Algorithms

Tushar B. Kute,
http://tusharkute.com

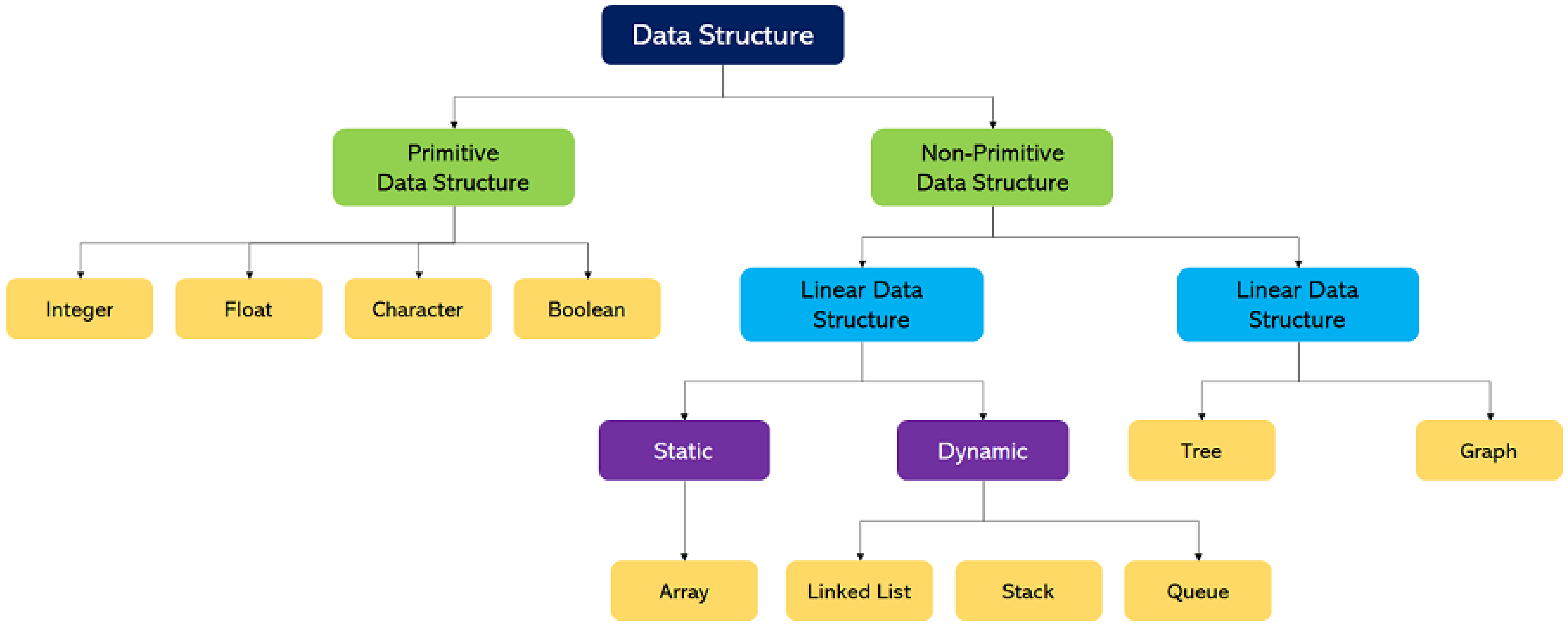
Data Structure

- The data structure name indicates itself that organizing the data in memory.
- There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language.
- Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner.
- This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory.

Data Structure

- Primitive Data structure
  - The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.
- Non-Primitive Data structure
  - The non-primitive data structure is divided into two types:
    - Linear data structure
    - Non-linear data structure

Data Structure



Algorithm

- An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer.
- The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task.
- It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

Algorithm : Characteristics

- Input: An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- Output: We will get 1 or more output at the end of an algorithm.
- Unambiguity: An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- Finiteness: An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- Effectiveness: An algorithm should be effective as each instruction in an algorithm affects the overall process.
- Language independent: An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

Algorithm : Dataflow

- Problem: A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- Algorithm: An algorithm will be designed for a problem which is a step by step procedure.
- Input: After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- Processing unit: The input will be given to the processing unit, and the processing unit will produce the desired output.
- Output: The output is the outcome or the result of the program.

Algorithm : Categories

- The major categories of algorithms are given below:
  - Sort: Algorithm developed for sorting the items in a certain order.
  - Search: Algorithm developed for searching the items inside a data structure.
  - Delete: Algorithm developed for deleting the existing element from the data structure.
  - Insert: Algorithm developed for inserting an item inside a data structure.
  - Update: Algorithm developed for updating the existing element inside a data structure.

Algorithm : Analysis

- Priori Analysis:
  - Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm. Various factors can be considered before implementing the algorithm like processor speed, which has no effect on the implementation part.
- Posterior Analysis:
  - Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing the algorithm using any programming language. This analysis basically evaluate that how much running time and space taken by the algorithm.

Time Complexity

- The time complexity of an algorithm is the amount of time required to complete the execution.
- The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity.
- The time complexity is mainly calculated by counting the number of steps to finish the execution.

Space Complexity

- Space complexity: An algorithm's space complexity is the amount of space required to solve a problem and produce an output.
- Similar to the time complexity, space complexity is also expressed in big O notation.

Types of Algorithms

- The following are the types of algorithm:
  - Search Algorithm
  - Sort Algorithm

Search Algorithms

- On each day, we search for something in our day to day life. Similarly, with the case of computer, huge data is stored in a computer that whenever the user asks for any data then the computer searches for that data in the memory and provides that data to the user.
- There are mainly two techniques available to search the data in an array:
  - Linear search
  - Binary search

Linear Search

- Linear search is a very simple algorithm that starts searching for an element or a value from the beginning of an array until the required element is not found.
- It compares the element to be searched with all the elements in an array, if the match is found, then it returns the index of the element else it returns -1.
- This algorithm can be implemented on the unsorted list.

Binary Search

- A Binary algorithm is the simplest algorithm that searches the element very quickly. It is used to search the element from the sorted list.
- The elements must be stored in sequential order or the sorted manner to implement the binary algorithm. Binary search cannot be implemented if the elements are stored in a random manner.
- It is used to find the middle element of the list.

Sorting Algorithms

- Sorting algorithms are used to rearrange the elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements.
- Why do we need a sorting algorithm?
  - An efficient sorting algorithm is required for optimizing the efficiency of other algorithms like binary search algorithm as a binary search algorithm requires an array to be sorted in a particular order, mainly in ascending order.
  - It produces information in a sorted order, which is a human-readable format.
  - Searching a particular element in a sorted list is faster than the unsorted list.

Asymptotic Analysis

- As we know that data structure is a way of organizing the data efficiently and that efficiency is measured either in terms of time or space.
- So, the ideal data structure is a structure that occupies the least possible time to perform all its operation and the memory space.
- Our focus would be on finding the time complexity rather than space complexity, and by finding the time complexity, we can decide which data structure is the best for an algorithm.

How to calculate complexity?

- How to calculate f(n)?
  - Calculating the value of f(n) for smaller programs is easy but for bigger programs, it's not that easy. We can compare the data structures by comparing their f(n) values.
  - We can compare the data structures by comparing their f(n) values.
  - We will find the growth rate of f(n) because there might be a possibility that one data structure for a smaller input size is better than the other one but not for the larger sizes.

How to calculate complexity?

- Let's look at a simple example.

  $f(n) = 5n^2 + 6n + 12$

  where n is the number of instructions executed, and it depends on the size of the input.

- When n=1

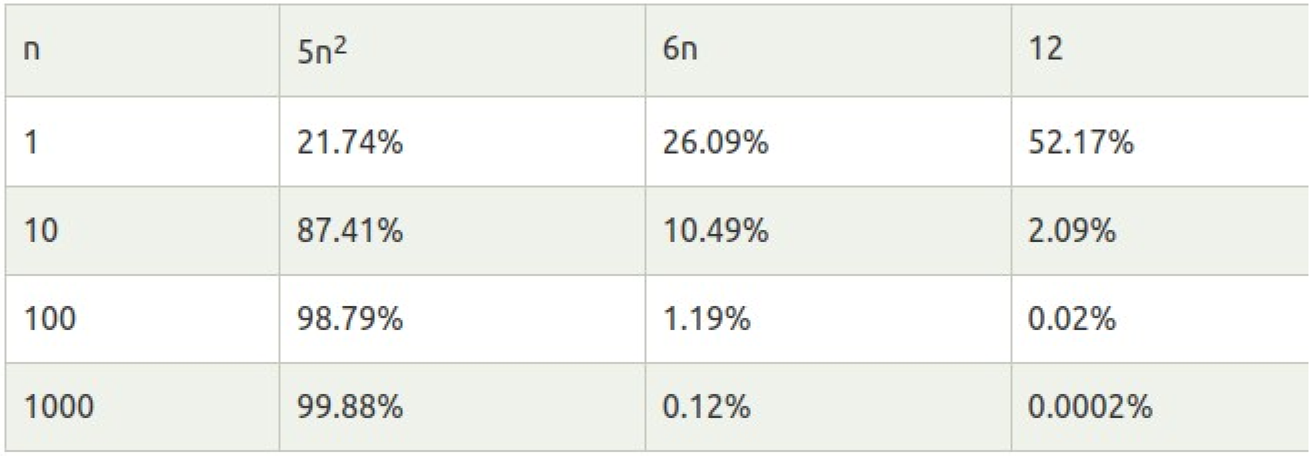
  % of running time due to $5n^2 = \frac{5}{5+6+12} * 100 = 21.74\%$

  % of running time due to $6n = \frac{5}{5+6+12} * 100 = 26.09\%$

  % of running time due to $12 = \frac{5}{5+6+12} * 100 = 52.17\%$

How to calculate complexity?

- From the above calculation, it is observed that most of the time is taken by 12.
- But, we have to find the growth rate of f(n), we cannot say that the maximum amount of time is taken by 12. Let's assume the different values of n to find the growth rate of f(n).



n	$5n^2$	6n	12
1	21.74%	26.09%	52.17%
10	87.41%	10.49%	2.09%
100	98.79%	1.19%	0.02%
1000	99.88%	0.12%	0.0002%

How to calculate complexity?

- Example: Running time of one operation is x(n) and for another operation, it is calculated as f(n2). It refers to running time will increase linearly with an increase in 'n' for the first operation, and running time will increase exponentially for the second operation. Similarly, the running time of both operations will be the same if n is significantly small.
- Usually, the time required by an algorithm comes under three types:
- Worst case: It defines the input for which the algorithm takes a huge time.
- Average case: It takes average time for the program execution.
- Best case: It defines the input for which the algorithm takes the lowest time

Asymptotic Notations

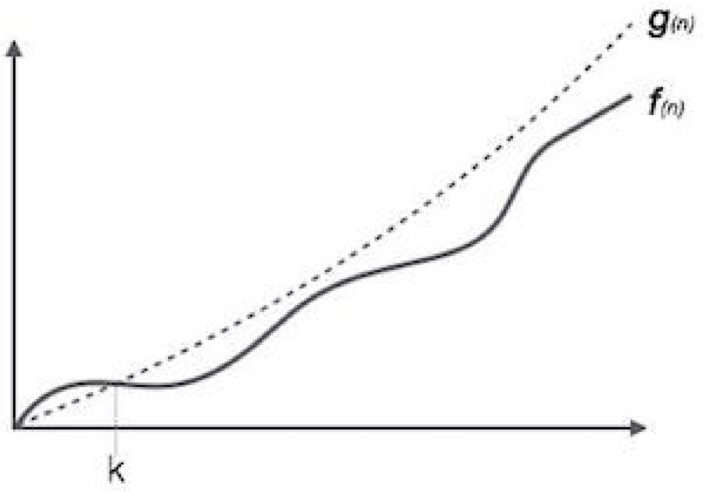
- The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:
  - Big oh Notation (O)
  - Omega Notation (Ω)
  - Theta Notation (θ)

Big oh Notation (O)

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound.
- So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

Big oh Notation (O)

- It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:



Big oh Notation (O)

- If f(n) and g(n) are the two functions defined for positive integers,
- then f(n) = O(g(n)) as f(n) is big oh of g(n) or f(n) is on the order of g(n)) if there exists constants c and no such that:

  $$f(n) \leq c.g(n) \text{ for all } n \geq no$$

- This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n).
- In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

Big oh Notation (O)

- Let's understand through examples
- Example 1: $f(n)=2n+3$, $g(n)=n$

  Now, we have to find Is $f(n)=O(g(n))$?

- To check $f(n)=O(g(n))$, it must satisfy the given condition:

$$f(n)<=c.g(n)$$

Big oh Notation (O)

- First, we will replace f(n) by 2n+3 and g(n) by n.

  2n+3 <= c.n
- Let's assume c=5, n=1 then

  2*1+3<=5*1

  5<=5
- For n=1, the above condition is true.

  If n=2

  2*2+3<=5*2

  7<=10

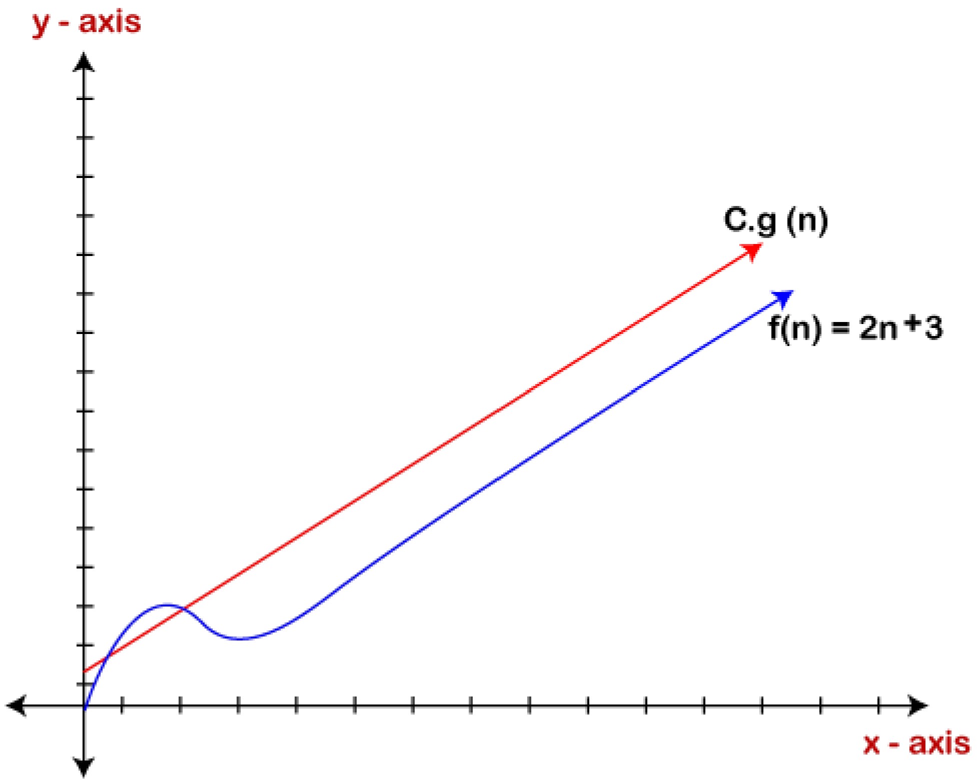
  For n=2, the above condition is true.

Big oh Notation (O)

- We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n.
- We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n.
- As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n).

Big oh Notation (O)

- It can be represented graphically as:



Big oh Notation (O)

- The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity.
- It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

Omega Notation (Ω)

- It basically describes the best-case scenario which is opposite to the big o notation.
- It is the formal way to represent the lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity.
- It determines what is the fastest time that an algorithm can run.
- If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big-Ω notation i.e. the Greek letter "omega". It is used to bound the growth of running time for large input size.

Theta Notation (θ)

- The theta notation mainly describes the average case scenarios.
- It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm.
- Big theta is mainly used when the value of worst-case and the best-case is same.
- It is the formal way to express both the upper bound and lower bound of an algorithm running time.

Common Asymptotic Notations

constant	-	?(1)
linear	-	?(n)
logarithmic	-	?(log n)
n log n	-	?(n log n)
exponential	-	2?(n)
cubic	-	?(n3)
polynomial	-	n?(1)
quadratic	-	?(n2)

Searching Algorithms

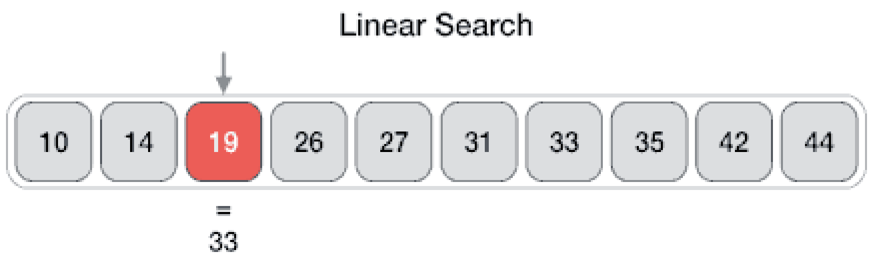
- A search algorithm is an algorithm designed to solve a search problem. Search algorithms work to retrieve information stored within particular data structure, or calculated in the search space of a problem domain, with either discrete or continuous values.
- Although search engines use search algorithms, they belong to the study of information retrieval, not algorithmics.
- The appropriate search algorithm often depends on the data structure being searched, and may also include prior knowledge about the data.
- Search algorithms can be made faster or more efficient by specially constructed database structures, such as search trees, hash maps, and database indexes.

Basic Searching Algorithms

- 1. Linear / Sequential Search
- 2. Binary Search

Linear Search

- Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one.
- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Algorithm

- Linear Search (Array A, Value x)
  - Step 1: Set i to 1
  - Step 2: if i > n then go to step 7
  - Step 3: if A[i] = x then go to step 6
  - Step 4: Set i to i + 1
  - Step 5: Go to Step 2
  - Step 6: Print Element x Found at index i and go to step 8
  - Step 7: Print element not found
  - Step 8: Exit

Pseudo-Code

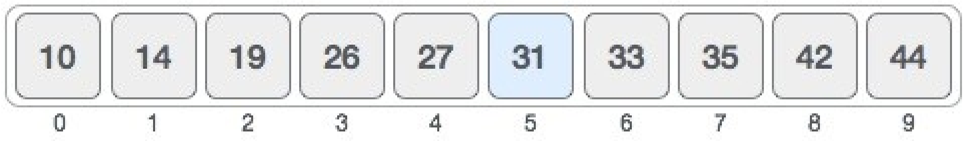
```
procedure linear_search (list, value)

   for each item in the list
      if match item == value
         return the item's location
      end if
   end for

end procedure
```

Binary Search

- Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.
- Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item.
- Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero

Binary Search

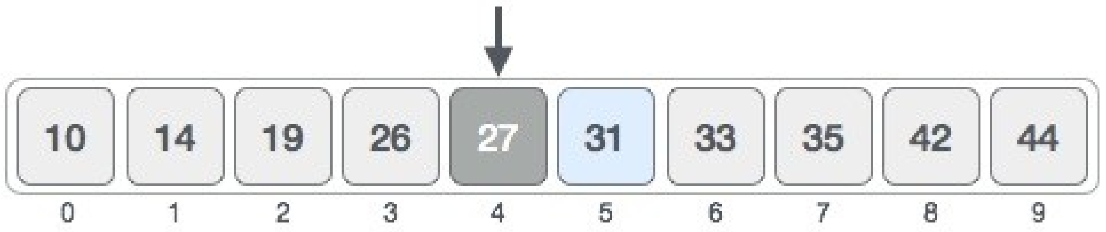
- For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example.
- The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



- First, we shall determine half of the array by using this formula –
- mid = low + (high - low) / 2

Binary Search

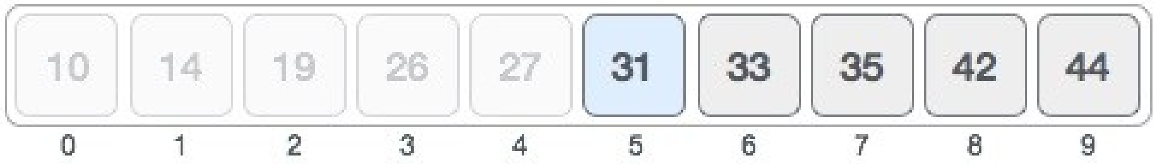
- Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



- Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match.
- As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

Binary Search

- Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



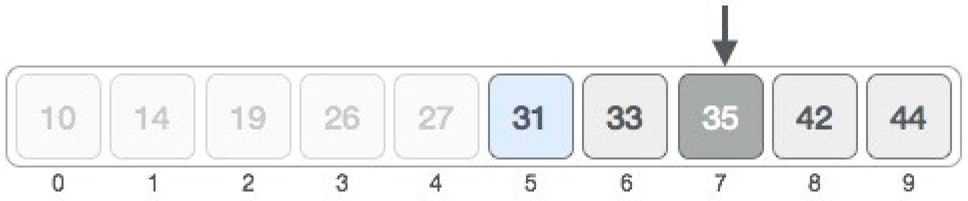
- We change our low to mid + 1 and find the new mid value again.

  low = mid + 1

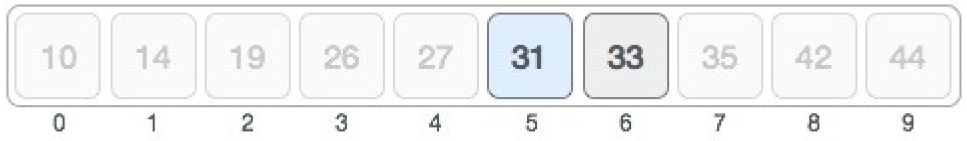
  mid = low + (high - low) / 2

Binary Search

- Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

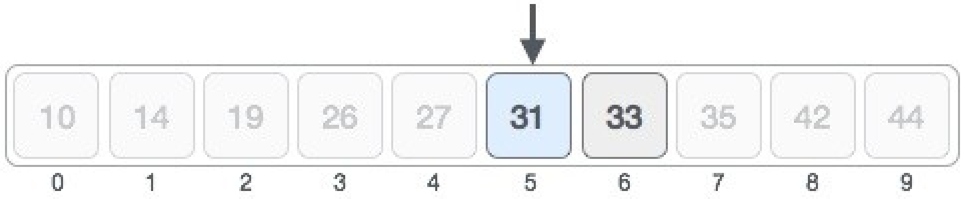


- The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

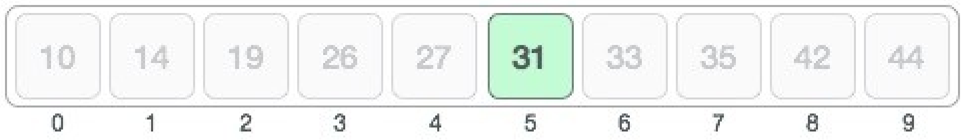


Binary Search

- Hence, we calculate the mid again. This time it is 5.



- We compare the value stored at location 5 with our target value. We find that it is a match.



- We conclude that the target value 31 is stored at location 5.
- Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudo-code

```
Procedure binary_search
  A ← sorted array
  n ← size of array
  x ← value to be searched

  Set lowerBound = 1
  Set upperBound = n

  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.

    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

    if A[midPoint] < x
      set lowerBound = midPoint + 1

    if A[midPoint] > x
      set upperBound = midPoint - 1

    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while

end procedure
```

Sorting Algorithms

- Sorting refers to arranging data in a particular format.
- Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.
- The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner.
- Sorting is also used to represent data in more readable formats.

Sorting Algorithms

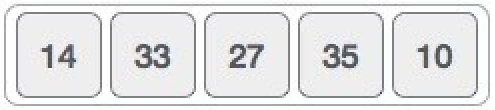
- Bubble Sort
- Quick Sort
- Heap Sort
- Merge Sort

Bubble Sort

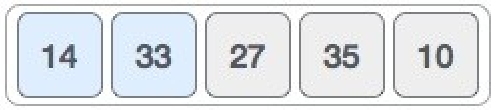
- Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.

Bubble Sort: Working

- We take an unsorted array for our example. Bubble sort takes O(n2) time so we're keeping it short and precise.



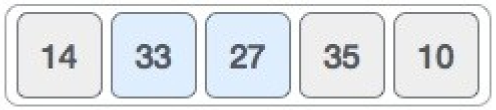
- Bubble sort starts with very first two elements, comparing them to check which one is greater.



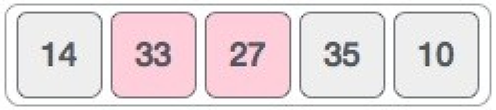
- In this case, value 33 is greater than 14, so it is already in sorted locations.

Bubble Sort: Working

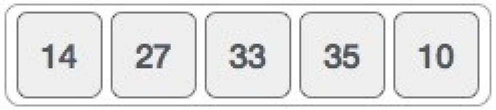
- Next, we compare 33 with 27.



- We find that 27 is smaller than 33 and these two values must be swapped.

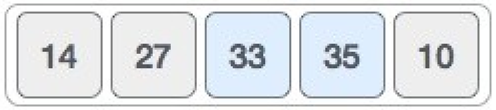


- The new array should look like this –

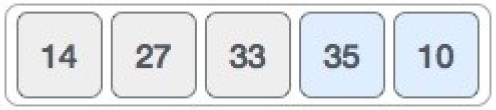


Bubble Sort: Working

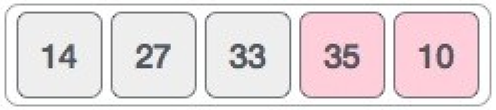
- Next we compare 33 and 35. We find that both are in already sorted positions.



- Then we move to the next two values, 35 and 10.

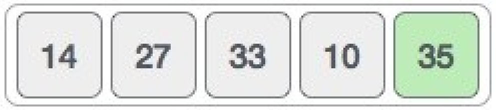


- We know then that 10 is smaller 35. Hence they are not sorted.

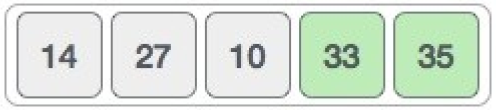


Bubble Sort: Working

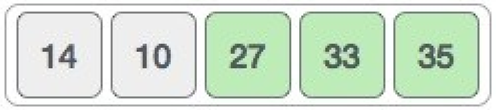
- We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –



- To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

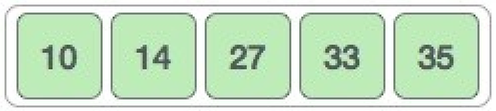


- Notice that after each iteration, at least one value moves at the end.



Bubble Sort: Working

- And when there's no swap required, bubble sorts learns that an array is completely sorted.



Bubble Sort Algorithm

- We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
begin BubbleSort(list)
   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for
   return list
end BubbleSort
```

Bubble Sort Pseudo-code

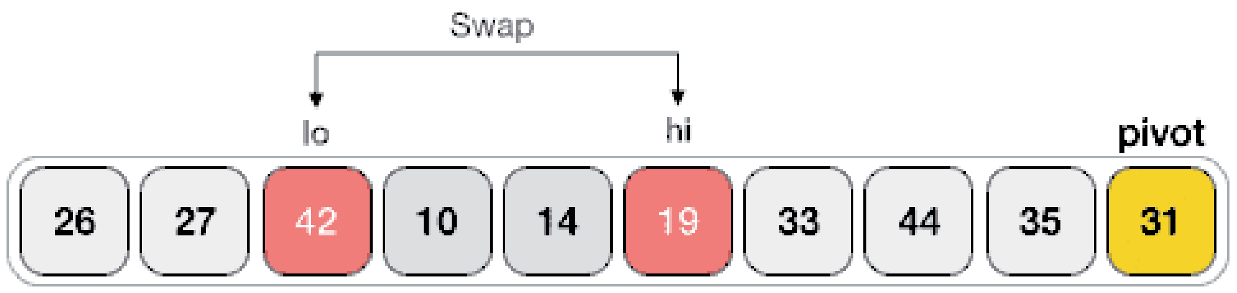
```
procedure bubbleSort( list : array of items )
  loop = list.count;
  for i = 0 to loop-1 do:
    swapped = false
    for j = 0 to loop-1 do:
      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
      end if
    end for
    /*if no number was swapped that means
    array is sorted now, break the loop.*/
    if(not swapped) then
      break
    end if
  end for
end procedure return list
```

Quick Sort

- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
- Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.
- This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n^2), respectively.

Quick Sort

- Following animated representation explains how to find the pivot value in an array.



- The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort

- Step 1 – Choose the highest index value has pivot
- Step 2 – Take two variables to point left and right of the list excluding pivot
- Step 3 – left points to the low index
- Step 4 – right points to the high
- Step 5 – while value at left is less than pivot move right
- Step 6 – while value at right is greater than pivot move left
- Step 7 – if both step 5 and step 6 does not match swap left and right
- Step 8 – if left ≥ right, the point where they met is new pivot

Quick Sort: Pseudo-code

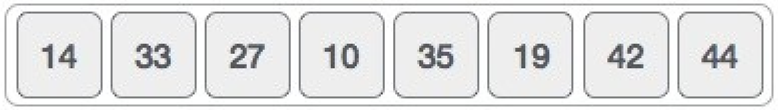
```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1

   while True do
      while A[++leftPointer] < pivot do
         //do-nothing
      end while

      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while

      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
      end if

   end while

   swap leftPointer,right
   return leftPointer

end function
```

Merge Sort

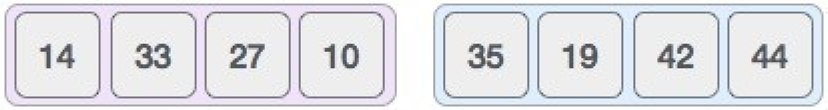
- Merge sort is a sorting technique based on divide and conquer technique.
- With worst-case time complexity being O(n log n), it is one of the most respected algorithms.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Merge Sort

- To understand merge sort, we take an unsorted array as the following –

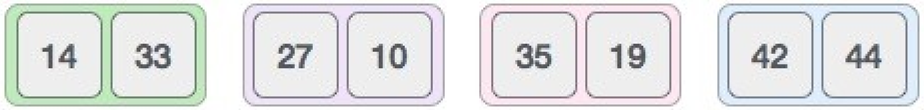


- We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

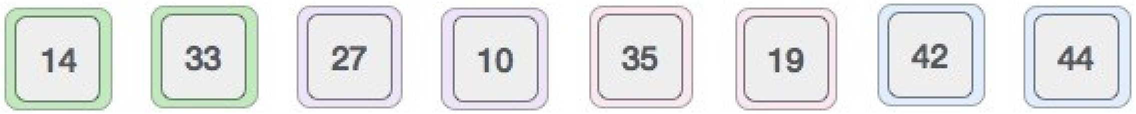


Merge Sort

- This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

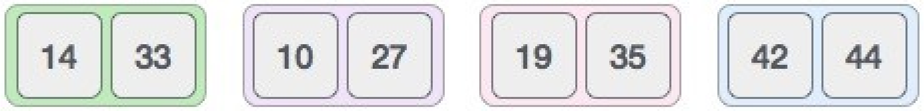


- We further divide these arrays and we achieve atomic value which can no more be divided.



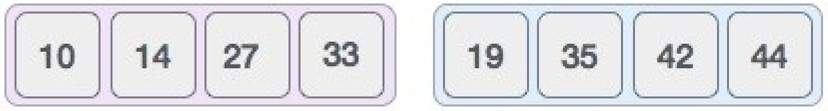
Merge Sort

- Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.
- We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

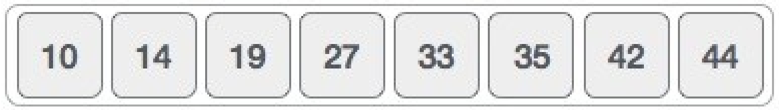


Merge Sort

- In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



- After the final merging, the list should look like this –



- Now we should learn some programming aspects of merge sorting.

Algorithm

- Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted.
- Then, merge sort combines the smaller sorted lists keeping the new list sorted too.
  - Step 1 – if it is only one element in the list it is already sorted, return.
  - Step 2 – divide the list recursively into two halves until it can no more be divided.
  - Step 3 – merge the smaller lists into new list in sorted order.

Heap Sort

- Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array.
- Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.
- Heap sort basically recursively performs two main operations -
  - Build a heap H, using the elements of array.
  - Repeatedly delete the root element of the heap formed in 1st phase.

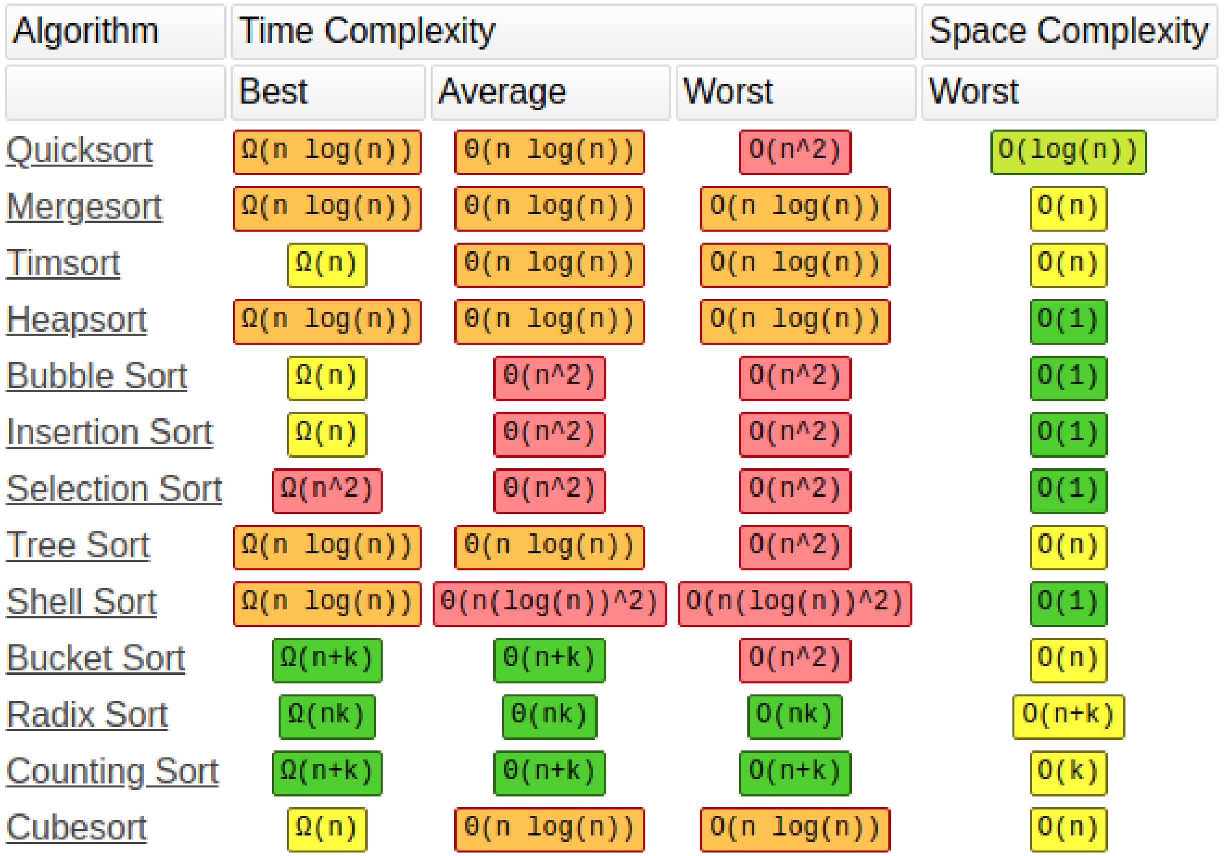
What is Heap?

- A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children.
- A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

What is Heap Sort?

- Heapsort is a popular and efficient sorting algorithm.
- The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.
- Heapsort is the in-place sorting algorithm.

Summary



Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Thank you

@mitu_skillologies @mITuSkillologies @mitu_group @mitu-skillologies @MITUSkillologies

kaggle
@mituskillologies

Web Resources
https://mitu.co.in
http://tusharkute.com

@mituskillologies

contact@mitu.co.in
tushar@tusharkute.com