UNIT-I

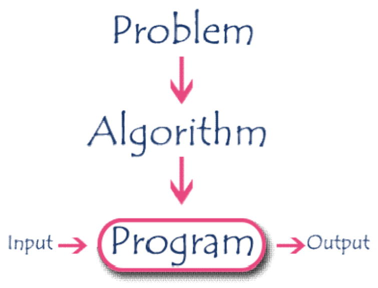
Introduction to Algorithm

What is an algorithm?

An algorithm is a step by step procedure to solve a problem. In normal language, algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer

Algorithms are used to convert our problem solution into step by step statements. These statements can be converted into computer programming instructions which forms a program. This program is executed by computer to produce solution. Here, program takes required data as input, processes data according to the program instructions and finally produces result as shown in the following picture.



Algorithm Specifications

Every algorithm must satisfy the following specifications...

Input - Every algorithm must take zero or more number of input values from external.

Output - Every algorithm must produce an output as result.

Definiteness - Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation)

Finiteness - For all different cases, the algorithm must produce result within a finite number of steps.

Effectiveness - Every instruction must be basic enough to be carried out and it also must be feasible.

Example of Algorithm

Let us consider the following problem for finding the largest value in a given list of values.

Problem Statement : Find the largest number in the given list of numbers?
Input : A list of positive integer numbers. (List must contain at least one number).
Output : The largest number in the given list of positive integer numbers.

Consider the given list of numbers as 'L' (input), and the largest number as 'max' (Output).

Algorithm

Step 1: Define a variable 'max' and initialize with '0'.

Step 2: Compare first number (say 'x') in the list 'L' with 'max', if 'x' is larger than 'max', set 'max' to 'x'.

Step 3: Repeat step 2 for all numbers in the list 'L'.

Step 4: Display the value of 'max' as a result.

Code in C Programming

```
int findMax(L)
{
  int max = 0,i;
  for(i=0; i < listSize; i++)
  {
    if(L[i] > max)
       max = L[i];
  }
  return max;
}
```

Recursive Algorithm

In computer science, all algorithms are implemented with programming language functions. We can view a function as something that is invoked (called) by another function. It executes its code and then returns control to the calling function. Here, a function can call themselves (by itself) or it may call another function which again call same function inside it.

The function which calls by itself is called as Direct Recursive function (or Recursive function)

A recursive algorithm can also be defined as follows...

The function which calls a function and that function calls its called function is called Indirect Recursive function (or Recursive function)

Most of the computer science students think that recursive is a technique useful for only a few special problems like computing factorials, Ackermann's function etc., This is unfortunate because any function implemented using assignment or if-else or while or looping statements can also be implemented using recursive functions. This recursive function is very easier to understand when compared to its iterative counterpart.

Performance Analysis

What is Performance Analysis of an algorithm?

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us.

Similarly, in computer science there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyse them and pick the one which is best suitable for our requirements. Formal definition is as follows...

Performance of an algorithm is a process of making evaluative judgement about algorithms.

It can also be defined as follows...

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.

We compare all algorithms with each other which are solving same problem, to select best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, execution speed of that algorithm, easy to understand, easy to implement, etc.,

Generally, the performance of an algorithm depends on the following elements...

Whether that algorithm is providing the exact solution for the problem?

Whether it is easy to understand?

Whether it is easy to implement?

How much space (memory) it requires to solve the problem?

How much time it takes to solve the problem? Etc.,

When we want to analyse an algorithm, we consider only the space and time required by that particular algorithm and we ignore all remaining elements.

Based on this information, performance analysis of an algorithm can also be defined as follows...

Performance analysis of an algorithm is the process of calculating space required by that algorithm and time required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures...

Space required to complete the task of that algorithm (Space Complexity). It includes program space and data space

Time required to complete the task of that algorithm (Time Complexity)

Space Complexity

What is Space complexity?

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

Memory required to store program instructions

Memory required to store constant values

Memory required to store variable values

And for few other things

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

Instruction Space: It is the amount of memory used to store compiled version of instructions.

Environmental Stack: It is the amount of memory used to store information of partially executed functions at the time of function call.

Data Space: It is the amount of memory used to store all the variables and constants.

NOTE

☀When we want to perform analysis of an algorithm based on its Space complexity, we consider only Data Space and ignore Instruction Space as well as Environmental Stack.
That means we calculate only the memory required to store Variables, Constants, Structures, etc.,

To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following...

2 bytes to store Integer value,

4 bytes to store Floating Point value,

1 byte to store Character value,

6 (OR) 8 bytes to store double value

Example 1

Consider the following piece of code...

```
int square(int a)

{

        return a*a;

}
```

In above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for return value.

That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be Constant Space Complexity.

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity

Example 2

Consider the following piece of code...

```
int sum(int A[], int n)
{
   int sum = 0, i;
   for(i = 0; i < n; i++)
      sum = sum + A[i];
   return sum;
}
```

In above piece of code it requires

'n*2' bytes of memory to store array variable 'a[]'
2 bytes of memory for integer parameter 'n'
4 bytes of memory for local integer variables 'sum' and 'i' (2 bytes each)
2 bytes of memory for return value.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the amount of memory depends on the input value of 'n'. This space complexity is said to be Linear Space Complexity.

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity

Time Complexity

What is Time complexity?

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity.

Time complexity of an algorithm can be defined as follows...

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, running time of an algorithm depends upon the following...

Whether it is running on Single processor machine or Multi processor machine.

Whether it is a 32 bit machine or 64 bit machine

Read and Write speed of the machine.

The time it takes to perform Arithmetic operations, logical operations, return value and assignment operations etc.,

Input data

NOTE

☀ When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent. We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc.,

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because, the configuration changes from one system to another system. To solve this problem, we must assume a model machine with specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.

To calculate time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

Single processor machine

32 bit Operating System machine

It performs sequential execution

It requires 1 unit of time for Arithmetic and Logical operations

It requires 1 unit of time for Assignment and Return value

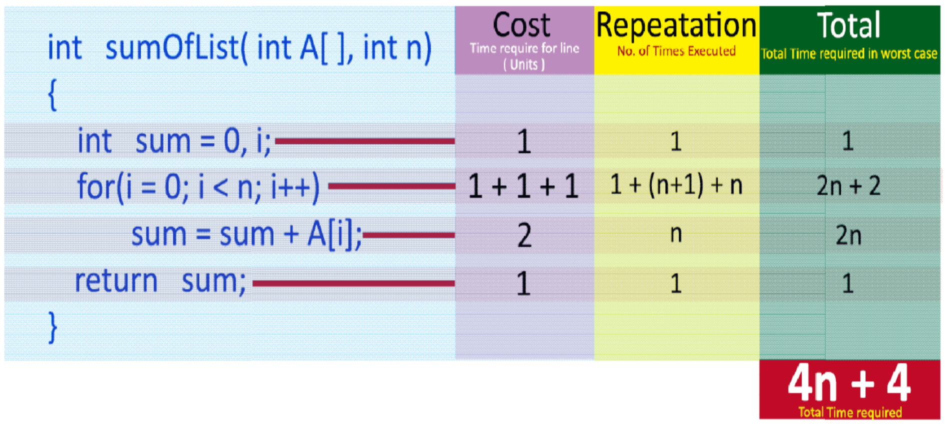
It requires 1 unit of time for Read and Write operations

Now, we calculate the time complexity of following example code by using the above defined model machine...

In above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b. That means for all input values, it requires same amount of time i.e. 2 units.

If any program requires fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

For the above code, time complexity can be calculated as follows...



	Cost Time require for line (Units)	Repeatation No. of Times Executed	Total Total Time required in worst case
`int  sumOfList( int A[ ], int n)`			
`{`			
`int  sum = 0, i;`	1	1	1
`for(i = 0; i < n; i++)`	1 + 1 + 1	1 + (n+1) + n	2n + 2
`sum = sum + A[i];`	2	n	2n
`return  sum;`	1	1	1
`}`			
			4n + 4 Total Time required

In above calculation
Cost is the amount of computer time required for a single operation in each line.
Repeatation is the amount of computer time required by each operation for all its repeatations.
Total is the amount of computer time required by each operation to execute.

So above code requires '4n+4' Units of computer time to complete the task. Here the exact time is not fixed. And it changes based on the n value. If we increase the n value then the time required also increases linearly.

Totally it takes '4n+4' units of time to complete its execution and it is Linear Time Complexity.

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity

Asymptotic Notation

What is Asymptotic Notation?

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity

NOTE

☀ In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity may be Space Complexity or Time Complexity).

For example, consider the following time complexities of two algorithms...

Algorihtm 1 : 5n2 + 2n + 1

Algorihtm 2 : 10n2 + 8n + 3

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. 'n' value). In above two time complexities, for larger value of 'n' the term in algorithm 1 '2n + 1' has least significance than the term '5n2', and the term in algorithm 2 '8n + 3' has least significance than the term '10n2'.

Here for larger value of 'n' the value of most significant terms (5n2 and 10n2) is very larger than the value of least significant terms (2n + 1 and 8n + 3). So for larger value of 'n' we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

Big - Oh (O)

Big - Omega (Ω)

Big - Theta (Θ)

Big - Oh Notation (O)

Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.

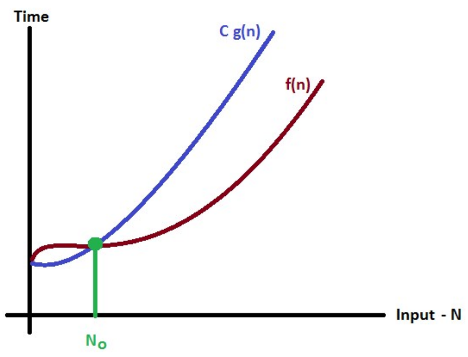
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as O(g(n)).

f(n) = O(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2

g(n) = n
If we want to represent f(n) as O(g(n)) then it must satisfy f(n) <= C x g(n) for all values of C > 0 and n0>= 1

f(n) <= C g(n)
⇒3n + 2 <= C n

Above condition is always TRUE for all values of C = 4 and n >= 2.
By using Big - Oh notation we can represent the time complexity as follows...
3n + 2 = O(n)

Big - Omege Notation (Ω)

Big - Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.

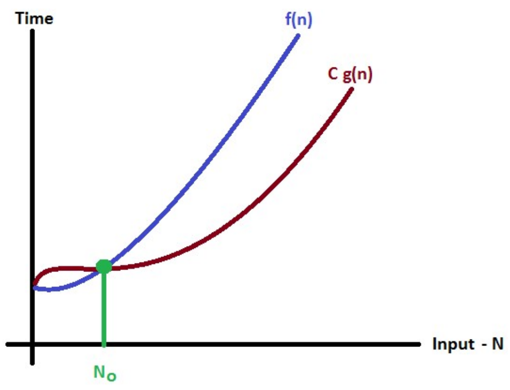
That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C x g(n) for all n >= n0, C > 0 and n0 >= 1. Then we can represent f(n) as Ω(g(n)).

f(n) = Ω(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n0, always C x g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent f(n) as Ω(g(n)) then it must satisfy f(n) >= C g(n) for all values of C > 0 and n0>= 1

f(n) >= C g(n)
⇒3n + 2 <= C n

Above condition is always TRUE for all values of C = 1 and n >= 1.
By using Big - Omega notation we can represent the time complexity as follows...
3n + 2 = Ω(n)

Big - Theta Notation (Θ)

Big - Theta notation is used to define the average bound of an algorithm in terms of Time Complexity.

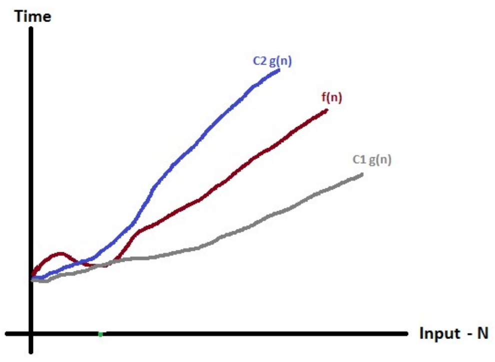
That means Big - Theta notation always indicates the average time required by an algorithm for all input values.
That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If C1 g(n) <= f(n) >= C2 g(n) for all n >= n0, C1, C2 > 0 and n0 >= 1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n0, always C1 g(n) is less than f(n) and C2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent f(n) as Θ(g(n)) then it must satisfy C1 g(n) <= f(n) >= C2 g(n) for all values of C1, C2 > 0 and n0>= 1

C1 g(n) <= f(n) >= ⇒C2 g(n)
C1 n <= 3n + 2 >= C2 n

Above condition is always TRUE for all values of C1 = 1, C2 = 4 and n >= 1.
By using Big - Theta notation we can represent the time compexity as follows...

Linear Search Algorithm
(Sequential Search)

What is Search?

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Linear Search Algorithm (Sequential Search Algorithm)

Linear search algorithm finds given element in a list of elements with O(n) time complexity where n is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching then results with element found otherwise search element is compared with next element in the list. If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

Linear search is implemented using following steps...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the first element in the list.

Step 3: If both are matching, then display "Given element found!!!" and terminate the function

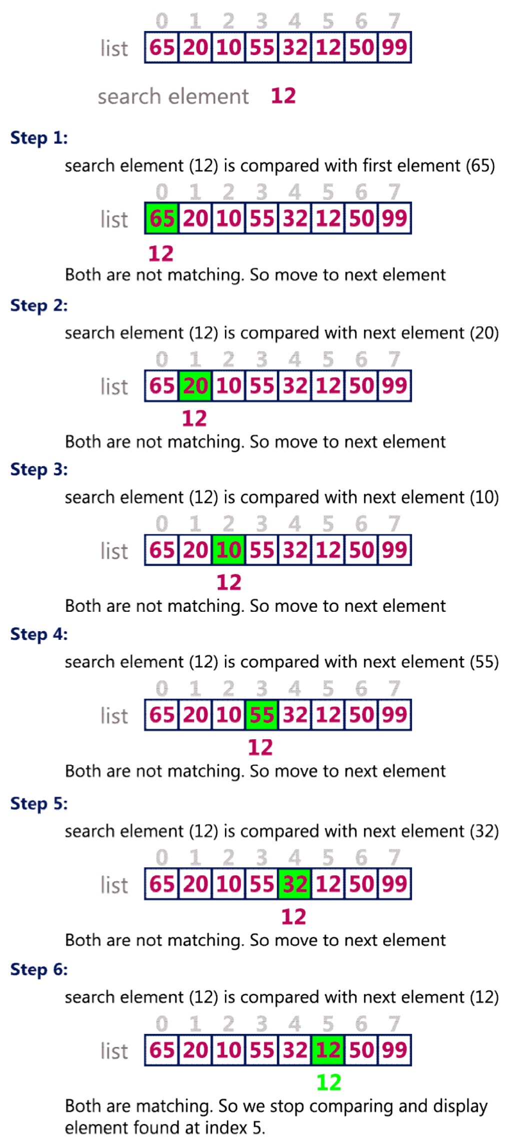
Step 4: If both are not matching, then compare search element with the next element in the list.

Step 5: Repeat steps 3 and 4 until the search element is compared with the last element in the list.

Step 6: If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Example

Consider the following list of element and search element...



Step 1:

search element (12) is compared with first element (65)

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

Binary Search Algorithm

What is Search?

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Binary Search Algorithm

Binary search algorithm finds given element in a list of elements with O(log n) time complexity where n is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with lkist of element which are already arraged in a order. The binary search can not be used for list of element which are in random order. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps...

Step 1: Read the search element from the user

Step 2: Find the middle element in the sorted list

Step 3: Compare, the search element with the middle element in the sorted list.

Step 4: If both are matching, then display "Given element found!!!" and terminate the function

Step 5: If both are not matching, then check whether the search element is smaller or larger than middle element.

Step 6: If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.

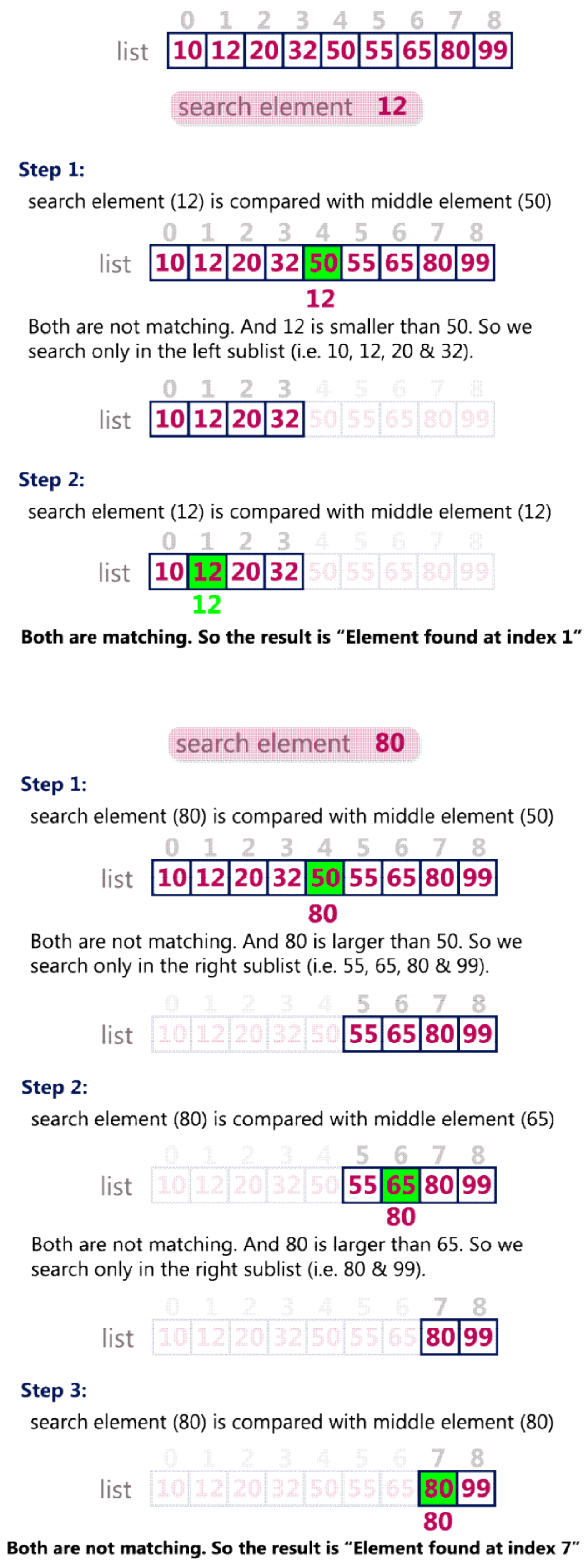
Step 7: If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.

Step 8: Repeat the same process until we find the search element in the list or until sublist contains only one element.

Step 9: If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

Example

Consider the following list of element and search element...


0 1 2 3 4 5 6 7 8
list 10 12 20 32 50 55 65 80 99
search element 12

Step 1:

search element (12) is compared with middle element (50)

list 10 12 20 32 50 55 65 80 99
12

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

Step 1:

search element (80) is compared with middle element (50)

list 10 12 20 32 50 55 65 80 99
80

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

search element (80) is compared with middle element (80)

Both are not matching. So the result is "Element found at index 7"

Static Hashing

In all search techniques like linear search, binary search and search trees, the time required to search an element is depends on the total number of element in that data structure. In all these search techniquies, as the number of element are increased the time required to search an element also increased linearly.

Hashing is another approach in which time required to search an element doesn't depend on the number of element. Using hashing data structure, an element is searched with constant time complexity. Hashing is an effective way to reduce the number of comparisions to seach an element in a data structure.

Hashing is defined as follows...

Hashing is the process of indexing and retrieving element (data) in a datastructure to provide faster way of finding the element using the hash key.

Here, hash key is a value which provides the index value where tha actual data is likely to store in the datastructure.

In this datastructure, we use a concept called Hash table to store data. All the data values are inserted into the hash table based on the hash key value. Hash key value is used to map the data with index in the hash table. And the hash key is generated for every data using a hash function. That means every entry in the hash table is based on the key value generated using a hash function.

Hash Table is defined as follows...

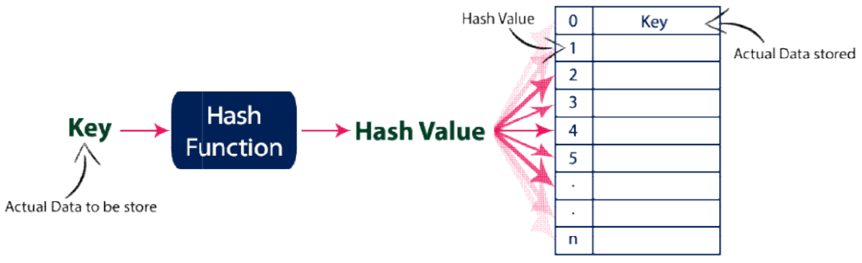
Hash table is just an array which maps a key (data) into the datastructure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity (i.e. O(1)).

Hash tables are used to perform the operations like insertion, deletion and search very quickly in a datastructure. Using hash table concept insertion, deletion and search operations are accoplished in constant time. Generally, every hash table make use of a function, which we'll call the hash function to map the data into the hash table.

A hash function is defined as follows...

Hash function is a function which takes a piece of data (i.e. key) as input and outputs an integer (i.e. hash value) which maps the data to a particular index in the hash table.

Basic concept of hashing and hash table is shown in the following figure...



Insertion Sort

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

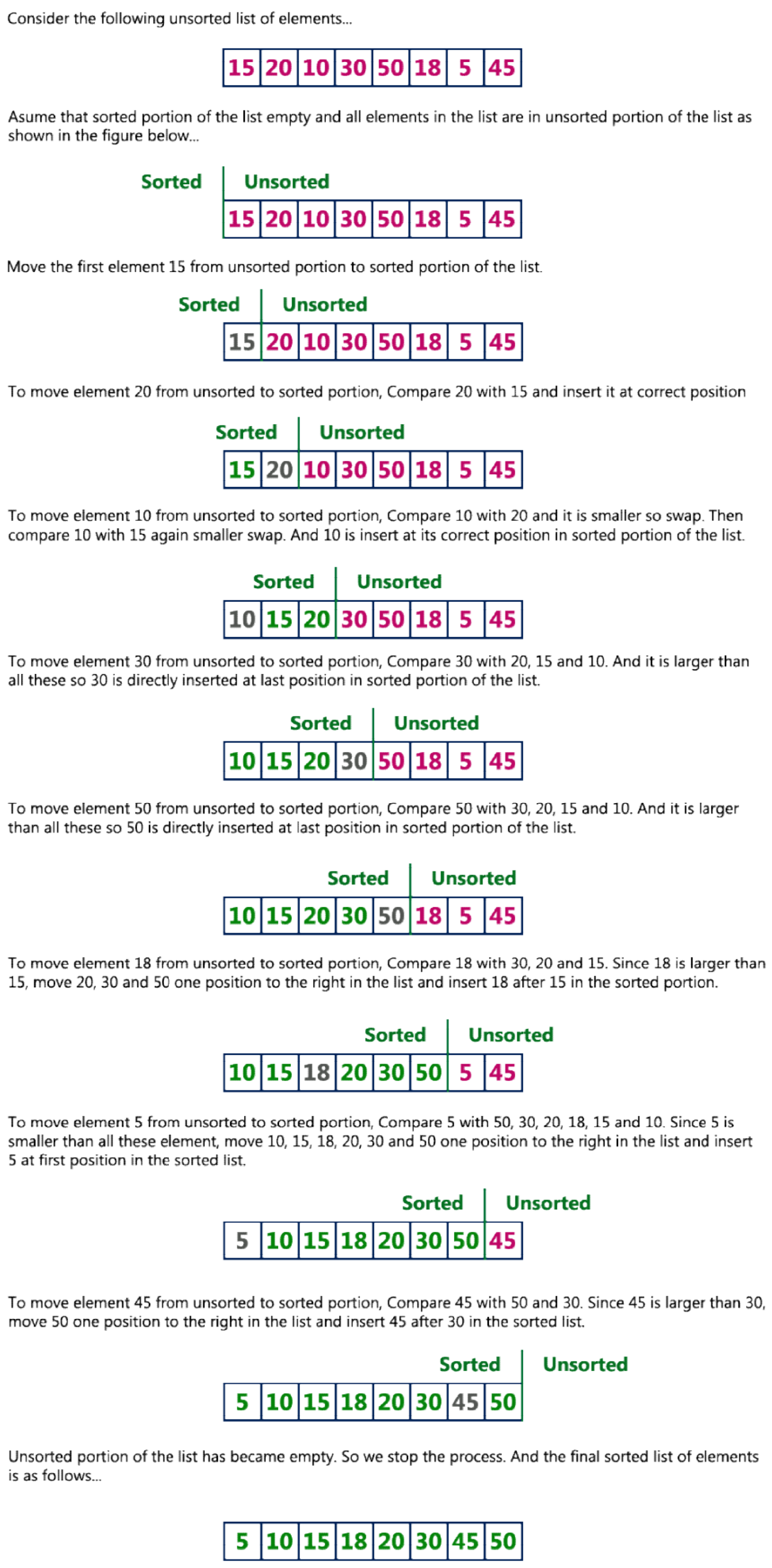
Step by Step Process

The insertion sort algorithm is performed using following steps...

Step 1: Asume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.

Step 2: Consider first element from the unsorted list and insert that element into the sorted list in order specified.

Step 3: Repeat the above process until all the elements from the unsorted list are moved into the sorted list.



Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45

Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Sorted | Unsorted

15	20	10	30	50	18	5	45

Move the first element 15 from unsorted portion to sorted portion of the list.

Sorted | Unsorted

15	20	10	30	50	18	5	45

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

Sorted | Unsorted

15	20	10	30	50	18	5	45

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

Sorted | Unsorted

10	15	20	30	50	18	5	45

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

Sorted | Unsorted

10	15	18	20	30	50	5	45

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

Sorted | Unsorted

5	10	15	18	20	30	50	45

To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.

Sorted | Unsorted

5	10	15	18	20	30	45	50

Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...

5	10	15	18	20	30	45	50

Complexity of the Insertion Sort Algorithm

To sort a unsorted list with 'n' number of elements we need to make (1+2+3+......+n-1) = (n (n-1))/2 number of comparisions in the worst case. If the list already sorted, then it requires 'n' number of comparisions.

Worst Case : O(n2)
Best Case : Ω(n)
Average Case : Θ(n2)

Selection Sort

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped. Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.

Step by Step Process

The selection sort algorithm is performed using following steps...

Step 1: Select the first element of the list (i.e., Element at first position in the list).

Step 2: Compare the selected element with all other elements in the list.

Step 3: For every comparision, if any element is smaller than selected element (for Ascending order), then these two are swapped.

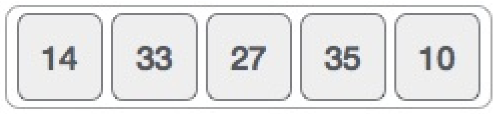
Step 4: Repeat the same procedure with next position in the list till the entire list is sorted.

Example

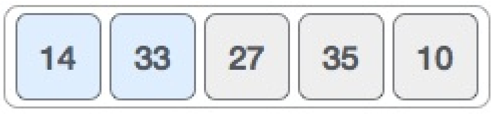
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of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

How Bubble Sort Works?

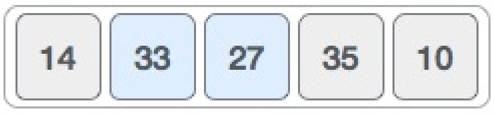
We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



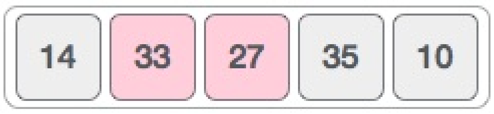
Bubble sort starts with very first two elements, comparing them to check which one is greater.



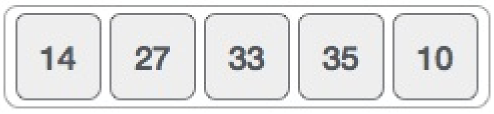
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



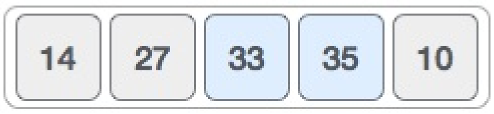
We find that 27 is smaller than 33 and these two values must be swapped.



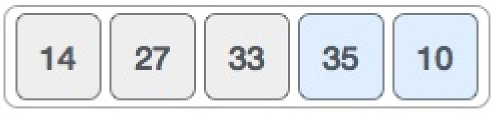
The new array should look like this −



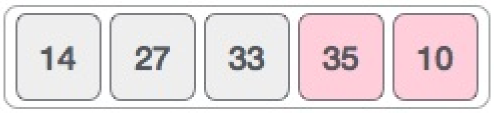
Next we compare 33 and 35. We find that both are in already sorted positions.



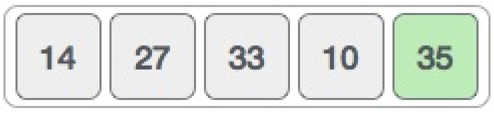
Then we move to the next two values, 35 and 10.



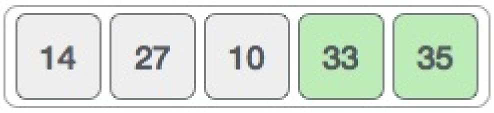
We know then that 10 is smaller 35. Hence they are not sorted.



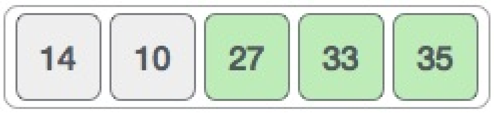
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



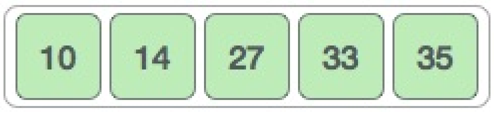
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



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



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



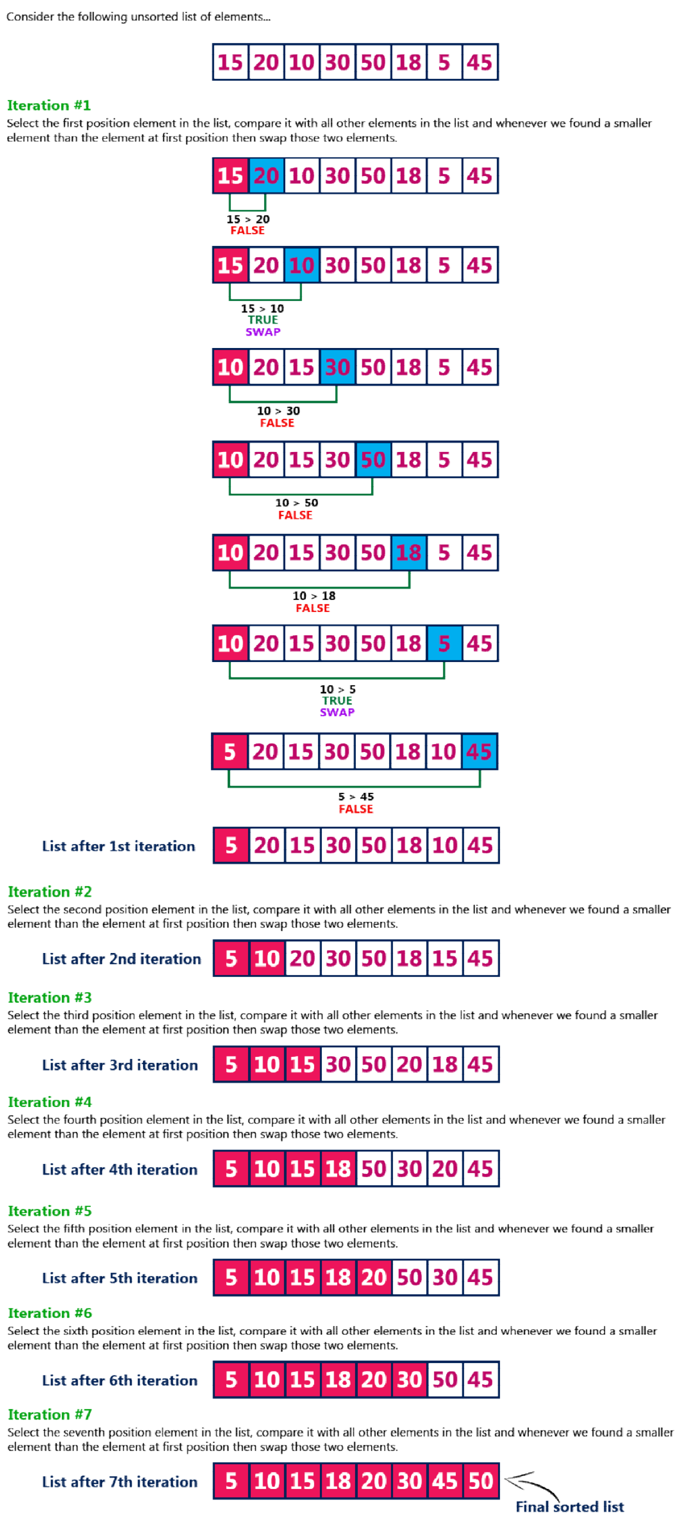
Now we should look into some practical aspects of bubble sort.

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
```


Consider the following unsorted list of elements...

Iteration #1
Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Iteration #2
Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Iteration #3
Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Iteration #4
Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Iteration #5
Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Iteration #6
Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Iteration #7
Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Complexity of the Insertion Sort Algorithm

To sort a unsorted list with 'n' number of elements we need to make ((n-1)+(n-2)+(n-3)+......+1) = (n (n-1))/2 number of comparisions in the worst case. If the list already sorted, then it requires 'n' number of comparisions.

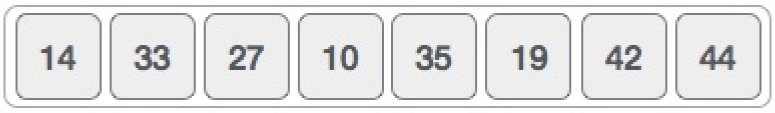
Worst Case : O(n2)
Best Case : Ω(n2)
Average Case : Θ(n2)

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.

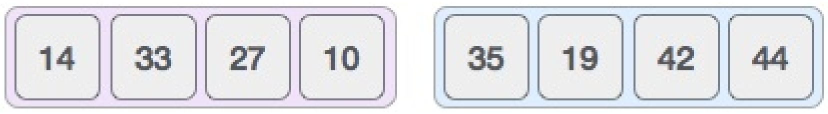
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

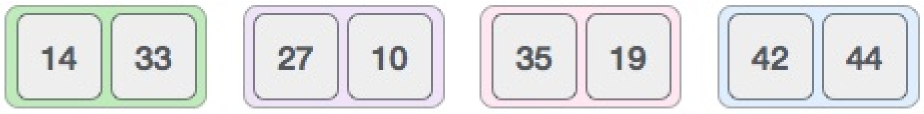
To understand merge sort, we take an unsorted array as the following −



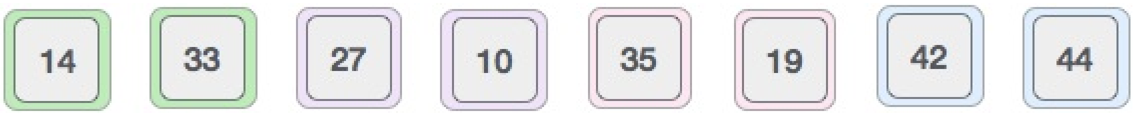
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



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

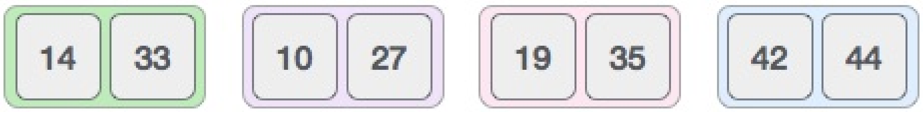


We further divide these arrays and we achieve atomic value which can no more be divided.

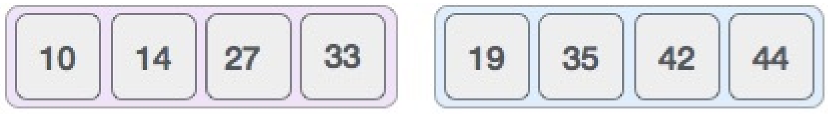


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

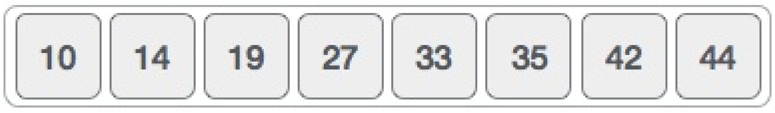
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



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.
```

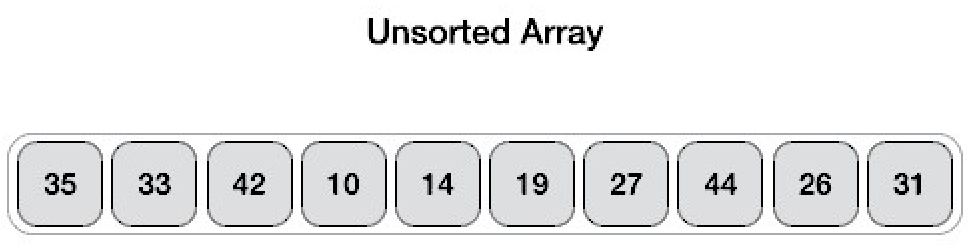
```
Step 2 − divide the list recursively into two halves until
it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted
order.
```

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.

Quick sort 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 of $O(n^2)$, where **n** is the number of items.

Partition in 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 Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

```
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 Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

```
Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively
```

Radix sort is a small method that many people intuitively use when alphabetizing a large list of names. Specifically, the list of names is first sorted according to the first letter of each name, that is, the names are arranged in 26 classes.

Intuitively, one might want to sort numbers on their most significant digit. However, Radix sort works counter-intuitively by sorting on the least significant digits first. On the first pass, all the numbers are sorted on the least significant digit and combined in an array. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combined in an array and so on.

```
Algorithm: Radix-Sort (list, n)
shift = 1
for loop = 1 to keysize do
   for entry = 1 to n do
      bucketnumber = (list[entry].key / shift) mod 10
      append (bucket[bucketnumber], list[entry])
   list = combinebuckets()
   shift = shift * 10
```

Analysis

Each key is looked at once for each digit (or letter if the keys are alphabetic) of the longest key. Hence, if the longest key has **m** digits and there are **n**keys, radix sort has order **O(m.n)**.

However, if we look at these two values, the size of the keys will be relatively small when compared to the number of keys. For example, if we have six-digit keys, we could have a million different records.

Here, we see that the size of the keys is not significant, and this algorithm is of linear complexity **O(n)**.

Example

Following example shows how Radix sort operates on seven 3-digits number.

Input	1st Pass	2nd Pass	3rd Pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

In the above example, the first column is the input. The remaining columns show the list after successive sorts on increasingly significant digits position. The code for Radix sort assumes that each element in an array ***A*** of ***n***elements has ***d*** digits, where digit ***1*** is the lowest-order digit and ***d*** is the highest-order digit.