www.ignousite.com
Course Code : BCS-042
Course Title : Introduction to Algorithm design
Assignment Number : BCA(4)/042/Assignment/2022-23
Maximum Marks : 100
Weightage : 25% www.ignousite.com
Last date of Submission : 31st October, 2022 (For July Session)
: 15th April, 2023 (For January Session)

Q1: Define time and space complexity of linear search algorithm. Calculate how many times the assignment operation will execute in the following code fragment

```
for ( i = 0, i < n, i++)
for (j = 0, j < m, j++)
  {
    x = x + 7;
    }
```

Ans. Time Complexity: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

In order to calculate time complexity on an algorithm, it is assumed that a constant time c is taken to execute one operation, and then the total operations for an input length on N are calculated. Consider an example to understand the process of calculation: Suppose a problem is to find whether a pair (X, Y) exists in an array, A of N elements whose sum is Z. The simplest idea is to consider every pair and check if it satisfies the given condition or not.

Space Complexity: The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input. Consider an example: Suppose a problem to find the frequency of array elements.

Algorithm	Time unit
for (i = 0, i < n, i++)	n + 1
for (j = 0, j < m, j++)	mx(n + 1) = (mn + m)
{	
x = x + 7;	mxn
}	**Total:** = (n + 1) + (mn +m) + mn
	= n + 1 + mn + n + mn
	= 2mn + m + n + 1 = O(mn)

Q2. For the function defined by f(n) = $3n^2 + 4n + 6$ and g(n) = n^2, Check if the following are true or not?

(i) f(n) = Ω(g(n))
(ii) n^3= Ω(g(n))
(iii) f(n) ≠Ω(n^4)

Ans. (i) f(n) = Ω(g(n))

Given, f(n) = $3n^2 + 4n + 6$ and g(n) = n^2

For $f(n) = \Omega(g(n))$ i.e. $3n^2 + 4n + 6 = \Omega(n^2)$, it must satisfy the following

$f(n) \geq C \times g(n) \geq 0$ for $\forall n \geq n_0$ where n, C are positive constants.

Now, we can write $f(n) = 3n^2 + 4n + 6 \geq 3n^2 \geq 0$ for $\forall n \geq 0$

$\Rightarrow f(n) \geq 3 \times g(n)$ for $\forall n \geq 0$..... (1) [substituting f(n) and g(n)]

Therefore, it is clear from inequation (1), C = 3 and $n_0 = 0$ Hence, $f(n) = \Omega(g(n))$ is true.

(ii) $n^3 = \Omega(g(n))$

For $n^3 = \Omega(g(n))$ i.e.n3 =Ω (n2), it must satisfy the following

$3 \geq C \times g(n) \geq 0$ for $\forall n \geq n_0$ where n, C are positive constants.

$\Rightarrow n^3 \geq C \times n^2$..... (1)

Clearly inequation (1) is satisfied for C = 1 and $\forall n \geq 0$ (i.e. $n_0 = 0$).

Since, we have found that $n^3 \geq C \times n^2$ for C = 1 and $n_0 = 0$. Hence, n3 = Ω (g (n)) is true.

(iii) $f(n) \neq \Omega(n^4)$

We assume contradict i.e. $f(n) = \Omega(n^4)$

$\Rightarrow 3n^2 + 4n + 6 \geq C \times n^4$..... (1)

As, $(3 + 4 + 6)\, n^3$ i.e. $13n^3 \geq 3n^2 + 4n + 6 \forall n \geq 1$ (i.e. $n_0 = 1$)

$\therefore 13n^3 \geq C \times n^4 \; \forall n \geq 1$

$\Rightarrow 13 \geq C \times n$ ..... (2)

$\Rightarrow \frac{13}{C} \geq n$

But, if n > 1 like $n = \frac{13}{C} + 1$ [C > 1] inequations (1) and (2) do not satisfied which directly mean our assumption was wrong i.e f(n) = Ω (n4) is not true. Hence, $f(n) \neq \Omega(n4)$ is true.

Q3. (a) Write important features of Quick Sort algorithm. How is it different from Selection Sort and Insertion Sort algorithms in terms of the sorting processes? Apply Quick Sort algorithm to do sorting of the following array of integer numbers in ascending order.

29	16	28	14	7	35	10	22	15	4

Write the pseudo-code and show all the intermediate processes of the running of the code/algorithm.

Ans. Important features of Quick Sort algorithm:

- Quick Sort is useful for sorting arrays.
- In efficient implementations Quick Sort is not a stable sort, meaning that the relative order of equal sort items is not preserved.
- Overall time complexity of Quick Sort is O(nLogn). In the worst case, it makes O(n2) comparisons, though this behavior is rare.
- The space complexity of Quick Sort is O(nLogn). It is an in-place sort (i.e. it doesn't require any extra storage)

Quick Sort: This is the best sort Technique. Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quick Sort that pick pivot in different ways.

Time Complexity :
Best Case : O(nlogn) #Means array is already sorted.
Average Case : O(nlogn) #Means array with random numbers.
Worst Case : O(n^2) #Means array with descending order.

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.
Time Complexity of Insertion Sort
Best Case : $O(n^2)$ #Means array is already sorted.
Average Case : $O(n^2)$ #Means array with random numbers.
Worst Case : $O(n^2)$ #Means array with descending order.

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.

Time Complexity of Insertion Sort
Best Case : O(n) #Means array is already sorted.
Average Case : $O(n^2)$ #Means array with random numbers.
Worst Case : $O(n^2)$ #Means array with descending order.

Given, unsorted array is:

29	16	28	14	7	35	10	22	15	4

Intermediate Steps of the Partition Procedure of Quick Sort algorithm:

29	16	28	14	7	35	10	22	15	4
1	2	3	4	5	6	7	8	9	10

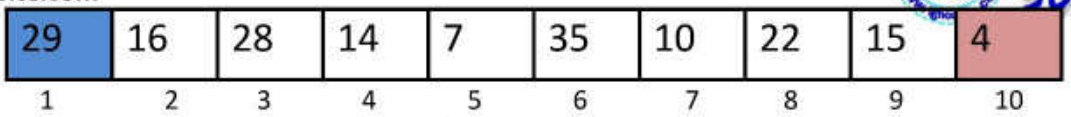
Feg. 1

Partitioning begins by locating two position markers —let's call them low (l) and high (h)— at beginning and end of the remaining items in the list (positions 1 and 10 in Figure 2). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. Figure 2 shows this process as we locate the position of 29.

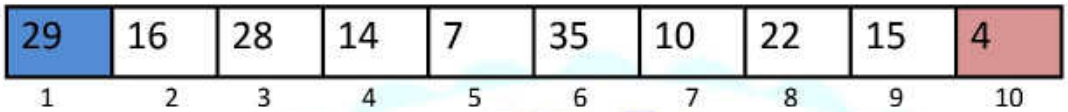
29	16	28	14	7	35	10	22	15	4
1	2	3	4	5	6	7	8	9	10

> 29 move to right

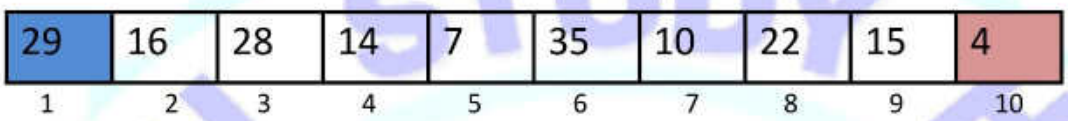
< 29 stop



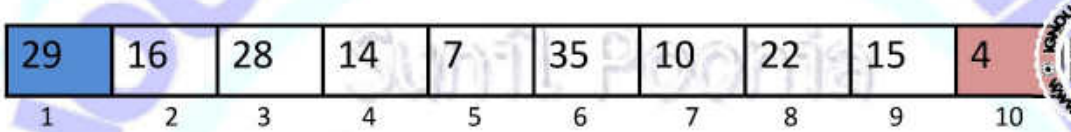
> 29 move to right



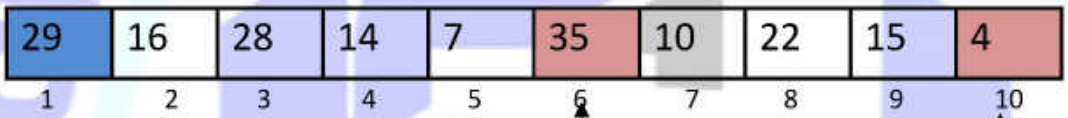
> 29 move to right



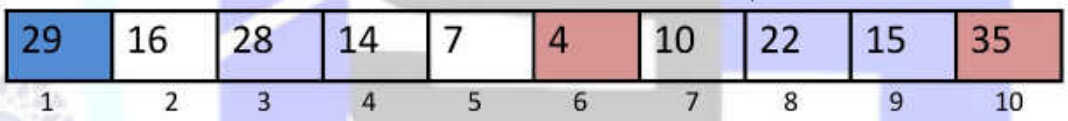
> 29 move to right



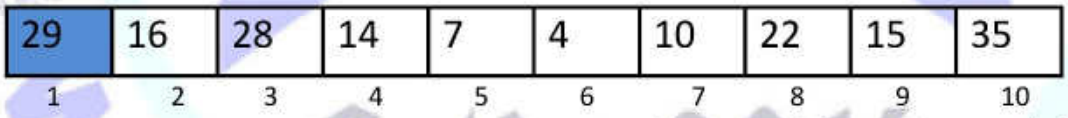
> 29 stop



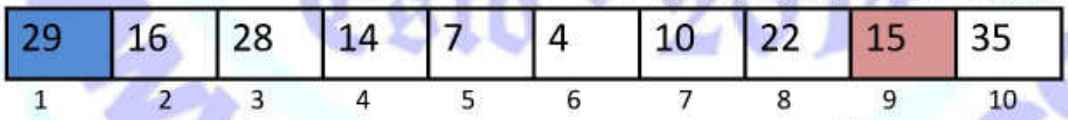
Swap 35 and 4



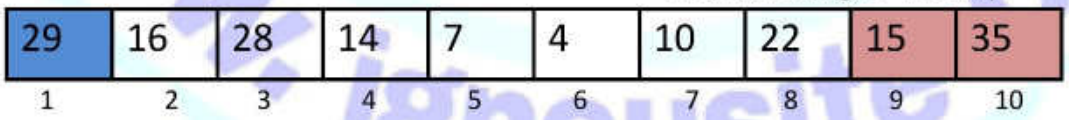
Now continue moving low and high.



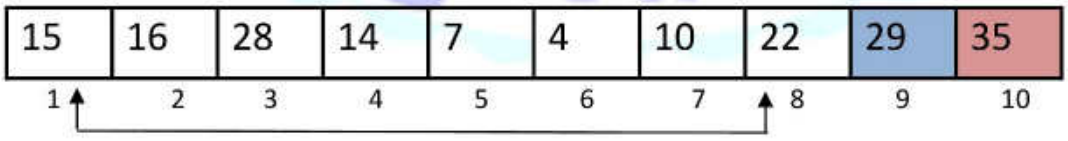
> 29 move to right < 29 move to left



> 29 move to right < 29 stop



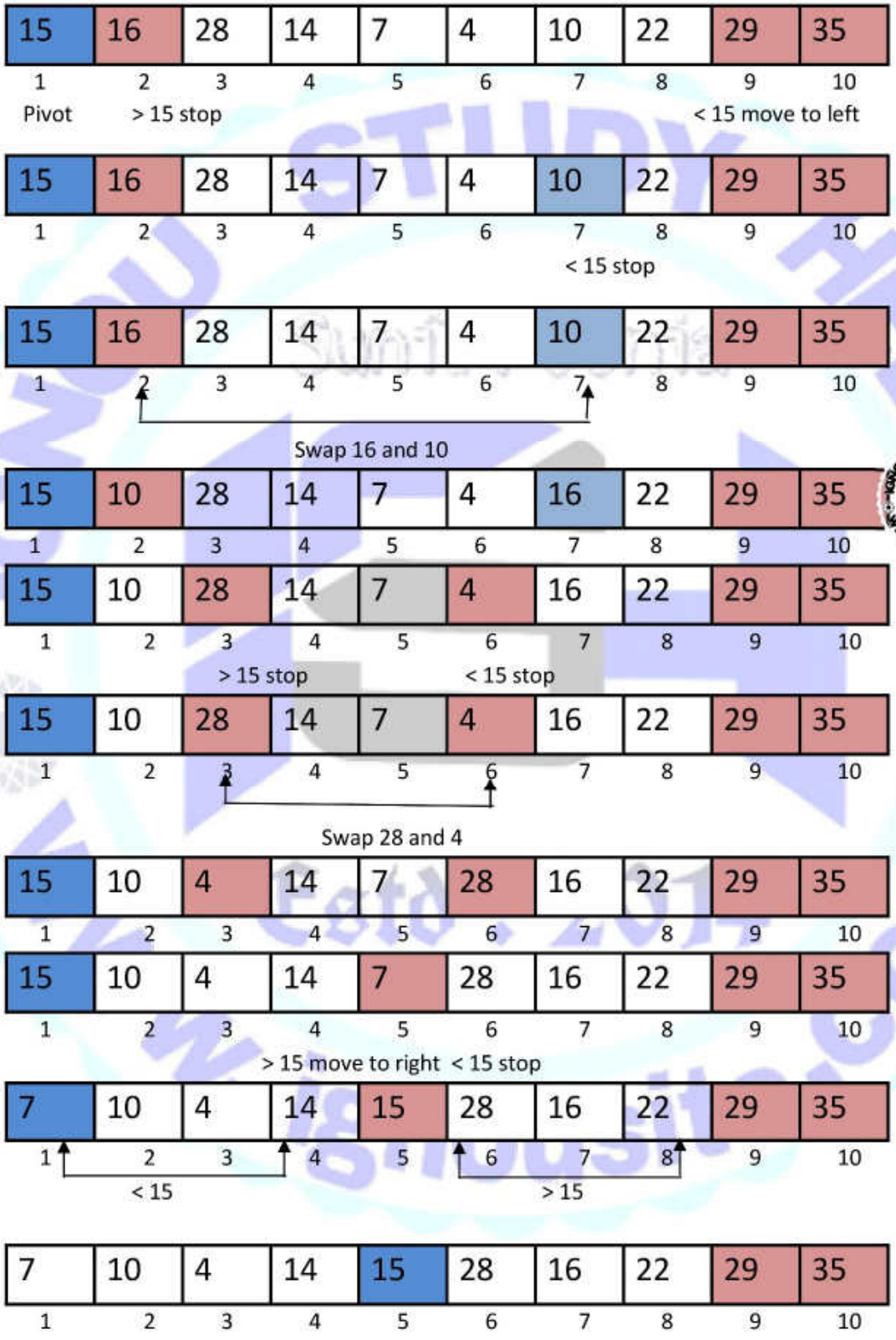
> 29 stop

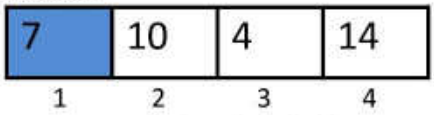


< 29

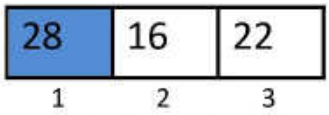
Here, 29 is a splitting point. On left of 29, elements are not sorted and hence the partition process must be repeat for the array left of 29 but on right of 29 only 35 have found. So for the right of 29, array has already been sorted.

Now, on repeating the partition process, we got the following results which are shown in figures below:



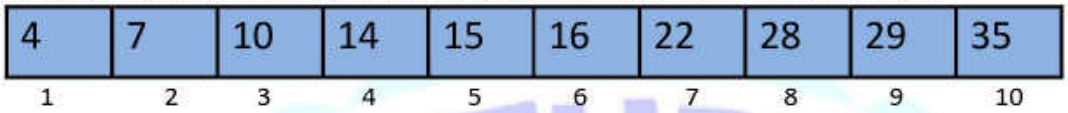


Quick Short left half



Quick Short right half

Now on repairing the partition process on both the sides, we got the final sorted array below:



Quick Short Pseudocode:

```
function partition Func (left, right, pivot)
  left Pointer = left
right Pointer = right -1
while True do
while A[++left Pointer] < pivot do
//do-nothing
  end while
while right Pointer > 0&& A[--right Pointer] > pivot do
//do-nothing
end while if left Pointer >= right Pointer break
else
swap left Pointer, right Pointer
end if
 end while
 swap left Pointer, right
return left Pointer
end function
```

Quick Short pivot Pseudocode

```
procedure quick Sort(left, right)
if right-left <= 0
return
else
pivot = A[right]
partition = partition Func(left, right, pivot)
quick Sort(left,partition-1)
quick Sort(partition+1,right)
end if
end procedure
```

(b) Compute worst case and best case time complexities of Binary Search algorithm. When the worst and best cases of Binary Search algorithm would occur? Explain.

Ans. Compute worst case and best case time complexities of Binary Search algorithm:

The space complexity of binary search algorithm:

- O(1) in the case of the iterative method.
- O(log N) in the case of the recursive method.

Space complexity illustrates memory usage (RAM) while executing the program with more significant inputs. Our program never needs additional memory in the most typical implementation of the binary search (i.e., iterative binary search). We would search the target element in the array during every iteration by updating the start and end index. Therefore, the size of our input does not affect the additional space required for the binary search algorithm, and the space complexity is constant O(1), i.e., space complexity is independent of input N (size of the array).

The time complexity of binary search algorithm:

- O(1) in the best case.
- O(log N) in the worst case.

An algorithm is O(log N) if it takes a constant time to cut the problem size by a fraction (usually by 1/2). Let's look at this mathematically:

In binary search, Length of the array: N (at iteration 1)
After iteration 1: N / 2 (size of the array is decreased by 1/2 after each iteration)
After iteration 2: (N / 2) / 2 = N / 4 = N / 2^2
After iteration 3: (N / 4) / 2 = N / 8 = N / 2^3
. . Suppose after k iterations, array size becomes 1, so
After iteration k: N / 2^k = 1

Therefore, $N = 2^k$

Taking $\log_2$ both sides,

$\Rightarrow \log_2(N) = \log_2(2^k)$

Since, $\log(x^y) = y * \log(x)$

$\Rightarrow \log_2(N) = k * \log_2(2)$

Since, $\log_a(a) = 1$

$\Rightarrow \log_2(N) = k * 1$

or,

$\Rightarrow k = \log_2(N)$

This proves that the time complexity of the binary search algorithm is O(log N).

The worst and best cases of Binary Search algorithm would occur: Time complexity describes the amount of time the algorithm takes on input provided.

Best Case: The best case is when the target element is present in the middle of the list. Binary search always begins its search with the middle whether the list comprises five or one million makes no difference. Therefore, the best case time complexity of the binary search for a list of size N is O(1), i.e., independent of the size of the input N.

Worst Case: The worst-case scenario is that the target element is not present in the list. The time taken to search the target element in the list is determined by the number of times we need to halve the list until we reach the final element. Therefore, the worst-case time complexity of the binary search for a list of size N is O(log N).

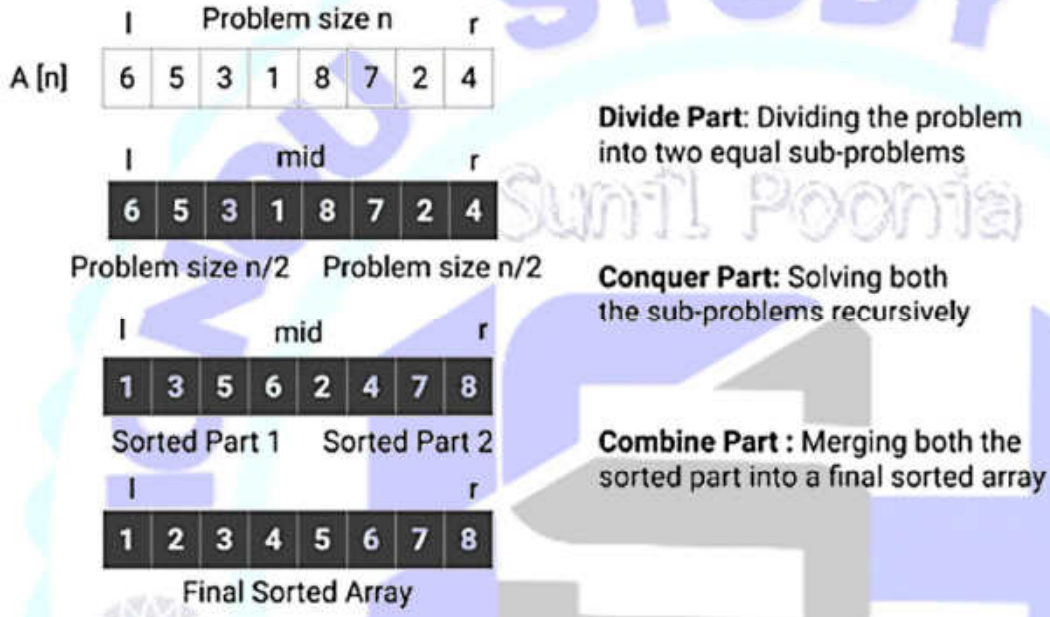
Q4. (a) Describe the followings problems in brief and define its recurrence relation: (4)

(i) Merge Sort Problem

(ii) Karatsuba Method

Ans.

(i) Merge Sort Problem: Suppose we need to sort an array A [l...r] of n integers starting from index l and ending at index r. The critical question is: can we solve sorting problem of size n using solution of smaller sub-problems or by applying divide and conquer strategy?



If we observe the above diagram, divide and conquer idea looks like this:

Divide part: Divide sorting problem of size n into two different and equal sub-problems of size n/2. We can easily divide the problem by calculating the mid.

Left subproblem: Sorting A[] from l to mid
Right subproblem: Sorting A[] from mid + 1 to r

Conquer part: We solve sub-problems recursively and sort both smaller halves. We need not worry about solution to the sub-problems because recursion will do this work for us.

Combine part: We merge both sorted halves to generate the final sorted array. In other words, we combine solution of both the sub-problems of size n/2 to solve sorting problems of size n.

(ii) Karatsuba Method:

Karatsuba discovered a method to compute X.Y $\left(as\ \ X.Y = a.c.10^{2\lfloor \frac{n}{2} \rfloor} + (bc + ad)\, 10^{\lfloor \frac{n}{2} \rfloor} + b.d\right)$ in only 3 multiplications, at the cost of few extra additions; as follows:

Let U = (a+b).(c+d)

V = a.c

W = b.d

Now $X.Y = V.10^{2\lfloor \frac{n}{2} \rfloor} + (U + V - W)\, 10^{\lfloor \frac{n}{2} \rfloor} + W$ .....(1)

Now, here, X.Y (as computed in equation (1)) requires only 3 multiplications of size n/2, which satisfy the following recurrence:

$$T(n) = \begin{cases} O(1) & if\ n = 1 \\ 3T\left(\frac{n}{2}\right) + O(n) & if\ n > 1 \end{cases}$$

Otherwise

Where **O (*n*)** is the cost of addition, subtraction and digit shift (multiplications by power of 10's), all these takes time proportional to '***n***'.

Recurrence relation of Karatsuba Algorithm:

$$T(n) = \begin{cases} O(1) & if\ n = 1 \\ 3T\left(\frac{n}{2}\right) + O(n) & if\ n > 1 \end{cases}$$

(b) Define the recurrence relation of the following function

```
f(n)
{
If ( n= = 1)
return 2
else
return f(n/4) + f(n/4) + f(n/4) + f(n/4) + 10
}
```

Ans.

T(n) ---------- Algorithm Test (f(n))
{
If (n= = 1)
return 2
else
T(n/4)+T(n/4)+T(n/4)+T(n/4)+C ---------- return f(n/4) + f(n/4) + f(n/4) + f(n/4) + 10
}
Therefore, T(n)= T(n/4)+T(n/4)+T(n/4)+T(n/4)+C

⇒ T(n) = 4T(n/4) + C where C = constant (in this case 10) and T(1) = 1

Hence, we obtain the recurrence relation as

$$T(n) = \begin{cases} 1 & if\ n = 1 \\ 4T\left(\frac{n}{4}\right) + C & if\ n > 1 \end{cases}$$

(c) Solve the following recurrence relation using both Substitution and Master methods.

T(n) = 3T (n/2) + 0(n)

Ans.

Master Method:

$T(n) = 3T(\frac{n}{2}) + O(n)$

$a = 3$

$b = 2$

$f(n) = n$

$n^{\log_b a} = n^{\log_2 3}$

$f(n) = n = O(n^{\log_2 3 - c})$

$\Rightarrow$ case 1 master method

$\Rightarrow T(n) = O(n^{\log_2 3})$

$\Rightarrow O(n^{1.59})$

Substitution method

$T(n) = 3T(\frac{n}{2}) + O(n)$

$= T(n) = 3T(\frac{n}{2}) + c \cdot n$ —— (1)

$= 3[3T(\frac{n}{2^2}) + c \cdot \frac{n}{2}] + c \cdot n$ (using eqn 2)

$= 3^2 T(\frac{n}{2^2}) + 3c \cdot \frac{n}{2} + c \cdot n$

$= T(n) = 3^2[3T(\frac{n}{2^3}) + c \cdot \frac{n}{2^2}] + 3c \cdot \frac{n}{2} + c \cdot n$ [using eqn 3]

$= 3^3 T(\frac{n}{2^3}) + 3^2 \cdot \frac{n}{2^2} \cdot c + 3 \cdot \frac{n}{2} \cdot c + 3 \cdot \frac{n}{2^0} \cdot c \quad (a^0 = 1)$

So, the general equation may be -

$T(n) = 3^k T(\frac{n}{2^k}) + 3^{k-1} \cdot \frac{n}{2^{k-1}} \cdot c + 3^{k-2} \cdot \frac{n}{2^{k-2}} \cdot c + \cdots + 3^0 \cdot \frac{n}{2^0} c$

Now, let $2^k = n$

Taking log arithm with base 2 on both sides, we have

$\log_2 2^k = \log_2 n$

$= k = \log_2 n \quad (\log_a a = 1)$

$T(n) = 3^k T(\frac{n}{n}) + 3^{k-1} \cdot \frac{n}{2^{k-1}} \cdot c + \frac{3^{k-2}}{2^{k-2}} \cdot nc + \cdots \frac{3^{k-k}}{2^{k-k}} \cdot nc$

$= 3^k \cdot T(1) + nc[(\frac{3}{2})^{k-1} + (\frac{3}{2})^{k-2} + \cdots + (\frac{3}{2})^{k-k}]$

$$= 3^k \cdot T(1) + nc \left(\frac{3}{2}\right)^k \left[\left(\frac{2}{3}\right) + \left(\frac{2}{3}\right)^2 + \cdots + \left(\frac{2}{3}\right)^k\right]$$

$$= 3^k \cdot T(1) + nc \cdot \frac{3^k}{2^k} \cdot \frac{2}{3} \cdot \frac{\left(1 - \left(\frac{2}{3}\right)^k\right)}{1 - \frac{2}{3}}$$

$$= 3^k \cdot T(1) + nc \cdot \frac{3^k}{2^k} \cdot \frac{2}{3} \times \frac{3}{1} \times \left(1 - \frac{2^k}{3^k}\right)$$

$$= 3^k \cdot T(1) + 2nc \cdot \frac{3^k}{2^k} - 2nc \cdot \frac{3^k}{2^k} \times \frac{2^k}{3^k}$$

$$T(n) = 3^k \cdot T(1) + 2nc \cdot \frac{3^k}{2^k} - 2nc$$

$$= 3^k \cdot T(1) + 2nc \cdot \frac{3^k}{n} - 2nc$$

$$= 3^k \cdot T(1) + 2c \cdot 3^k - 2nc$$

$$= 3^k \left[T(1) + 2c\right] - 2nc$$

$$= 3^{\log_2 n} \left[T(1) + 2c\right] - 2nc \quad (k = \log_2 n)$$

$$= n^{\log_2 3} \left[T(1) + 2c\right] - 2nc \quad (a^{\log_b c} = c^{\log_b a})$$

$$= O(n^{\log_2 3})$$

Q5. Write a Linear Search algorithm to search for number 36 in the following list of integer numbers. How many search operations will be required in this example. Show all the intermediate steps.

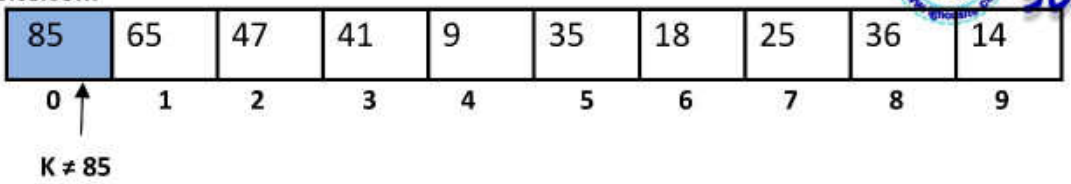
85	65	47	41	9	35	18	25	36	14

Which are the other algorithms which perform better than the linear search algorithm? Discuss the worst case and average case time complexity of the algorithm.

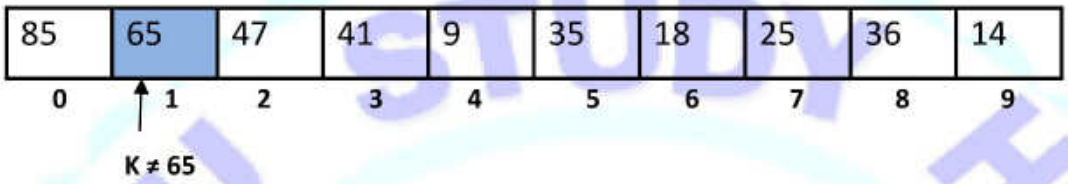
Ans. The given elements of array are:

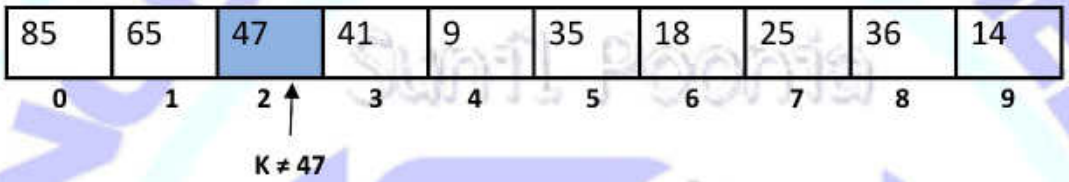
85	65	47	41	9	35	18	25	36	14
0	1	2	3	4	5	6	7	8	9

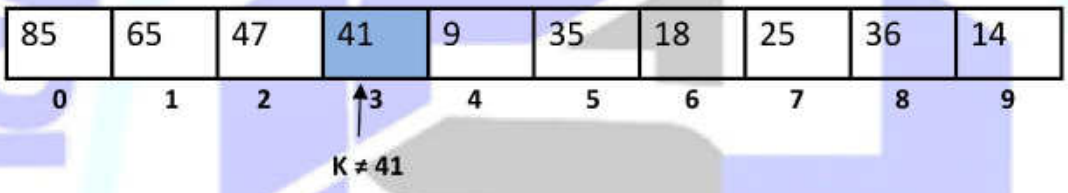
The element to be searched is K = 36. Now, start from the first element and compare K with each element of the array.

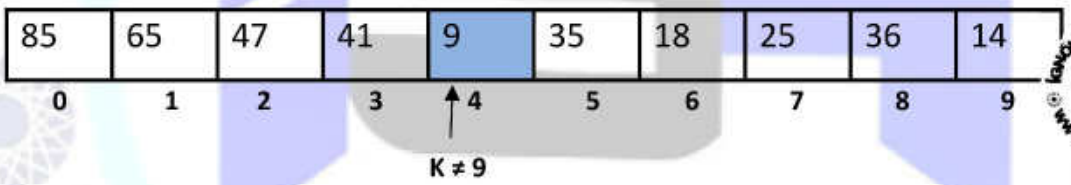


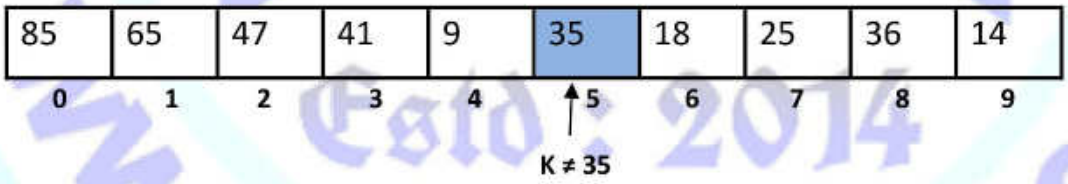
The value of K,i.e., 36,is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.

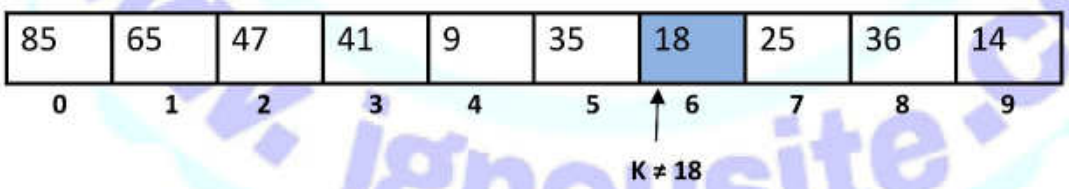


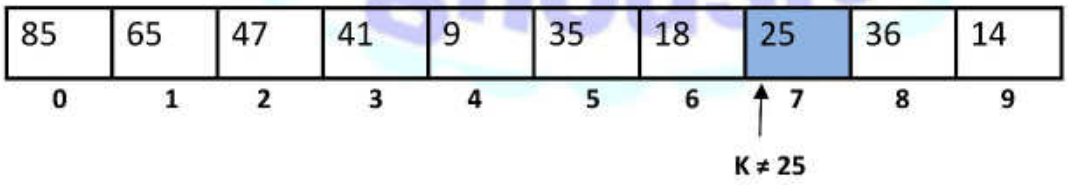


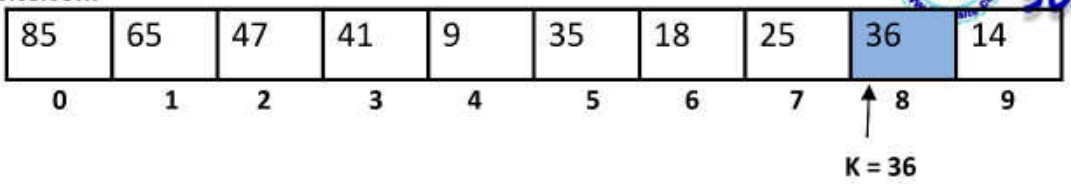












Now, the element to be searched is found at 8th position. So algorithm will return the index of the element matched. There are 9 search operation required to find 36 from the given array.

Binary search is better than linear search if the given array already sort.

Average Case Complexity: The average case time complexity of Binary search is O(logn).

Worst Case Complexity: In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is O(logn).

Q6. What is the idea behind binary exponent evaluation? Write pseudo-code to compute a^n using right to left and left to right binary exponentiation algorithm and perform its complexity analysis. Apply the algorithm to compute a^{55} and calculate the total number of multiplication operations in this case. How many multiplication operations are required if brute force multiplication method is used in this example? Show all the intermediate steps.

Ans. The idea behind binary exponent evaluation: Binary exponentiation (also known as exponentiation by squaring) is a trick which allows to calculate \(a^n\) using only \(O(\log n)\) multiplications (instead of \(O(n)\) multiplications required by the naive approach).

It also has important applications in many tasks unrelated to arithmetic, since it can be used with any operations that have the property of associativity:

\[(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)\]

The idea of binary exponentiation is, that we split the work using the binary representation of the exponent.

Left to Right Binary Exponentiation

Step I: Pseudo code to compute a^n by left to right binary exponentiation method:

// An array A of sizes with binary string equal to exponent n, where s is length of binary string n.
Step 1: Set result = a
Step 2: Set i = s – 2
Step 3: Compute result = result * result
Step 4: If A[i] =1 then compute result = result* a
Step 5: i = i – 1 and if i ≤ 0 then go to step 4.
Step 6: return computed value as result

Step II: Algorithm to compute a^n by left to right binary exponentiation method is as follows:

Input: a^n and binary string of length s for exponent n as an array A[s]
Output: Final value of a^n

To illustrate the above algorithm to compute a^{55}

In this case exponent n= 55 which is equivalent to binary string 110111 i.e. s = 6

i = 5	i = 4	i = 3	i = 2	i = 1	i = 0
1	1	0	1	1	1

Step by step illustration of the left to right binary exponentiation algorithm for a^{55}:

S = 6

Result = a

Iteration 1:

i = s − 2 = 6 − 2 = 4

result = a ∗ a = a^2

A[4] = 1 [True]

result = a^2∗a = a^3

Iteration 2:

i = 4 − 1 = 3

result = a^3∗a^3 = a^6

A[3] = 0 [False]

result = a^2∗a = a^3

Iteration 3:

i = 3 − 1 = 2

result = a^6∗a^6 = a^{12}

A[2] = 1 [True]

result = a^{12}∗a = a^{13}

Iteration 4:

i = 2 − 1 = 1

result = a^{13}∗a^{13} = a^{26}

A[1] = 1 [True]

result = a^{26}∗a = a^{27}

Iteration 5:

i = 1 − 1 = 0

result = a^{27}∗a^{27} = a^{54}

A[0] = 1 [True]

result = a^{54}∗a = a^{55}

Here, total number of multiplication is 9 instead of 54 multiplications in brute force algorithm i.e. n − 1 = 55 −1 = 54

Complexity Analysis: This algorithm performs either one multiplication or two multiplications in each iteration of for loop as shown in step 2 of Step II. Hence, total number of multiplications in the algorithm for computing a^n= a^{55} will be in the range of s − 1 ≤f(n) ≤ 2(s − 1) where s = 6 is length of the binary string equivalent to exponent n = 55 and f is function that represent number of multiplication in terms of exponent n = 55.

So complexity of the algorithm will be O($\log_2$n) = O($\log_2$ 55). As n = 55 can be representation in binary by using maximum of s = 6 bits i.e n =2^s which further implies s = O($\log_2$n) = O($\log_2$ 55).

Right to Left Binary Exponentiation

Step I: Pseudo code to compute a^n by right to left binary exponentiation method:

// An array A of sizes with binary string equal to exponent n, where s is length of binary string n.

Step 1: Set x = a

Step 2: If A [0] = 1 then set result = a

Step 3: else set result = 1

Step 4: Initialize i = 1

Step 5: Computex = x∗x

Step 6: If A[i] = 1 then compute result = result∗x

Step 7: Increment i by 1 as i = i + 1 and if i ≤ s − 1 then go to step 4.

Step 8: return computed value as result.

Step II: Algorithm to compute a^n by left to right binary exponentiation method is as follows:

Input: a^n and binary string of lengths for exponent n as an array A[s]

Output: Final value of a^n

To illustrate the above algorithm to compute a^{55}

In this case exponent n = 55 which is equivalent to binary string 110111 i.e. s = 6

i = 5	i = 4	i = 3	i = 2	i = 1	i = 0
1	1	0	1	1	1

Step by step illustration of the left to right binary exponentiation algorithm for a^{55}:

s = 6result = a

A[0] = 1

Iteration 1:

i = 1

$x = a*a = a^2$

A[1] = 1 [True]

result = result ∗ $x = a^2 * a = a^3$

Iteration 2:

i = 2

$x = a^2*a^2 = a^4$

A[2] = 1 [True]

result = result ∗ x = a3∗a4 = a7

Iteration 3:

i = 3

$x = a^4*a^4 = a^8$

A[3] = 0 [False]

Iteration 4:

i = 4

$x = a^8*a^8 = a^{16}$

A[4] = 1 [True]

result = result ∗ $x = a^7*a^{16} = a^{23}$

Iteration 5:

i = 5

$x = a^{16}*a^{16} = a^{32}$

A[5] = 1 [True]

result = result ∗ $x = a^{23}*a^{32} = a^{55}$

Here, total number of multiplications is 9 instead of 54 multiplications in brute force algorithm i.e. n − 1 = 55 −1 = 54

Complexity Analysis: This algorithm performs either one multiplication or two multiplications in each iteration of for loop as shown in step 2 of Step II

Hence, total number of multiplications in the algorithm for computing $a^n = a^{55}$ will be in the range of $s - 1 \leq f(n) \leq 2(s - 1)$ where $s = 6$ is length of the binary string equivalent to exponent $n = 55$ and f is function that represent number of multiplication in terms of exponent $n = 55$. So complexity of the algorithm will be $O(\log_2 n) = O(\log_2 55)$. As $n = 55$ can be representation in binary by using maximum of $s = 6$ bits i.e $n = 2^s$ which further implies $s = O(\log_2 n) = O(\log_2 55)$.

Q7. (a) Discuss the working of DFS and BFS algorithms with suitable examples.

Ans. DFS algorithms:

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

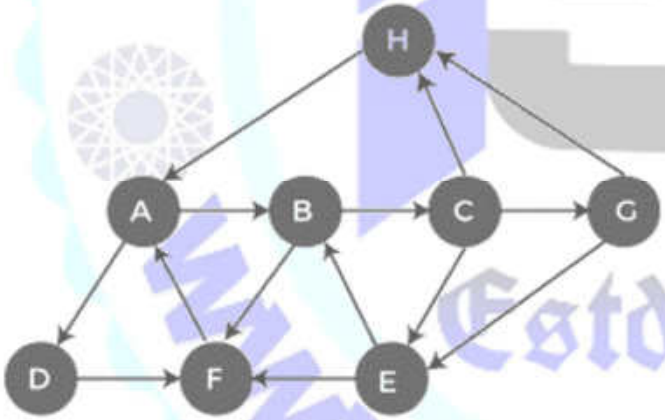
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

Example: In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Step 1 - First, push H onto the stack.
STACK: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.
Print: H]STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.
Print: A
STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.
Print: D
STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.
Print: F
STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.
Print: B
STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.
Print: C
STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.
Print: G
STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.
Print: E
STACK:
Now, all the graph nodes have been traversed, and the stack is empty.

BFS algorithms:

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

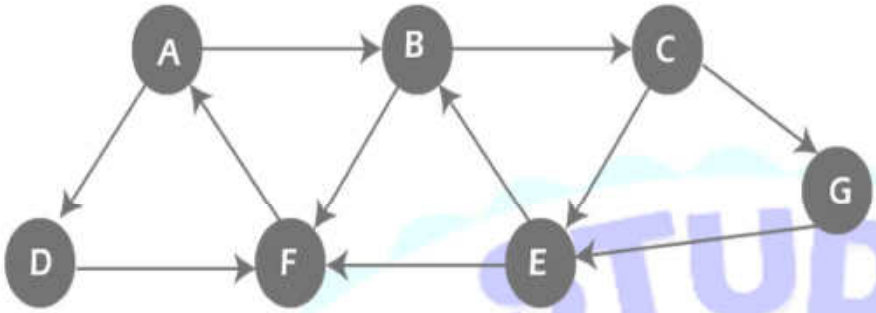
Step 5: Enqueue all the neighbors of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

(waiting state)

[END OF LOOP]

Example: In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

In the above graph, minimum path 'P' can be found by using the BFS that will start from Node A and end at Node E. The algorithm uses two queues, namely QUEUE1 and QUEUE2. QUEUE1 holds all the nodes that are to be processed, while QUEUE2 holds all the nodes that are processed and deleted from QUEUE1.

Now, let's start examining the graph starting from Node A.

Step 1 - First, add A to queue1 and NULL to queue2.
QUEUE1 = {A}
QUEUE2 = {NULL}

Step 2 - Now, delete node A from queue1 and add it into queue2. Insert all neighbors of node A to queue1.
QUEUE1 = {B, D}
QUEUE2 = {A}

Step 3 - Now, delete node B from queue1 and add it into queue2. Insert all neighbors of node B to queue1.
QUEUE1 = {D, C, F}
QUEUE2 = {A, B}

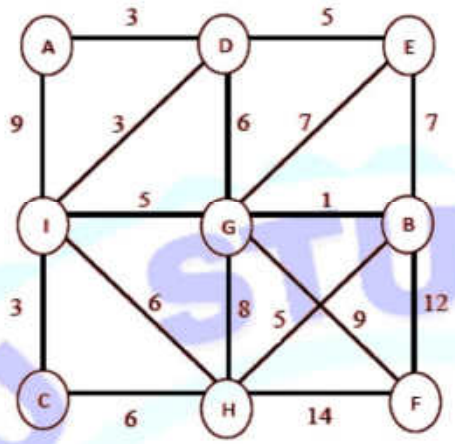
Step 4 - Now, delete node D from queue1 and add it into queue2. Insert all neighbors of node D to queue1. The only neighbor of Node D is F since it is already inserted, so it will not be inserted again.
QUEUE1 = {C, F}
QUEUE2 = {A, B, D}

Step 5 - Delete node C from queue1 and add it into queue2. Insert all neighbors of node C to queue1.
QUEUE1 = {F, E, G}
QUEUE2 = {A, B, D, C}

Step 6 - Delete node F from queue1 and add it into queue2. Insert all neighbors of node F to queue1. Since all the neighbors of node F are already present, we will not insert them again.
QUEUE1 = {E, G}
QUEUE2 = {A, B, D, C, F}

Step 7 - Delete node E from queue1. Since all of its neighbors have already been added, so we will not insert them again. Now, all the nodes are visited, and the target node E is encountered into queue2.
QUEUE1 = {G}
QUEUE2 = {A, B, D, C, F, E}

(b) Write Kruskal's algorithm to calculate the minimum cost spanning tree of the following graph and calculate the time complexity of the algorithm. Show all the intermediate steps.



Ans.

The graph G(V, E) given below contains 9 vertices and 17 edges. And you will create a minimum spanning tree T(V', E') for G(V, E) such that the number of vertices in T will be 9 and edges will be 8 (= 9 – 1).

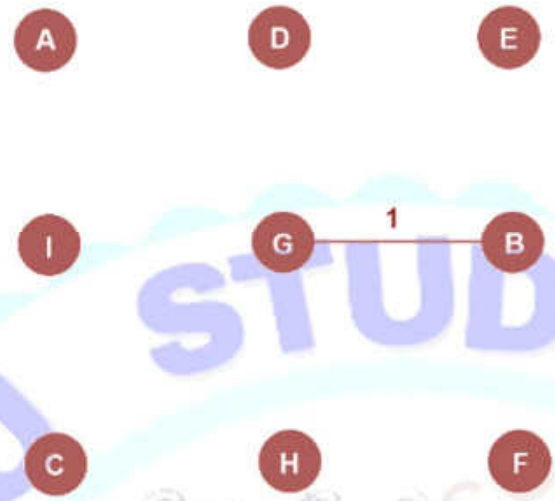
The given graph has no loops or parallel edges, so now we are going to next step.

The next step that we will proceed with arranging all edges in a sorted list by their edge weights.

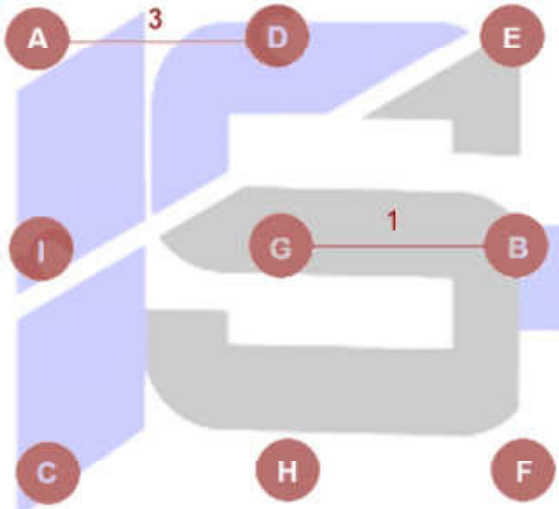
B, G	1
A, D	3
D, I	3
I, C	3
D, E	5
G I	5
B, H	5
D, G	6
I, H	6
G, H	6
E, G	7
B, E	7
G, H	8
A, I	9
G, F	9
B, F	12
H, F	14

After this step,

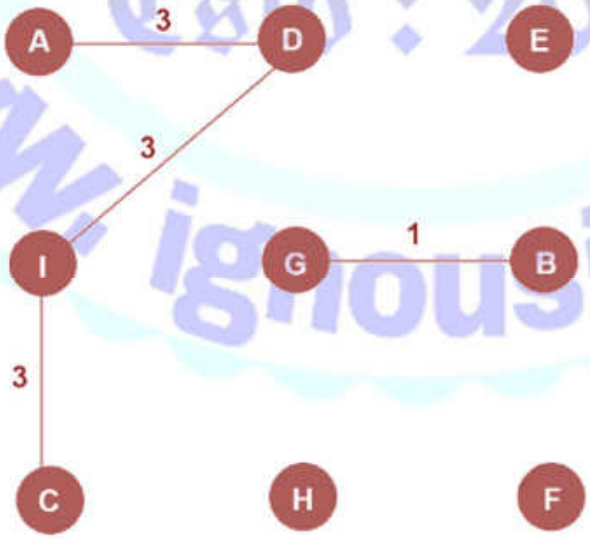
Step: 1 we will include edges in the MST such that the included edge would not form a cycle in our tree structure. The first edge that we will pick is edge BG, as it has a minimum edge weight that is 1.



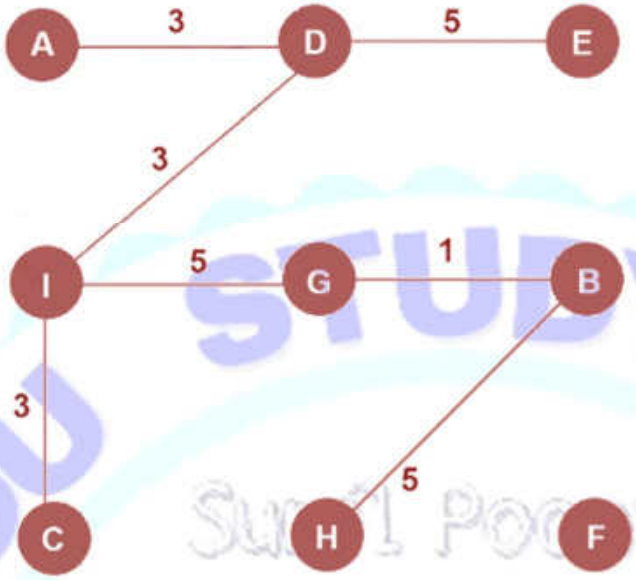
Step: 2 Add edge AD to the spanning tree.



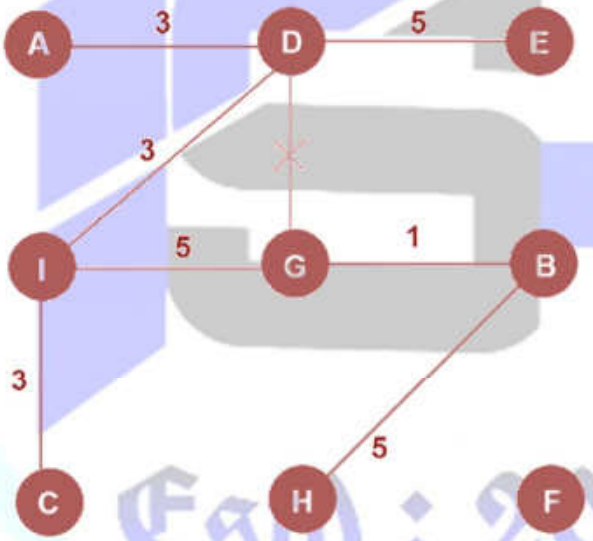
Step: 3 Add edge DI and edge IC to the spanning tree as it does not generate any loop.



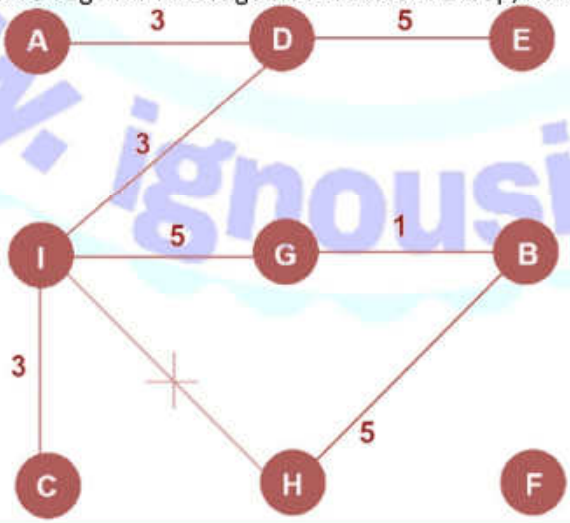
Step: 4 Add edge DE, GI and edge BH to the spanning tree as it does not generate any loop.



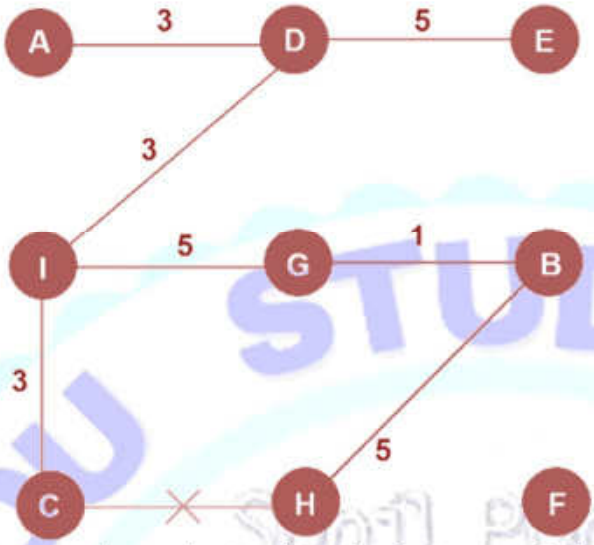
Step: 5 Next up is edge DG. This edge generates the loop in our tree structure. Thus, we will discard this edge.



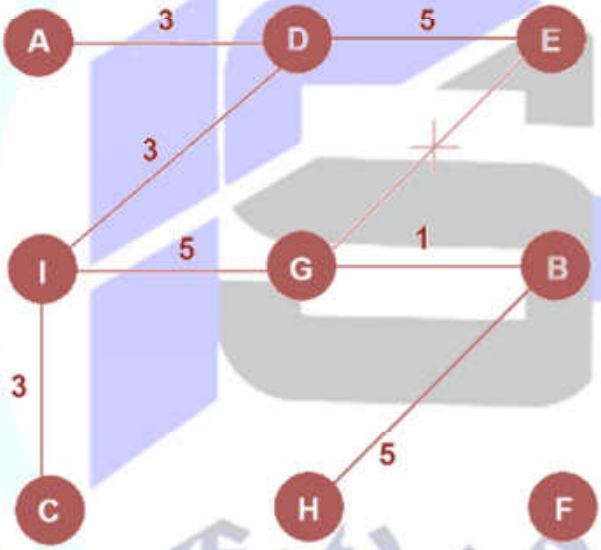
Step: 6 Following edge DG, we have edge HI. This edge also creates the loop; hence we will discard it.



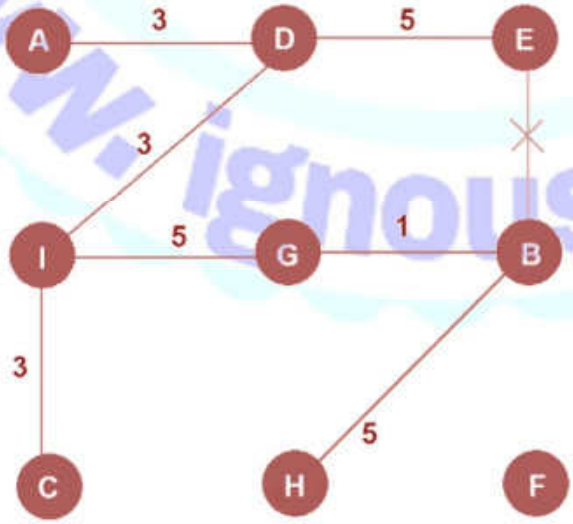
Step: 7 Following edge HI, we have edge CH. This edge also creates the loop; hence we will discard it.



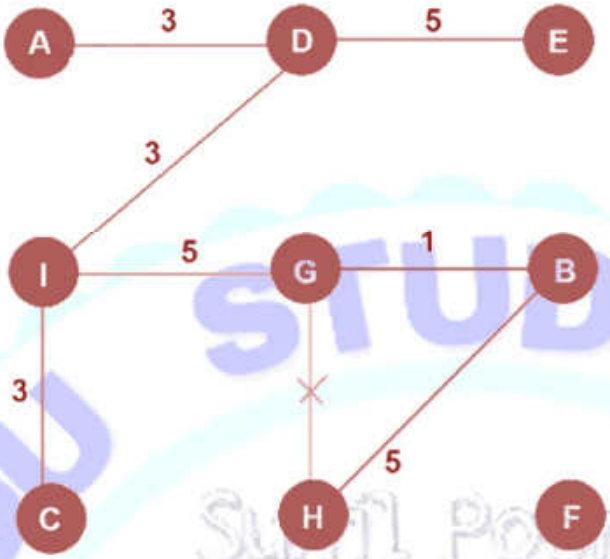
Step: 8 Following edge CH, we have edge EG. This edge also creates the loop; hence we will discard it.



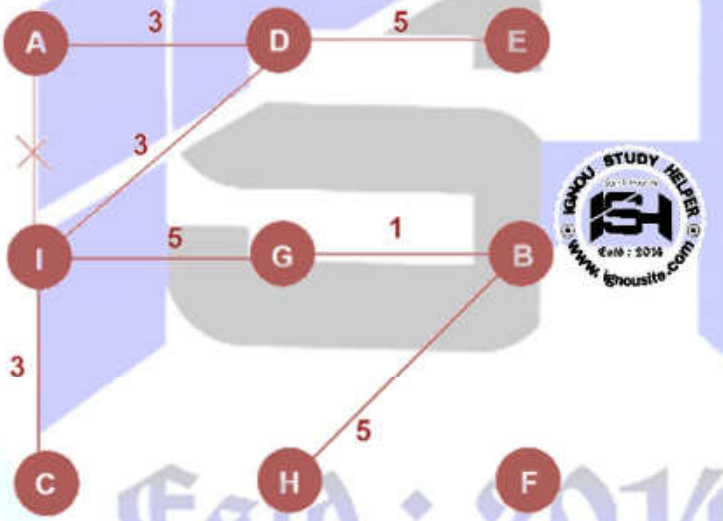
Step: 9 Following edge EG, we have edge BE. This edge also creates the loop; hence we will discard it



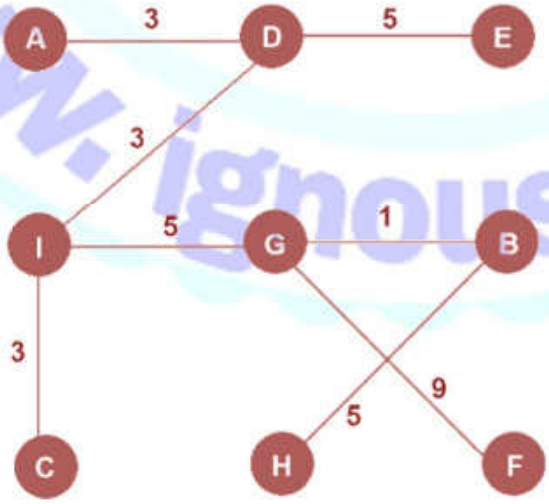
Step: 10 Following edge BE, we have edge GH. This edge also creates the loop; hence we will discard it.



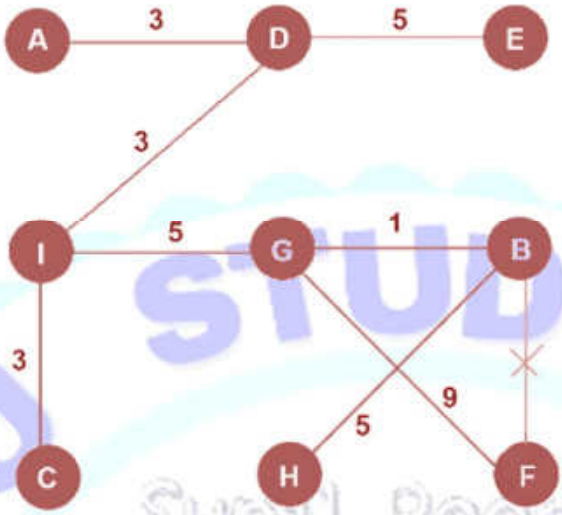
Step: 11 Following edge GH, we have edge AI. This edge also creates the loop; hence we will discard it.



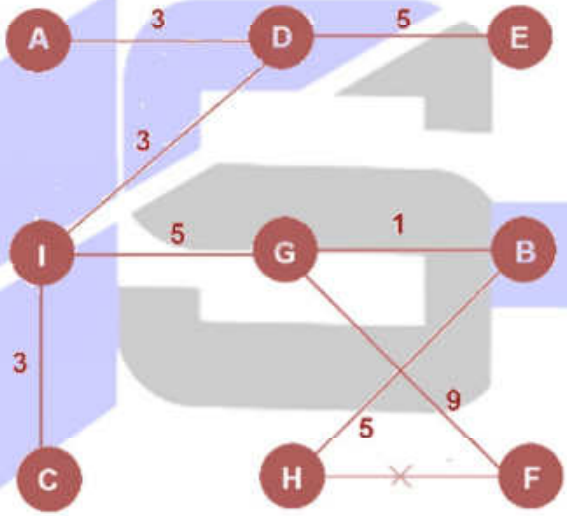
Step: 12 Add edge GF as it does not generate any loop.



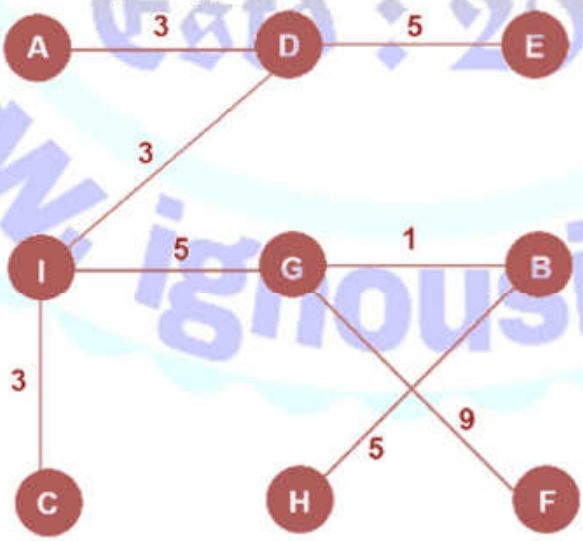
Step: 13 Following edge GF, we have edge BF. This edge also creates the loop; hence we will discard it.



Step: 14 Following edge BF, we have edge HF. This edge also creates the loop; hence we will discard it.



Step: 15 Therefore, the final minimum spanning tree is as follows:



Hence, the minimum cost = 1 + 3 + 3 + 3 +5 + 5 + 5 + 9 = 34.

Q8. Sort the following sequence using Bubble Sort algorithm. Show all the intermediate steps/ passes involved in sorting.

25	27	15	35	13	50	33	14	40	27

Ans.

Input: arr[] = {25, 27, 15, 35, 13, 50, 33, 14, 40, 27}

First Pass:

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

(**25, 27**, 15, 35, 13, 50, 33, 14, 40, 27) –> (**25, 27**, 15, 35, 13, 50, 33, 14, 40, 27),

Here, algorithm compares the first two elements, and does not swap them since 27 > 25.

(25, **27, 15**,35, 13, 50, 33, 14, 40, 27) –> (25, **15, 27**, 35, 13, 50, 33, 14, 40, 27),

Swap since 27 > 15

(25, 15, **27, 35**, 13, 50, 33, 14, 40, 27) –> (25, 15, **27, 35**, 13, 50, 33, 14, 40, 27),

No Swap since 35> 27

(25, 15, 27, **35, 13**, 50, 33, 14, 40, 27) –> (25, 15, 27,**13, 35**, 50, 33, 14, 40, 27),

Swap since 35 >13.

(25, 15, 27, 13, **35, 50**, 33, 14, 40, 27) –> (25, 15, 27, 13, **35, 50**, 33, 14, 40, 27),

No swap Since 50> 35

(25, 15, 27, 13, 35, **50, 33**, 14, 40, 27) –> (25, 15, 27, 13,35**, 33, 50**, 14, 40, 27),

Swap Since 50 >33

(25, 15, 27, 13, 35, 33, **50, 14**, 40, 27) –> (25, 15, 27, 13, 35, 33, **14, 50**, 40, 27),

Swap Since 50 >14

(25, 15, 27, 13, 35, 33,14, **50, 40**, 27) –> (25, 15, 27, 13, 35, 33, 14, **40, 50**, 27) ,

Swap Since 50 >40

(25, 15, 27, 13, 35, 33, 14, 40, **50, 27**) –> (25, 15, 27, 13, 35, 33, 14, 40, **27, 50**).

Swap Since 50 > 27

Second Pass:

Now, during second iteration it should look like this:

(**25, 15**, 27, 13, 35, 33, 14, 40, 27, 50) –> (**15, 25**, 27, 13, 35, 33, 14, 40, 27, 50),

Swap since 25 > 15

(15, **25, 27**, 13, 35, 33, 14, 40, 27, 50) –> (15, **25, 27**, 13, 35, 33, 14, 40, 27, 50),

No swap since 27> 25

(15, 25, **27, 13**, 35, 33, 14, 40, 27, 50) –> (15, 25, **13, 27**, 35, 33, 14, 40, 27, 50),

Swap since 27 > 13

(15, 25,13, **27, 35**, 33, 14, 40, 27, 50) –> (15, 25, 13, **27, 35**, 33, 14, 40, 27, 50),

No swap since 35> 27

(15, 25, 13, 27, **35, 33**, 14, 40, 27, 50)–> (15, 25, 13, 27, **33, 35**, 14, 40, 27, 50),

Swap since 35 >33

(15, 25, 13, 27, 33, **35, 14**, 40, 27, 50)–> (15, 25, 13,27, 33, **14, 35**, 40, 27, 50),

Swap since 35 >14

(15, 25, 13, 27, 33, 14, **35, 40,** 27, 50)–> (15, 25, 13, 27, 33, 14, **35, 40**, 27, 50),

No swap since 40> 35

(15, 25, 13, 27, 33, 14, 35, **40, 27**, 50)–> (15, 25, 13,27, 33, 14, 35 ,**27, 40**, 50),

Swap since 40 >27

(15, 25, 13, 27, 33, 14, 35, 27, **40, 50**)–> (15, 25, 13,27, 33, 14, 35,27, **40, 50**),

No swap since 50> 40

Third Pass:

Now, during third iteration it should look like this:

(**15, 25**, 13, 27, 33, 14, 35, 27, 40, 50) –> (**15, 25**, 13, 27, 33, 14, 35, 27, 40, 50),

No swap since 25> 15

(15, **25, 13**, 27, 33, 14, 35, 27, 40, 50) –> (15, **13, 25**, 27, 33, 14, 35, 27, 40, 50) ,

Swap since 25 >13

(15, 13, **25, 27**, 33, 14, 35, 27, 40, 50)–> (15, 13, **25, 27**, 33, 14, 35, 27, 40, 50),

No swap since 27 >25

(15, 13, 25, **27, 33**, 14, 35, 27, 40, 50)–>(15, 13, 25, **27, 33**, 14, 35, 27, 40, 50),

No swap since 33> 27

(15, 13, 25, 27, **33, 14**, 35, 27, 40, 50)–> (15, 13, 25, 27, **14, 33**, 35, 27, 40, 50),

Swap since 33 >14

(15, 13, 25, 27, 14, **33, 35**, 27, 40, 50) –> (15, 13, 25, 27, 14, **33, 35**, 27, 40, 50),

No swap since 35> 33

(15, 13, 25, 27, 14, 33, **35, 27**, 40, 50)–> (15, 13, 25, 27, 14, 33, **27, 35**, 40, 50),

Swap since 35 >27

(15, 13, 25, 27, 14, 33, 27, **35, 40**, 50) –> (15, 13, 25, 27, 14, 33, 27, **35, 40**, 50),

No swap since 40> 35

(15, 13, 25, 27, 14, 33, 27, 35, **40, 50**)–> (15, 13, 25, 27, 14, 33, 27, 35, **40, 50**),

No swap since 50> 40

Fourth Pass:

Now, during fourth iteration it should look like this:

(**15, 13**, 25, 27, 14, 33, 27, 35, 40, 50) –> (**13, 15**, 25, 27, 14, 33, 27, 35, 40, 50),

Swap since 15 > 13

(13, **15, 25**, 27, 14, 33, 27, 35, 40, 50) –> (13, **15, 25**, 27, 14, 33, 27, 35, 40, 50),

No swap since 25> 15

(13, 15, **25, 27**,14, 33, 27, 35, 40, 50)–> (13, 15, **25, 27**, 14, 33, 27, 35, 40, 50),

No swap since 27 >25

(13, 15, 25,**27,14**, 33, 27, 35, 40, 50)–> (13, 15, 25, **14,27**, 33, 27, 35, 40, 50),

Swap since 27 > 14

13, 15, 25, 14, 27,**33, 27**, 35, 40, 50) –> (13, 15, 25, 14,27,**33, 27**, 35, 40, 50)

No swap since 33 >27

(13, 15, 25, 14, 27, **33,27**,35, 40, 50) –> (13, 15, 25, 14, 27, **27,33**,35, 40, 50),

Swap since 33 > 27

(13, 15, 25, 14, 27, 27, **33,35**, 40, 50)–> (13, 15, 25, 14, 27, 27, **33,35**, 40, 50),

No swap since 35> 33

(13, 15, 25, 14, 27, 27, 33,**35, 40**, 50) –> (13, 15, 25, 14, 27, 27, 33, **35, 40**,50),

No swap since 40> 35

(13, 15, 25, 14, 27, 27, 33, 35, **40, 50**)–> (13, 15, 25, 14, 27, 27, 33, 35, **40, 50**),

No swap since 50> 40

Fifth Pass:

Now, during fourth iteration it should look like this:

(**13, 15**,25, 14, 27, 27, 33, 35, 40, 50) –> (**13, 15**,25, 14, 27, 27, 33, 35, 40, 50),

No swap since 15 >13

(13, **15, 25**, 14, 27, 27, 33, 35, 40, 50) –> (13, **15, 25**, 14, 27, 27, 33, 35, 40, 50),

No swap since 25> 15

(13, 15, **25,14**, 27, 27, 33, 35, 40, 50)–> (13, 15, **14,25**, 27, 27, 33, 35, 40, 50),

Swap since 25 > 14

(13, 15, 14, **25, 27**, 27, 33, 35, 40, 50)–> (13, 15, 14, **25, 27**, 27, 33, 35, 40, 50),

No swap since 27 >25

(13, 15, 14, 25, **27,27**, 33, 35, 40, 50) –> (13, 15, 14, 25, **27,27**, 33, 35, 40, 50)

No swap since 27 >27

(13, 15, 14, 25, 27, **27, 33**, 35, 40, 50) –> (13, 15, 14, 25, 27,**27, 33**, 35, 40, 50),

No swap since 33 >27

(13, 15, 14, 25, 27, 27, **33, 35**, 40, 50) –> (13, 15, 14, 25, 27, 27, **33, 35**, 40, 50),

No swap since 35> 33

(13, 15, 14, 25, 27, 27, 33, **35, 40**, 50) –> (13, 15, 14, 25, 27, 27, 33, **35, 40**, 50),

No swap since 40> 35

(13, 15, 14, 25, 27, 27, 33, 35, **40, 50**)–> (13, 15, 14, 25, 27, 27, 33, 35, **40, 50**),

No swap since 50> 40

Sixth Pass:

Now, during fourth iteration it should look like this:

(**13, 15**,14, 25, 27, 27, 33, 35, 40, 50) –> (**13, 15**,14, 25, 27, 27, 33, 35, 40, 50),

No swap since 15 >13

(13, **15, 14**, 25, 27, 27, 33, 35, 40, 50) –> (13, **14, 15**, 25, 27, 27, 33, 35, 40, 50),

Swap since 15 > 14

(13, 14, **15,25**, 27, 27, 33, 35, 40, 50)–> (13, 14, **15,25**, 27, 27, 33, 35, 40, 50),

No swap since 25> 15

(13, 14, 15, **25, 27**, 27, 33, 35, 40, 50)–> (13, 14, 15, **25, 27**, 27, 33, 35, 40, 50),

No swap since 27 >25

(13, 14, 15, 25, **27,27**, 33, 35, 40, 50) –> (13, 14, 15, 25, **27,27**, 33, 35, 40, 50)

No swap since 27 >27

(13, 14, 15, 25, 27, **27, 33**, 35, 40, 50) –> (13, 14, 15, 25, 27, **27, 33**, 35, 40, 50),

No swap since 33> 27

(13, 14, 15, 25, 27, 27**, 33, 35**, 40, 50) –> (13, 14, 15, 25, 27, 27, **33, 35**, 40, 50),

No swap since 35> 33

(13, 14, 15, 25, 27, 27, 33, **35, 40**, 50) –> (13, 14, 15, 25, 27, 27, 33, **35, 40**, 50),

No swap since 40> 35

(13, 14, 15, 25, 27, 27, 33, 35, **40, 50**)–> (13, 14, 15, 25, 27, 27, 33, 35, **40, 50**),

No swap since 50> 40

Seventh Pass:

Now, the array is already sorted, but our algorithm does not know if it is completed.
The algorithm needs one whole pass without any swap to know it is sorted.
(**13, 14**,15, 25, 27, 27, 33, 35, 40, 50) –> (**13, 14**,15, 25, 27, 27, 33, 35, 40, 50),

No swap since 14 >13

(13, **14, 15**, 25, 27, 27, 33, 35, 40, 50) –> (13, **14, 15**, 25, 27, 27, 33, 35, 40, 50),

No swap since 15 >14

(13, 14, **15,25**, 27, 27, 33, 35, 40, 50)–> (13, 14, **15,25**, 27, 27, 33, 35, 40, 50),

No swap since 25> 15

(13, 14, 15, **25, 27**, 27, 33, 35, 40, 50)–> (13, 14, 15, **25, 27**, 27, 33, 35, 40, 50),

No swap since 27 >25

(13, 14, 15, 25, **27,27**, 33, 35, 40, 50) –> (13, 14, 15, 25, **27,27**, 33, 35, 40, 50)

No swap since 27 >27

(13, 14, 15, 25, 27, **27, 33**, 35, 40, 50) –> (13, 14, 15, 25, 27, **27, 33**, 35, 40, 50),

No swap since 33> 27

(13, 14, 15, 25, 27, 27, **33, 35**, 40, 50) –> (13, 14, 15, 25, 27, 27**, 33, 35**, 40, 50),

No swap since 35> 33

(13, 14, 15, 25, 27, 27, 33, **35, 40**, 50) –> (13, 14, 15, 25, 27, 27, 33, **35, 40**, 50),

No swap since 40> 35

(13, 14, 15, 25, 27, 27, 33, 35, **40, 50**)–> (13, 14, 15, 25, 27, 27, 33, 35, **40, 50**),

No swap since 50> 40

Hence, the final sorted array is the following:

{13, 14, 15, 25, 27, 27, 33, 35, 40, 50}

Q9. (a) Write general form of Divide & Conquer and Greedy Techniques.

Ans. Divide And Conquer: This technique can be divided into the following three parts:

1. Divide: This involves dividing the problem into smaller sub-problems.
2. Conquer: Solve sub-problems by calling recursively until solved.
3. Combine: Combine the sub-problems to get the final solution of the whole problem.

```
DAC(a, i, j)
{
  if(small(a, i, j))
   return(Solution(a, i, j))
  else
   m = divide(a, i, j)          // f1(n)
   b = DAC(a, i, mid)           // T(n/2)
   c = DAC(a, mid+1, j)         // T(n/2)
   d = combine(b, c)            // f2(n)
  return(d)
}
```

Greedy Techniques: Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are the best fit for Greedy.

```
Algorithm Greedy (C, n)

/* Input: A input domain (or Candidate set ) C of size n, from which solution is to be Obtained. */

// function select (C: candidate_set) return an element (or candidate).

// function solution (S: candidate_set) return Boolean

// function feasible (S: candidate_set) return Boolean

/* Output: A solution set S, where S⊆C, which maximize or minimize the selection

criteria w. r. t. given constraints */
{
S ← Φ          // Initially a solution set S is empty.
```

While (not solution (S) and C ≠ Φ)

{

x ← select (C) /* A "best" element x is selected from C which maximize or minimize the selection criteria. */

C ← C – {x} /* once x is selected, it is removed from C

if (feasible (S ∪ {x}) then /* x is now checked for feasibility

S ← S– {x}

}

If (solution (S))

return S;

else

return "No Solution"

} // end of while

(b) Formulate a Knapsack problem and apply it to find an optimal solution for the following Knapsack problem. Use any two approaches:

- **Kapacity of Knapsack: 23**
- **Number of Objects: 7**

Profits P_i and weights W_i of objects are defined as follows:

$(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (13, 11, 18, 9, 7, 22, 5)$

$(W_1, W_2, W_3, W_4, W_5, W_6, W_7) = (7, 6, 8, 4, 6, 3, 5)$

Ans.

Given, number of object, $n = 7$

Capacity of knapsack, $m = 23$

$(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (13, 11, 18, 9, 7, 22, 25)$

$(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (7, 6, 8, 4, 6, 3, 5)$

To Solve this problem, Greedy method may apply any one of the following strategies:

→ From the remaining object, select the object with maximum profit that fit into the knapsack.

→ From the remaining object, select the object that has minimum weight and also fits into knapsack.

→ From the ~~remaining~~ remaining object, select the object with maximum that fits into the knapsack.

As per the question's requirement, I am going to solve this problem by the 3rd approache which ~~the~~ are mentioned above.

Approach 3 - From the remaining object, select the object with maximum $\frac{p_i}{w_i}$ that fits into the knapsack.

In this approach, we select those object first which has maximum value of $\frac{p_i}{w_i}$ that is we select those ~~of~~ object first which has maximum profit per unit weight.

Since, $\left(\frac{p_1}{w_1}, \frac{p_2}{w_2}, \frac{p_3}{w_3}, \frac{p_4}{w_4}, \frac{p_5}{w_5}, \frac{p_6}{w_6}, \frac{p_7}{w_7}\right) = (1.86, 1.83, 2.25, 2.25, 1.17, 7.33, 5.0)$.

Thus, we must select first 6th object then 7th after that 3rd then 4th after that 1st then 2nd and at last 5th object

The Solution of the problem is given below:

Knapsack problem (fractional)

n=7 object Oi 1 2 3 4 5 6 7

profit Pi 13 11 18 9 7 22 25

weight wi 7 6 8 4 6 3 5

$\frac{P_i}{w_i}$ = 1.86 1.83 2.25 2.25 1.17 7.33 5.0

x_i = $\frac{3}{7}$ 0 1 1 0 1 1

where $0 \le x_i \le 1$ (To get the maximum profit)

constraints: Total capacity of knapsack:

$\sum_{i=1}^{7} x_i w_i \le m (=23)$

objective: maximized total profit:

$\sum_{i=1}^{7} x_i P_i$

Total weight $\sum_{i=1}^{7} x_i w_i = x_1 w_1, x_2 w_2, x_3 w_3, x_4 w_4, x_5 w_5, x_6 w_6, x_7 w_7$

$= (\frac{3}{7} \times 7) + (0 \times 6) + (1 \times 8) + (1 \times 4) + (0 \times 6) + (1 \times 3) + (1 \times 5)$

$= 3 + 8 + 4 + 3 + 5 = 23$ unit

maximum total profit,

$\sum_{i=1}^{7} x_i P_i = x_1 P_1, x_2 P_2, x_3 P_3, x_4 P_4, x_5 P_5, x_6 P_6, x_7 P_7$

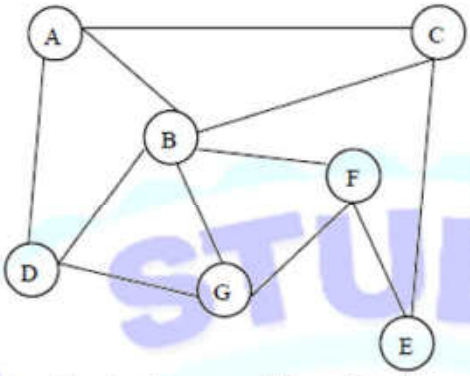
$= (\frac{3}{7} \times 13) + (0 + 11) + (1 \times 18) + (1 \times 9) + (0 \times 7) + (1 \times 22) + (1 \times 25)$

$= \frac{557}{7} =$ Rs. 79.57

In this approach we have total profit $\sum_{i=1}^{7} x_i P_i$ = Rs 79.57, and the optimal solution set

$(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = (\frac{3}{7}, 0, 1, 1, 0, 1, 1)$ Ans

Q10. What is graph? List few applications of graph traversal schemes. For the following graph write adjacency list and adjacency matrix.



Ans. Graph: A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices (V) and a set of edges(E). The graph is denoted by G(E, V).

List few applications of graph traversal schemes: The graph has two types of traversal schemes. These are called the Breadth First Search and Depth First Search.

Breadth First Search (BFS):

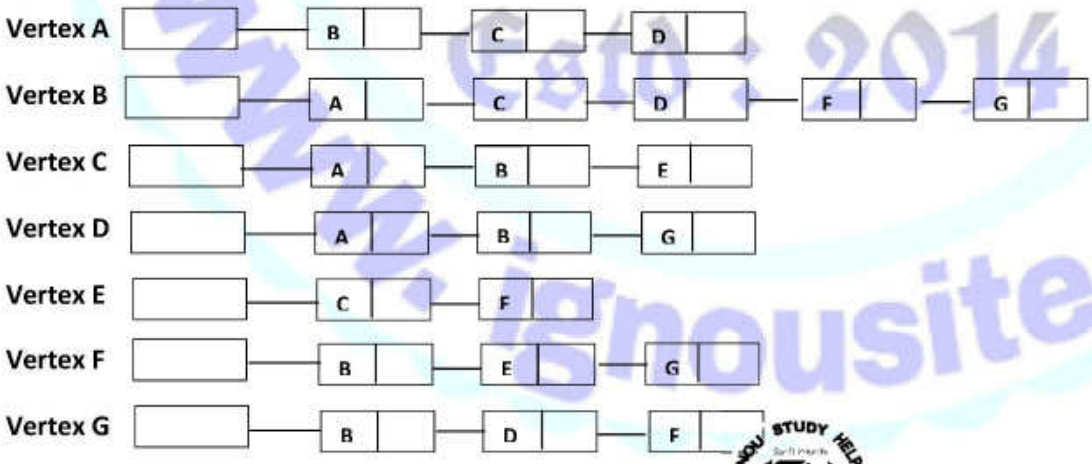
The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.

Depth First Search (DFS):

The Depth First Search (DFS) is a graph traversal algorithm. In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

Adjacency list and adjacency matrix:

Adjacency list:



Adjacency matrix:

	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	1	1	0	1	1
C	1	1	0	0	1	0	0
D	1	1	0	0	0	0	1
E	0	0	1	0	0	1	0
F	0	1	0	0	1	0	1
G	0	1	0	1	0	1	0