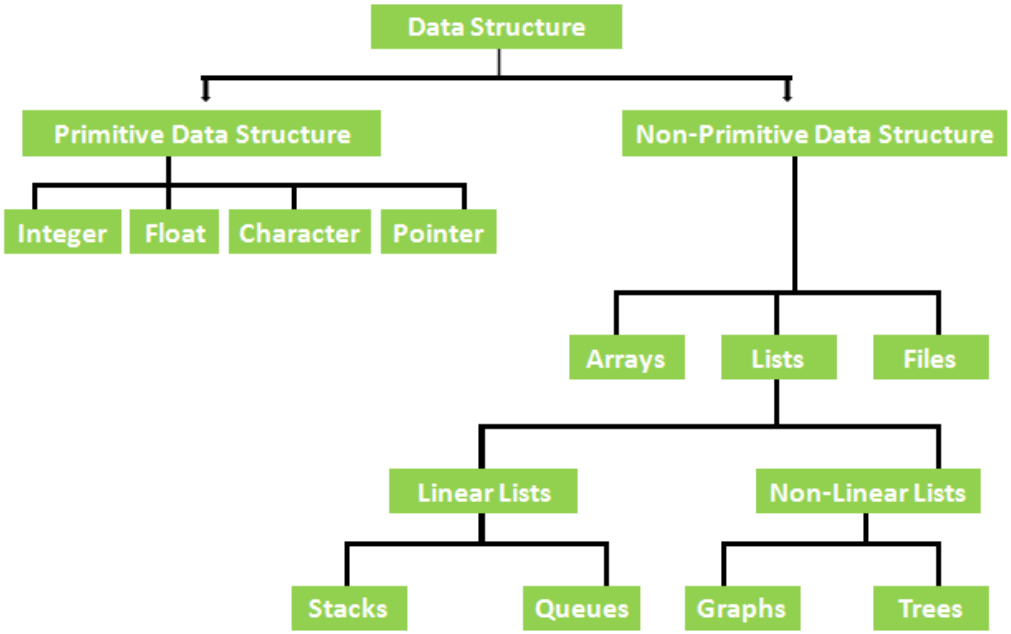
Algorithm

- From the data structure point of view (more on that later), the following are some important categories of algorithms −
  - **Search** − Algorithm to search an item in a data structure.
  - **Sort** − Algorithm to sort items in a certain order.
  - **Insert** − Algorithm to insert item in a data structure.
  - **Update** − Algorithm to update an existing item in a data structure.
  - **Delete** − Algorithm to delete an existing item from a data structure.

Data Structure

- a **data structure** is a data organization, management, and storage format that enables efficient access and modification. More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data, i.e., it is an algebraic structure about data.
- Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.
- Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Types of Data Structure



1.2 How To Write An Algorithm

- Algorithms are never written to support a particular programming code.
- basic code constructs like loops (do, for, while), flow-control (if-else), etc can be used.
- We write algorithms in a step-by-step manner.

Example

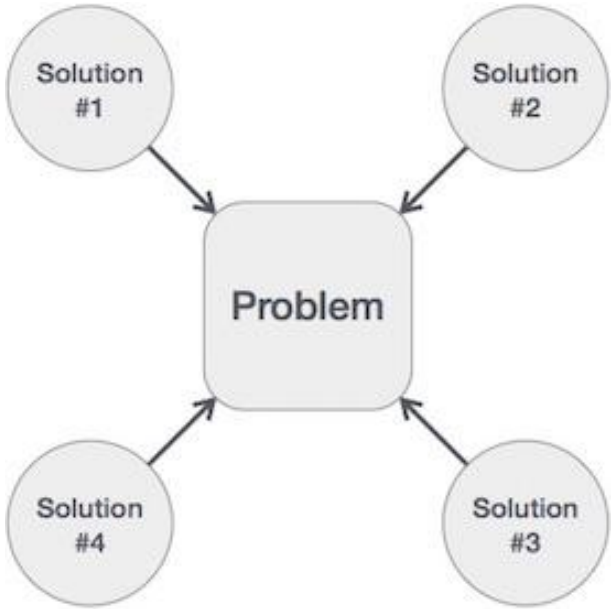
Algorithms tell the programmers how to code the program.

Problem − Design an algorithm to add two numbers and display the result.

```
Step 1 – START ADD
Step 2 – get values of a & b
Step 3 – c ← a + b
Step 4 – display c
Step 5 – STOP
```

- Writing step numbers is optional.
- We design an algorithm to get a solution for a given problem. A problem can be solved in more than one way.

- Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.



Time Complexity

- the **time complexity** is the computational complexity that describes the amount of computer time it takes to run an algorithm.
- Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform.

Step count method

- The step count method is one of the method to analyze the algorithm.
- In this method, we count number of times one instruction is executing. From that we will try to find the complexity of the algorithm.

Example

- Suppose we have one algorithm to perform sequential search. Suppose each instruction will take c1, c2, …. amount of time to execute, then we will try to find out the time complexity of this algorithm

Algorithm	Number of times	Cost

seqSearch(arr, n, key)
i <- 1
while i <= n do
  if arr[i++] == key, then
  break
 	
return i
	
	1
	n+1
	n
	0/1
	
	1
		
		c1
		c2
		c3
		c4
		
		c5

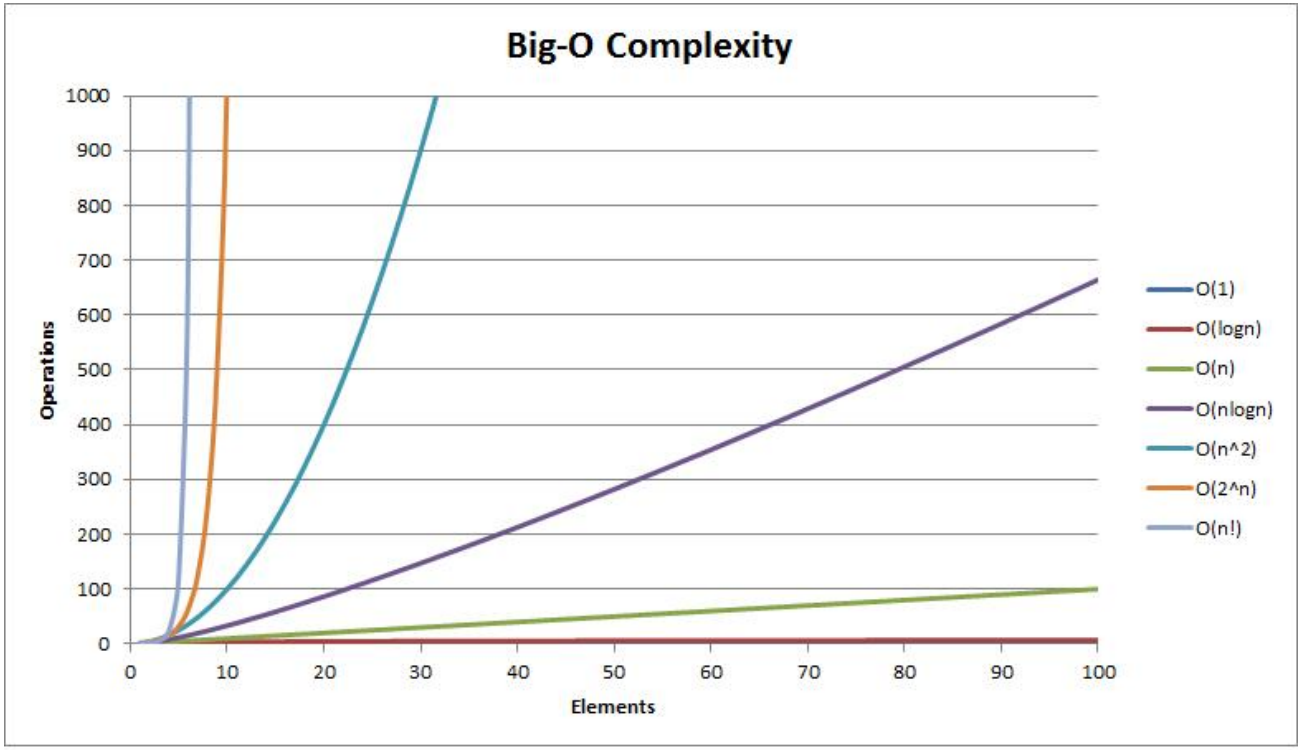
Example (cont.)

Now if we add the cost by multiplying the number of times it is executed, (considering the worst case situation), we will get

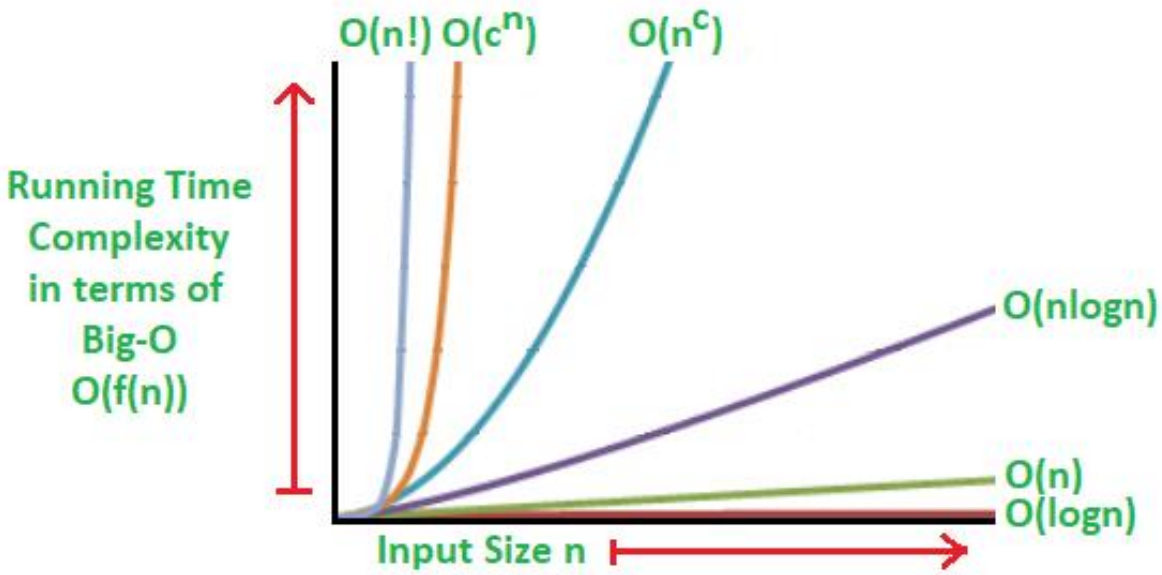
$$Cost = c1 + (n+1)c\ 2 + nc3 + c4 + c\ 5$$
$$Cost = c1 + nc\ 2 + c2 + nc3 + c4 + c5$$
$$Cost = n(c\ 2 + c3) + c1 + c4 + c5$$
$$Cost = n(c\ 2 + c3) + C$$

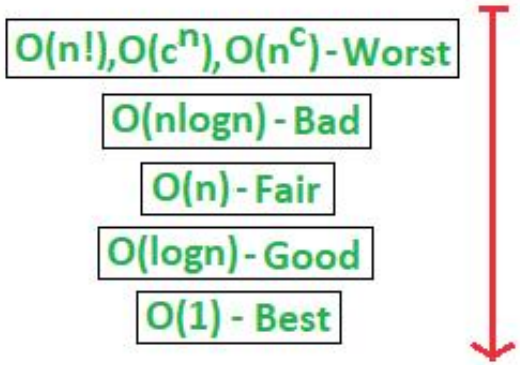
Considering the $c1\ +\ c4\ +\ c5$ is C, so the final equation is like straight line $y\ =\ mx\ +\ b$. So we can say that the function is linear. The complexity will be $O(n)$. (more on that later)

Example


Big-O Complexity
Operations
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Big-O Complexity





1.3 Sorting Algorithm

- A sorting algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of the element in the respective data structure.

- Input: A sequence of n numbers (A_1, A_2, … , A_n)
- Output: A premutation (recording) of the input sequence (A'_1, A'_2, … , A'_n) such that $A'_1 \leq A'_2 \leq \ldots \leq A'_n$

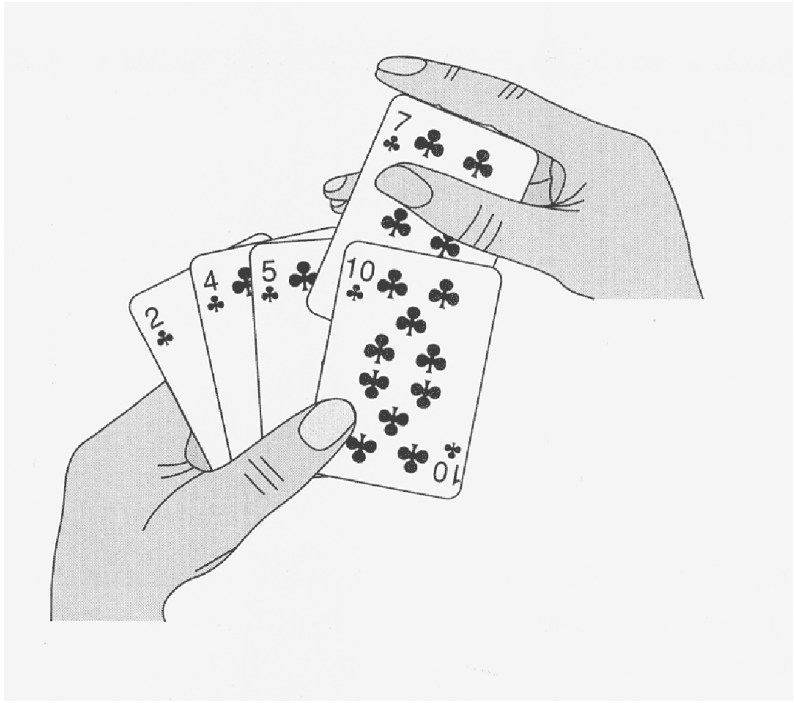
Example

8 5 3 1 4 7 9

Sorting Algorithm (cont.)

- The **sorting problem** → a **problem**
- **Insertion sort** → an algorithm (**solution**)
- **Bubble sort** → an algorithm (**solution**)
- …

Insertion sort



Insertion sort

- A good algorithm for sorting small sets of numbers
- Exactly the same way you arrange a hand of cards
  1. First, all the cards are on the back of the table and your left hand is empty
  2. Take a card off the table and place it in the appropriate sorted place in your left hand
  3. To find the right place for the next card, compare it with all the cards in the left hand one by one (from right to left)
  4. All the cards in your left hand are sorted at any time

Algorithm

For each element $A[i]$, if $A[i] > A[i+1]$, swap the elements until $A[i] < A[i+1]$.

Pseudocode

```
Insertion_Sort(A)
1 for j <- 2 to length[A]
2     do  key <- A[j]
3         Insert A[j] into the stored sequence A[1 ... j - 1]
4         i <- j - 1
5         while i > 0 and A[i] > key
6             do  A[i+1] <- A[i]
7                 i < i - 1
8         A[i+1] <- key
```

Complexity

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 ▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

t_j is the number of times the while loop test in line 5 is executed for that value of j

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2..n} t_j + c_6 \sum_{j=2..n} (t_j - 1) + c_7 \sum_{j=2..n} (t_j - 1) + c_8(n-1)$$

Complexity (cont.)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2..n} t_j + c_6 \sum_{j=2..n} (t_j - 1) + c_7 \sum_{j=2..n} (t_j - 1) + c_8(n-1)$$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

Complexity (cont.)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2..n} t_j + c_6 \sum_{j=2..n} (t_j - 1) + c_7 \sum_{j=2..n} (t_j - 1) + c_8(n-1)$$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

$$\boldsymbol{T(n) = An + B \sum_{j=2}^{n} t_j + C}$$

Question

- What are the *best-case* and *worst-case* running times of INSERTION-SORT?
- How about *average-case*?

Best case

The array is already sorted

$$Array = [2, 5, 7, 9, 11, 13]$$

Step 1	2 5 7 9 11 13	
Step 2	2 5 | 7 9 11 13	j = 2
Step 3	2 5 7 | 9 11 13	j = 3
Step 4	2 5 7 9 | 11 13	j = 4
Step 5	2 5 7 9 11 | 13	j = 5
Step 6	2 5 7 9 11 13 |	j = 6
Step 7	2 5 7 9 11 13	

$A[i] > key$ is never satisfied

Best case (cont.)

If the input array is already sorted, insertion sort compares n elements and performs no swaps. Therefore, in the *best-case*, insertion sort runs in ***n*** time. Why?

Worst case

The array is sorted in reverse order:

$$Array = [13, 11, 9, 7, 5, 2]$$

Step 1	13 11 9 7 5 2	
Step 2	13 11 | 9 7 5 2	j = 2
Step 3	11 13 9 | 7 5 2	j = 3
Step 4	9 11 13 7 | 5 2	j = 4
Step 5	7 9 11 13 5 | 2	j = 5
Step 6	5 7 9 11 13 2 |	j = 6
Step 7	2 5 7 9 11 13	

$A[i] > key$ is always satisfied

Worst case (cont.)

The *worst-case* for insertion sort occurs when the input list is in decreasing order. To insert the last element, we need *n-1* comparisons and *n-1* swaps. To insert the second to last element, we need at most *n-2* comparisons and at most *n-2* swaps, and so on.

Therefore, in the *worst-case*, insertion sort runs in $\boldsymbol{n^2}$ time. How come?

Average-case

When analyzing algorithms, the *average-case* often has the same complexity as the worst case. So insertion sort, on average, takes $\boldsymbol{n^2}$ time.

Worst-case and average-case analysis

Usually, we use the *worst-case* running-time to analyze an algorithm, which is actually the longest runtime for each input of size n.

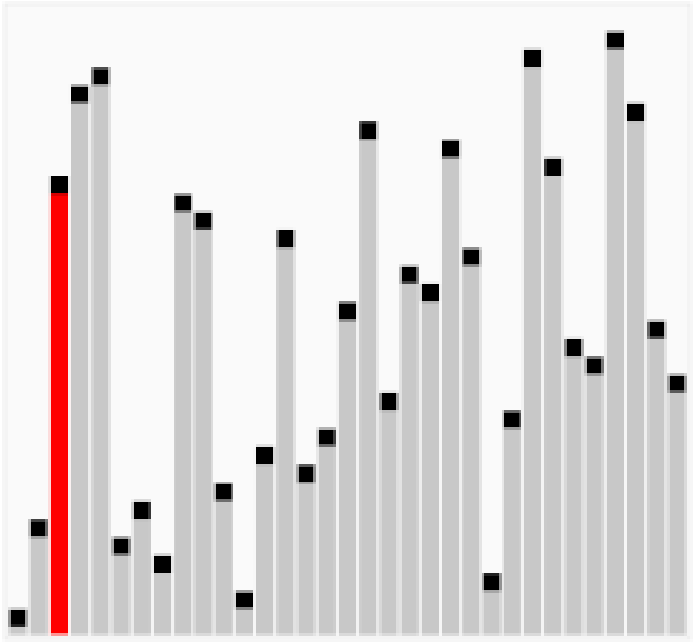
Why do we do this?

1. *Worst-case* gives an upper-bound guarantee of run-time
2. For most algorithms, the worst case occurs more often during execution (e.g., In searching, It often happens that the target element is missing)

Why we don't use *average-case*?

1. False estimate
2. To compute a useful average-case complexity you need to have a probability distribution over your inputs. Often you simply don't know what that is, so you compute it from a uniform (or some other appropriate maximum-entropy) distribution. That's still only going to be an approximation.

Bubble sort



Bubble sort

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

This algorithm is not suitable for large data sets as its average and worst case complexity are of n^2 where ***n*** is the number of items.

Algorithm

The **bubble sort algorithm** is as follows:

1. Compare $A[0]$ and $A[1]$. If $A[0]$ is bigger than $A[1]$ swap the elements.
2. Move to the next element, $A[1]$ (which might now contain the result of a swap from the previous step), and compare it with $A[2]$. If $A[1]$ is bigger than $A[2]$, swap the elements. Do this for every pair of elements until the end of the list.
3. Do steps 1 and 2 for n times.

Pseudocode

```
Bubble-sort(A)
1  for i <- n - 1 down to 1 do
2       for j <- 1 to i do
3                if A[j]>A[j+1]
4                         swap(A[j],A[j+1]);
5                end if
```

Complexity

To calculate the complexity of the bubble sort algorithm, it is useful to determine how many comparisons each loop performs. For each element in the array, bubble sort does $n - 1$ comparisons. Because the array contains *n* elements, it does $n(n - 1)$ comparisons. In other words, bubble sort performs n operations on n number of elements, leading to a total running time of n^2.

Complexity (cont.)

When $i = 1$, no comparisons are made by the program. When $i = 2$, one comparison is made by the program. When $i = 3$, two comparisons are made, and so on. Thus, we can conclude that when $i = m,\ m - 1$ comparisons are made. Hence, in an array of length *n*, it does $1 + 2 + 3 + 4 + \cdots + (n - 2) + (n - 1)$ comparisons.

Using the previous formula to calculate $1 + 2 + 3 + 4 + \ \ldots\ + (n - 2) + (n - 1)$, it follows that

$$\frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$$

As expected, the algorithm's complexity is n^2.

Complexity (cont.)

n is the *best-case* running time for bubble sort. It is possible to modify bubble sort to keep track of the number of swaps it performs. If an array is already in sorted order, and bubble sort makes no swaps, the algorithm can terminate after one pass. With this modification, if bubble sort encounters a list that is already sorted, it will finish in n time.