



COMPUTER SCIENCE AND ENGINEERING

DATA STRUCTURES

Name of the Faculty

Dr. M. Lakshmi Prasad

Professor, CSE.

UNIT –I

INTRODUCTION TO DATA STRUCTURES, SEARCHING AND SORTING

Contents



- Introduction to Data Structures
- Classification and Operations on Data Structures
- Preliminaries of Algorithm
- Algorithm Analysis and Complexity
- Recursive Algorithms
- Searching Techniques - Linear, Binary search
- Sorting Techniques- Bubble, Selection, Insertion

Data Structure



Data Structure is a way of storing data in a computer so that it can be used efficiently and it will allow the most efficient algorithm to be used.

A **data structure** is representation of the logical relation ship existing between individual elements of data.

A data structure should be seen as a logical concept that must address two fundamental concerns.

1. First, how the data will be stored and
2. Second, what operations will be performed on it.

Overview of Data Structures



The structure of input and output data can be used to derive the structure of a program. Data structure affects the design of both structural and functional aspects of a program.

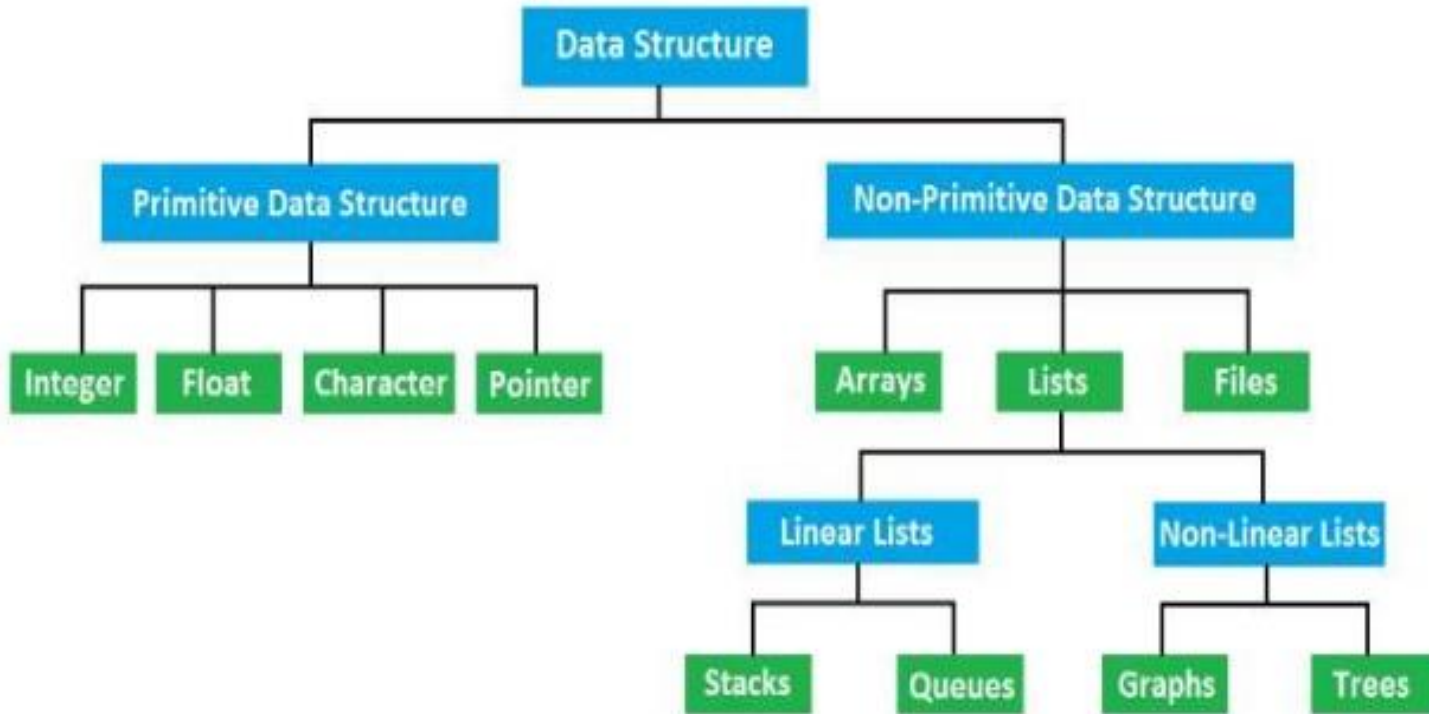
Algorithm + Data structure=Program.

Classification of data structure

Data structures are normally divided into two categories:

- Primitive data structures
- Non-primitive data structures

Classification of data structure



Primitive data structure



The primitive data structures are known as basic data structures. These data structures are directly operated upon by the machine instructions. Normally, primitive data structures have different representation on different computers.

Example of primitive data structure :

- **integer**
- **float**
- **character**
- **pointer**

Non-Primitive data structure



The non-primitive data structures are highly developed complex data structures. Basically, these are developed from the primitive data structure. The non-primitive data structure is responsible for organizing the group of homogeneous and heterogeneous data elements.

Example of Non-primitive data structure :

- **Arrays**
- **Lists**
- **Trees**
- **Graphs**

Linear and Non-linear Data Structures



- **Linear Data Structures:** Linear data structures can be constructed as a continuous arrangement of data elements in the memory. It can be constructed by using array data type. In the linear data structure the relationship of adjacency is maintained between the data elements.
- **Non-linear data structures:** Non-linear data structure can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear data structure the relationship of adjacency is not maintained between the data items.

Operations on Data Structures



- Add an element
- Delete an element
- Traverse
- Sort the elements
- Search for a data element
- Update an element
- Merging

Algorithm



Definition:

An algorithm is a series of step-by-step instructions that aim to solve a particular problem.

- The word algorithm originates from the Arabic word Algorism which is linked to the name of the Arabic Mathematician “Abu Abdullah Muhammad ibn Musa Al-Khawarizmi”
- Khawarizmi is considered to the first algorithm designer for adding numbers.

Structure of an Algorithm



- An algorithm has the following structure:
 - Input Step
 - Assignment Step
 - Decision Step
 - Repetitive Step
 - Output Step

Properties of an Algorithm



- **Finiteness:** The algorithm must always terminate after a finite number of steps.
- **Definiteness:** Each step must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
- **Input:** An algorithm has zero or more inputs, taken from a specified set of objects.
- **Output:** An algorithm has one or more outputs, which have a specified relation to the inputs.
- **Effectiveness:** It must be possible to perform each step of the algorithm correctly and in a finite amount of time. That is, its steps must be basic enough so that, for example, someone using a pencil and a paper could carry out exactly, and in a finite amount of time.

Algorithm Analysis and Complexity



- The performance of algorithms can be measured on the scales of Time and Space
- The Time Complexity of an algorithm or a program is a function of the running time of the algorithm or a program.
- The Space Complexity of an algorithm or a program is a function of the space needed by the algorithm or program to run to completion.

Performance Analysis



The performance of the program based on efficient usage of primary and secondary memory as well as the execution time of the program. The performance evaluation of a program is done using:

- Performance analysis (machine independent)
- Performance measurement (machine dependent)

Definition : The process of estimating time and space consumed by the program that is independent of the machine is called performance analysis. So the efficiency of the program depends on two factors:

- Space complexity
- Time complexity

Space complexity/efficiency



Space complexity is the **total amount of memory space used by algorithm/program including the space of input values for execution.**

- So to find space complexity, it is enough to calculate the space occupied by the variables used in an algorithm/program.
- Auxiliary space is just a temporary or extra space and it is not the same as space-complexity.
- The space requirement S of a program P denoted by $S(P)$ is given by:

$$S(P) = c + Sp(I)$$

- c is fixed space requirement
- I variable space requirement

Space Complexity



In simpler terms,

Space Complexity = Auxiliary space + Space use by input values

Important Note: The best algorithm/program should have the least space-complexity. The lesser the space used, the faster it executes.

How to calculate Space Complexity of an Algorithm?



Let us understand the Space-Complexity calculation through examples.

Example - 1:

```
#include<stdio.h>
int main()
{
    int a = 5, b = 5, c;
    c = a + b;
    printf("%d", c);
}
```

How to calculate Space Complexity of an Algorithm?



In the above program, 3 integer variables are used. The size of the integer data type is 2 or 4 bytes which depends on the compiler. Now, let's assume the size as 4 bytes. So, the total space occupied by the above-given program is $4 * 3 = 12$ bytes. Since no additional variables are used, no extra space is required.

Hence, **space complexity for the above-given program is $O(1)$, or constant.**

How to calculate Space Complexity of an Algorithm?

Example -2:

```
#include <stdio.h>
int main()
{
    int n, i, sum = 0;
    scanf("%d", &n);
    int arr[n];
    for(i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
        sum = sum + arr[i];
    }
    printf("%d", sum);
}
```

How to calculate Space Complexity of an Algorithm?

In the above-given code, the array consists of n integer elements. So, the space occupied by the array is $4 * n$. Also we have integer variables such as n , i and sum . Assuming 4 bytes for each variable, the total space occupied by the program is $4n + 12$ bytes.

Since the highest order of n in the equation $4n + 12$ is n , so **the space complexity is $O(n)$ or linear.**

Summary

Big O Notation	Space Complexity details
$O(1)$	Constant Space Complexity occurs when the program doesn't contain any loops, recursive functions or call to any other functions.
$O(n)$	Linear space complexity occurs when the program contains any loops.

Time complexity

- The time complexity of an algorithm is the amount of computer time required to run the program till the completion.
- Normally, the time complexity is calculated using step count. i.e., how many times a program step is executed.

Total Frequency Count of Program

Segment A

• Program Statements	• Frequency Count
.....	
$x = x + 2$	1
.....	
Total Frequency Count	1

Time Complexity of Program Segment A is **$O(1)$** .

Total Frequency Count of Program Segment B

- Program Statements

.....

for i = 1 to n do

 x = x + 2;

end

.....

Total Frequency Count

- Frequency Count

(n+1)

n

n

.....

3n+1

Time Complexity of Program Segment B is **O(n)**.

Total Frequency Count of Program Segment C



- Program Statements

.....

```
for i= 1 to n do
  for j = 1 to n do
    x = x+ 2;
  end
end
end
```

.....

Total Frequency Count

- Frequency Count

(n+1)

(n+1)n

N^2

N^2

N

.....

...

$3n^2 + 3n + 1$

Time Complexity of Program Segment is **$O(n^2)$** .

Asymptotic Notations

- **Big oh(O):** $f(n) = O(g(n))$ (read as f of n is big oh of g of n), if there exists a positive integer n_0 and a positive number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$.

$f(n)$	$g(n)$	
$16n^3 + 45n^2 + 12n$	n^3	$f(n) = O(n^3)$
$34n - 40$	n	$f(n) = O(n)$
50	1	$f(n) = O(1)$

- Here $g(n)$ is the upper bound of the function $f(n)$.

Asymptotic Notations

- **Omega(Ω):** $f(n) = \Omega(g(n))$ (read as f of n is omega of g of n), if there exists a positive integer n_0 and a positive number c such that

$$|f(n)| \geq c |g(n)| \text{ for all } n \geq n_0.$$

$f(n)$	$g(n)$	
$16n^3+8n^2+2$	n^3	$f(n) = \Omega(n^3)$
$24n+9$	n	$f(n) = \Omega(n)$

- Here $g(n)$ is the lower bound of the function $f(n)$.

Asymptotic Notations

- **Theta(Θ):** $f(n) = \Theta(g(n))$ (read as f of n is θ of g of n), if there exists a positive integer n_0 and two positive constants c_1 and c_2 such that $c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)|$ for all $n \geq n_0$.

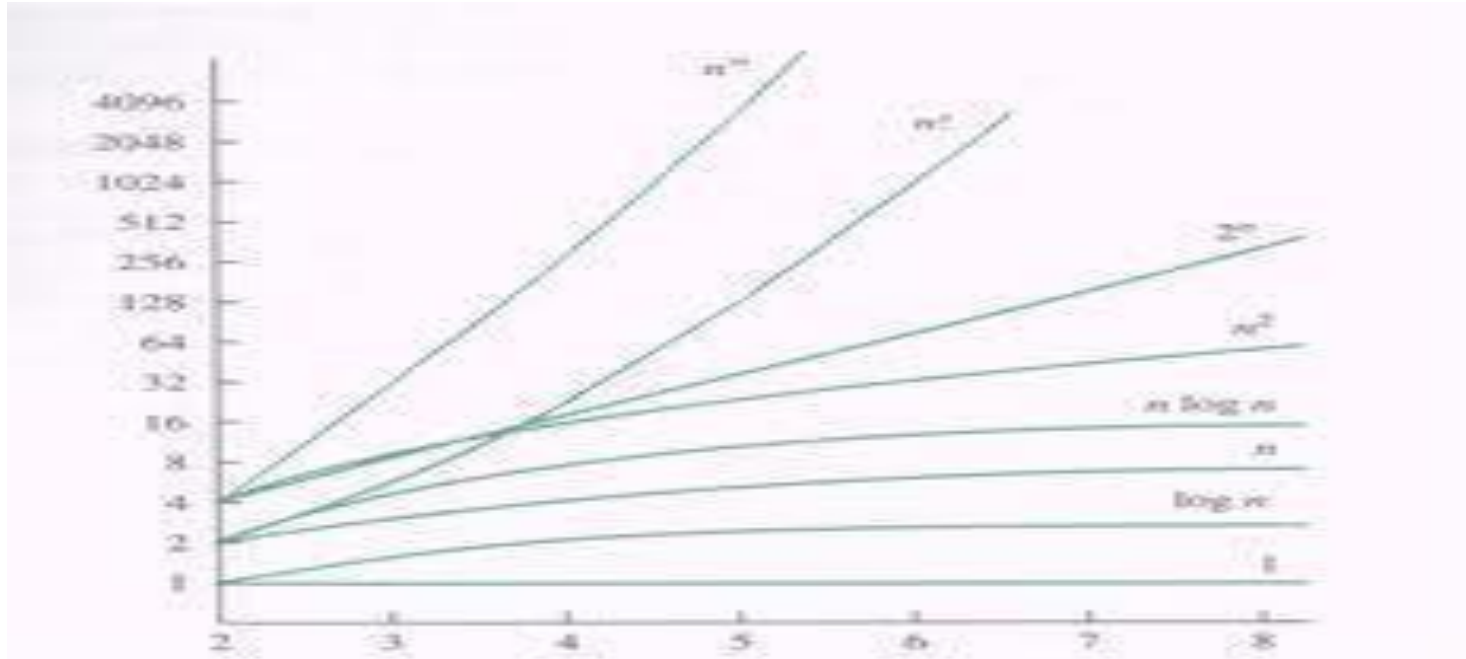
$f(n)$	$g(n)$	
$16n^3 + 30n^2 - 90$	n^2	$f(n) = \Theta(n^2)$

- The function $g(n)$ is both an upper bound and a lower bound for the function $f(n)$ for all values of n , $n \geq n_0$

Time Complexity

Complexity	Notation	Description
Constant	$O(1)$	Constant number of operations, not depending on the input data size.
Logarithmic	$O(\log n)$	Number of operations proportional of $\log(n)$ where n is the size of the input data.
Linear	$O(n)$	Number of operations proportional to the input data size.
Quadratic	$O(n^2)$	Number of operations proportional to the square of the size of the input data.
Cubic	$O(n^3)$	Number of operations proportional to the cube of the size of the input data.
Exponential	$O(2^n)$	Exponential number of operations, fast growing.
	$O(k^n)$	
	$O(n!)$	

Time Complexities of various Algorithms



Recursion

Definition:

The process of **recursion** involves solving a problem by turning it into smaller varieties of itself.

Types of Recursions

- Direct Recursion
 - a function calls itself from within itself.
- Indirect Recursion
 - two functions call one another mutually.

Differences Between Recursion and Iteration

- A function is said to be recursive if it calls itself again and again within its body whereas iterative functions are loop based imperative functions.
- Recursion uses stack whereas iteration does not use stack.
- Recursion uses more memory than iteration as its concept is based on stacks.
- Recursion is comparatively slower than iteration due to overhead condition of maintaining stacks.
- Recursion makes code smaller and iteration makes code longer.
- Iteration terminates when the loop-continuation condition fails whereas recursion terminates when a base case is recognized.
- Infinite recursion can crash the system whereas infinite looping uses CPU cycles repeatedly.
- Recursion uses selection structure whereas iteration uses repetition structure.

Types of Recursion

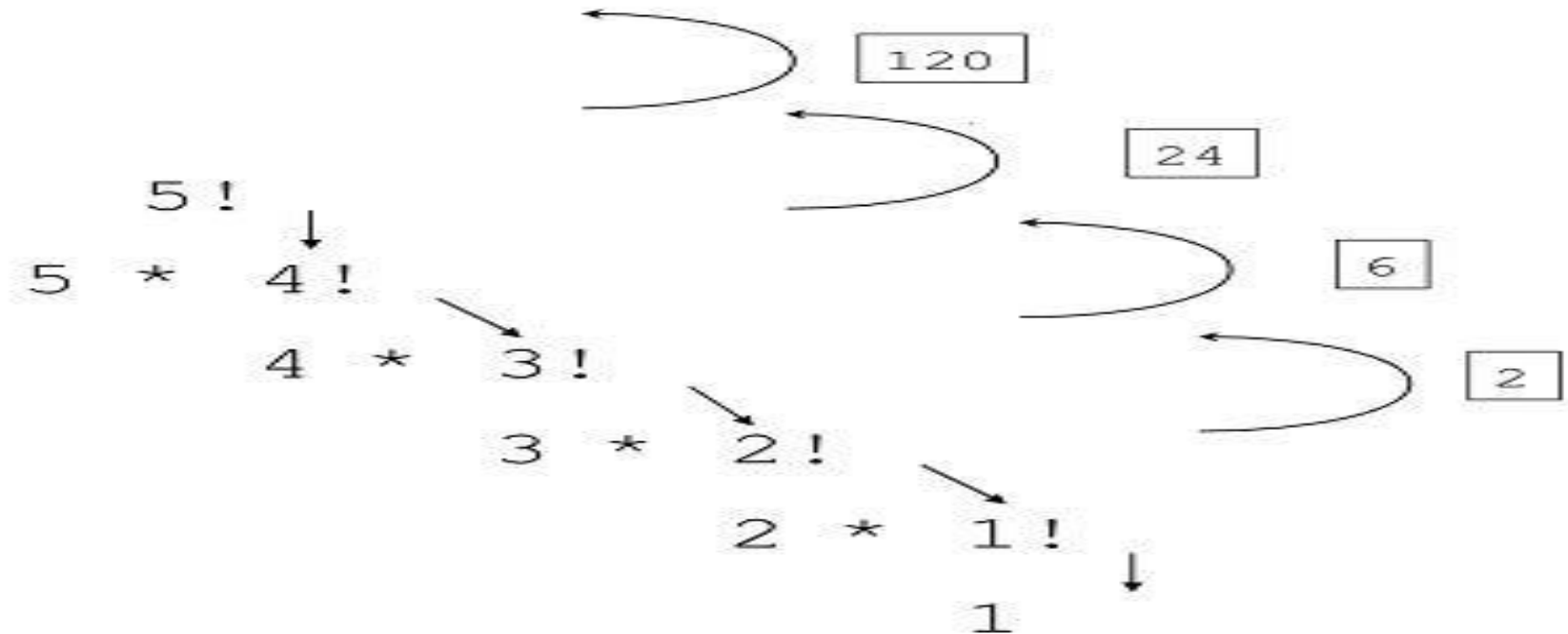
Recursion may be further categorized as:

- Linear Recursion
- Binary Recursion
- Multiple Recursion

Linear Recursion

It is the most common type of Recursion in which function calls itself repeatedly until base condition [termination case] is reached. Once the base case is reached the results are return to the caller function. If a recursive function is called only once then it is called a linear recursion.

Linear Recursion



Binary Recursion

Some recursive functions don't just have one call to themselves; they have two (or more). Functions with two recursive calls are referred to as binary recursive functions.

Example1: The Fibonacci function fib provides a classic example of binary recursion. The Fibonacci numbers can be defined by the rule:

$$\begin{aligned}\text{fib}(n) &= 0 \text{ if } n \text{ is } 0, \\ &= 1 \text{ if } n \text{ is } 1, \\ &= \text{fib}(n-1) + \text{fib}(n-2) \text{ otherwise}\end{aligned}$$

Binary Recursion

For example, the first seven Fibonacci numbers are

$$\text{Fib}(0) = 0$$

$$\text{Fib}(1) = 1$$

$$\text{Fib}(2) = \text{Fib}(1) + \text{Fib}(0) = 1$$

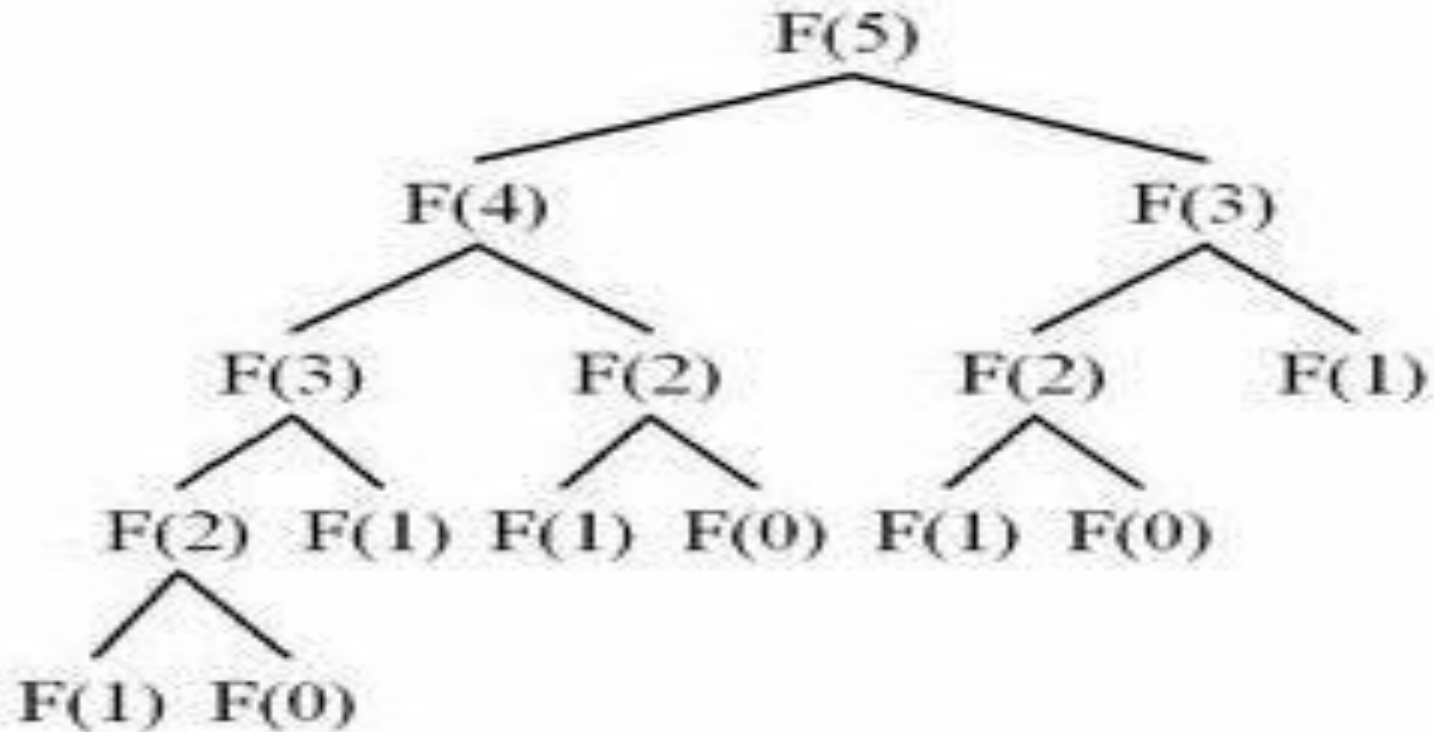
$$\text{Fib}(3) = \text{Fib}(2) + \text{Fib}(1) = 2$$

$$\text{Fib}(4) = \text{Fib}(3) + \text{Fib}(2) = 3$$

$$\text{Fib}(5) = \text{Fib}(4) + \text{Fib}(3) = 5$$

$$\text{Fib}(6) = \text{Fib}(5) + \text{Fib}(4) = 8$$

Binary Recursion



Tail Recursion

A recursive function is tail recursive when recursive call is the last thing executed by the function.

- As such, tail recursive functions can often be easily implemented in an iterative manner; by taking out the recursive call and replacing it with a loop, the same effect can generally be achieved.
- In fact, a good compiler can recognize tail recursion and convert it to iteration in order to optimize the performance of the code.

Tail Recursion

Example:

```
void print(int n)
{
    if (n < 0) return;
    cout << " " << n;

    // The last executed statement is recursive call
    print(n-1);
}
```

Tail Recursion

Factorial of a given number using tail recursion.

```
long fact(long n, long a)
{
    if(n == 0)
        return a;
    return fact(n-1, a*n);
}
```

Searching

- **Search:** A search algorithm is a method of locating a specific item of information in a larger collection of data.
- The following are the different ways to search the contents of an array:
 1. Linear or Sequential Search
 2. Binary Search

Linear Search

- In this technique, key is compared with each item in a given array sequentially one after the other.
- If the key is found, the corresponding record may be retrieved or the position of the key may be returned.
- If the key is not found the function return 0 or -1. this technique is usually used to search in an unordered list.
- Also called **sequential or serial search**.

Algorithm

```
1.Read array A of n elements and key
2 flag=0
3.for i in range(n)
    if A[i]==key
        flag=1
    return flag
if flag=1
    print search is successful
else
    print search is unsuccessful
```

Complexity of Linear Search

For linear search, we need to count the number of comparisons performed, but each comparison may or may not search the desired item.

Case	Best Case	Worst Case	Average Case
If item is present	1	n	$n / 2$
If item is not present	n	n	n

Binary Search

- Binary search is an extremely efficient algorithm. This search technique searches the given item in minimum possible comparisons. To do the binary search first we had to sort the array elements.

Logic

- First find the middle element of the array i.e.,
 $\text{mid} = (\text{low} + \text{high}) / 2$ here $\text{low} = 0$ $\text{mid} = n - 1$
- compare the mid element with key element.

Binary Search

There are three cases:

- (a) if $A[\text{mid}]$ is a desired key element then the search is successful.
- (b) if $A[\text{mid}]$ is less than key element then search only the first half of the array. i.e., **high = mid-1**
- (c) if $A[\text{mid}]$ is greater than the key element search in the second half of the array. i.e., **low = mid+1**

Binary Search

- Repeat the same steps until an element is found. In this algorithm every time we are reducing the search area.
- So the number of comparisons keep on decreasing.
- In worst case the number of comparisons is atmost $\log(n)$.
- So it is an efficient algorithm when compared linear search but the array has to be sorted before doing binary search

Algorithm

Here A is an array of n elements.

Step 1: low=0, high=n-1, mid=(low + high)/2

Step 2: repeat step 3 and step 4 while low <= high and A[mid] != key

Step 3: if item < A[mid], then

 set high = mid-1

 else

 set low=mid +1

Step 4: set mid=(low + high)/2

Step 5: if A[mid]= key then

 set loc=mid

 else

 set loc=NULL

Step 6: exit

Example

Binary Search

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
	L=0	1	2	3	M=4	5	6	7	8	H=9
23 > 16 take 2 nd half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5	6	M=7	8	H=9
23 > 56 take 1 st half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5, M=5	H=6	7	8	9
Found 23, Return 5	2	5	8	12	16	23	38	56	72	91

Complexity of Binary Search

In Binary Search, each comparison eliminates about half of the items from the list. Consider a list with n items, then about $n/2$ items will be eliminated after first comparison. After second comparison, $n/4$ items of the list will be eliminated.

- If this process is repeated for several times, then there will be just one item left in the list.
- The number of comparisons required to reach to this point is $n/2^i = 1$. If we solve for i , then it gives us $i = \log n$.
- The maximum number of comparisons is logarithmic in nature, hence the time complexity of binary search is $O(\log n)$.

Complexity of Binary Search

Case	Best Case	Worst Case	Average Case
If item is present	1	$O(\log n)$	$O(\log n)$
If item is not present	$O(\log n)$	$O(\log n)$	$O(\log n)$

Sorting Techniques

Sorting in general refers to various methods of arranging or ordering things based on criteria's (numerical, chronological, alphabetical etc.).

There are many approaches to sorting data and each has its own merits and demerits.

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort

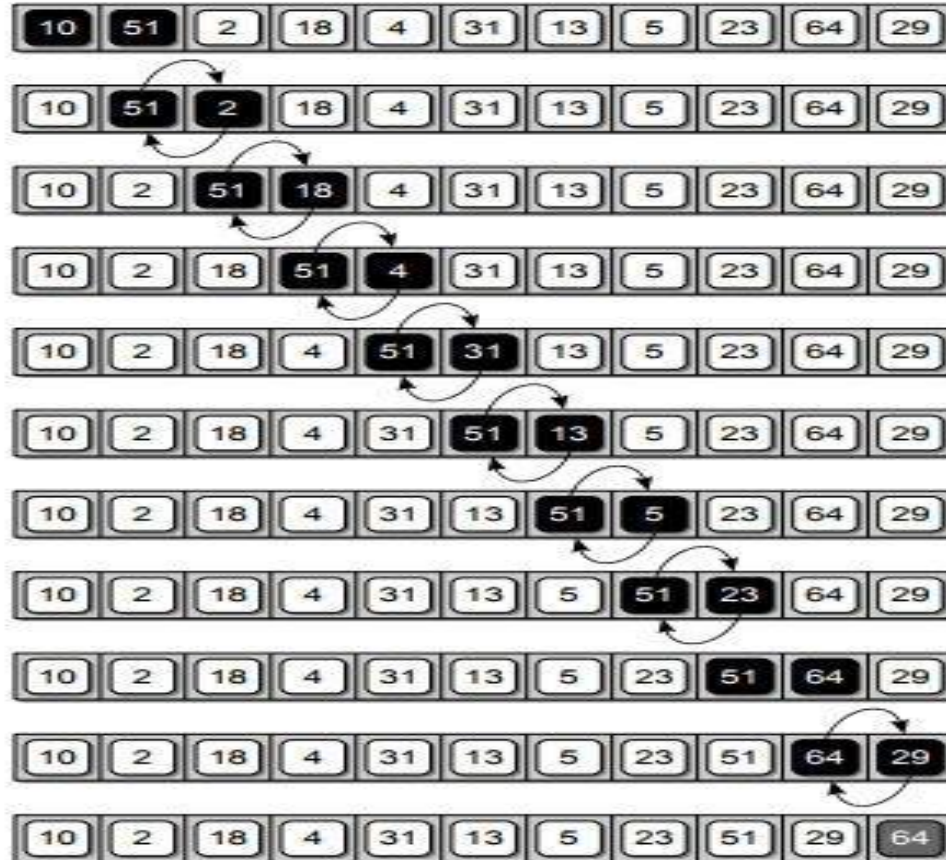
Bubble Sort

This sorting technique is also known as *exchange sort*, which arranges values by iterating over the list several times and in each iteration the larger value gets bubble up to the end of the list.

- This algorithm uses multiple passes and in each pass the first and second data items are compared.
- if the first data item is bigger than the second, then the two items are swapped.
- Next the items in second and third position are compared and if the first one is larger than the second, then they are swapped, otherwise no change in their order.
- This process continues for each successive pair of data items until all items are sorted.

Example

Pass - 1



Algorithm

Step 1: Repeat Steps 2 and 3 for $i=1$ to $n-1$

Step 2: Set $j=1$

Step 3: Repeat while $j \leq n-i-1$

(A) if $a[i] < a[j]$

Then interchange $a[i]$ and $a[j]$

[End of if]

(B) Set $j = j+1$

[End of Inner Loop]

[End of Step 1 Outer Loop]

Step 4: Exit

Program

```
def bubbleSort(arr):
    n = len(arr)
    # Traverse through all array elements for i in range(n):
    for i in range(n-1):
        # Last i elements are already in place for j in range(0, n-i-1):
        for j in range(0,n-i-1):
            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element if arr[j] > arr[j+1] :
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
        #print(arr)
# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]
print("Unsorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i], end=' ')
bubbleSort(arr)
print ("\nSorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i], end=' ')
```

Time Complexity of Bubble Sort

The efficiency of Bubble sort algorithm is independent of number of data items in the array and its initial arrangement. If an array containing n data items, then the outer loop executes $n-1$ times as the algorithm requires $n-1$ passes. In the first pass, the inner loop is executed $n-1$ times; in the second pass, $n-2$ times; in the third pass, $n-3$ times and so on. The total number of iterations resulting in a run time of $O(n^2)$.

Worst Case Performance $O(n^2)$

Best Case Performance $O(n)$

Average Case Performance $O(n^2)$

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- 1) The subarray which is already sorted.
- 2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Example

```
arr[] = 64 25 12 22 11
```

```
// Find the minimum element in arr[0...4]  
// and place it at beginning
```

```
11 25 12 22 64
```

```
// Find the minimum element in arr[1...4]  
// and place it at beginning of arr[1...4]
```

```
11 12 25 22 64
```

```
// Find the minimum element in arr[2...4]  
// and place it at beginning of arr[2...4]
```

```
11 12 22 25 64
```

```
// Find the minimum element in arr[3...4]  
// and place it at beginning of arr[3...4]
```

```
11 12 22 25 64
```

Algorithm

```
For i = 1 to n-1 /*where n is the size of the array */  
    set: min_index=1 /*here we took starting index as 1 */  
    For j=i+1 to n  
        If arr[j]<arr[min_index]  
            set: min_index=j  
        EndIf  
    EndFor  
    swap (arr[i] and swap[min_index])  
EndFor  
). Exit.
```

Program

```
A = [64, 25, 12, 22, 11]
# Traverse through all array elements
for i in range(len(A)):
    # Find the minimum element in remaining unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]
# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
    print("%d" %A[i])
```

Time Complexity of Selection Sort

Selecting the lowest element requires scanning all n elements (this takes $n - 1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n - 1$ elements and so on, for $(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1) / 2 \in O(n^2)$ comparisons. Each of these scans requires one swap for $n - 1$ elements (the final element is already in place).

Worst Case Performance	$O(n^2)$
Best Case Performance	$O(n^2)$
Average Case Performance	$O(n^2)$

Insertion Sort

- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- The array is virtually split into a sorted and an unsorted part.
- Values from the unsorted part are picked and placed at the correct position in the sorted part.
- Simple implementation
- Efficient for (quite) small data sets
- More efficient in practice than most other simple quadratic (i.e., $\underline{O}(n^2)$) algorithms such as selection sort or bubble sort; the best case (nearly sorted input) is $\underline{O}(n)$

Example



Example

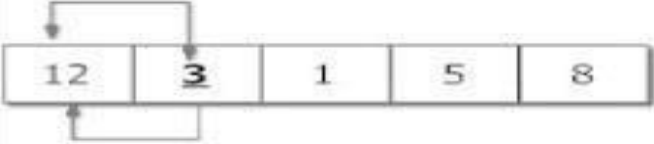
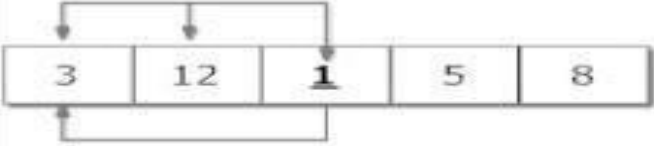
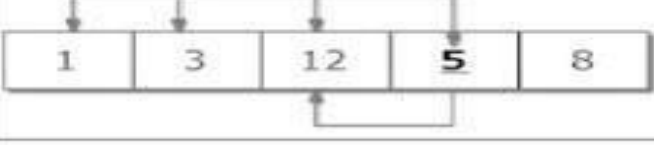
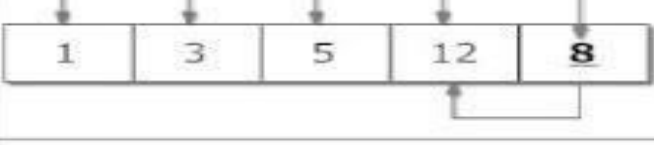

Step 1		Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2		Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3		Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4		Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
		Sorted Array in Ascending Order

Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Algorithm

To sort an array of size n in ascending order:

Step 1: Iterate from $\text{arr}[1]$ to $\text{arr}[n]$ over the array.

Step 2: Compare the current element (key) to its predecessor.

Step 3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Program

```
def insertionSort(arr):  
    # Traverse through 1 to len(arr)  
    for i in range(1, len(arr)):  
        key = arr[i]  
  
        # Move elements of arr[0..i-1], that are  
        # greater than key, to one position ahead  
        # of their current position  
        j = i-1  
        while j >= 0 and key < arr[j] :  
            arr[j + 1] = arr[j]  
            j -= 1  
        arr[j + 1] = key  
  
# Driver code to test above  
arr = [12, 11, 13, 5, 6]  
insertionSort(arr)  
for i in range(len(arr)):  
    print ("%d" % arr[i])
```

Time Complexity of Insertion Sort

If there are n elements to be sorted. Then, this procedure is repeated $n-1$ times to get sorted list of array.

Worst Case Performance	$O(n^2)$
Best Case Performance(nearly)	$O(n)$
Average Case Performance	$O(n^2)$

Quick Sort

Quick sort is a divide and conquer algorithm. Quick sort first divides a large list into two smaller sub- lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists.

The steps are:

1. Pick an element, called a **pivot**, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

The base case of the recursion is lists of size zero or one, which never need to be sorted.

Algorithm to find Pivot Element

Step – 1: Consider first element as a pivot $a[\text{low}]$

Step – 2: Initialize i to low index, and j to high index

Step – 3: Repeat the following steps until $i < j$

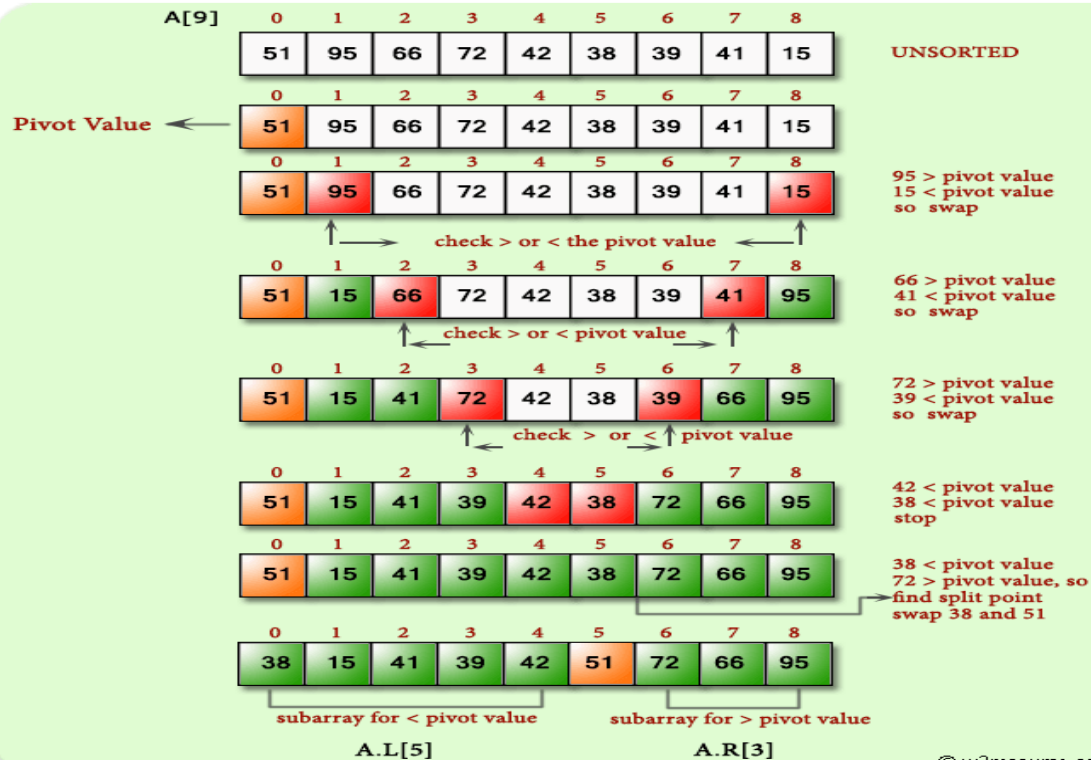
a) Keep on incrementing ' i ' value while $a[i] \leq \text{pivot}$

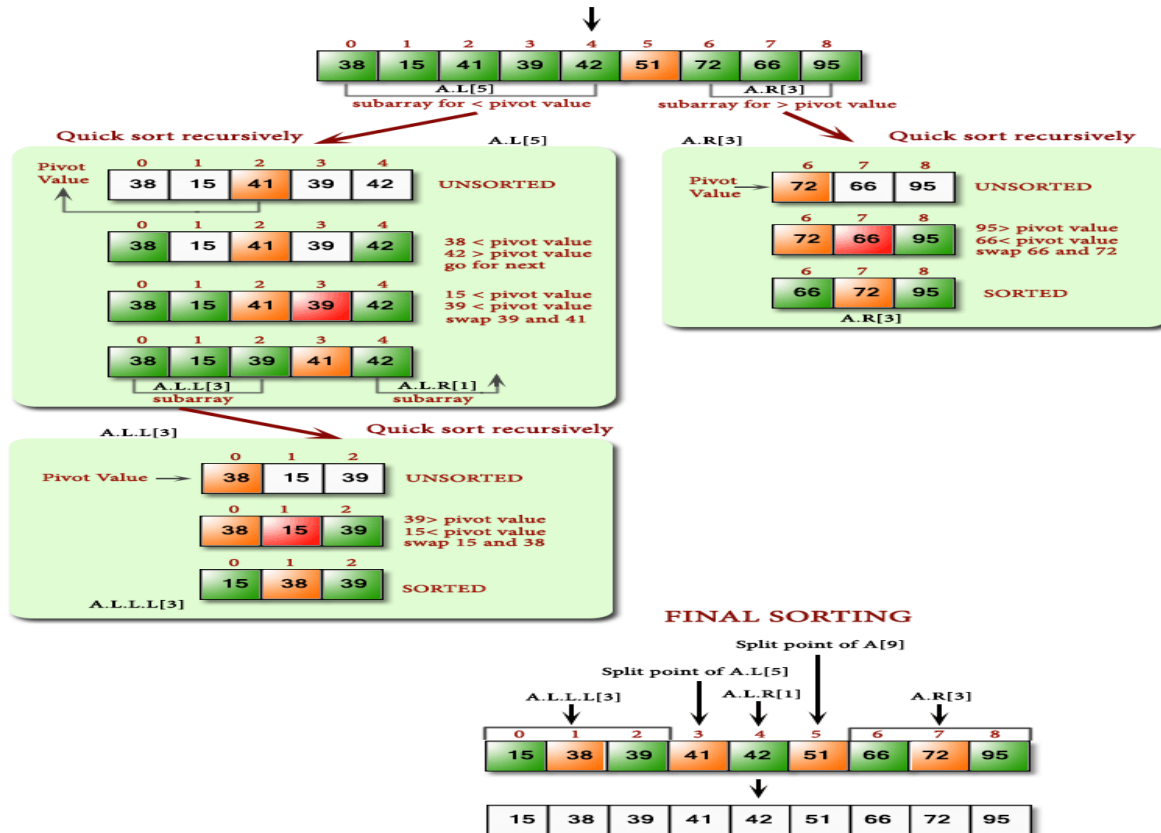
b) Keep on decrementing ' j ' value while $a[j] > \text{pivot}$

c) if $i < j$ then $\text{swap}(a[i], a[j])$

Step – 4: if $i > j$ then $\text{swap}(a[j], \text{pivot})$, j is the position of pivot

Quick Sort





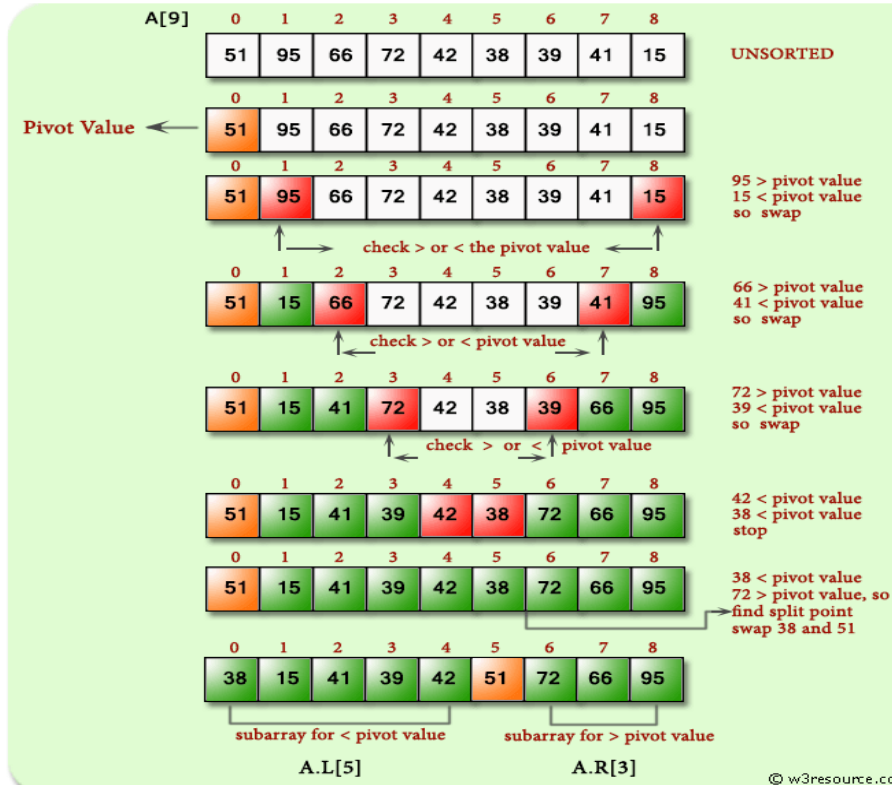
Program

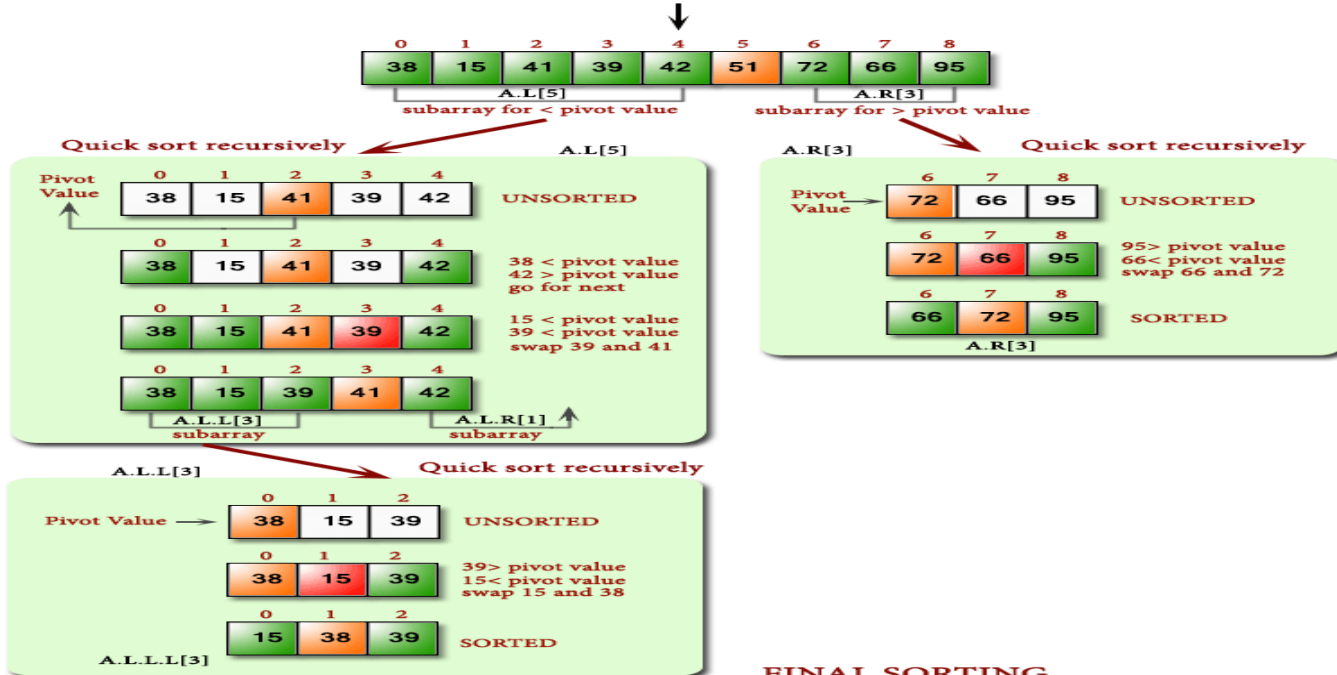
```
def partition(Array,low,up):
    i = low+1
    j = up
    pivot = Array[low]
    while(i<=j):
        while(Array[i]<pivot and i<up):
            i = i+1
        while(Array[j]>pivot):
            j = j-1
        if(i<j):
            Array[i],Array[j] = Array[j],Array[i]
            i = i+1
            j = j-1
        else:
            i = i+1
    Array[low] = Array[j]
    Array[j] = pivot
    return j
```

Program

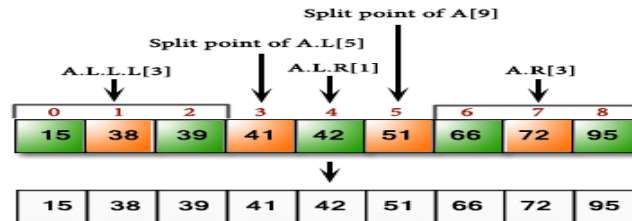
```
def quick(Array,low,up):  
    if(low>=up):  
        return  
    piv_loc = partition(Array,low,up)  
    #print(Array[low:piv_loc])  
    quick(Array,low,piv_loc-1)  
    #print(Array[piv_loc+1:up+1])  
    quick(Array,piv_loc+1,up)  
Array = [48,44,19,59,72,80,42,65,82,8,95,68]  
#Array=[30,20,10,50,60,40]  
low = 0  
up = len(Array) - 1  
print("Unsorted elements:")  
for i in Array:  
    print(i,end=' ')  
quick(Array,low,up)  
print("\nSorted elements:")  
for i in Array:  
    print(i,end=' ')
```

Quick Sort





FINAL SORTING



Time Complexity of Quick Sort

on average, makes $O(n \log n)$ comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons, though this behavior is rare. Quick sort is often faster in practice than other $O(n \log n)$ algorithms.

Worst Case Performance

$O(n^2)$

Best Case Performance(nearly)

$O(n \log_2 n)$

Average Case Performance

$O(n \log_2 n)$

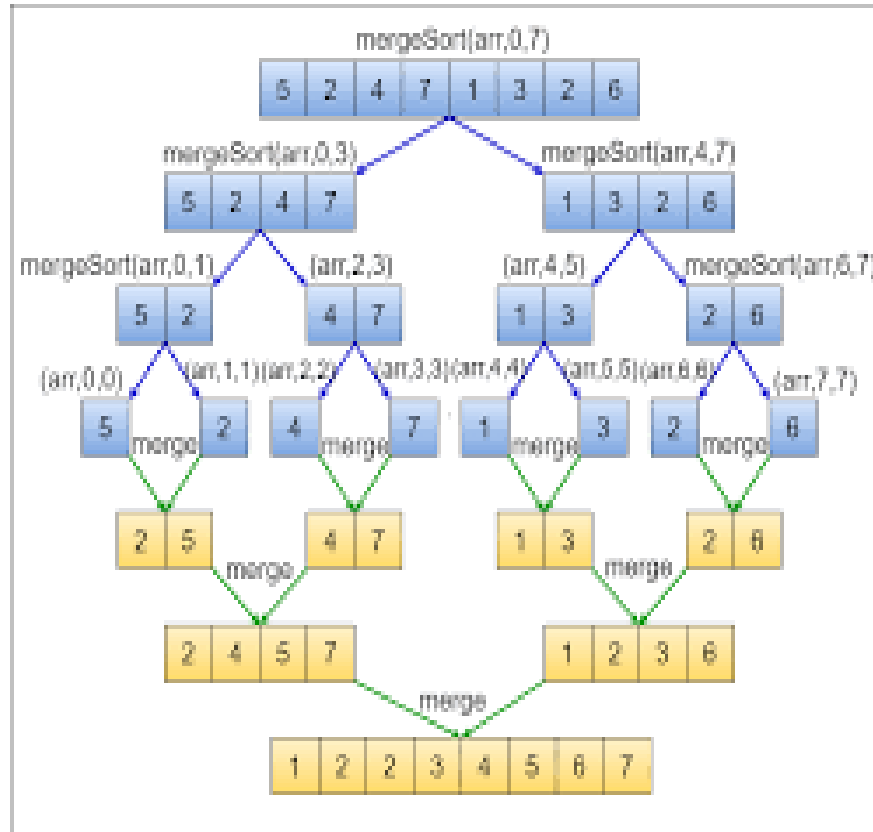
Merge Sort

Merge sort is based on Divide and conquer method. It takes the list to be sorted and divide it in half to create two unsorted lists. The two unsorted lists are then sorted and merged to get a sorted list. The two unsorted lists are sorted by continually calling the merge-sort algorithm; we eventually get a list of size 1 (One) which is already sorted. The two lists of size 1 (One) are then merged.

Merge Sort Procedure:

1. Divide the input which we have to sort into two parts in the middle. Call it the left part and right part.
2. Sort each of them separately. Note that here sort does not mean to sort it using some other method. We use the same function recursively.
3. Then merge the two sorted parts.

Merge Sort Example



Merge Sort Algorithm

MergeSort() function:

- It takes the array, left-most and right-most index of the array to be sorted as arguments.
- Middle index (mid) of the array is calculated as $(\text{left} + \text{right})/2$.
- Check if $(\text{left} < \text{right})$ cause we have to sort only when $\text{left} < \text{right}$ because when $\text{left} = \text{right}$ it is anyhow sorted.
- Sort the left part by calling MergeSort() function again over the left part MergeSort(array, left, mid) and the right part by recursive call of MergeSort function as MergeSort(array, mid + 1, right). Lastly merge the two arrays using the Merge function.

Merge Sort Algorithm

Merge() function:

- It takes the array, left-most , middle and right-most index of the array to be merged as arguments.
- Finally copy back the sorted array to the original array.

Merge Sort Algorithm

MergeSort(arr[], l, r)

If $l < r$:

1. Find the middle point to divide the array into two halves:
middle $m = l + (r-l)/2$
2. Call mergeSort for first half: Call mergeSort(arr, l, m)
3. Call mergeSort for second half: Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
Call merge(arr, l, m, r)

Time Complexity of Merge Sort

Worst Case Performance

$O(n \log_2 n)$

Best Case Performance(nearly)

$O(n \log_2 n)$

Average Case Performance

$O(n \log_2 n)$

Time Complexity Comparison of Sorting Algorithms

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Quick Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$	$O(N)$
Merge Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(N)$