Data Structures

Introduction of Data structures:

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

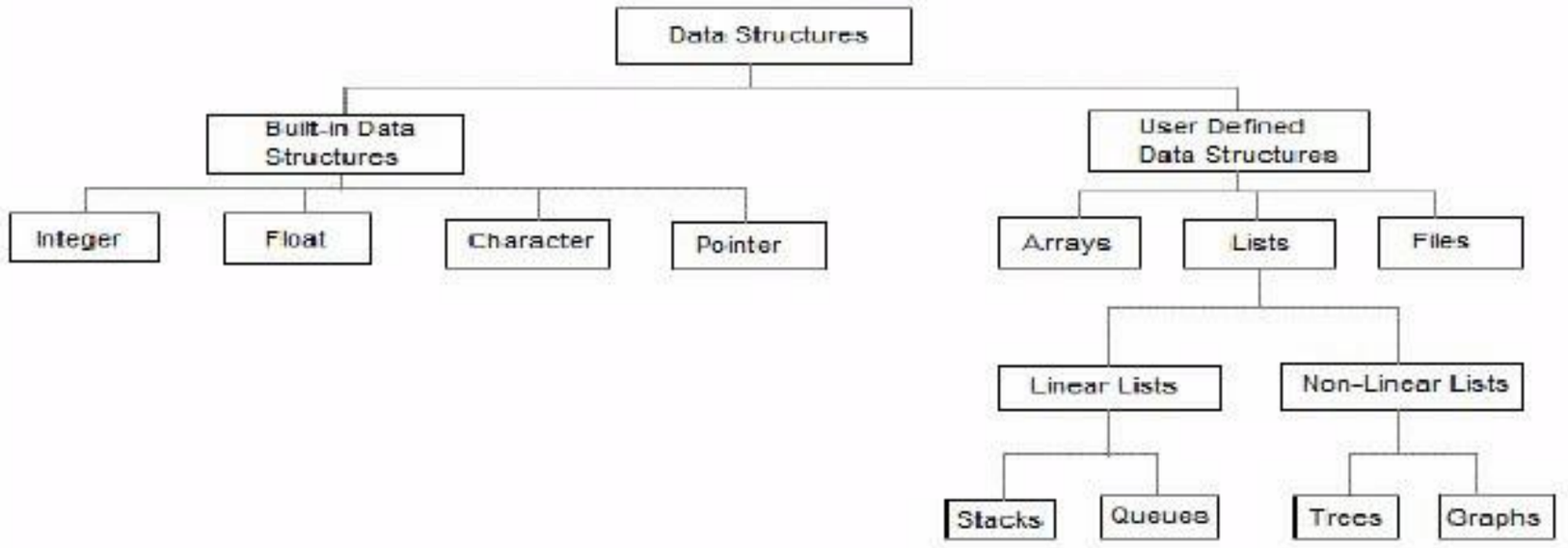
Basic types of Data Structures

As we discussed above, anything that can store data can be called as a data strucure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

What is Algorithm ?

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic (solution) of a problem, which can be expressed either as an informal high level description as **pseudo code** or using a **flowchart**.
An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties:

1. Time Complexity
2. Space Complexity

Space Complexity

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...

1. Memory required to store program instructions
2. Memory required to store constant values
3. Memory required to store variable values
4. 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...

1. Instruction Space: It is the amount of memory used to store compiled version of instructions.
2. Environmental Stack: It is the amount of memory used to store information of partially executed functions at the time of function call.
3. Data Space: It is the amount of memory used to store all the variables and constants.

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 data type values (according to the compiler). For example, the C Programming Language compiler requires the following...

1. 2 bytes to store Integer value,
2. 4 bytes to store Floating Point value,
3. 1 byte to store Character value,
4. 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

Time Complexity is a way to represent the amount of time needed by the program to run to completion.
Time complexity of an algorithm signifies the total time required by the program to run to completion. The time complexity of algorithms is most commonly expressed using the **big O notation**.
Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)
{
  statement;
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.


```
for(i=0; i < N; i++)
{
  for(j=0; j < N;j++)
  {
    statement;
  }
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while(low <= high)
{
  mid = (low + high) / 2;
  if (target < list[mid])
    high = mid - 1;
  else if (target > list[mid])
    low = mid + 1;
```

```
  else break;
}
```

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)
{
  int pivot = partition(list, left, right);
  quicksort(list, left, pivot - 1);
  quicksort(list, pivot + 1, right);
}
```

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE : In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

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 : $5n^2 + 2n + 1$
- Algorihtm 2 : $10n^2 + 8n + 3$

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 '$5n^2$', and the term in algorithm 2 '8n + 3' has least significance than the term '$10n^2$'.

Here for larger value of 'n' the value of most significant terms ($5n^2$ and $10n^2$) 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...

1. Big - Oh (O)
2. Big - Omega (Ω)
3. 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.

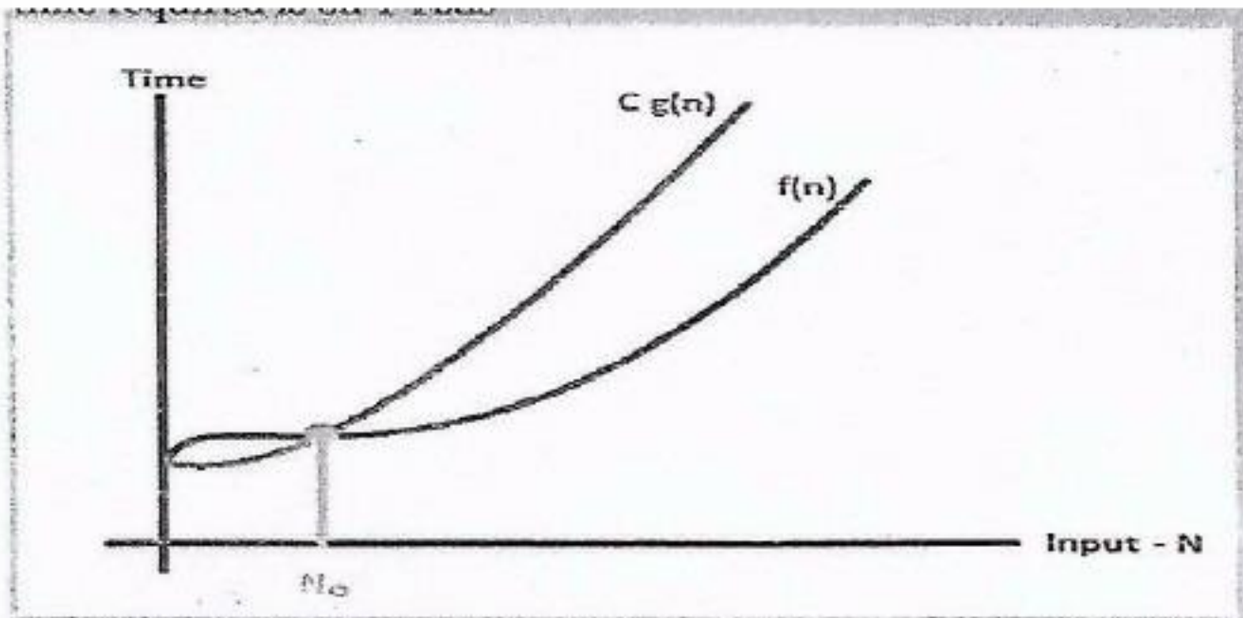
That means Big - Oh notation always indi
cates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

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 $n >= n_0$, $C > 0$ and $n_0 >= 1$. 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 n_0, 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 $f(n) <= C \times g(n)$ for all values of $C > 0$ and $n_0 >= 1$

$f(n) <= C\ g(n)$

$\Rightarrow 3n + 2 <= C\ n$

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.

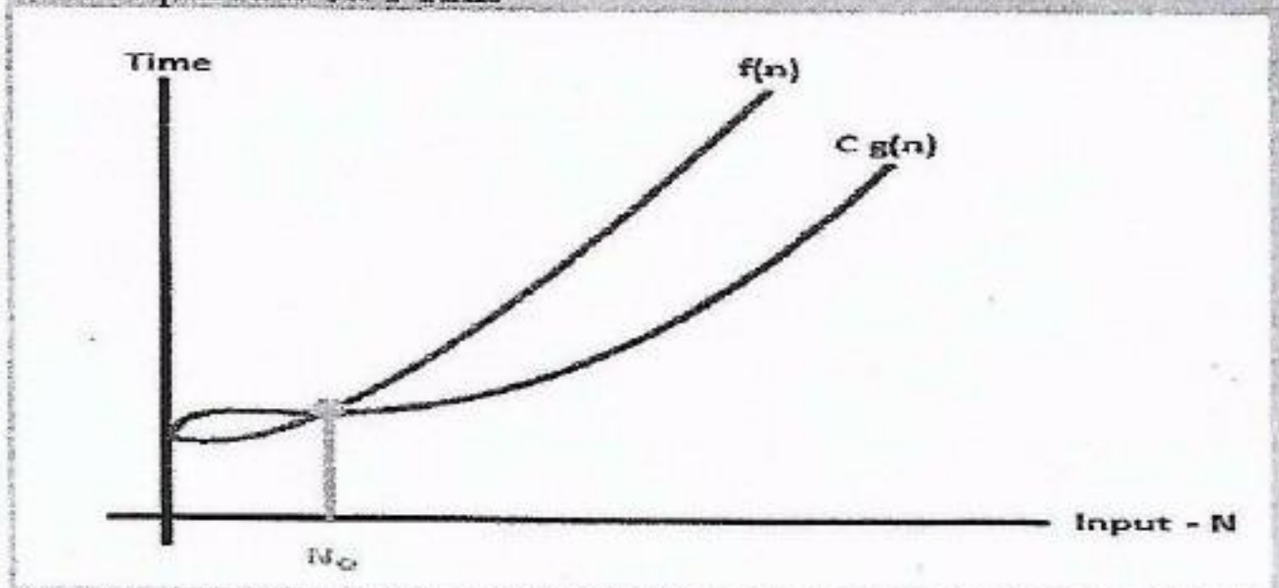
That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

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 $f(n) >= C \times g(n)$ for all $n >= n_0$, $C > 0$ and $n_0 >= 1$. Then we can represent f(n) as $\Omega(g(n))$.

$f(n) = \Omega(g(n))$

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 n_0, 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 $\Omega(g(n))$ then it must satisfy $f(n) >= C\ g(n)$ for all values of $C > 0$ and $n_0 >= 1$

$f(n) >= C\ g(n)$

$\Rightarrow 3n + 2 <= C\ n$

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...

$3n + 2 = \Omega(n)$

Big - Theta Notation (Θ)

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

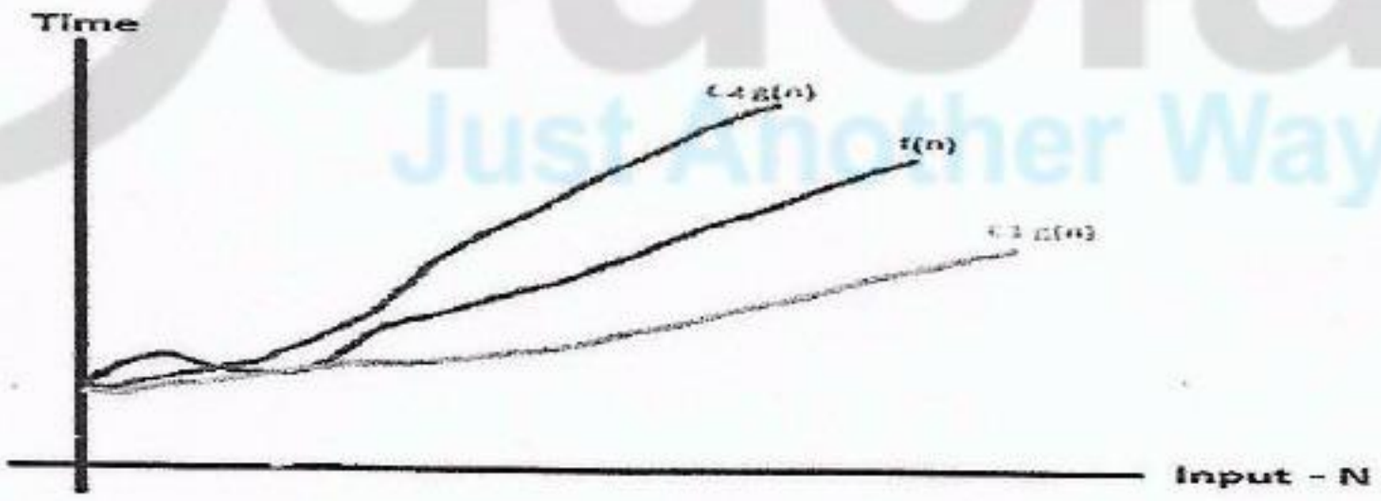
That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

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 $C_1\ g(n) <= f(n) >= C_2\ g(n)$ for all $n >= n_0$, $C_1, C_2 > 0$ and $n_0 >= 1$. Then we can represent f(n) as $\Theta(g(n))$.

$f(n) = \Theta(g(n))$

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 n_0, always $C_1\ g(n)$ is less than f(n) and $C_2\ g(n)$ 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 $\Theta(g(n))$ then it must satisfy $C_1\ g(n) <= f(n) >= C_2\ g(n)$ for all values of $C_1, C_2 > 0$ and $n_0 >= 1$

$C_1\ g(n) <= f(n) >= \Rightarrow C_2\ g(n)$

$C_1\ n <= 3n + 2 >= C_2\ n$

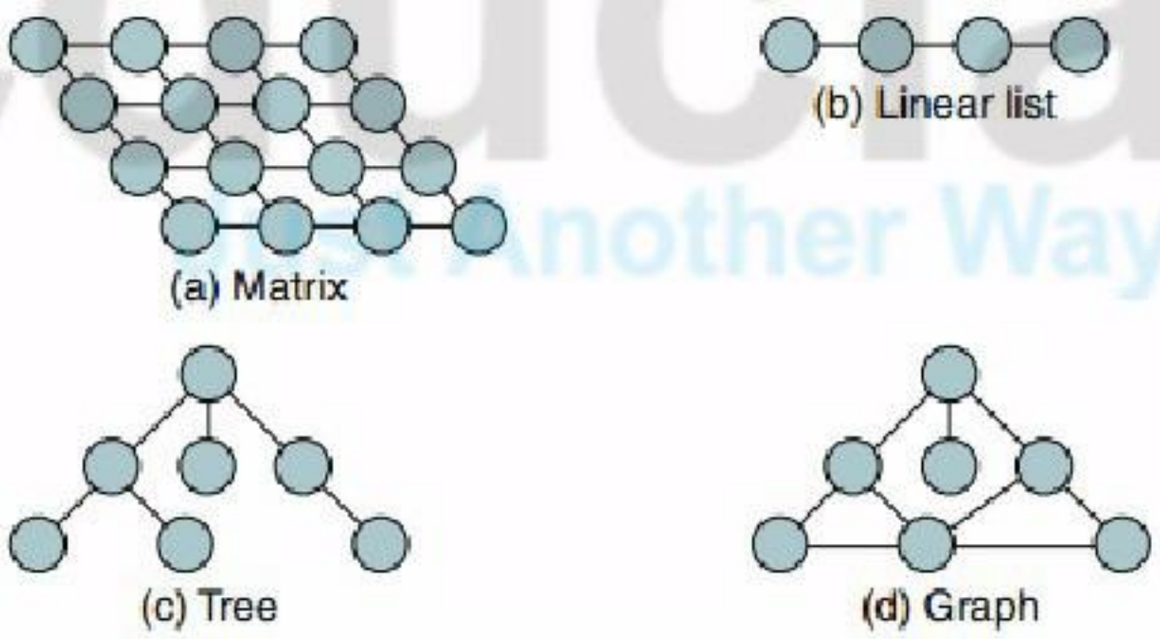
Above condition is always TRUE for all values of $C_1 = 1$, $C_2 = 4$ and $n >= 1$.

By using Big - Theta notation we can represent the time compexity as follows...

$3n + 2 = \Theta(n)$

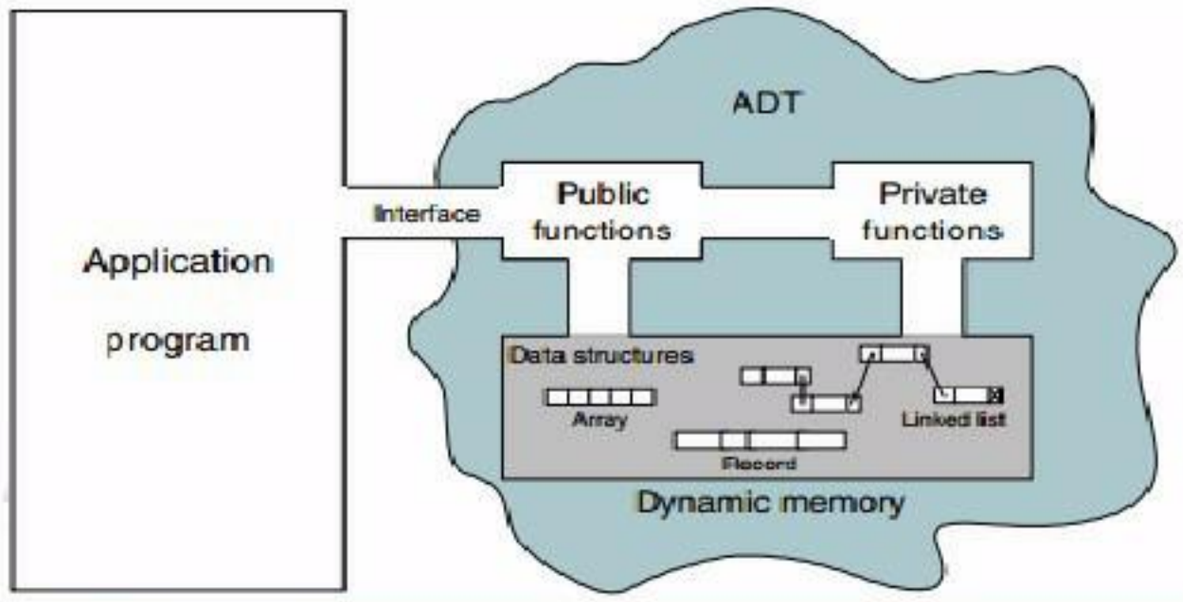
Abstract Data Types:

Abstract Data Type Generally speaking, programmers' capabilities are determined by the tools in their tool kits. These tools are acquired by education and experience. A knowledge of data structures is one of those tools. When we first started programming, there were no abstract data types. If we wanted to read a file, we wrote the code to read the physical file device. It did not take long to realize that we were writing the same code over and over again. So we created what is known today as an abstract data type (ADT). We wrote the code to read a file and placed it in a library for all programmers to use. This concept is found in modern languages today. The code to read the keyboard is an ADT. It has a data structure, a character, and a set of operations that can be used to read that data structure. Using the ADT we can not only read characters but we can also convert them into different data structures such as integers and strings. With an ADT users are not concerned with how the task is done but rather with what it can do. In other words, the ADT consists of a set of definitions that allow programmers to use the functions while hiding the implementation. This generalization of operations with unspecified implementations is known as abstraction. We abstract the essence of the process and leave the implementation details hidden. Consider the concept of a list. At least four data structures can support a list. We can use a matrix, a linear list, a tree, or a graph. If we place our list in an ADT, users should not be aware of the structure we use. As long as they can insert and retrieve data, it should make no difference how we store the data. Figure, shows four logical structures that might be used to hold a list.



Model for an Abstract Data Type:

The ADT model is shown in below Figure. The colored area with an irregular outline represents the ADT. Inside the ADT are two different aspects of the model: data structures and functions (public and private). Both are entirely contained in the model and are not within the application program scope. However, the data structures are available to all of the ADT's functions as needed, and a function may call on other functions to accomplish its task. In other words, the data structures and the functions are within scope of each other.



Abstract Data Type Model

ADT Operations:

Data are entered, accessed, modified, and deleted through the external interface drawn as a passageway partially in and partially out of the ADT. Only the public functions are accessible through this interface. For each ADT operation there is an algorithm that performs its specific task. Only the operation name and its parameters are available to the application, and they provide the only interface to the ADT.

ADT Data Structure:

When a list is controlled entirely by the program, it is often implemented using simple structures similar to those used in your programming class. Because the abstract data type must hide the implementation from the user, however, all data about the structure must be maintained inside the ADT. Just encapsulating the structure in an ADT is not sufficient. It is also necessary for multiple versions of the structure to be able to coexist. Consequently, we must hide the implementation from the user while being able to store different data.

ADT Implementations:

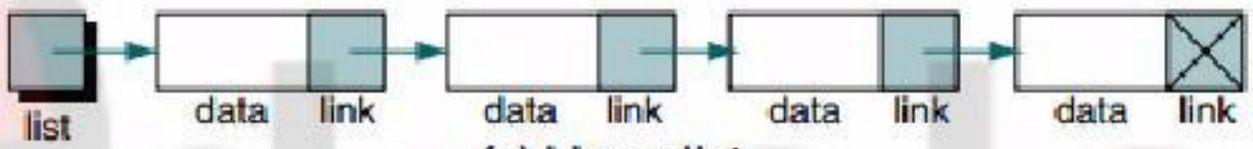
There are two basic structures we can use to implement an ADT list: arrays and linked lists. Array Implementations In an array, the sequentiality of a list is maintained by the order structure of elements in the array (indexes). Although searching an array for an individual element can be very efficient, addition and deletion of elements are complex and inefficient processes. For this reason arrays are seldom used, especially when the list changes frequently. In addition, array implementations of nonlinear lists can become excessively large, especially when there are several successors for each element. Appendix F provides array implementations for two ADTs.

Linked List Implementations:

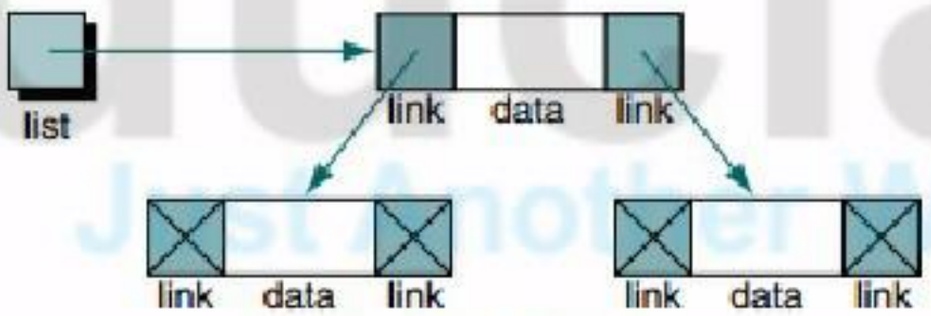
A linked list is an ordered collection of data in which each element contains the location of the next element or elements. In a linked list, each element contains two parts: data and one or more links. The data part holds the application data—the data to be

processed. Links are used to chain the data together. They contain pointers that identify the next element or elements in the list. We can use a linked list to create linear and non-linear structures. In linear linked lists, each element has only zero or one successor. In non-linear linked lists, each element can have zero, one, or more successors. The major advantage of the linked list over the array is that data are easily inserted and deleted. It is not necessary to shift elements of a linked list to make room for a new element or to delete an element. On the other hand, because the elements are no longer physically sequenced, we are limited to sequential searches:1 we cannot use a binary search.2

As below fig. shows a linked list implementation of a linear list. The link in each element, except the last, points to its unique successor; the link in the last element contains a null pointer, indicating the end of the list. As in below Figure part (b) shows a linked list implementation of a non-linear list. An element in a non-linear list can have two or more links. Here each element contains two links, each to one successor. As in below Figure part (c) contains an example of an empty list, linear or non-linear. We define an empty list as a null list pointer.



(a) Linear list



(b) Non-linear list

(c) Empty list

Divide and conquer approach:

The divide-and-conquer approach Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub problems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several sub problems that are similar to the original problem but smaller in size, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

Divide the problem into a number of sub problems that are smaller instances of the same problem.

Conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straightforward manner.

Combine the solutions to the sub problems into the solution for the original problem.

The **merge sort** algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows.

Divide: Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.

Conquer: Sort the two subsequences recursively using merge sort.

Combine: Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order. The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure MERGE.A; p; q; r/, where A is an array and p, q, and r are indices into the array such that p<=q<r.

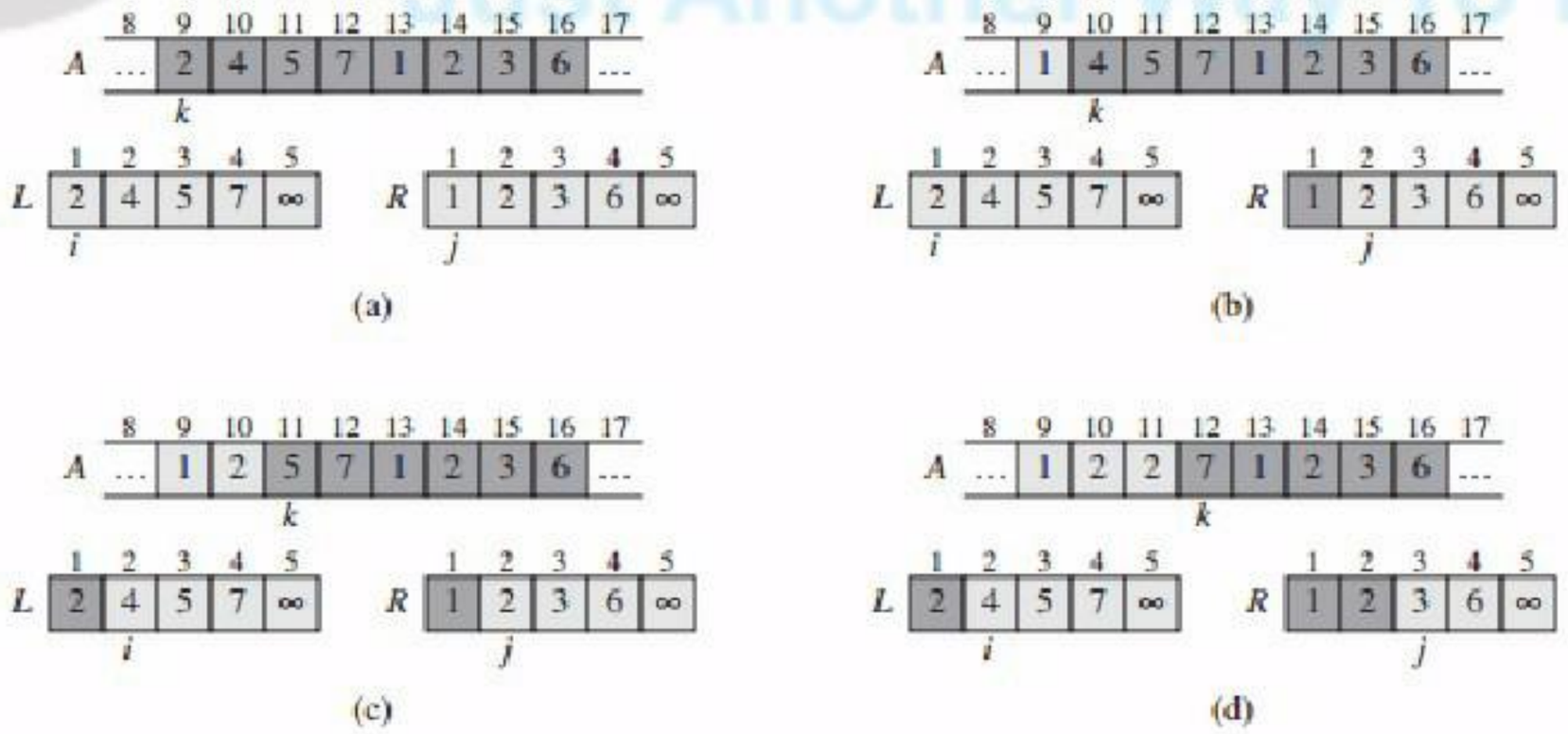
The procedure assumes that the subarrays A[p….q] and A[q+1…r] are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray A[p…r].

Our MERGE procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged, and it works as follows. Returning to our card-playing motif, suppose we have two piles of cards face up on a table. Each pile is sorted, with the smallest cards on top. We wish to merge the two piles into a single sorted output pile, which is to be face down on the table. Our basic step consists of choosing the smaller of the two cards on top of the face-up piles, removing it from its pile (which exposes a new top card), and placing this card face down onto the output pile. We repeat this step until one input pile is empty, at which time we just take the remaining input pile and place it face down onto the output pile. Computationally, each basic step takes constant time, since we are comparing just the two top cards. Since we perform at most n basic steps, merging takes $\Theta(n)$ time.

The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step. We place on the bottom of each pile a ***sentinel*** card, which contains a special value that we use to simplify our code. Here, we use ∞ as the sentinel value, so that whenever a card with ∞ is exposed, it cannot be the smaller card unless both piles have their sentinel cards exposed. But once that happens, all the nonsentinel cards have already been placed onto the output pile. Since we know in advance that exactly $r - p + 1$ cards will be placed onto the output pile, we can stop once we have performed that many basic steps.

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

In detail, the MERGE procedure works as follows. Line 1 computes the length n_1 of the subarray $A[p..q]$, and line 2 computes the length n_2 of the subarray $A[q+1..r]$. We create arrays L and R ("left" and "right"), of lengths $n_1 + 1$ and $n_2 + 1$, respectively, in line 3; the extra position in each array will hold the sentinel. The **for** loop of lines 4–5 copies the subarray $A[p..q]$ into $L[1..n_1]$, and the **for** loop of lines 6–7 copies the subarray $A[q+1..r]$ into $R[1..n_2]$. Lines 8–9 put the sentinels at the ends of the arrays L and R. Lines 10–17, illus-



Analyzing divide-and-conquer algorithms:

When an algorithm contains a recursive call to itself, we can often describe its running time by a recurrence equation or recurrence, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. We can then use

mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm.

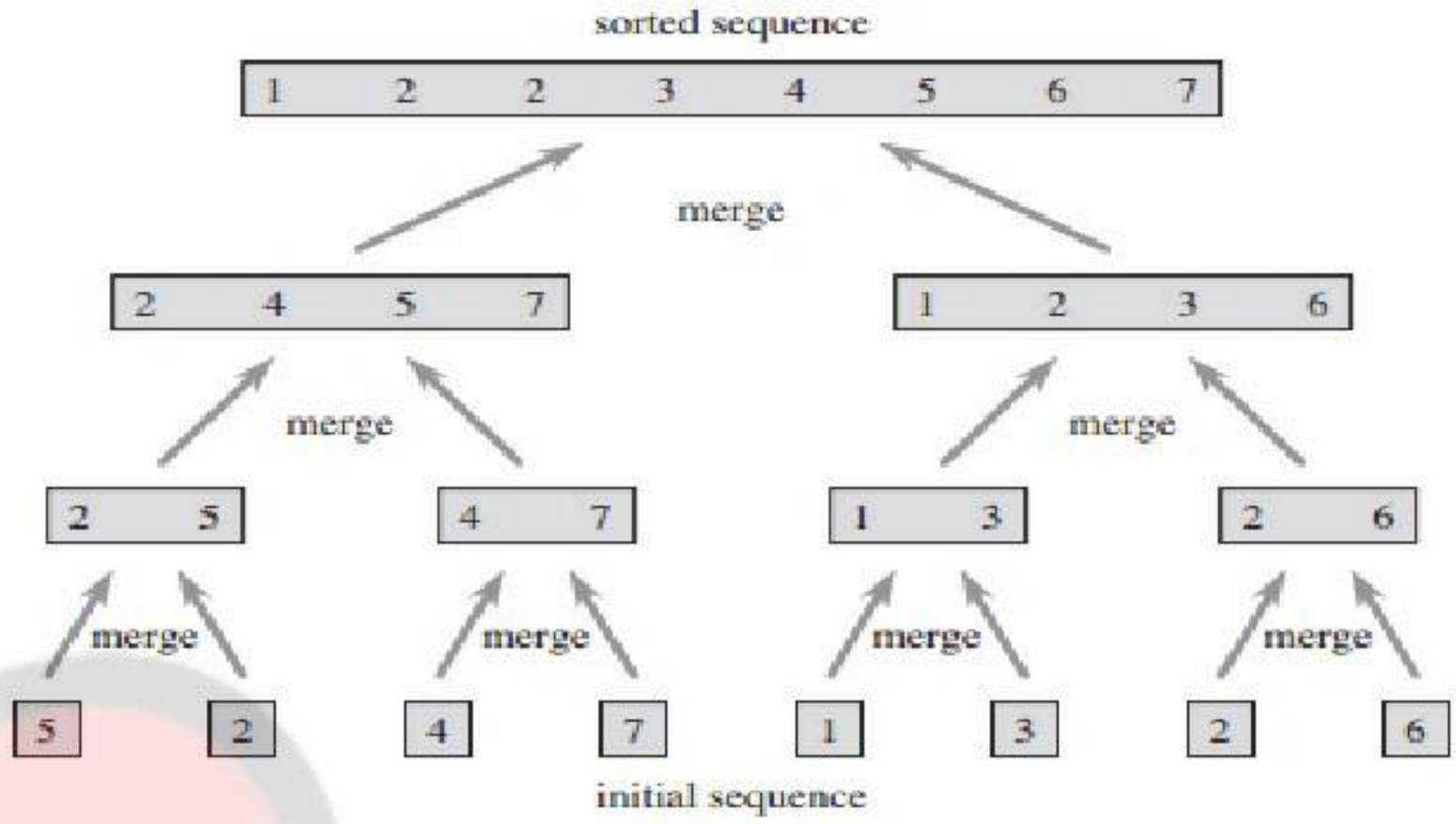


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

A recurrence for the running time of a divide-and-conquer algorithm falls out from the three steps of the basic paradigm. As before, we let $T(n)$ be the running time on a problem of size n. If the problem size is small enough, say $n \leq c$ for some constant c, the straightforward solution takes constant time, which we write as $\Theta(1)$. Suppose that our division of the problem yields a subproblems, each of which is $1/b$ the size of the original. (For merge sort, both a and b are 2, but we shall see many divide-and-conquer algorithms in which $a \neq b$.) It takes time $T(n/b)$ to solve one subproblem of size n/b, and so it takes time $aT(n/b)$ to solve a of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \text{ ,} \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

Dynamic Programming:

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to sub problems. (“Programming” in this context refers to a tabular method, not to writing computer code.) As we saw in Chapters 2 and 4, divide-and-conquer algorithms partition the problem into disjoint sub problems, solve the sub problems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the sub problems overlap—that is, when sub problems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

We typically apply dynamic programming to optimization problems. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution, typically in a bottom-up fashion.

4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4. When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

Unit 2:

Sorting:

Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:

- Roll No.
- Name
- Age
- Class

Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

Sorting Efficiency

There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters. First parameter is the execution time of program, which means time taken for execution of program. Second is the space, which means space taken by the program.

Types of Sorting Techniques

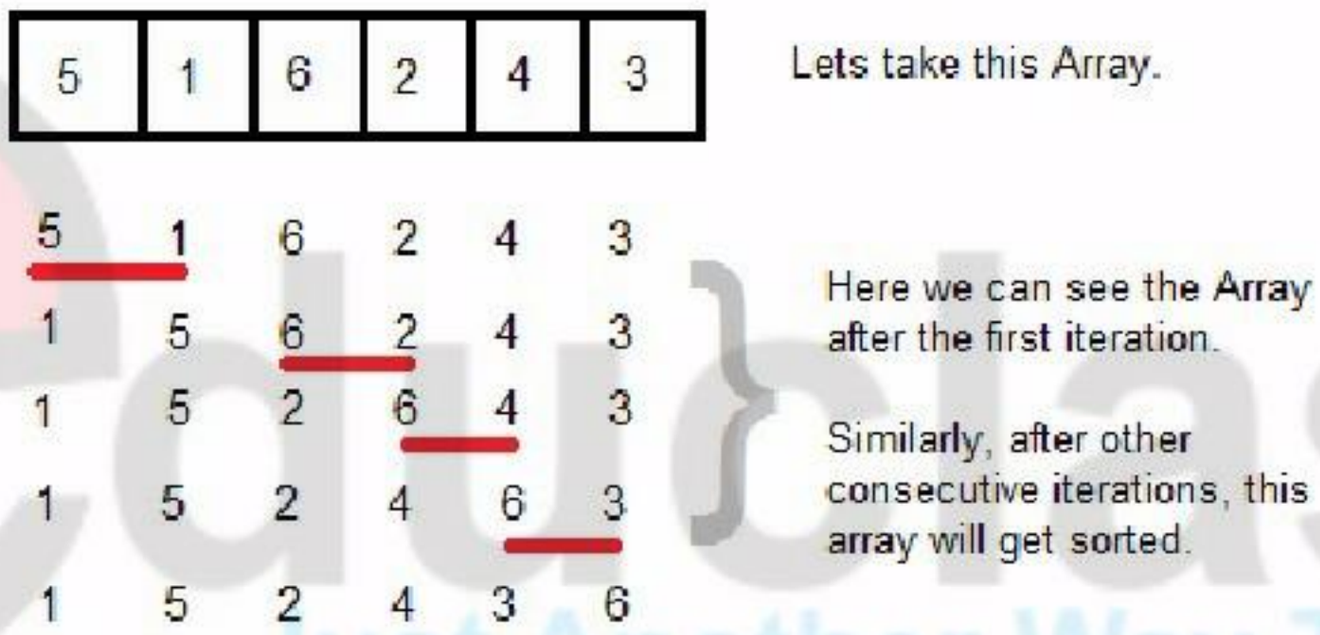
There are many types of Sorting techniques, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next sections.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort

4. Quick Sort
5. Merge Sort
6. Heap Sort

Bubble Sorting

Bubble Sort is an algorithm which is used to sort **N** elements that are given in a memory for eg: an Array with **N** number of elements. Bubble Sort compares all the element one by one and sort them based on their values.
It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.
Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.



Sorting using Bubble Sort Algorithm

Let's consider an array with values {5, 1, 6, 2, 4, 3}

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6; i++)
{
  for(j=0; j<6-i-1; j++)
  {
    if( a[j] > a[j+1])
    {
      temp = a[j];
      a[j] = a[j+1];
      a[j+1] = temp;
    }
```

```
  }
}
//now you can print the sorted array after this
```

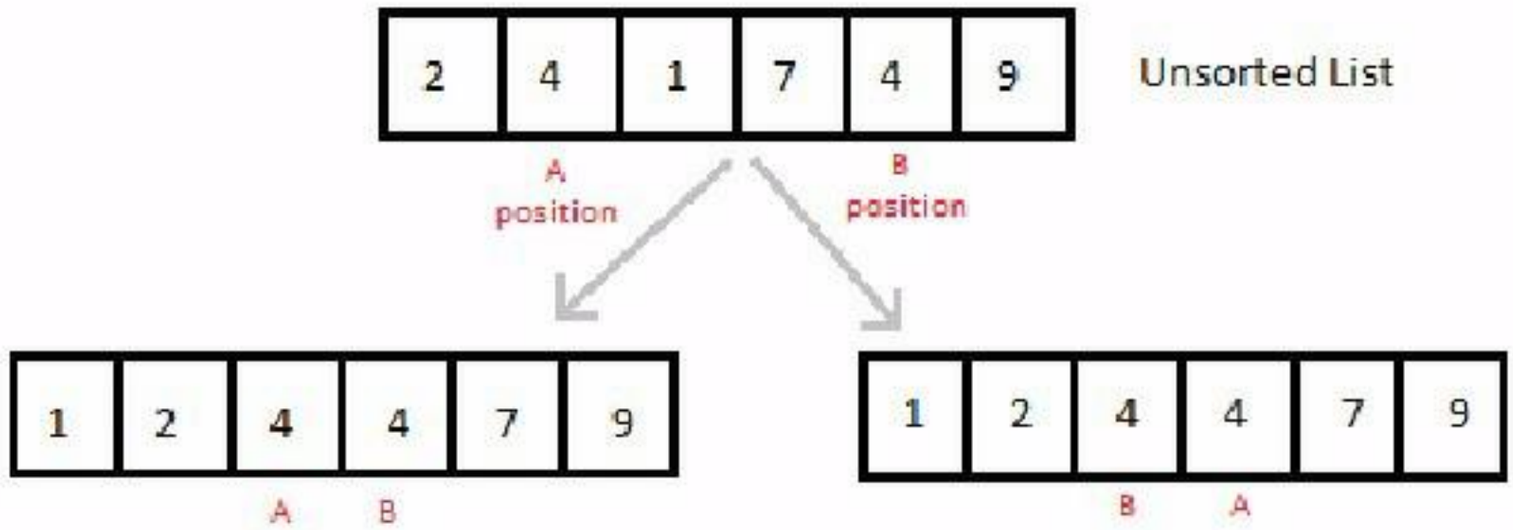
Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort and unsorted array, still the above algorithm isn't efficient and can be enhanced further. Because as per the above logic, the for loop will keep going for six iterations even if the array gets sorted after the second iteration.
Hence we can insert a flag and can keep checking whether swapping of elements is taking place or not. If no swapping is taking place that means the array is sorted and wew can jump out of the for loop.

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6; i++)
{
  int flag = 0;        //taking a flag variable
  for(j=0; j<6-i-1; j++)
  {
    if( a[j] > a[j+1])
    {
      temp = a[j];
      a[j] = a[j+1];
      a[j+1] = temp;
      flag = 1;        //setting flag as 1, if swapping occurs
    }
  }
  if(!flag)            //breaking out of for loop if no swapping takes place
  {
    break;
  }
}
```

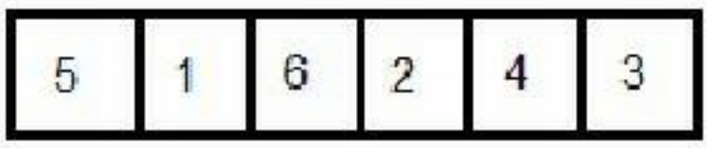
In the above code, if in a complete single cycle of j iteration(inner for loop), no swapping takes place, and flag remains 0, then we will break out of the for loops, because the array has already been sorted.

Insertion Sorting
It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

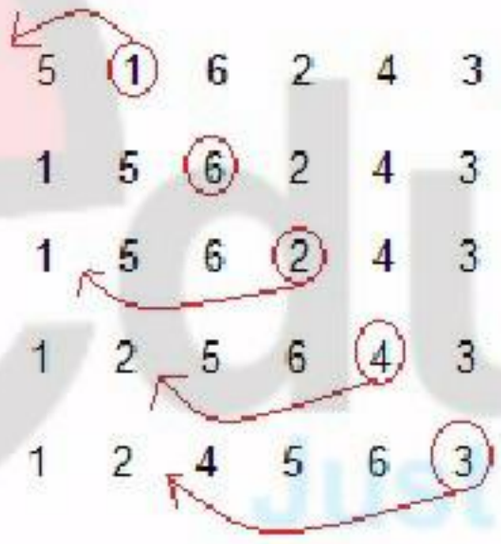
1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, inerstion sort also requires a single additional memory space.
6. It is **Stable**, as it does not change the relative order of elements with equal keys



How Insertion Sorting Works



Lets take this Array.



(Always we start with the second element as key.)

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

Sorting using Insertion Sort Algorithm

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1; i<6; i++)
{
  key = a[i];
  j = i-1;
  while(j>=0 && key < a[j])
  {
```

```
    a[j+1] = a[j];
    j--;
  }
  a[j+1] = key;
}
```

Now lets, understand the above simple insertion sort algorithm. We took an array with 6 integers. We took a variable **key**, in which we put each element of the array, in each pass, starting from the second element, that is **a[1]**.

Then using the while loop, we iterate, until **j** becomes equal to zero or we find an element which is greater than **key**, and then we insert the key at that position.

In the above array, first we pick 1 as key, we compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and is compared with, 6 and 5, and then 2 is placed after 1. And this goes on, until complete array gets sorted.

Insertion Sorting in C++

```
#include <stdlib.h>
#include <iostream.h>

using namespace std;

//member functions declaration
void insertionSort(int arr[], int length);
void printArray(int array[],int size);

int main() {
        int array[5]= {5,4,3,2,1};
        insertionSort(array,5);
        return 0;
}

void insertionSort(int arr[], int length) {
        int i, j ,tmp;
        for (i = 1; i < length; i++) {
                j = i;
                while (j > 0 && arr[j - 1] > arr[j]) {
                        tmp = arr[j];
                        arr[j] = arr[j - 1];
                        arr[j - 1] = tmp;
                        j--;
                }
        printArray(arr,5);
        }
}

void printArray(int array[], int size){
```

```
        cout<< "Sorting tha array using Insertion sort... ";
        int j;
        for (j=0; j < size;j++)
                for (j=0; j < size;j++)
                        cout <<" "<< array[j];
        cout << endl;
}
```

Radix Sort:

QuickSort, MergeSort, HeapSort are comparison based sorting algorithms.
CountSort is not comparison based algorithm. It has the complexity of $O(n+k)$O(n+k), where kk is the maximum element of the input array.
So, if k is $O(n)$,CountSort becomes linear sorting, which is better than comparison based sorting algorithms that have O(nlogn) time complexity. The idea is to extend the CountSort algorithm to get a better time complexity when k goes $O(n^2)$. Here comes the idea of Radix Sort.

Algorithm:
For each digit ii where ii varies from the least significant digit to the most significant digit of a number
    Sort input array using countsort algorithm according to ith digit.

We used count sort because it is a stable sort.

Example: Assume the input array is:
10,21,17,34,44,11,654,123
Based on the algorithm, we will sort the input array according to the **one's digit** (least significant digit).
0: 10
1: 21 11
2:
3: 123
4: 34 44 654
5:
6:
7: 17
8:
9:

So, the array becomes 10,21,11,123,24,44,654,17
Now, we'll sort according to the **ten's digit**:
0:
1: 10 11 17
2: 21 123
3: 34
4: 44
5: 654
6:
7:
8:
9:

Now, the array becomes : 10,11,17,21,123,34,44,654
Finally , we sort according to the **hundred's digit** (most significant digit):
0: 010 011 017 021 034 044
1: 123
2:
3:
4:
5:
6: 654
7:
8:
9:

The array becomes : 10,11,17,21,34,44,123,654 which is sorted. This is how our algorithm works.

implementation:

```
void countsort(int arr[],int n,int place)
{
	int i,freq[range]={0};		//range for integers is 10 as digits range from 0-9
	int output[n];
	for(i=0;i<n;i++)
		freq[(arr[i]/place)%range]++;
	for(i=1;i<range;i++)
		freq[i]+=freq[i-1];
	for(i=n-1;i>=0;i--)
	{
		output[freq[(arr[i]/place)%range]-1]=arr[i];
		freq[(arr[i]/place)%range]--;
	}
```

```
        for(i=0;i<n;i++)
                arr[i]=output[i];
}
void radixsort(ll arr[],int n,int maxx)          //maxx is the maximum element in the array
{
        int mul=1;
        while(maxx)
        {
                countsort(arr,n,mul);
                mul*=10;
                maxx/=10;
        }
}
```

Complexity Analysis:
The complexity is O((n+b)∗logb(maxx))O((n+b)∗logb(maxx)) where bb is the base for representing numbers and maxx is the maximum element of the input array. This is clearly visible as we make (n+b)(n+b) iterations logb(maxx)logb(maxx) times (number of digits in the maximum element) . If maxx≤ncmaxx≤nc,then the complexity can be written as O(n∗logb(n))O(n∗logb(n)).

Advantages :
1. Fast when the keys are short i.e. when the range of the array elements is less.
2. Used in suffix array construction algorithms like Manber's algorithm and DC3 algorithm.

Disadvantages:
1. Since Radix Sort depends on digits or letters, Radix Sort is much less flexible than other sorts. Hence, for every different type of data it needs to be rewritten.
2. The constant for Radix sort is greater compared to other sorting algorithms.
3. It takes more space compared to Quicksort which is in place sorting.

The Radix Sort algorithm is an important sorting algorithm that is integral to suffix -array construction algorithms. It is also useful on parallel machines.

Quick Sort:

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule

of **Divide and Conquer** (also called *partition-exchange sort*). This algorithm divides the list into three main parts:

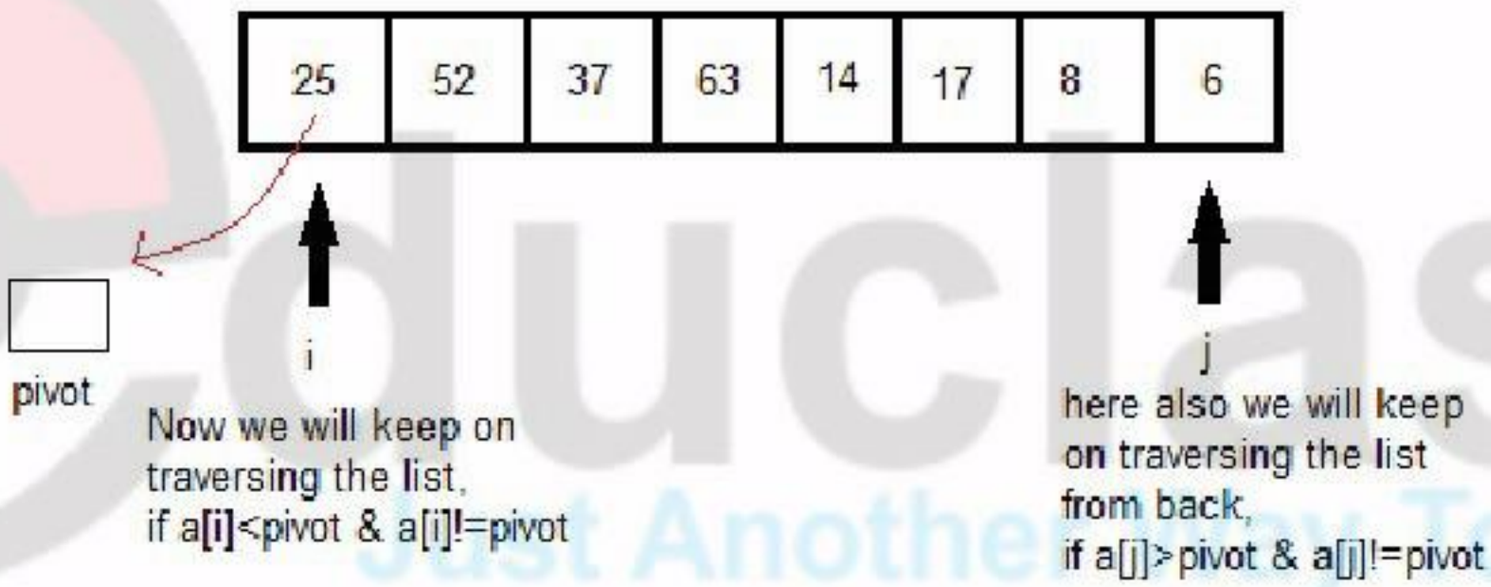
1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken **25** as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 **25** 63 37 52

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

How Quick Sorting Works



DIVIDE AND CONQUER - QUICK SORT

Sorting using Quick Sort Algorithm

```
/*  a[] is the array, p is starting index, that is 0,
and r is the last index of array.  */
```

```
void quicksort(int a[], int p, int r)
{
  if(p < r)
  {
    int q;
    q = partition(a, p, r);
    quicksort(a, p, q);
    quicksort(a, q+1, r);
  }
}

int partition(int a[], int p, int r)
{
  int i, j, pivot, temp;
  pivot = a[p];
  i = p;
  j = r;
  while(1)
  {
   while(a[i] < pivot && a[i] != pivot)
   i++;
   while(a[j] > pivot && a[j] != pivot)
   j--;
   if(i < j)
   {
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
   }
   else
   {
    return j;
   }
  }
}
```

Complexity Analysis of Quick Sort

Worst Case Time Complexity : $O(n^2)$
Best Case Time Complexity : $O(n \log n)$
Average Time Complexity : $O(n \log n)$
Space Complexity : $O(n \log n)$

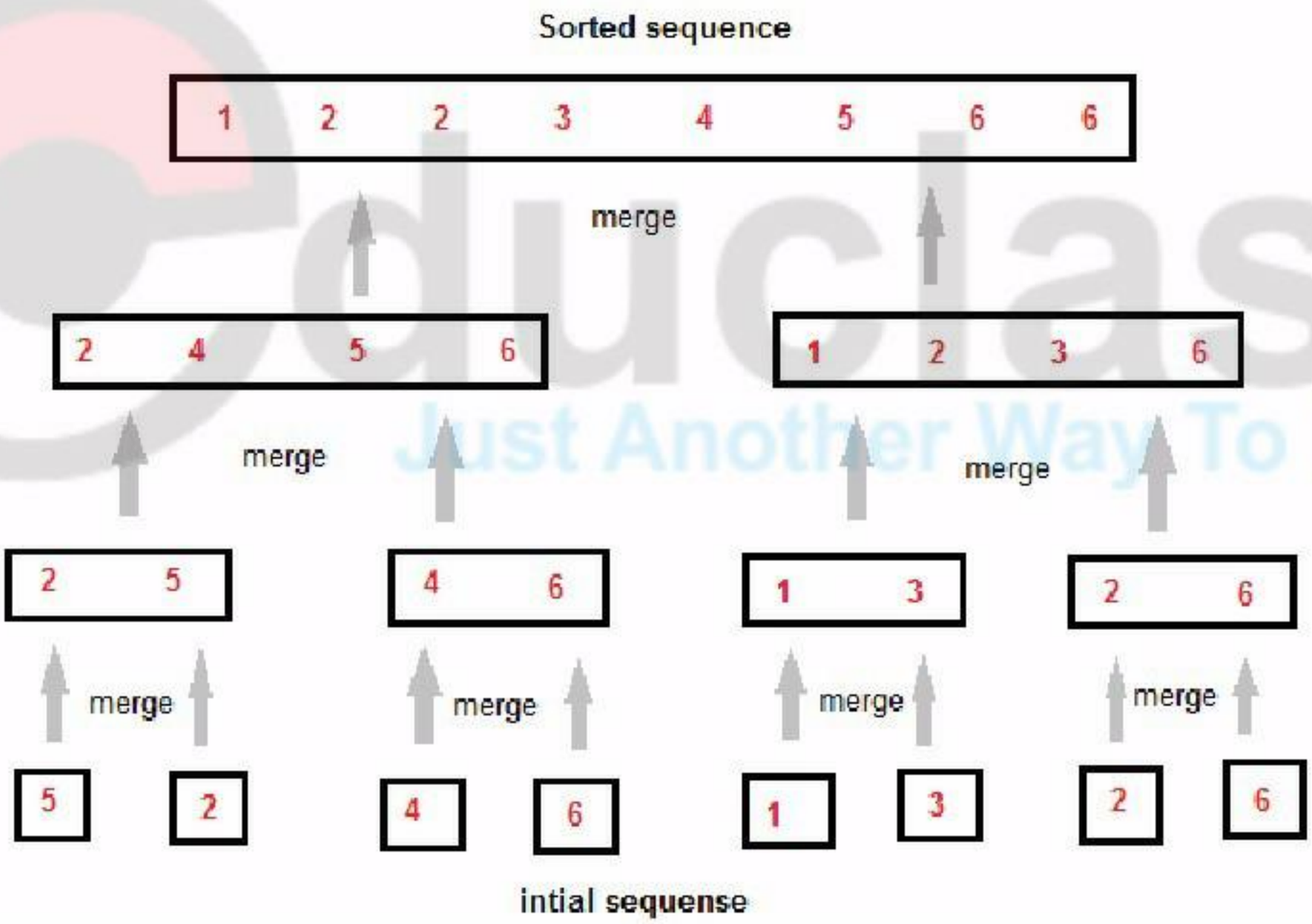
- Space required by quick sort is very less, only $O(n \log n)$ additional space is required.

- Quick sort is not a stable sorting technique, so it might change the occurrence of two similar elements in the list while sorting.

Merge Sort:

Merge Sort follows the rule of **Divide and Conquer**. But it doesn't divides the list into two halves. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at lasts one sorted list is produced. Merge Sort is quite fast, and has a time complexity of **O(n log n)**. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

How Merge Sort Works



Like we can see in the above example, merge sort first breaks the unsorted list into sorted sublists, and then keep merging these sublists, to finlly get the complete sorted list.

Sorting using Merge Sort Algorithm

```
/*  a[] is the array, p is starting index, that is 0,
and r is the last index of array.  */

Lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

void mergesort(int a[], int p, int r)
{
  int q;
  if(p < r)
  {
    q = floor( (p+r) / 2);
    mergesort(a, p, q);
    mergesort(a, q+1, r);
    merge(a, p, q, r);
  }
}

void merge(int a[], int p, int q, int r)
{
  int b[5];     //same size of a[]
  int i, j, k;
  k = 0;
  i = p;
  j = q+1;
  while(i <= q && j <= r)
  {
    if(a[i] < a[j])
    {
      b[k++] = a[i++];       // same as b[k]=a[i]; k++; i++;
    }
    else
    {
      b[k++] = a[j++];
    }
  }

while(i <= q)
{
  b[k++] = a[i++];
}

while(j <= r)
{
  b[k++] = a[j++];
}
```

```
  for(i=r; i >= p; i--)
  {
    a[i] = b[--k];        // copying back the sorted list to a[]
  }

}
```

Complexity Analysis of Merge Sort

Worst Case Time Complexity : O(n log n)
Best Case Time Complexity : O(n log n)
Average Time Complexity : O(n log n)
Space Complexity : O(n)

- Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.
- It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists.
- It is the best Sorting technique for sorting **Linked Lists**.

Heap Sort:

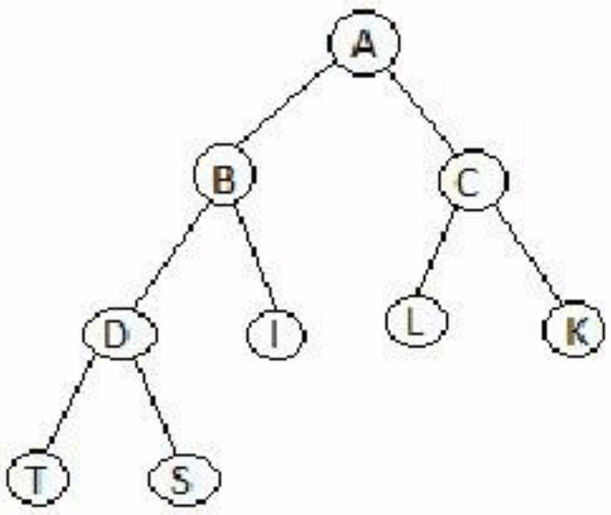
Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts :

- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

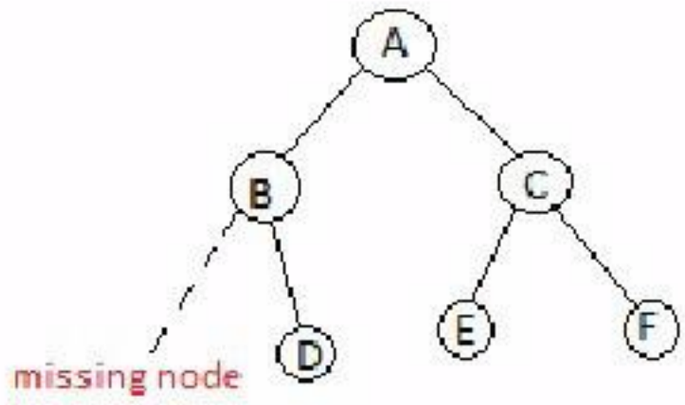
What is a Heap?

Heap is a special tree-based data structure,that satisfies the following special heap properties:

1. **Shape Property:** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

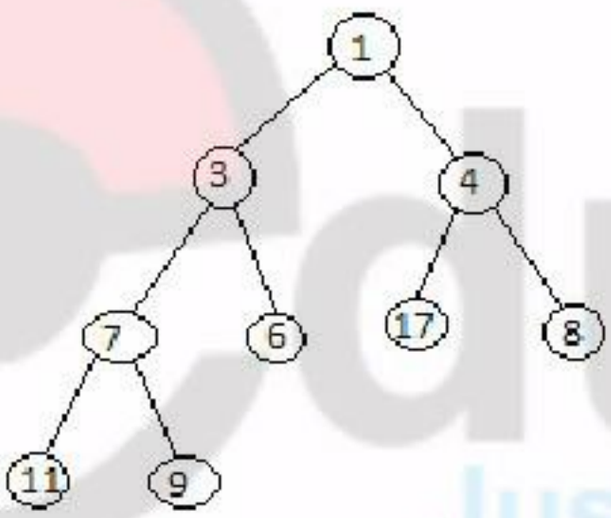


Complete Binary Tree



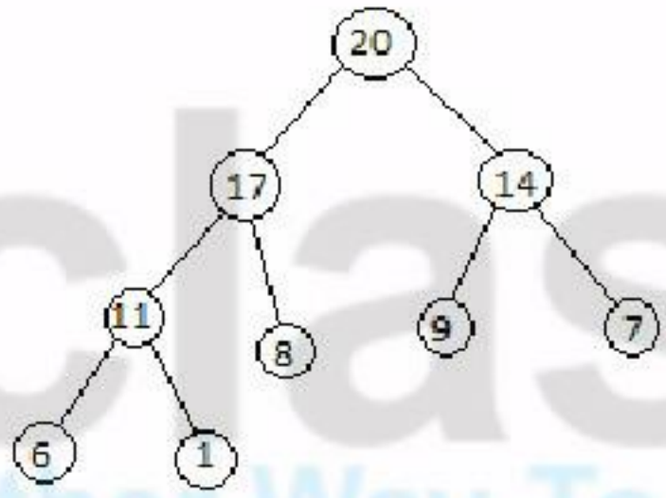
In-Complete Binary Tree

2. **Heap Property:** All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their children, heap is called a **Max-Heap**, and if the parent nodes are smalled than their child nodes, heap is called **Min-Heap**.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

How Heap Sort Works

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining

elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.
In the below algorithm, initially **heapsort()** function is called, which calls **buildheap()** to build heap, which inturn uses **satisfyheap()** to build the heap.

Sorting using Heap Sort Algorithm

```
/*  Below program is written in C++ language  */

void heapsort(int[], int);
void buildheap(int [], int);
void satisfyheap(int [], int, int);

void main()
{
  int a[10], i, size;
  cout << "Enter size of list";    // less than 10, because max size of array is 10
  cin >> size;
  cout << "Enter" << size << "elements";
  for( i=0; i < size; i++)
  {
    cin >> a[i];
  }
  heapsort(a, size);
  getch();
}

void heapsort(int a[], int length)
{
  buildheap(a, length);
  int heapsize, i, temp;
  heapsize = length - 1;
  for( i=heapsize; i >= 0; i--)
  {
    temp = a[0];
    a[0] = a[heapsize];
    a[heapsize] = temp;
    heapsize--;
    satisfyheap(a, 0, heapsize);
  }
  for( i=0; i < length; i++)
  {
    cout << "\t" << a[i];
  }
}
```

```
void buildheap(int a[], int length)
{
  int i, heapsize;
  heapsize = length - 1;
  for( i=(length/2); i >= 0; i--)
  {
    satisfyheap(a, i, heapsize);
  }
}

void satisfyheap(int a[], int i, int heapsize)
{
  int l, r, largest, temp;
  l = 2*i;
  r = 2*i + 1;
  if(l <= heapsize && a[l] > a[i])
  {
    largest = l;
  }
  else
  {
    largest = i;
  }
  if( r <= heapsize && a[r] > a[largest])
  {
    largest = r;
  }
  if(largest != i)
  {
    temp = a[i];
    a[i] = a[largest];
    a[largest] = temp;
    satisfyheap(a, largest, heapsize);
  }
}
```

Complexity Analysis of Heap Sort

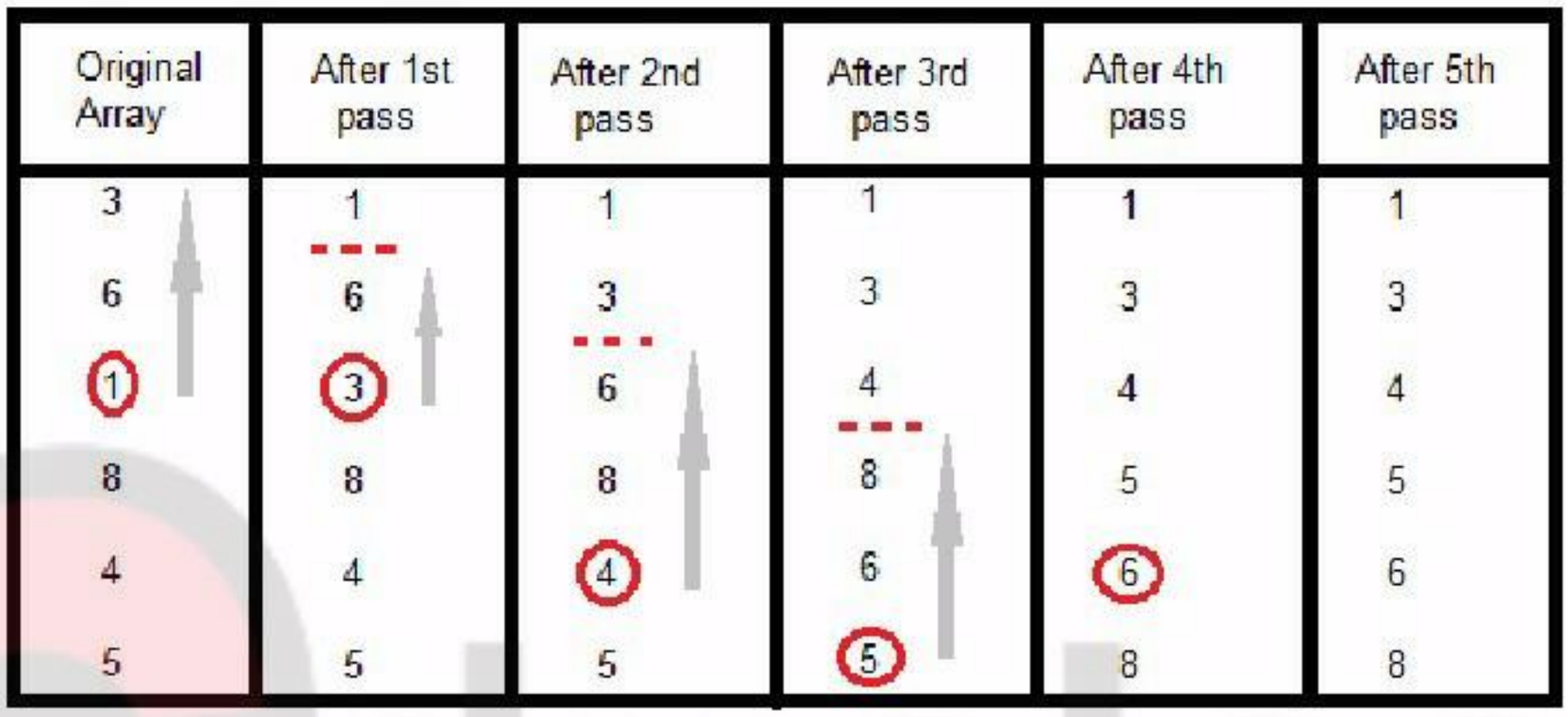
Worst Case Time Complexity : O(n log n)
Best Case Time Complexity : O(n log n)
Average Time Complexity : O(n log n)
Space Complexity : O(n)

- Heap sort is not a Stable sort, and requires a constant space for sorting a list.
- Heap Sort is very fast and is widely used for sorting.

Selection Sorting:

Selection sorting is conceptually the most simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

How Selection Sorting Works



In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 nad 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted.

Sorting using Selection Sort Algorithm

```
void selectionSort(int a[], int size)
{
  int i, j, min, temp;
  for(i=0; i < size-1; i++ )
  {
    min = i;   //setting min as i
    for(j=i+1; j < size; j++)
    {
      if(a[j] < a[min])   //if element at j is less than element at min position
      {
       min = j;    //then set min as j
      }
    }
```

```
    temp = a[i];
    a[i] = a[min];
    a[min] = temp;
  }
}
```

Complexity Analysis of Selection Sorting

Worst Case Time Complexity : $O(n^2)$
Best Case Time Complexity : $O(n^2)$
Average Time Complexity : $O(n^2)$
Space Complexity : $O(1)$

Shell Sort:

In this sorting, we compare elements that are distance apart rather than adjacent.

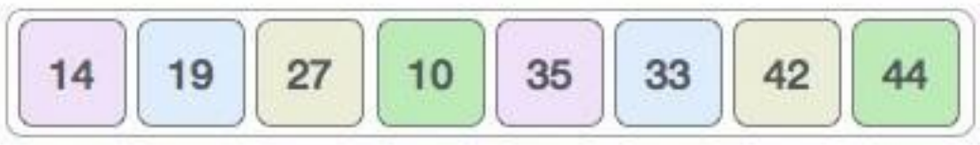
We calculate "Gap" for each pass, and then select the elements towards the right of gap.

How Shell Sort Works?

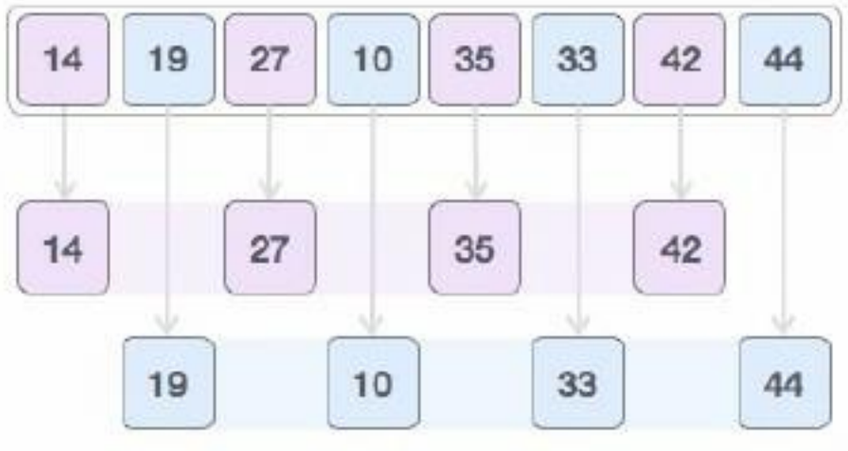
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding.

we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 14}

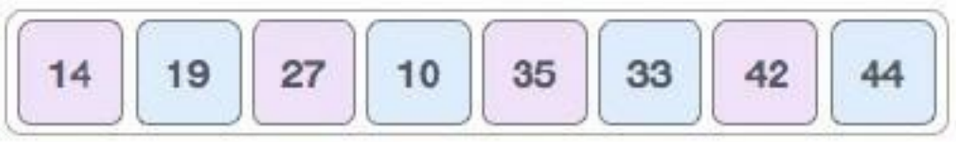
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this –



Then, we take interval of 2 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}

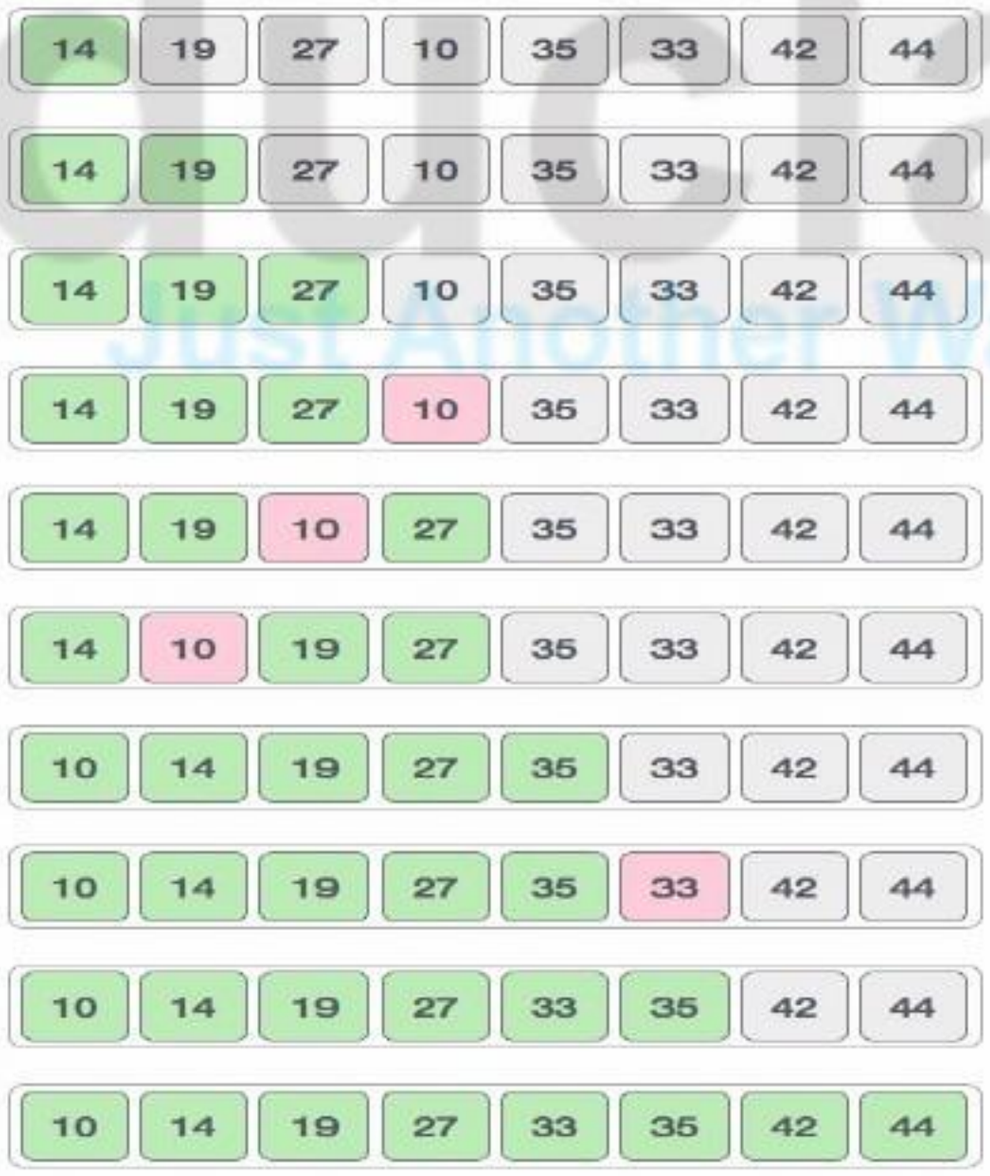


We compare and swap the values, if required, in the original array. After this step, the array should look like this –



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction –



We see that it required only four swaps to sort the rest of the array.

Algorithm

Following is the algorithm for shell sort.

Step 1 − Initialize the value of *h*
Step 2 − Divide the list into smaller sub-list of equal interval *h*
Step 3 − Sort these sub-lists using **insertion sort**
Step 3 − Repeat until complete list is sorted