Unit 1: Fundamentals of Algorithm

Unit at a glance:

1.1 Classification of Data Type

Data type can be classified as ***primitive data type*** and ***abstract data type.***

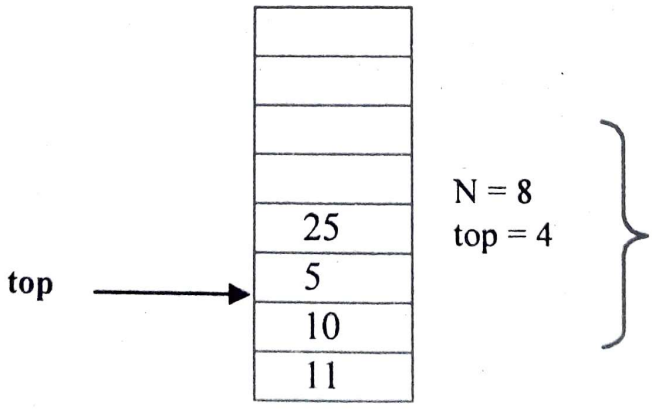
- **Primitive data type:** In the phrase *primitive data type* the word **primitive** means "a fundamental component that is used to create other, larger parts." In terms of the programming language, **primitive data type** is either of the following:
- **Abstract data type** (ADT): *Abstract Data Types* are a set of data values and associated operations that are precisely independent of any particular implementation. The term abstract signifies that the data type will only set the rule of it's usage but how it will be used depends on the implementation. For example stacks and queues are called abstract data types. The stack data type defines two abstract methods PUSH and POP.
- **Atomic data types:** An atomic data type can contain content whose values are not composed of values of other data types.

1.2 Stack

A stack is an abstract data type which follows LIFO (Last In First Out) data structure. It declares two methods PUSH and POP. Stacks are implemented either by an array or by a linked list.

1.3 Array Representation

Stacks are abstract data types. They are implemented either by an array or a linked list. In array representation, a pointer or a variable top is used to keep track of the stack. A variable N is used to store the maximum number of elements in the stack.



1.4 Various Types of Expressions

A mathematical expression involves constants (operands) and operations (operators).

- Infix notation: **operand1 *operator* operand2,** **A + B**
- Prefix notation: ***operator*** **operand1 operand2,** **+ A B**

- Postfix notation: **operand operator**

Conversion from INFIX to POSTFIX

In order to convert the infix to its corresponding postfix form; we have to follow the following steps:

i) Fully parenthesize the expression according to the priority of different operators.
ii) Move all operators so that they replace their corresponding right parentheses.
iii) Delete all parentheses.

The priorities of different operators are given below:

Operators	Priority
Unary - , unary + , not (!)	4
* , / , % , and (& / &&)	3
+ , - , or (| / ||)	2
<, <=, >, >=, = =, !=	1

Let us consider the following expression:

4 / 2 + 3 * 7 – 12 % 2

As per the priority table, after parenthesizing the expression we get

(((4 / 2) + (3 * 7)) - (12 % 2))

After moving the operators corresponding to the parentheses we get

(((4 2 /) + (3 7 *)) - (12 2 %))

= 4 2 / 3 7 * + 12 2 % -

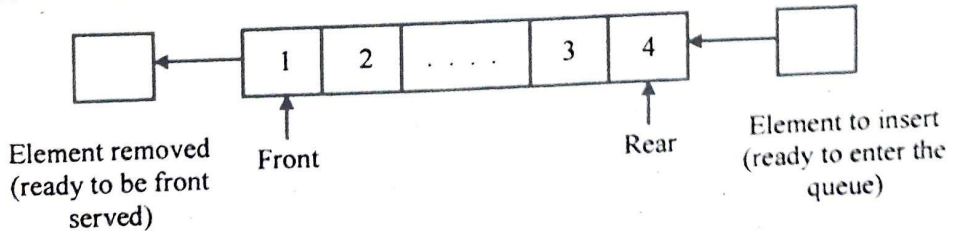
which is the required postfix form.

1.5 Queues

A queue is an abstract data type, which follows FIFO (First In First Out) data structure. It declares two methods QINSERT and QDELETE. Queues are implemented either by an array or by a linked list.

Array Representation

Queues are abstract data types. They are implemented either by an array or a linked list. In array representation, a pointer or a variable top is used to keep track of the stack. A variable N is used to store the maximum number of elements in the QUEUE.



value, which denotes the priority of that element. The priority determines the order in which they exit the queue. The element, having the highest priority, will be removed first. ***Dequeue*** means double-ended queue. In these types of queues insertion and deletion can be done from both ends.

1.7 Algorithm

An algorithm is a finite sequential set of instructions which, if followed, accomplish a particular task or a set of tasks in finite time. Algorithms are used for calculation, data processing, and automated reasoning.

1.8 Complexity

There are two types of complexities of an algorithm, time complexity and space complexity.

- The ***time complexity*** of an algorithm is the amount of time the computer requires to execute the algorithm.
- The ***space complexity*** of an algorithm is the amount of memory space the computer requires, completing the execution of the algorithm.

1.9 Big OH Notation

The Big Oh (the "O" stands for "order of") notation is used to classify functions by their asymptotic growth function and hence finding the time complexity of an algorithm

Short Answer Type Questions

A. Choose the correct answer from the given alternatives in each of the following:

1. Which of the following is asymptotically smallest? **[WBSCTE 2022]**
   (a) n (b) log n (c) n log n (d) 2n
   Answer: (b)

2. A linear list in which elements can be added or removed at either end but not in the middle is known as **[Model Question]**
   (a) queue (b) deque (c) stack (d) tree
   Answer: (b)

3. The prefix expression for the infix expression a * (b + c) / e – f is [Model Question]
(a) /*a + bc –ef (b) –/* + abcef (c) –/*a + bcef (d) None of these
Answer: (a)

4. The number of stacks required to implement mutual recursion is [Model Question]
(a) 3 (b) 2 (c) 1 (d) none of these
Answer: (c)

5. Priority queue can be implemented using [Model Question]
(a) array (b) linked list (c) heap (d) all of these
Answer: (d)

6. Reserve Polish notation is often known as [Model Question]
(a) Infix (b) Prefix (c) Postfix (d) none of these
Answer: (c)

7. If $f(n) = 5n^3 + 4n^2 - 8n + 100$, then $f(n) =$ _____ [Model Question]
(a) O(1) (b) O(n) (c) $O(n^2)$ (d) $O(n^3)$
Answer: (d)

B. Fill in the blanks in the following statements:

8. LIFO scheme is used in ***queue*** data structure. [WBSCTE 2022]

9. An algorithm is a ***procedure*** to solve a problem. [WBSCTE 2022]

10. Element of queue will be deleted from ***front*** end [WBSCTE 2022]

11. O-notation provides an asymptotic ***notation***. [WBSCTE 2022]

12. Big O notation derives the ***worst*** case. [WBSCTE 2022]

13. Ω notation gives the lower bound of the function f(n). [WBSCTE 2022]

14. If algorithm takes $O(n^2)$, it is faster for sufficiently larger n than if it had taken $O(2^n)$. [WBSCTE 2022]

15. ***GCD (m,n)= GCD (n, m mod n)*** is the formula used in Euclid's algorithm for finding the greatest common divisor of two number. [WBSCTE 2022]

C. Answer the following questions:

16. What is the Greedy method? [WBSCTE 2022]
Answer: A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.

17. What is data abstraction? [WBSCTE 2022]

Answer:

An Abstract Data Type in data structure is a kind of a data type whose behavior is defined with the help of some attributes and some functions.

18. Describe best case time complexity. [WBSCTE 2022]

Answer:

It defines the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.

19. Define 'Big Theta' Notation. [WBSCTE 2022]

Answer:

Big – Theta(Θ) notation specifies asymptotic bounds (both upper and lower) for a function f(n) and provides the average time complexity of an algorithm.

20. Define Dynamic Programming algorithm. [WBSCTE 2022]

Answer:

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property.

21. What do you mean by space complexity? [WBSCTE 2022]

Answer:

Space complexity refers to the total amount of memory space used by an algorithm/program, including the space of input values for execution. Calculate the space occupied by variables in an algorithm/program to determine space complexity.

22. What is Omega Notation & give example? [WBSCTE 2022]

Answer:

Omega notation specifically describes best case scenario. It represents the lower bound running time complexity of an algorithm. So if we represent a complexity of an algorithm in Omega notation, it means that the **algorithm cannot be completed in less time than this**, it would at-least take the time represented by Omega notation or it can take more (when not in best case scenario).

Definition (Big-Omega, Ω()): Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers. We say that $f(n)$ is $\Omega(g(n))$ (or $f(n) \in \Omega(g(n))$) if there exists a real constant c>0 and there exists an integer constant $n_0 \geq 1$ such that $f(n) \geq c.g(n)$ for every integer $n \geq n_0$.

Definition (Little-Omega, ω()): Let $f(n)$ and $g(n)$ be functions that map positive integers to positive real numbers. We say that $f(n)$ is $\omega g(n)$ (or $f(n) \in \Omega(g(n))$) if for any real constant c>0, there exists an integer constant $n_0 \geq 1$ such that $f(n) > c.g(n)$ for every integer $n \geq n_0$.

23. What is the difference between greedy method and divide conquer method?

[Model Question]

Answer:

In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

24. Define dequeue?

OR,

What is dequeue?

[Model Question]

Answer:

Dequeue is a linear data structure, where insertions and deletions are made to or from either end of the structure.

Long Answer Type Questions

➲ 1. What is an algorithm and what is the Complexity of the Algorithm?

[WBSCTE 2022]

Answer:

Algorithm

An algorithm is a finite set of instructions, those if followed, accomplishes a particular task. It is not language specific, we can use any language and symbols to represent instructions.

The criteria of an algorithm

- **Input:** Zero or more inputs are externally supplied to the algorithm.

- **Output:** At least one output is produced by an algorithm.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Finiteness:** In an algorithm, it will be terminated after a finite number of steps for all different cases.
- **Effectiveness:** Each instruction must be very basic, so the purpose of those instructions must be very clear to us.

Analysis of algorithms

Algorithm analysis is an important part of computational complexities. The complexity theory provides the theoretical estimates for the resources needed by an algorithm to solve any computational task. Analysis of the algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of the analysis of the algorithm is the required time or performance.

Complexities of an Algorithm

The complexity of an algorithm computes the amount of time and spaces required by an algorithm for an input of size (n). The complexity of an algorithm can be divided into two types. The **time complexity** and the **space complexity**.

Time Complexity of an Algorithm

The time complexity is defined as the process of determining a formula for total time required towards the execution of that algorithm. This calculation is totally independent of implementation and programming language.

Space Complexity of an Algorithm

Space complexity is defining as the process of defining a formula for prediction of how much memory space is required for the successful execution of the algorithm. The memory space is generally considered as the primary memory.

➲ **2. Write Divide and Conquer algorithm with detailed explanation.**

[WBSCTE 2022]

Answer: ***Refer to Question no. 6 of Long Answer Type Questions.***

➲ **3. a) What is a Stack ADT?**

b) Write a C function of popping an element from a stack implemented using linked list.

c) Explain three uses of stack data structure. **[Model Question]**

Answer:

a) A Stack ADT is a (ordered) collection of items, where all insertions are made to the end of the sequence and all deletions always are made from the end of the sequence. In principle a stack is a container of data items, from which we get data items out in reverse order compared to the order they have been put into the container. We can also say that the item that has been put last in is coming first out. That's why a stack is also called LIFO ((Last In First Out list). We can as well say that the item, which is put first in the container is get last out (First In Last Out: FILO).

b) Assume that the list is defined as below:

```
typedef struct node *nptr;
struct node
{
int data;
nptr next;
};
/* function pop */
int pop(nptr s) /*Function to pop the elements*/
{
nptr temp;
int y;
if(s->next==NULL)
          {
printf("Underflow on Pop");
return(-1);
}
Else
{
y=s->next->data;
temp=s->next;
s->next=temp->next;
free(temp);
return(y);
}
}
```

c) Use of stack

Reversing Data: We can use stacks to reverse data.
(example: files, strings). It is very useful for finding palindromes.
Consider the following pseudocode:

```
1)     read (data)
2)     loop (data not EOF and stack not full)
              1) push (data)
              2) read (data)
3)     Loop (while stack notEmpty)
              1) pop (data)
              2) print (data)
```

```
Converting Decimal to Binary:
Consider the following pseudocode
Read (number)
Loop (number > 0)
      1) digit = number modulo 2
      2) print (digit)
      3) number = number / 2
```

The problem with this code is that it will print the binary number backwards. (ex: 19 becomes 11001000 instead of 00010011.)

To remedy this problem, instead of printing the digit right away, we can push it onto the stack. Then after the number is done being converted, we pop the digit out of the stack and print it.

Evaluating arithmetic expressions.

In high level languages, infix notation cannot be used to evaluate expressions. We must analyze the expression to determine the order in which we evaluate it. A common technique is to convert a infix notation into postfix notation, then evaluating it. This is done using stack.

➲ **4. i) What is Circular queue?**

ii) Write Q-insert algorithm for the circular queue.

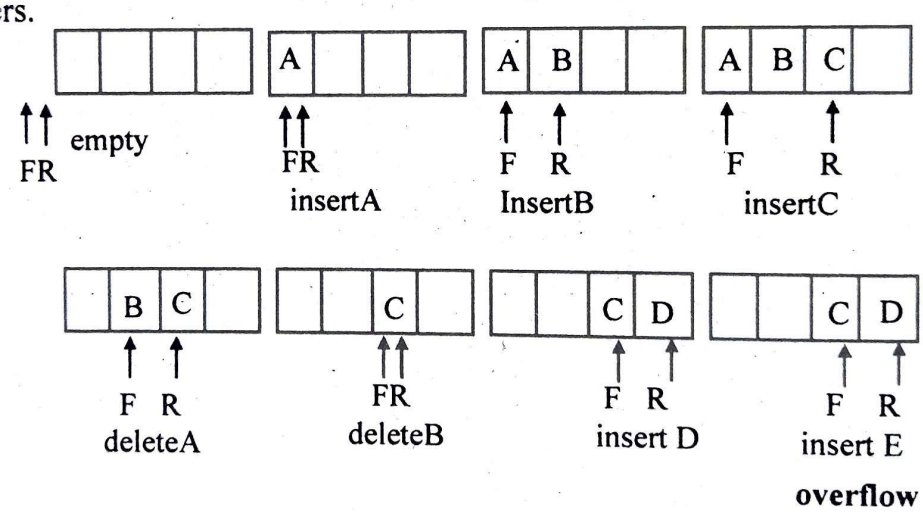
OR,

Write an algorithm to insert an element into circular queue. **[Model Question]**

Answer:

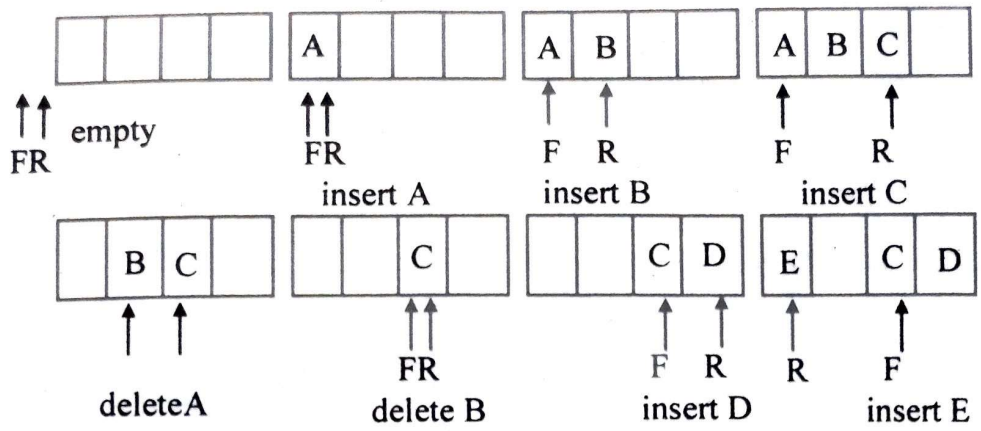
i) The two algorithms QINSERT & QDELETE can be very wasteful of storage if the front pointer F never manages to catch up the rear pointer. Actually an arbitrary large amount of memory would be required to accommodate the elements. The method of performing operations on a Queue should only be used when the queue is emptied at certain intervals.

Let us consider the following sequence of operations. F and R represents the front and rear pointers.



To avoid this problem we can think an alternative representation of a queue, which prevents an excessive use of memory. In this representation elements are arranged as in a circular fashion where the rear again points to the front. This type of queue is known ***circular queues***.

So the above sequence of operations can be represented are shown below (in circular queue).



As we can see that the overflow issue in the previous case while inserting E is resolved by redirecting the rear to the front. This is the essence of circular queue.

ii) Let us now write the insert and delete functions of the circular queue implementation. Let us also assume that F and R represents front and rear pointers respectively

```
void CQInsert(int item)
{
  if (rear == N-1)
  {
    printf("Queue Is Full");
    return;
  }
  CQ[++rear] = item;
  if (front ==  - 1)
    front = 0;
}
int CQDelete()
{
  int x;
  if (front ==  - 1)
  {
    printf("Queue Is Empty");
    exit(0);
  }
  x = CQ[front];
  CQ[front] =  - 1;
  if (front == rear)
    front = rear =  - 1;
  else
    front++;
  return x;
}
```

In linear queue the condition for queue full is

QREAR==MAXLIMIT

Suppose maxlimit is ten and queue is full now if we delete 9 element from queue then inspite of queue is empty we cannot insert any element in the queue. This wastage of memory is solved through circular queue where queue full condition is

QREAR==Qfront+1

➲ 5. Discuss different types of asymptotic notation. **[Model Question]**

Answer:

Big-O notation : Big-O notation is a theoretical measure of resource requirement of an algorithm. Usually time and memory needed, given the problem size n (which is usually the number of inputs) do count as resource. Informally speaking, some equation

f(n) = O(g(n)) means, f(n) is less than some constant multiple of g(n).

So, f(n) = O(g(n)) means there are positive constants c and k, such that

$0 \leq f(n) \leq cg(n)$ for all $n \geq k$. The value of c and k must be fixed for the function f and must not depend on n.

Ω *notation:* Ω notation is also a theoretical measure of the resource requirement of an algorithm. Usually time and memory needed, given the problem size n (which is usually the number of inputs) are treated as resource as before. Informally, saying some equation f(n) = Ω(g(n)) means it is more than some constant multiple of g(n). So, f(n) = Ω(g(n)) means there are positive constants c and k, such that $0 \leq cg(n) \leq f(n)$ for all $n \geq k$. The values of c and k must be fixed for the function f and must not depend on n.

***ω-notation*:** ω notation is also a theoretical measure of the resource requirement of an algorithm. Usually the time and memory needed, matter as resources, given the problem size n, (which is usually the number of inputs). Informally, saying some equation $f(n) = \omega(g(n))$ means $g(n)$ relative to $f(n)$ becomes insignificant as n goes to infinity. The notation is read as, "*f of n is little omega of g of n*".

Given any ε > 0 however small, there exists a $k > 0$ such that $\left|\frac{g(n)}{f(n)}\right| < \varepsilon$, whenever $n \geq k$. The value of k must not depend on n, but may depend on c. In other words, $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

Θ *notation*: This is also a theoretical measure of the execution of an algorithm, in terms of the time or memory needed, given the problem size n. Informally, saying some equation f(n) = Θ (g(n)) means it is within a spectrum induced by a pair of constant multiples of g(n). So, f(n) = Θ (g(n)) means there are positive constants c_1, c_2 and k, such

that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge k$. The values of c_1, c_2, and k must be fixed for the function f and must not depend on n.

Little-o notation: This is yet another theoretical measure of the execution of an algorithm with respect to time and / or memory resources needed, given the problem size n, which is usually the number of input items. Informally, f(n) = o(g(n)) means f(n) becomes insignificant relative to g(n) as n approaches infinity. The notation is read as, "*f of n is little oh of g of n*".

Given any $\varepsilon > 0$ however small, there exists a $k > 0$ such that $\left|\frac{f(n)}{g(n)}\right| < \varepsilon$, whenever $n \ge k$.

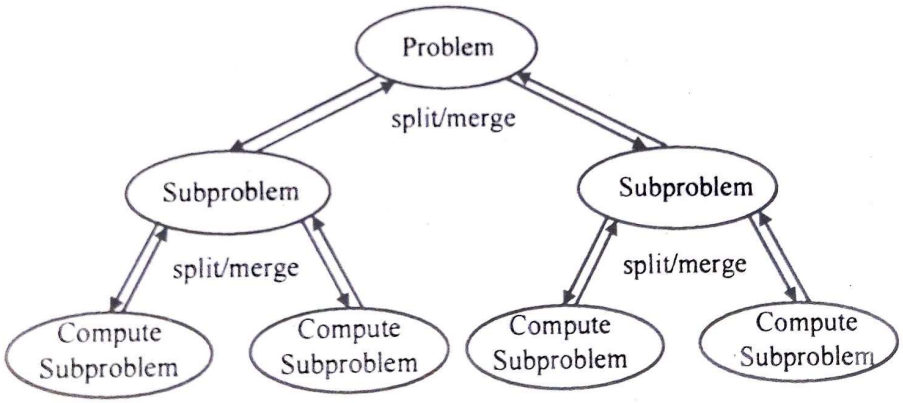
The value of k must not depend on n, but may depend on c. This is equivalent to $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.

➲ 6. Explain the basic concept of a divide-and-conquer algorithm.

[Model Question]

Answer:

The divide and conquer approach as the name suggests divides the given problem in parts and then each problem is solved independently. When we keep dividing the problem into smaller parts a moment comes when the problem cannot be divided further into smaller part, then those smaller parts are solved and the solution of all those smaller parts or sub-parts is finally merged to obtain the solution of the original problem.



Divide and Conquer Algorithm contains the following steps:

1) Divide

The first and foremost process for solving any problem using Divide and Conquer paradigm. As suggested by the name it's function is just to divide the problem into sub-problem which in turn if are more complex then are again divided into more sub-parts. Basically, if we consider for example binary search (an example of Divide and Conquer approach) the given list is broken (under some specified consider defined by its algorithm and user input) into single elements among which then the element to search is compared and the user gets a prompt whether the element is in the list.

2) Conquer

This process is referred to as ' Conquer ' because this process is what which performs the basic operation of a defined algorithm like sort in cases of various sorts, finds the element to be searched in case of binary search, multiplying of the numbers in Karatsuba Algorithm and etc. But almost among algorithms at least the most basic ones that we study mostly are considered to be solved in this part since the divide part breaks them into single elements which can be simply solved.

3) Merge

This is the last process of the **'Divide'** and **'Conquer'** approach whose function is to recursively rebuild the original problem but what we get now is the solution to that problem. In merge procedure, the solved elements of the 'Conquer 'are recursively merged together like the way they divided in the first step i.e. 'Divide'.

Application of divide and conquer approach include binary search, merge sort, quick sort etc.

➲ **7. What is the basic characteristic of a Greedy algorithm?** **[Model Question]**

Answer:

The characteristic of greedy method is basically same as that of solving a typical optimization problem preferably of time infeasible nature. The components of a typical greedy method are:

1. A set of elements like nodes, edges in a graph.
2. A set of elements which have already been used.
3. To test whether a given set of elements provide a solution or not. However, the solutions may not be optimal.
4. A selection function that picks up some elements which have not yet been used.
5. An objective function which associates a value to a solution.

➲ **8. a) What do you mean by dynamic programming?**

b) What is the difference between dynamic programming and greedy method?

[Model Question]

Answer:

a) Dynamic programming is a method for reducing the runtime of algorithms exhibiting the properties of overlapping sub problems and optimal substructure. Dynamic Programming is an approach developed to solve sequential, or multi-stage, decision problems; hence, the name "dynamic" programming. But, as we shall see, this approach is equally applicable for decision problems where sequential property is induced solely for computational convenience.

Dynamic programming is both a mathematical optimization method and a computer programming method. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.

Algorithms which can be solved by Dynamic programming are, Matrix-chain multiplication, All pair shortest paths, Single source shortest path, Travelling Salesman problem.

b) Greedy method is an algorithm that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems.

So the difference between Greedy method and Dynamic Programming are,

- Both techniques are optimization techniques, and both build solutions from a collection of choices of individual elements.
- The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.
- Dynamic programming computes its solution bottom up by synthesizing them from smaller subsolutions, and by trying many possibilities and choices before it arrives at the optimal set of choices.
- There is no a priori test by which one can tell if the Greedy method will lead to an optimal solution.
- By contrast, there is a prior test for Dynamic Programming, called The Principle of Optimality(A problem is said to satisfy the Principle of Optimality if the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems.), which can say if Dynamic Programming will lead to an optimal solution.

➲ **9. Show that the function $f(n)$ defined by:**

$$f(1) = 1$$

$$f(n) = f(n-1) + \frac{1}{n} \text{ for } n > 1$$

has the complexity O (log n).

Define Big – O, Ω, θ notations. Explain the conceptual differences among these three representatives.

And, Define Big O notation. **[Model Question]**

Answer:

Answer:

1st Part:

f(n) = f(n-1)+1/n

= f(n-2)+1/(n-1)+1/n

= f(n-3)+1/(n-2)+1/(n-1)+1/n

.....

.....

= f(2-1)+1/2+1/3+.....+1/(n-1)+1/n

= 1/1 + 1/2 +1/3 +.....+ 1/n

These types of numbers are called Harmonic numbers.

We can evaluate this type of series just by integrating 1/x from ½ to n + ½.

After integrating, we get the result as $\ln(n + ½) - \ln ½$

$$\approx \ln n - 0.7$$

Hence, we can write the complexity as O(log n).

2nd part:

Big O notation:

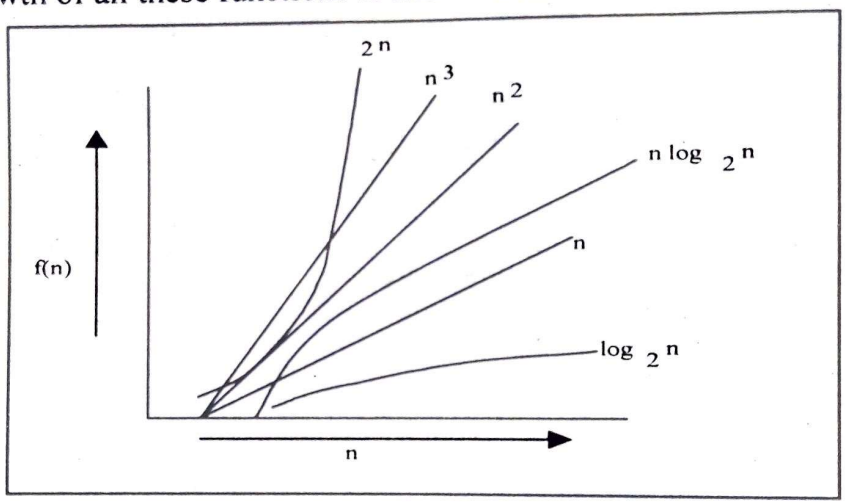
The Big Oh (the "O" stands for "order of") notation is used to classify functions by their asymptotic growth function and hence finding the time complexity of an algorithm.

There are two types of complexities of an algorithm, time complexity and space complexity. The time complexity of an algorithm is the amount of time the computer requires to execute the algorithm. The space complexity of an algorithm is the amount of memory the computer needs to run to complete the execution of the algorithm. The algorithms are compared based on their performances. The performance is measured in terms of time complexity & space complexity. Based on the nature of the input, time complexity can be of three different types: best case, average case and worst case.

The different computing functions are measured as:

$n, n^2, n^3, \log_2 n, n\log_2 n, 2^n$.

The rate of growth of all these functions is shown below:



Ω:

Let $f(n)$ and $g(n)$ be functions, where n is a positive integer. We write $f(n) = \Omega(g(n))$ if and only if $g(n) = O(f(n))$.

θ:

Let $f(n)$ and $g(n)$ be functions, where n is a positive integer. We write $f(n) = \theta(g(n))$ if and only if $g(n) = O(f(n))$. and $g(n) = \Omega(f(n))$.

➲ **10. Convert the infix expression $9+5*7-6\,\hat{}\,2+15/3$ into its equivalent postfix expression and evaluate that postfix expression, clearly showing the state of the stack.** **[Model Question]**

Answer:

Symbol scanned	Stack	Output
9		9
+	+	9
5	+	95
*	+*	95

Symbol scanned	Stack	Output
7	+*	957
-	-	957*+
6	-	957*+6
^	-^	957*+6
2	-^	957*+62
+	+	957*+62^-
15	+	957*+62^15
/	+/	957*+62^15
3	+/	957*+62^153
NONE	NONE	957*+62^153/+

Unit 2: Sorting

Unit at a glance:

2.1 Classification of Sorting

Sorting can be broadly classified into two categories.

- ***Internal sorting:*** It is the method when the sorting takes place within the main memory. The time required for read and write operations are considered to be insignificant e.g. Bubble Sort, Selection sort, Insertion sort etc.
- ***External sorting:*** It is the method when the sorting takes place with the secondary memory. The time required for read and write operations are considered to be significant e.g. sorting with disks, sorting with tapes.

2.2 Different Type of Sorting

2.2.1 Bubble Sort

Given an array of unsorted elements, Bubble sort performs a sorting operation on the first two adjacent elements in the array, then between the second & third, then between third & fourth & so on.

2.2.2 Selection Sort

In selection sorting we select the first element and compare it with rest of the elements. The minimum value in each comparison is swapped in the first position of the array. During each pass elements with minimum value are placed in the first position, then second then third and so on. We continue this process until all the elements of the array are sorted.

2.2.3 Insertion Sort

In insertion sort data is sorted data set by identifying an element that is out of order relative to the elements around it. It removes that element from the list, shifting all other elements up one place.

Finally it places the removed element in its correct location.

For example, when holding a hand of cards, players will often scan their cards from left to right, looking for the first card that is out of place. If the first three cards of a player's hand are 4, 5, 2, he will often be satisfied that the 4 and the 5 are in order relative to each other, but, upon getting to the 2, desires to place it before the 4 and the 5. In that case, the player typically removes the 2 from the list, shifts the 4 and the 5 one spot to the right, and then places the 2 into the first slot on the left.

2.2.4 Quick Sort

In Quick-Sort we divide the array into two halves. We select a pivot element (normally the middle element of the array) and perform a sorting in such a manner that all the elements to the left of the pivot element is lesser than it & all the elements to it's right is greater than the pivot element. Thus we get two sub arrays. One which is to the left of the pivot element having all elements lesser than the pivot element & another sub array to the right of the pivot element having elements greater than the pivot element. Then we recursively call the quick sort function on these two sub arrays to perform the necessary sorting.

2.2.5 Merge Sort

In this method, we divide the array or list into two sub arrays or sublists as nearly equal as possible and then sort them separately. Then the subarrays are again divided into another sub arrays. This process will continue until we get the subarrays containing single element. Then we merge the subarrays into a single unit in the same way as they are sorted. That is, we find 2 arrays of 1 element each; merge them into a sorted array of 2 elements. Each pair of resulting 2 elements array is then merged into a 4-element array. This process continues with larger and larger arrays until the entire array is sorted.

2.2.6 Heap Sort

A heap takes the form of a binary tree with the feature that the maximum or minimum element is placed in the root. Depending upon this feature the heap is called max-heap or min-heap respectively. After the heap construction the elements from the root are taken out from the tree and the heap structure is reconstructed. This process continues until the heap is empty.

2.2.7 Radix Sort

Radix sorting is a technique for ordering a list of positive integer values. The values are successively ordered on digit positions, from right to left. This is accomplished by copying the values into "buckets," where the index for the bucket is given by the position of the digit being sorted. Once all digit positions have been examined, the list must be sorted.

2.2.8 Shell Sort

Shell sort is named after its discoverer D.L. Shell. It is also called diminishing-increment sort. In case of insertion sort, we can move entries only one position because it compares only adjacent keys. To modify this method, we first compare keys that are far apart and then sort the entries far apart. Afterwards, we sort the entries, which are closer together, finally reducing the increment between keys being sorted to 1, to ensure that the file is completely sorted.

Short Answer Type Questions

A. Choose the correct answer from the given alternatives in each of the following:

1. Which of the following sorting algorithms is the fastest? **[WBSCTE 2022]**
   (a) Merge sort (b) Quick sort (c) Insertion sort (d) Shell sort

Answer: (b)

2. The given array is arr = {1, 2, 4, 3}. Bubble sort is used to sort the array elements. How many iterations will be done to short array? **[WBSCTE 2022]**
   (a) 4 (b) 2 (c) 1 (d) 0

Answer: (a)

3. Choose the incorrect statement about merge sort from the following **[WBSCTE 2022]**
   (a) it is a comparison based sort
   (b) it is an adaptive algorithm
   (c) it is not an in-place algorithm
   (d) it is a stable algorithm

Answer: (b)

4. What is a random QuickSort? **[WBSCTE 2022]**
   (a) The leftmost element is chosen as the pivot
   (b) The right most element is chosen as the pivot
   (c) Any element in the array is chosen as the pivot
   (d) A random number is generated which is used as the pivot

Answer: (c)

5. Which of the following algorithm implementations is similar to that of an insertion sort? **[WBSCTE 2022]**
   (a) Binary heap (b) Quick sort (c) Merge sort (d) Radix sort

Answer: (a)

6. Which of the following cases occurs when searching an array using linear search the value to be searched is equal to the first element of the array? **[Model Question]**
   (a) Worst case
   (b) Average case
   (c) Best case
   (d) Amortized case

Answer: (c)

7. In which sorting, consecutive adjacent pairs of elements in the array are compared with each other? **[Model Question]**
   (a) Bubble sort
   (b) Selection sort
   (c) Merge sort
   (d) Radix sort

Answer: (a)

8. The breadth first search algorithm for a connected graph requires [Model Question]
(a) a stack (b) a queue
(c) an array (d) none of these

Answer: (b)

9. In the worst case, a binary search tree will take how much time to search an element? [Model Question]
(a) $O(n)$ (b) $O(\log n)$ (c) $O(n^2)$ (d) $O(n \log n)$

Answer: (a)

B. Fill in the blanks in the following statements:

10. Running time complexity of heap sort is ***O(n logn)***. [WBSCTE 2022]

11. Time complexity of insertion sort is ***$O(n^2)$***. [WBSCTE 2022]

12. Worst case complexity of quick sort is ***n^2.*** [Model Question]

13. The best case time complexity of binary search is ***O(logn)***. [Model Question]

14. ***Merge sort*** is a type of external sort. [Model Question]

15. An item property positioned at each call to Quick sort is called a ***pivot.*** [Model Question]

16. Merge sort technique based on ***divide and conquer*** method. [Model Question]

17. ***Bubble*** sorting is called internal sorting. [Model Question]

C. Answer the following questions:

18. Define insertion sort. [WBSCTE 2022]

Answer:

Insertion sort is a sorting algorithm in which the elements are transferred one at a time to the right position.

19. What is pivot in Quick Sort? [WBSCTE 2022]

Answer:

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

20. What is Linear Time Sorting? [WBSCTE 2022]

Answer:

Although all comparision sorting algorithm requires at least $\Omega(n \log n)$ comparisons, in certain conditions, we are able to sort an array in $O(n)$ complexity.

21. What do you mean by sorting? [WBSCTE 2022]

Answer:

Sorting is the process of arranging data into meaningful order so that you can analyze it more effectively.

22. Why a sorting technique is called stable? [WBSCTE 2022]

Answer:

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. This means a sorting algorithm is called stable if two identical elements do not change the order during the process of sorting.

23. Which one is more efficient – Insertion Sort or Merge Sort? [WBSCTE 2022]

Answer:

Considering average time complexity of both algorithms we can say that Merge Sort is efficient in terms of time and Insertion Sort is efficient in terms of space.

24. "Sometimes bubble sort may work in linear time, though it is in $O(n^2)$ sort" – justify. [Model Question]

Answer:

When the array is already sorted, there is no sorting required and bubble sort will work in $O(n)$ complexity or in linear time, though it is in $O(n^2)$ sort.

25. When the quick sort algorithm takes $O(n^2)$ time? [Model Question]

Answer:

In worst case the quick sort algorithm takes $O(n^2)$.

Long Answer Type Questions

➲ **1. (a) Write an algorithm for the bubble sort method.**

(b) What down the complexity of merge sort? [WBSCTE 2022]

Answer:

a) ***Refer to Question No. 11 of Long Answer Type Questions.***

b) ***Refer to Question No. 6 of Long Answer Type Questions.***

➲ **2. Write an external sorting algorithm and explain its functionality with an example.**

OR,

Explain Merge Sort Algorithm with suitable explanation and its use.

[Model Question]

Answer:

Merge Sort is an External Sorting Technique.

Algorithm:

Algorithm of Merge Sort:

```
MERGE (M) //M is the list to be sorted //
VARIABLES LEFT , RIGHT , RESULT
    1.if  length(M) = 1
      1.1 return(M)
     Else
      1.2 VARIABLE  middle  ←length(M) / 2
      1.3 for each x in M up to middle add x to LEFT.
       1.4 for each x in M after middle add x to RIGHT.
    2. LEFT    ← MERGE (LEFT)
    3. RIGHT   ← MERGE (RIGHT)
    4. RESULT  ← MERGE (LEFT,RIGHT)
    5. return RESULT.
```

Example:

Consider the array of ten elements a[10]={44, 33, 11, 88, 55, 66, 99, 22, 75, 77}

Pictorially the file can now be viewed as

44	33	11	88	55	66	99	22	75	77
0	1	2	3	4	5	6	7	8	9

where vertical solid bars indicate the boundaries of the subarrays. Elements a[0] and a[1] are merged to yield

33	44	11	88	55	66	99	22	75	77
0	1	2	3	4	5	6	7	8	9

Then a[2] is merged with a[0], a[1] and

11	33	44	88	55	66	99	22	75	77
0	1	2	3	4	5	6	7	8	9

Is produced, Next, elements a[3] and a[4] are merged:

11	33	44	55	88	66	99	22	75	77
0	1	2	3	4	5	6	7	8	9

and then a[0]....a[2] and a[3], a[4]:

11	33	44	55	88	66	99	22	75	77
0	1	2	3	4	5	6	7	8	9

At this point the algorithm has returned to the first invocation of MergeSort and is about to process the second recursive call. Repeated recursive calls are invoked producing the following subarrays:

At this point there are two sorted subarrays and the final merge produces the fully sorted result.

11	22	33	44	55	66	75	77	88	99
0	1	2	3	4	5	6	7	8	9

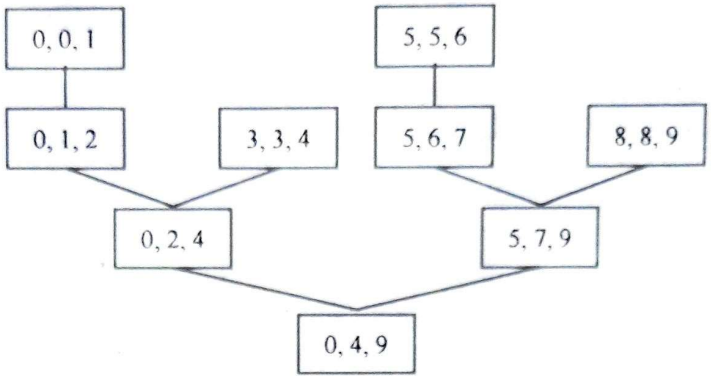


Fig: Tree of calls of Merge

The Mergesort algorithm can be used to sort a collection of objects. Mergesort is so called divide and conquer algorithm. Divide and conquer algorithms divide the original data into smaller sets of data to solve the problem.

Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation. $T(n) = 2T(n/2) + O(n)$

The solution of the above recurrence is $O(n \log n)$. The list of size N is divided into a max of $\log n$ parts, and the merging of all sublists into a single list takes $O(n)$ time, the worst-case run time of this algorithm is $O(n \log n)$.

Best Case Time Complexity: $O(n * \log n)$

Worst Case Time Complexity: $O(n * \log n)$

Average Time Complexity: $O(n * \log n)$

The time complexity of MergeSort is $O(n * \log n)$ in all the 3 cases (worst, average and best) as the mergesort always divides the array into two halves and takes linear time to merge two halves.

➲ **3. Differentiate between internal and external sorting.** **[Model Question]**

Answer:

Internal Sorting	External Sorting
It is the method when the sorting takes place within the main memory.	It is the method when the sorting takes place with the secondary memory
The time required for read and write operations are considered to be insignificant	The time required for read and write operations are considered to be significant

Internal Sorting	External Sorting
The internal sorting methods are applied to small collection of data.	The External sorting methods are applied only when the number of data elements to be sorted is too large.
Internal sorting takes small input	External sorting can take as much as large input
Internal sorting is simpler than external sorting	External sorting is complex than internal sorting
Example: Bubble Sort, Selection sort, Insertion sort etc.	Example: Merge Sort

➲ **4. Write a note on Internal and external sorting.** **[Model Question]**

Answer:

Internal sorting: It is the method when the sorting takes place within the main memory. The time required for read and write operations are considered to be insignificant.

The internal sorting methods are applied to small collection of data and it takes small input. Example: Bubble Sort, Selection sort, Insertion sort etc.

Algorithm of Bubble Sort:

```
if a[0] > a[1] then
     swap a[0] & a[1]
else
     compare (a[1] & a[2])
             if ( a[1] > a[2] )
                 then again
                 swap (a[1] & a[2])
              and so on
```

External sorting:

External sorting is the method when the sorting takes place with the secondary memory e.g. sorting with disks, sorting with tapes. External sort continues with larger and larger arrays until the entire array is sorted. Example: Merge sort.

Algorithm of Merge Sort:

```
MERGE ( M) //M is the list to be sorted //
VARIABLES LEFT , RIGHT , RESULT
      1.if length(M) = 1
        1.1 return(M)
       Else
        1.2 VARIABLE middle ← length(M) / 2
        1.3 for each x in M up to middle add x to LEFT.
        1.4 for each x in M after middle add x to RIGHT.
      2. LEFT ← MERGE (LEFT)
      3. RIGHT ← MERGE (RIGHT)
      4. RESULT ← MERGE (LEFT,RIGHT)
      5. return RESULT.
```

5. Describe the quick sort algorithm.

OR,

What is the role of pivot? Write an algorithm for performing sorting by Quick sort. **[Model Question]**

Answer:

CODE:

```
void quick_sort(int a[], int l, int r)
{
  int temp, left, right, pivot;
  left = l;
  right = r;
  pivot = (left + right) / 2;
  while (left <  = right)
  {
    while (pivot > a[left])
      left++;
    while (pivot < a[right])
      right--;
    if (left <  = right)
    {
      temp = a[right];
      a[left] = a[right];
      a[right] = temp;
      left++;
      right--;
    }
  }
  if (l < right)
    quick_sort(a, l, right); // recursive call to the left sub
array
  else
    quick_sort(a, left, r);// recursive call to the right sub
array
}
```

In Quick-Sort we divide the array into two halves. We select a pivot element (normally the middle element of the array) & perform a sorting in such a manner that all the elements to the left of the pivot element is lesser than it & all the elements to it's right is greater than the pivot element. Thus we get two sub arrays. Then we recursively call the quick sort function on these two sub arrays to perform the necessary sorting.

Let us consider the following unsorted array:

a[]=45 26 77 14 **68** 61 97 39 99 90

Step 1:

We choose two indices as left = 0 and right = 9. We find the pivot element by the formula a[left + right] / 2 = a [0 + 9] / 2 = a [4]. So the **pivot element is a [4] = 68**. We also start with a [left] = a [0] = 45 and a [right] = a [9] = 90

Step 2:
We compare all the elements to the left of the pivot element. Then increase the value of left by 1 each time, when an element is less than 68. We stop when there is no such element. We record the latest value of left. In our example the left value is 2 i.e., **left=2.**

Step 3:
We compare all the elements to the right of the pivot element. In our example the right value is 7 i.e., **right=7.**

Step 4:
Since left < = right we swap between a [left] and a [right]. That is between 77 and 39.
The array now looks like

45 26 39 14 **68** 61 97 77 99 90.

Step 5:
We further increment the value of left and decrement the value of right by 1 respectively i.e., **left = 2 + 1 = 3 and right = 7 - 1 = 6.**
We repeat the above steps until left < = right.

➲ **6. What is the worst case time complexity of heap sort? Find the complexity of bubble sort in different case. Comment on the complexity of merge sort.**
[Model Question]

Answer:
1st part:
Worst case time complexity of Heap sort:

$O(n\log(n))$

2nd part:
Time Complexity of Bubble sort:
For the first pass the inner loop iterates n – 1 times. In the next iteration n – 2 times and in the last pass only once.
Hence the complexity of the bubble sort is

$$f(n)=(n-1)+(n-2)+(n-3)+\ldots\ldots+2+1$$
$$=\frac{\left[(n-1)*(n-1+1)\right]}{2}=\frac{\left[n(n-1)\right]}{2}=O(n^2)$$

This expression is true for **average case** as well as **worst case.**

Best Case: The minimum number of comparison needed to sort a certain list is:

$f(n)=(n-1)$ Only single pass.

$\leq 2n \quad \leq C_1 * g(n)$ where $C_1=2,\ g(n)=n$

$\therefore \quad f(n)=O(g(n))=O(n)$

That's why bubble sort is generally considered for small number of records.

3rd part:

Here at every step, the merge-sort considers only one array. In the next step, the algorithm splits the array into halves and then sorts and merges them. In the k^{th} iteration, the algorithm splits the arrays into sub-arrays, which are $2^k - 1$ in number. In worst case the number of steps required to break the array into sub-arrays of single elements is $\log_2 n$. At each iteration the maximum number of comparisons is $O(n)$. So, the time complexity is $O(n \log_2 n)$ (in best, average as well as in worst case also). A drawback of mergesort is that it needs an additional space of $\Theta(n)$ for the temporary array b. There are different possibilities to implement function **merge**. The most efficient of these is variant b. It requires only half as much additional space, it is faster than the other variants, and it is stable.

➲ 7. Compare the Marge sort and Quick sort with suitable example.

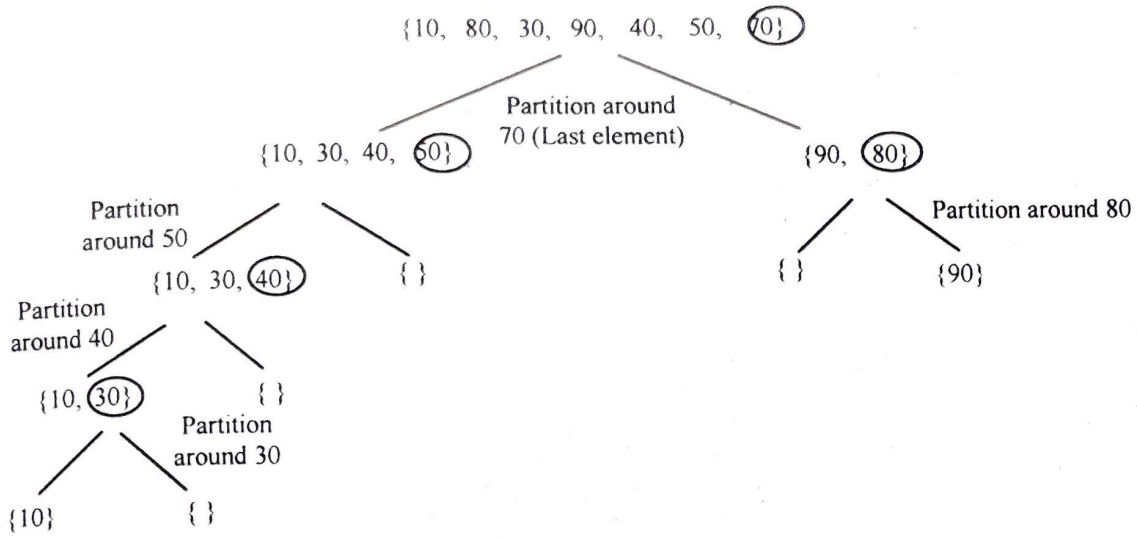
[Model Question]

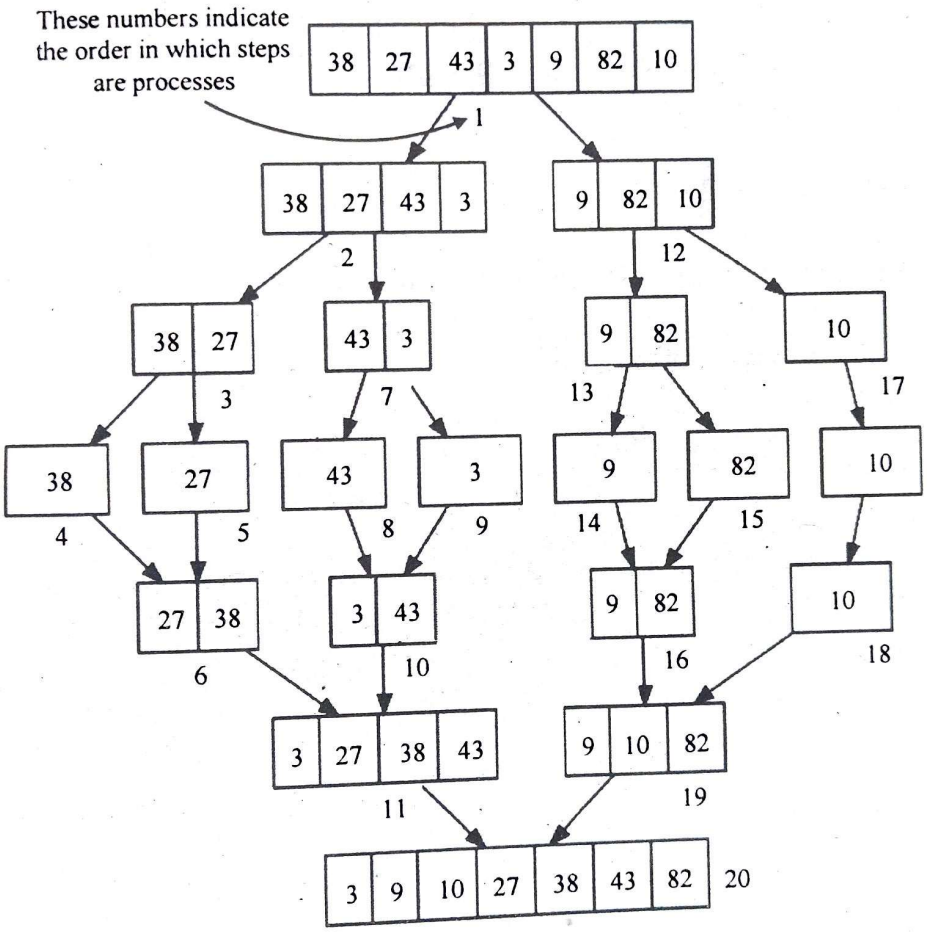
Answer:

Quick sort is an internal algorithm which is based on divide and conquer strategy. In this:

- The array of elements is divided into parts repeatedly until it is not possible to divide it further.
- It is also known as **"partition exchange sort"**.
- It uses a key element (pivot) for partitioning the elements.
- One left partition contains all those elements that are smaller than the pivot and one right partition contains all those elements which are greater than the key element.

Merge sort is an external algorithm and based on divide and conquer strategy. In this:
The elements are split into two sub-arrays (n/2) again and again until only one element is left.
Merge sort uses additional storage for sorting the auxiliary array.
Merge sort uses three arrays where two are used for storing each half, and the third external one is used to store the final sorted list by merging other two and each array is then sorted recursively.
At last, the all sub arrays are merged to make it 'n' element size of the array.





➲ **8. Show the Sorting steps of the following elements stored in an integer array Sample = {2, 84, 15, 75, 39, 27, 85, 71, 63, 97, 27, 14} using selection sort.** **[Model Question]**

Answer:

Selection Sort:

Pass 0	75	39	27	85	71	63	97	27	14
Pass 1	14	39	27	85	71	63	97	27	75

Pass 2	14	27	39	85	71	63	97	27	75
Pass 3	14	27	27	85	71	63	97	39	75
Pass 4	14	27	27	39	71	63	97	85	75
Pass 5	14	27	27	39	63	71	97	85	75
Pass 6	14	27	27	39	63	71	97	85	75
Pass 7	14	27	27	39	63	71	75	85	97

➲ **9. Distinguish between bubble sort and Selection sort.** **[Model Question]**

Answer:

Selection sort

We have array of $10(N=10)$ integers with random number raging from 1 to 10 and we need to sort them. What selection sort does it looks for the lowest number in all N spots then it puts the lowest number in the first spot of array, while moving value in the first spot in older lowest number spot. Making exchange between values. When it does the same, but this time only with $N-1$ elements and so on.

Example (with 5 numbers):

2 8 4 6 3

2 8 4 6 3 /* first iteration, 2 is already the lowest number and it is in first spot, nothing to do */

2 3 4 6 8 /* We checked next N - 1 [8, 4, 6, 3] and lowest was 3, we exchanged values 3 <--> 8 */

2 3 4 6 8 /* 4 is the lowest, nothing to do */

2 3 4 6 8 /* same */

2 3 4 6 8 /* same, N = 0 we are done. */

Bubble sort

Again we have array of 10 integers. Bubble sort is done by comparing neighbors elements in array and if n > n + 1 then we exchange values. We do the same N - 1 times throughout full array.

Example (with 5 numbers):

2 8 4 6 3

2 8 4 6 3 /* 2 > 8 = NO */

2 4 8 6 3 /* 8 > 4 = YES */

2 4 6 8 3 /* 8 > 6 = YES */

2 4 6 3 8 /* 8 > 3 = YES */

2 4 6 3 8 /* 2 > 4 = NO */

2 4 6 3 8 /* 4 > 6 = NO */

2 4 3 6 8 /* 6 > 3 = YES */

2 4 3 6 8 /* 6 > 8 = NO */

2 4 3 6 8 /* 2 > 4 = NO */

2 3 4 6 8 /* 4 > 3 = YES */

....

➲ **10. Sort the elements 77, 49, 25, 12, 9, 33, 56, 81 using – (a) Bubble sort, (b) Selection sort, (c) Radix sort.** **[Model Question]**

OR,

Sort the elements 77, 49, 25, 12, 9, 33, 56, 81 using selection sort showing the passes. Write an algorithm to implement insertion sort. **[Model Question]**

Answer:

1st Part:

a) Bubble Sort:

77, 49, 25, 12, 9, 33, 56, 81

Pass1:

Step 1	49	77	25	12	9	33	56	81
Step 2	49	25	77	12	9	33	56	81
Step 3	49	25	12	77	9	33	56	81
Step 4	49	25	12	9	77	33	56	81
Step 5	49	25	12	9	33	77	56	81
Step 6	49	25	12	9	33	56	77	81
Step 7	49	25	12	9	33	56	77	81

Pass2:

Step 1	25	49	12	9	33	56	77	81
Step 2	25	12	49	9	33	56	77	81
Step 3	25	12	9	49	33	56	77	81
Step 4	25	12	9	33	49	56	77	81
Step 5	25	12	9	33	49	56	77	81
Step 6	25	12	9	33	49	56	77	81

Pass3:

Step 1	12	25	9	33	49	56	77	81
Step 2	12	9	25	33	49	56	77	81
Step 3	12	9	25	33	49	56	77	81
Step 4	12	9	25	33	49	56	77	81
Step 5	12	9	25	33	49	56	77	81

Pass4:

Step 1	9	12	25	33	49	56	77	81
Step 2	9	12	25	33	49	56	77	81
Step 3	9	12	25	33	49	56	77	81
Step 4	9	12	25	33	49	56	77	81

Next several pass yield the same output.

The final output is 9, 12, 25, 33, 49, 56, 77, 81

b) Selection sort:

77, 49, 25, 12, 9, 33, 56, 81

Iteration 1	9	49	25	12	77	33	56	81
Iteration 2	9	12	25	49	77	33	56	81

Iteration 3	9	12	25	49	77	33	56	81
Iteration 4	9	12	25	33	77	49	56	81
Iteration 5	9	12	25	33	49	77	56	81
Iteration 6	9	12	25	33	49	56	77	81
Iteration 7	9	12	25	33	49	56	77	81

c) Radix Sort:

77, 49, 25, 12, 9, 33, 56, 81

bucket	pass 1	pass 2
0	8[1]	[0]9
1	1[2]	[1]2
2	3[3]	[2]5
3	2[5]	[3]3
4	5[6]	[4]9
5	7[7]	[5]6
6	4[9], 0[9]	[7]7
7		[8]

2nd Part:

insertionSort(arr,n)

Loop from i = 1 to n-1.

……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

➲ **11. Explain time complexity of Bubble sort and Insertion sort.** **[Model Question]**

Answer:

Bubble sort algorithm:

Algorithm:

1. Read the total number of elements say n
2. Store the elements in the array
3. Set the i = 0.
4. Compare the adjacent elements.
5. Repeat step for all n elements.
6. Increment the value of i by 1 and repeat step 4,5 for i < n.
7. Print the sorted list of elements.
8. Stop.

Analysis: In the above algorithm basic operation if $(a[j] > a[j+1])$ which is executed within nested for loops. Hence time complexity of bubble sort is $\Theta(n)^2$

Insertion sort algorithm:

Algorithm Insert _sort$(A[0.....n-1])$

//Problem Description: This algorithm is for sorting the

//elements using insertion sort

//Input: An array of n elements

```
//Output: Sorted array A[0.....n-1] in ascending order
  for i ← 1 to n-1 do
{
          temp ← A[i]//mark A[i]th element
          j ← i-1//set j at previous element of A[i]
          while(j>=0)AND(A[j]>temp)do
          {
       //comparing all the previous elements of A[i] with
       //A[i].If any greater element is found then insert
       //it at proper position
           A[j+1] ← A[j]
           j ← j-1
           }
           A[j+1] ← temp //copy A[i] element at A[j+1]
}
```

Analysis:

When an array of elements is almost sorted then it is **best case** complexity. The best case time complexity of insertion sort is $O(n)$.

If an array is randomly distributed then it results in **average case** time complexity which is $O(n^2)$.

If the list of elements is arranged in descending order and if we want to sort the elements in ascending order then it results in **worst case** time complexity which is $O(n^2)$.

➲ **12. Compare the insertion sort and selection sort with i) Efficiency ii) Sort stability and iii) Passes.** **[Model Question]**

Answer:

i) Efficiency: The code for selection sort is as follows

```
for(i=0;i<n-1;i++)
          for(j=i+1;j<n;j++)
          {
               if(ai>aj)
                    swap ai and aj
           }
```

In the i^{th} pass, n-i comparisons will be needed to select the smallest element. $n-1$ passes are required to sort the array. Thus, the number of comparisons needed to sort an array having n elements

$$=(n-1)+(n-2)+...+2+1=\frac{n(n-1)}{2}=\frac{1}{2}(n^2-n)=O(n^2)$$

Insertion sort:

```
for(i=1;i<n;i++)
{
        k=a[i];
        for(j=i-1;j>=0&&y<a[i])
        a[j+1]=a[j];
        a[j+1]=1;
}
```

Inner loop is data sensitive. If the input list is presented for sorting is presorted then the test $a[i] > y$ in the inner loop will fail immediately. Thus, only one comparison will be made in each pass.

Thus the total number of comparisons (Best case)
$= n - 1 = n$ for large n.
If the numbers to be sorted are initially in descending order then the inner loop will make i iterations in i^{th} pass.
$\therefore$ Total number of comparisons.

$$= 1 + 2 + 3 + ... + (n-1) = \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n) = n^2 \text{ for large n.}$$

ii) Sort stability: Both the techniques are stable. Both of them have same time complexity.

iii) Passes: Both selection and insertion sort requires $n-1$ passes.
Insertion sort is more stable than selection sort.
Insertion sort requires less overhead while sorting data. Selection sort requires more number of comparisons. In both the algorithms, elements are read in liner fashion and adjacent elements are compared. Hence identical numbers will maintain their relative sequence even in the sort list.

➲ 13. Analyze quick sort algorithm.

[Model Question]

Answer:
The quick sort algorithm is performed using following two important functions – *Quick* and *partition*. Let us see them –

Algorithm Quick($A[0...n-1]$,low,high)
//Problem Description: This algorithm performs sorting of
//the elements given in Array $A[0...n-1]$
//Input: An array $A[0...n-1]$ in which unsorted elements are
//given. The low indicates the leftmost element in the list
//and high indicates the rightmost element in the list
//Output: Creates a sub array which is sorted in ascending
//order

```
if(low < high)then
//split the array into two sub arrays
   m ← partition(A[low...high)// m is mid of the array
Quick(A[low...m-1])
Quick(A[mid+1...high])
```

In the above algorithm call to partition algorithm is given. The *partition* performs arrangement of the elements in ascending order. The recursive *quick* routine is for dividing the list in two sub lists. The pseudo code for *Partition* is as given below-

```
Algorithm Partition (A[low...high])
//Problem Description: This algorithm partitions the
//subarray using the first element as pivot element
//Input: A subarray A with low as left most index of the
//array and high as the rightmost index of the array.
//Output: The partitioning of array A is done and pivot
//occupies its proper position. And the rightmost index of
//the list id returned
pivot ← A[low]
i ← low
j ← high+1
while(i<=j) do
{
 while(A[i]<=pivot) do
   i ← i+1
 while(A[j]>=pivot) do
   j ← j-1
 if(i<=j) then
 swap(A[i],A[j]//swaps A[i] and A[j]
 }
 Swap(A[low]],A[j]//when i crosses j swap A[low] and A[j]
 return j//rightmost index of the list
```

The partition function is called to arrange the elements such that all the elements that are less than pivot are at the left side of pivot and all the elements that are greater than pivot are all at the fight of pivot. In other words pivot is occupying its proper position and the partitioned list is obtained in an ordered manner.

Analysis

When pivot is chosen such that the array gets divided at the mid then it gives the best case time complexity. The best case time complexity of quick sort is $O(n \log_2 n)$.

The worst case for quick sort occurs when the pivot is minimum or maximum of all the elements in the list. This can be graphically represented as -

This ultimately results in $O(n^2)$ time complexity. When array elements are randomly distributed then it results in average case time complexity, and it is $O(n \log_2 n)$.

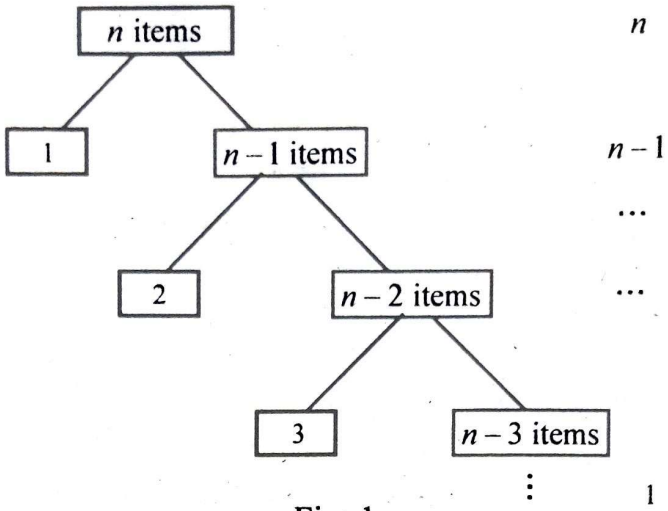


Fig: 1

➲ **14. Write short notes on the following:** **[Model Question]**

a) Counting sort

b) Radix sort

c) Bucket sort

d) Shell sort

Answer:

a) In counting sort, it assumes that each of the n input elements is an integer in the range 0 to k, for some integer k. When $k = O(n)$, the counting sort runs in $\theta(n)$ time.

Counting sort is based on assigning the position to an element into an array equal to its value) i.e., 18 goes to 18^{th} position) but if the number have same value it may create problem.

In counting sort, we assume that A[1.....n] is an input array and length [A] = n. Now we require two another arrays: The array B[1.....n] holds the sorted output and the array C[0.....k] provides temporary storage, where k is the largest number in array A.

COUNTING – SORT (A,B, k)

1. for $i \leftarrow 0$ to k
2. do $c[i] \leftarrow 0$
3. for $j \leftarrow 1$ to length [A]
4. do C[A[j]] $\leftarrow$ C[A[j]]+1
5. for $i \leftarrow 1$ to key
6. do $c[i] \leftarrow c[i] + c[i-1]$
7. for $j \leftarrow$ length [A] down to 1
8. do B[C[A[i]]] $\leftarrow$ A[j]
9. C[A[j] $\leftarrow$ C[A[j]]-1

Analysis: How much time does counting sort require? The for loop of lines 1-2 takes $\theta(k)$ times, the for loop of lines 3-4 takes time $\theta(n)$, the for loop of line 5-6 takes time $\theta(k)$, and the for loop of lines 7-9 takes $\theta(n)$ time. Thus, the total time is $\theta(k+n)$. We use counting sort when we have $k = O(n)$, in which case the running time is $\theta(n)$.

Example: Illustrate the operating of counting-sort on the array

$A = \{6,0,2,0,1,3,4,6,1,3,2\}$

Solution:

	1	2	3	4	5	6	7	8	9	10	11
A	6	0	2	0	1	3	4	6	1	3	2

Using counting – sort (A, B, k)

Here A $\rightarrow$ Given input array

B $\rightarrow$ Sorted output array

and $k \rightarrow 6$ (largest or big number in A)

Step I: for $i \leftarrow 0$ to 6

do $c[i] \leftarrow 0$

		0	1	2	3	4	5	6
i.e.,	C	0	0	0	0	0	0	0

Step II: for $j = 1$ to 11

do $c[A[j]] \leftarrow c[A[j]] + 1$

i.e., when $j = 1$

$c[A[1]] \leftarrow c[A[j]] + 1$

By given input array A, the value of A[1] is 6. Put the value of A[1] in the above,

$$c[6] \leftarrow c[6]+1$$

We will see the value of $c[6]$ in the array and put in the above line

$$c[6] \leftarrow 0+1$$

i.e., $$c[6] \leftarrow 1$$

We continue it till $j=11$, now final c array,

	0	1	2	3	4	5	6
C	2	2	2	2	1	0	2

Step III: for $i \leftarrow$ to 6

do $c[i] \leftarrow c[i]+c[i-1]$

i.e., when $i=1$

$$c[1] \leftarrow c[1]+c[1-1]$$
$$\leftarrow 2+2$$
$$c[1] \leftarrow 4$$

Again we continue it till $i=6$, after it final c will be

	0	1	2	3	4	5	6
C	2	4	6	8	9	9	11

Step IV: For $j \leftarrow 11$ to 1

$$B[c\,[A[j]]] \leftarrow A[j]$$
$$c[A[j]] \leftarrow c[a[j]]-1$$

i.e., when $j=11$

$$B[c[A[11]]] \leftarrow A[11]$$
$$c[A[11]] \leftarrow c[A[11]]-1$$
$$B[c[2]] \leftarrow 2$$
$$c[2] \leftarrow c[2]-1$$
$$B[6] \leftarrow 2$$
$$c[2] \leftarrow 5$$

	1	2	3	4	5	6	7	8	9	10	11
B						2					

	0	1	2	3	4	5	6
C			5				

Continue this process till $j = 1$, now the final sorted output array B and temporary array c is

	1	2	3	4	5	6	7	8	9	10	11
B	0	0	1	1	2	2	3	3	4	6	6

	0	1	2	3	4	5	6
C	0	2	4	6	8	9	9

Shortcut method: We solve the counting sort using shortcut method.

	1	2	3	4	5	6	7	8	9	10	11
Given A	6	0	1	0	1	3	4	6	1	3	2

First we choose the largest number in the given array A, i.e., $k = 6$.

Step I: We initialize for loop $i \leftarrow 0$ to 6 (largest number)

	0	1	2	3	4	5	6
C	0	0	0	0	0	0	0

Step II: Now, for $j \leftarrow 1$ to length [A]

i.e., for $j \leftarrow 1$ to 11 do $c[A[j]] \leftarrow c[A[j]] + 1$

In this step, we count the key in the given array A i.e., number of 0 in the array is 2 and number of 6 in the array is 2.

	0	1	2	3	4	5	6
C	2	2	2	2	1	0	2

Step III: For $i \leftarrow 1$ to 6

do $c[i] \leftarrow c[i] + c[i-1]$

In this step, suppose $i = 1$, we add the previous value of A[1] is A[0] with A[1], i.e., $2 + 2 = 4$

0	1	2	3	4	5	6
2	4	6	8	9	9	11

Step IV: We write sorted array

	1	2	3	4	5	6	7	8	9	10	11
B	0	0	1	1	2	2	3	3	4	6	6

We calculate c using step II and step III i.e., the value of $c[0]$ is 2 in step III and value of $c[0]$ is 2 in step II we subtract step III $c[0]$ – step II $c[0]$ i.e., $2-2=0$. Again, the value of $c[1]$ is 4 in step III and value of $c[1]$ in step II. We subtract step III $c[1]$ – step $c[1]$ i.e., $4-2=2$. The final c is

	0	1	2	3	4	5	6
C	0	2	4	6	8	9	9

b) In Radix sort, the lists contain of n-integers, and each integer has d-digit number. We start the sorting repeatedly, starting at the lower order digit, and finishing with the highest order. It is notice that the sorting is stable, if the numbers are already sorted. With respect to low order digits and then later, we sort with respect to high order digits, numbers that have same high order digit will remain sorted with respect to their low order digit.

Sometimes, radix sort used to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort, first on day, next on month, and finally on year.

The code for radix sort is straight forward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest order digit.

RADEX-SORT (A,d)

1. for $i \leftarrow 1$ to d
2. do use a stable sort to sort array A on digit i.

Analysis: Radix sort runs in linear time. When each digit is in the range 1 to k, and k is not too large, counting sort is the obvious choice. We know counting sort runs in $\theta(k+n)$ time i.e., each pass over n d-digit numbers take $A(k+n)$ times. There are d passed, so the total time for radix sort is $\theta(d(n+k))$, where d is constant and $k = O(n)$.

Example: Illustrate the operation of RADIX-SORT on the following list:

$A = \{329,457,657,839,436,720,355\}$

								Output
3 2 9		7 2 [0]		7 [2] 0		[3] 2 9
4 5 7		3 5 [5]		3 [2] 9		[3] 5 5
6 5 7	⇒	4 3 [6]	⇒	4 [3] 6	⇒	[4] 3 6
8 3 9		4 5 [7]		8 [3] 9		[4] 5 7
4 3 6		6 5 [7]		3 [5] 5		[6] 5 7
7 2 0		3 2 [9]		4 [5] 7		[7] 2 0
3 5 5		8 3 [9]		6 [5] 7		[8] 3 9
		↑		↑		↑

c) Like counting sort, bucket sort is fast because it assumes something about the input whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the interval [0,1]. The idea of bucket sort is to divide the interval [0,1] into n equal-sized subintervals, or buckets, and then distribute the n input numbers into the buckets. Since the inputs are uniformly distributed over [0,1], we don't expect many numbers to fall into each bucket. To produce the output, we simple sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.
In bucket sort, we assumes that the input is an n-element array A and that each element A[i] in the array satisfies $0 \le A[i] < 1$. The code requires an auxiliary array B [0.....$n-1$] of linked lists and assumes that there is a technique for maintaining such lists.

BUCKET-SORT(A)

1. $n \leftarrow$ length [A]
2. for $i \leftarrow 1$ to n
3. do insert A[i] into list B$\left[\lfloor nA[i] \rfloor\right]$
4. for $i \leftarrow 0$ to $n-1$
5. do sort list B[i] with insertion sort.
6. Concatenate the lists $B[0], B[1].....B[(n-1)]$ together in order.

To see that the procedure of this algorithm, consider two elements A[i] and A[j]. We assume without loss of generally that $A[i] \le A[j]$. Since $\lfloor nA[i] \rfloor \le \lfloor nA[j] \rfloor$, element A[$i$] is placed either into the same bucket as A[j] or into a bucket with a lower index. If A[i] and A[j] are placed into the same bucket, then the for loop of lines 4-5 puts them into the proper order. If A[i] and A[j] are placed into different buckets, then the line 6 puts them into the proper order. Hence, bucket sort works correctly.

Analysis: We observe that all lines except line 5 take $O(n)$ time in the worst case. It remains to balance the total time taken by the n calls to insertion sort in line 5. The cost of the calls to insertion sort, let n_i be the random variable denoting the number of elements placed in bucket B[i]. Since insertion sort runs in quadratic time. The running time of bucket sort is

$$T(n) = \theta(n) + \sum_{i=0}^{n-1} O\left(n_i^2\right)$$

Taking expectations of both sides and using linearity of expectations, we have

$$E[T(n)] = E\left[\theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right]$$

$$= \theta(n) + \sum_{i=0}^{n-1} E\left[O(n_i^2)\right] \qquad \text{(by linearity of expection)}$$

$$= \theta(n) + \sum_{i=0}^{n-1} O\left(E\left[(n_i^2\right]\right) \qquad \ldots.(A)$$

We known

$$E(n_i) = np \text{ and } p = \frac{1}{n}$$

$$E(n_i) = n \times \frac{1}{n} = 1$$

Vari $$(E) = np(1-p) = n \times \frac{1}{n}\left(1 - \frac{1}{n}\right) = \left(1 - \frac{1}{n}\right)$$

$$E(n_i^2) = \text{Vari}\,(E) + E(n_i) = 1 - \frac{1}{n} + 1 = 2 - \frac{1}{n} = \theta(1)$$

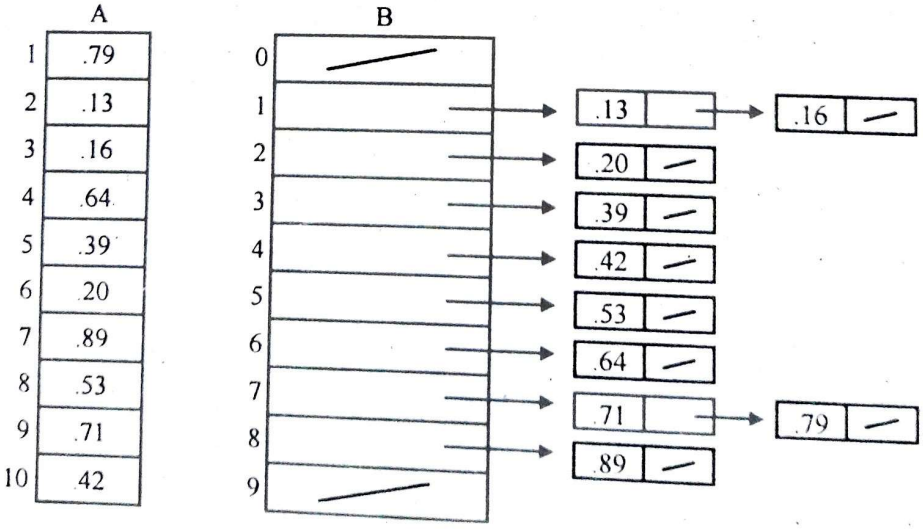
The value of $E(n_i^2)$ put in equation (A), we get

$$T(n) = \theta(n) + O\sum_{i=0}^{n-1} \theta(1) = \theta(n) + O[\theta(1) + \theta(1) + \ldots..] = \theta(n)$$

Thus, the entire bucket sort algorithm runs in linear expected time.

Example: *Illustrate the operation of BUCKET-SORT on the array*

$$A = \{.79,.13,.16,.64,.39,.20,.89,.53,.71,.42\}.$$



The sorted array B is

0	1	2	3	4	5	6	7	8	9
.13	.16	.20	.39	.42	.53	.64	.71	.79	.89

d) Shell sort is a sorting algorithm that is a generalization of insertion sort with two observations:

- Insertion sort is efficient if the input is "almost sorted".
- Insertion sort is typically inefficient because it moves values just one position at a time.

The shell sort is named after its inventor D.L. Shell in 1959. It is fast, easy to implement. However, its complexity analysis is more complicated. The idea of shell sort is to arrange the data sequence in a two-dimensional array and sort the columns of the array. The effect is that the data sequence is partially sorted. The process above is repeated, but each time with a narrower array, i.e. with a smaller number of columns. In the last step, the array consists of only one column. In each step, the sortedness of the sequence is increased until the last step is completely sorted. However, the number of sorting operations necessary in each step is limited due to the presortedness of the sequence obtained in the preceding steps.

Consider a small value that is initially stored in the incorrect position of the array which makes the array out of order. Using an $O(n^2)$ sort such as bubble sort or insertion sort, it will take roughly n comparisons and exchange to move this value all the way to the other end of the array. Shell sort first moves values using huge step sixes, so a small value will move a long way towards its final position with just a few comparisons and exchanges.

Actually, the data sequence is not arranged in a two-dimensional array, but held in a one-dimensional array that is indexed appropriately. For instance, data elements at positions 0, 5, 10, 15, etc. would form the first column of and array with five columns. The "columns" obtained by indexing in this way are sorted with insertion Algorithm 16.6 is the algorithm for shell sort.

Algorithm 16.6: SHELL-SORT

```
//An array DATA of length n with array elements numbered 0 to
n-1
1. Set inc = round (n/2)
2. Repeat while inc > 0 do
   1.  Repeat for i = inc to n-1 do
   2.  temp = DATA [i]
   3.  j = i
```

```
4.  Repeat while j ≥ inc and DATA [j - inc] > temp do
5.  DATA [j] = DATA [j-inc]
6.  j = j-inc
7.  DATA [j] = temp
8.  End step 4 loop
9.  inc = round (inc/2.2)
10. End step2 loop
3. End loop
4. End
```

Example 16.7

Let 3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2 be the data sequence to be sorted. First, it is arranged in an array with seven columns (left) and then the columns are sorted (right) (Fig. 1).

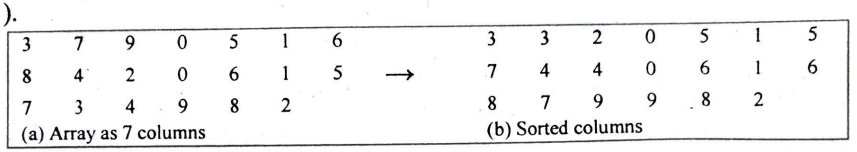


Fig: 1 Sorted array with seven columns

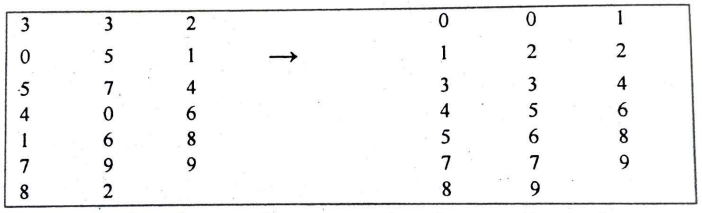


Fig: 2 Sorting elements by representing them as three columns

Data elements 8 and 9 have now already come to the end of the sequence, but a small element (2) is still there as shown in Fig. 1(b). In the next step, the sequence is arranged in three columns, which are again sorted.

Now the sequence is almost completely sorted as shown in Fig. 2. When arranging it in one column in the last step, it is only the elements 6, 8 and 9 that have to move a little bit to their correct positions.

Analysis:

Best Case: The best case in the shell sort is when the array is already sorted in the right order. The number of comparisons is less. In that case the inner loop does not need to any work and a simple comparison will be sufficient to skip the inner sort loop. The other loops $O(n \log n)$. the best case of $O(n)$ is reached by using a constant number of increments. Hence, the best case time complexity of shell sort is $O(n \log n)$.

Worst Case and Average Case: The running time of Shellsort depends on the choice of increment sequence. The problem will Shell's increments is that pairs of increments are not necessarily relatively prime and smaller increments can have little effect. The wort case and average case time complexity is $O(n)$.

Unit 3: Searching

Unit at a glance:

3.1 Searching

Searching is the term defined to retrieve a record with a particular value.
The different types of searching techniques are

- Linear (Sequential) Search
- Binary Search
- Interpolation Search
- Hashing

3.2 AVL Tree

A binary search tree structure that is balanced with respect to the heights of subtrees is called an AVL tree. As a result of the balanced nature of this type of tree, dynamic retrievals can performed in O(log n) times if the tree has n nodes in it.
If T is a non-empty binary search tree with T_L & T_R as its left & right subtrees respectively, then T is height balanced if
i) T_L & T_R are height balanced and
ii) $|h_L - h_R| <= 1$, where h_L & h_R are the heights of T_L & T_R respectively.
The Balanced Factor, BF (T), of a node T in a binary search tree is defined to be $h_L - h_R$ i.e. the height of its left sub tree minus the height of its right sub tree.

3.3 Hashing

- A **hash table** data structure is just like an array. Data is stored into this array at specific index generated by a hash function. A **hash function** hashes (converts) a number in a large range, into a number in a smaller range. This smaller range corresponds to the index numbers in an array. An array into which data is inserted using a hash function is called a **hash table.**
- In hash tables, there is always a possibility that two data elements will hash to the same integer value. When this happens, a **collision** occurs i.e. two data members try to occupy the same place in the hash table array. There are methods to deal with such situations like **Open Addressing and Chaining**.
- **Open addressing:** In this method, when a data item can't be placed at the index calculated by the hash function, another location is sought. This is termed as probe sequence. There are three open addressing methods, which vary in probe sequence to find the next vacant cell. These are **Linear probing, Quadratic probing and double hashing.**

- In Open addressing, collisions are resolved by looking for an open cell in the hash table. A different approach is to create a linked list at each index in the hash table. A data item's key is hashed to the index in the usual way, and the item is inserted into the linked list at that index. Other items that hashes to the same index are simply added to the linked list at that index. There is no need to search for empty cells in the primary hash table array. This is the **chaining** method.
- **Clustering** in hash tables occur when filled sequence in a hash table becomes longer. It means that positions are occupied by elements and we have to search a longer period of time to get an empty cell.

Short Answer Type Questions

A. Choose the correct answer from the given alternatives in each of the following:

1. Given an array arr = {45, 77, 89, 90, 94, 99, 100} and key = 99; what are the mid values (corresponding array elements) in the first and second levels of recursion? **[WBSCTE 2022]**

   (a) 90 and 99 (b) 90 and 94 (c) 89 and 99 (d) 89 and 94

Answer: (a)

2. Which of the following cases occurs when searching an array using linear search the value to be searched is equal to the first element of the array? **[Model Question]**

   (a) Worst case (b) Average case
   (c) Best case (d) Amortized case

Answer: (c)

3. In the worst case, a binary search tree will take how much time to search an element? **[Model Question]**

   (a) $O(n)$ (b) $O(\log n)$ (c) $O(n^2)$ (d) $O(n \log n)$

Answer: (a)

B. Fill in the blanks in the following statements:

4. Worst case for linear search is ***O(n)***. **[WBSCTE 2022]**

5. Each node in a binary tree has at most ***two*** child nodes. **[WBSCTE 2022]**

6. Deletion in heap requires ***two*** number of arrays. **[WBSCTE 2022]**

7. Balance factor of AVL tree is ***–1, 0 or +1***. **[WBSCTE 2022]**

8. The process where two rotations are required to balance a tree is called ***double rotation***. [WBSCTE 2022]

9. Best case time complexity of binary search is ***O(1)***. [Model Question]

10. The average case time complexity of binary search is ***$O(\log_n)$***. [Model Question]

11. ***Collision*** happens when two items tries to occupy the same location of the hash table. [Model Question]

12. Linear probing is a ***hash collision resolution*** technique. [Model Question]

C. Answer the following questions:

13. What is another name of height balanced binary search tree? [WBSCTE 2022]

Answer:

AVL tree.

14. Mention the various types of searching techniques in C. [WBSCTE 2022]

Answer:

Searching Algorithms: Linear Search · Binary Search · Jump Search · Interpolation Search · Exponential Search.

15. What is advantage of linked list representation of binary trees over arrays? [WBSCTE 2022]

Answer:

- dynamic size.
- B. ease of insertion/deletion.
- C. ease in randomly accessing a node.
- D. both dynamic size and ease in insertion/deletion.

16. What is Collision? [WBSCTE 2022]

Answer:

A collision occurs when more than one value to be hashed by a particular hash function hash to the same slot in the table or data structure (hash table) being generated by the hash function.

17. What is direct addressing? [WBSCTE 2022]

Answer:

Direct addressing is possible only when we can afford to allocate an array that has one position for every possible key.

18. Which is the best searching algorithm and why? [WBSCTE 2022]

Answer:
Binary search is a more efficient search algorithm which relies on the elements in the list being sorted.

19. Write the advantages of height balancing. **[WBSCTE 2022]**

Answer:
A **height-balanced binary tree** is defined as a binary tree in which the **height** of the left and the right subtree of any node differ by not more than 1. AVL tree, red-black tree are examples of height-balanced trees.

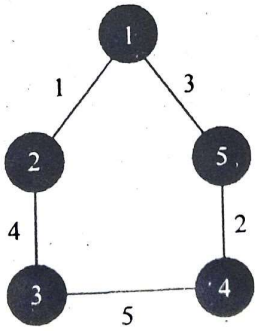
20. What is inorder traversal in BST? **[WBSCTE 2022]**

Answer:
It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal. So, in the inorder traversal, each node is visited in between of its subtrees.

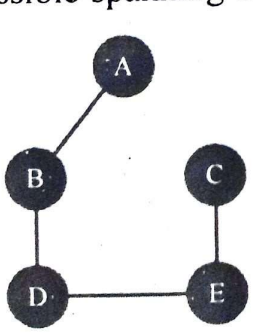
21. Describe spanning tree with an example. **[WBSCTE 2022]**

Answer:
A spanning tree can be defined as the subgraph of an undirected connected graph. It includes all the vertices along with the least possible number of edges. If any vertex is missed, it is not a spanning tree. A spanning tree is a subset of the graph that does not have cycles, and it also cannot be disconnected.

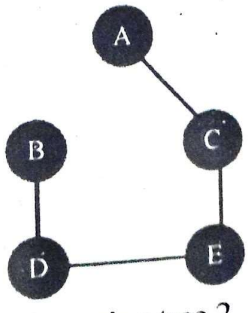
Suppose the graph be –



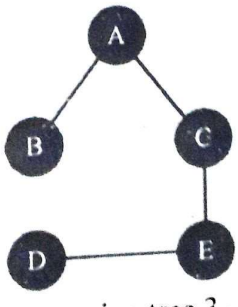
Some of the possible spanning trees that will be created from the above graph are given as follows:



Spanning tree 1



Spanning tree 2



Spanning tree 3

22. Classify the Hashing Functions based on the various methods by which the key value is found. [Model Question]

Answer:

1. Direct method,
2. Subtraction method,
3. Modulo-Division method,
4. Digit-Extraction method,
5. Mid-Square method,
6. Folding method,
7. Pseudo-random method.

23. What is sequential search? What is the average number of comparisons in a sequential search? [Model Question]

Answer:

Linear (Sequential) Search is a method where the search begins at one end of the list, scans the elements of the list from left to right (if the search begins from left) until the desired record is found or the other end is reached. Let us consider an array with the following elements:

a[ ]=33, 51, 27, 85, 66, 23, 13, 57

Suppose we want to search for the element 66. We scan the array from left starting from the first element comparing 66 with all other elements in the array. If a suitable match is not found we move on the next element for comparison. In our case the suitable match is found at a [4] position. The average number of comparisons in a sequential search is $n(n-1)/2$.

24. In binary search how the elements should be arranged? [Model Question]

Answer:

For binary search, the array should be arranged in ascending or descending order. Binary search relies on divide and conquer strategy in a sorted list.

25. Why binary search is better than linear search? [Model Question]

Answer:

It depends.

- If the list is large and changing often, with items constantly being added or deleted, then the time it takes to constantly re-order the list to allow for a binary search might be longer than a simple serial search in the first place.
- If the list is large and static e.g. telephone number database, then a binary search is very fast compared to linear search. (in maths terms it takes 2log2(n) for a binary search over n items)
- If the list is small then it might be simpler to just use a linear search
- If the list is random, then linear is the only way
- If the list is skewed so that the most often searched items are placed at the beginning, then on average, a linear search might be better.

26. The restriction while using the binary search is? [Model Question]

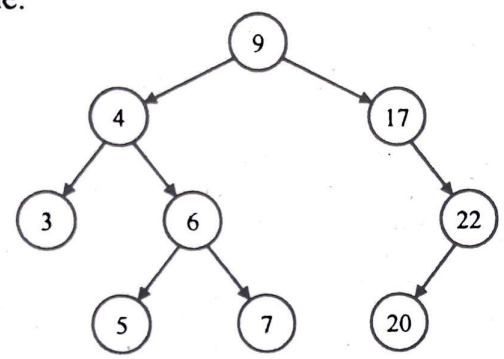
Answer:

A binary search tree is a binary tree with the restriction that the value at any node is bigger than the values of all the items in the left subtree and is smaller than the values of all the elements in the right subtree.

27. Explain the concept of binary search tree. [Model Question]

Answer:

A binary search tree (BST), also known as an ordered binary tree, is a node-based data structure in which each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree. The left sub-tree contains only nodes with keys less than the parent node; the right sub-tree contains only nodes with keys greater than the parent node.



The BST data structure is the basis for a number of highly efficient sorting and searching algorithms and it can be used to construct more abstract data structures including sets, multisets, and associative arrays.

28. Discuss the advantages of an AVL tree. [Model Question]

Answer:

An **AVL tree** is a self-balancing binary tree. It is similar in concept to a Red-Black tree. AVL tree does not waste much memory.

29. What do you mean by hash function? [Model Question]

Answer:

Hash Function: A hash function is a mapping between a set of input values and a set of integers, known as hash values. It is usually denoted by H.

Long Answer Type Questions

➲ 1. Write and explain recursive binary search algorithm. [WBSCTE 2022]

Answer:

Suppose we want to locate x in the sequence $a_1, a_2, \ldots\ldots a_n$ of integers in increasing order. To perform a binary search, we begin by comparing x with the middle term,

$a_{\lfloor (n+1)/2 \rfloor}$. Our algorithm will terminate if x equals this term and return the location of this term in the sequence. Otherwise, we reduce the search to a smaller search sequence, namely, the first half of the sequence if x is smaller than the middle term of the original sequence, and the second half otherwise. We have reduced the solution of the search problem to the solution of the same problem with a sequence at most half as long. If have we never encountered the search term x, our algorithm returns the value 0. We express this recursive version of a binary search algorithm as Algorithm 6.

ALGORITHM 6 A Recursive Binary Search Algorithm.

procedure *binary search* $(i, j, x : i, j, x$ integers, $1 \le i \le j \le n)$

m $:= \lfloor (i+j)/2 \rfloor$

if $x = a_m$ **then**

  return m

else if $(x < a_m$ and i < m$)$ **then**

  return *binary search*$(i, m-1, x)$

else if $(x > a_m$ and j > m$)$ **then**

  return *binary search*$(m+1, j, x)$

else return 0

{output is location of x in $a_1, a_2, \ldots\ldots a_n$ if it appears; otherwise it is 0}

➲ **2. What are the primary assumptions for application of binary search method? Write an algorithm for binary search. Find the time complexity of the binary search algorithm.** **[Model Question]**

OR,

Design an algorithm that finds an element X in a sorted list of N elements in $O(\log N)$ time. **[Model Question]**

Answer:

In Binary Search, the entire ***sorted*** list is divided into two parts. We first compare our input item with the mid element of the list and then restrict our attention to only the first or second half of the list depending on whether the input item comes before or after the mid-element. In this way we reduce the length of the list at each step until we are getting a single element in a list.

Let us consider a sorted array with the following.

a[ ]=13 23 27 33 51 66 85

We find the mid-element by the formula

mid = a [left + right] / 2; left and right are the leftmost and rightmost index of the array.

1. Initialize an ordered array, searcharray, searchno, length.
2. Initialize low=0 and high=length.
3. Repeat step 4 till low<=high.

```
4. Middle =(low + high) / 2.
5. if searcharray[middle]=searchno
Search is successful
return middle.
else if searcharray[middle]>searchno[high]
high=middle - 1
else
low=middle + 1
6. End
```

The time complexity of Binary Search is as follows:

In each iteration, the array is split into two halves. Thereby we can say that the binary search takes the form of a binary tree. The time complexity is thus O ($\log_2 n$) in worst case also.

➲ **3. What do you mean by bucket? What are the criteria to choose a good hash-function? Explain with example how module (mod) operator is used to obtain a hash function. Discuss how collision will be resolved in your example.**

[Model Question]

Answer:

1st Part:

A hash table internally contains Buckets in which it stores Keys / value pairs.

The rules for a good hash function are:

Rule 1: The hash value is fully determined by the data being hashed.

Rule 2: The hash function uses all the input data.

Rule 3: The hash function "uniformly" distributes the data across the entire set of possible hash values.

Rule 4: The hash function can generate different hash values for similar strings.

2nd Part:

Division Remainder Method:

According to this method, the key value is divided by an appropriate number, generally a prime number, and the division of remainder is used as the address for the record.

H (K) = K Mod M (Definition)

K → key

M→ size of the table (**usually taken as prime number**)

H (K)→ Hash Function

Hash Table: →

7
18
89

Let us consider an array of 7 elements
such as 89, 18, 49, 58, 9, and 7
H (K) = K mod M
Taking the elements as key & size of the array as M.
Thus
K = 89 (for the first element)
M = 7 (size of the array)

H (89) = 89 mod 7 = 5, so 89 will be placed in location 5

Next, 18
H (18) = 18 mod 7 = 4
Similarly H (7) = 7 mod 7 = 0, so 7 will be the first element of the Hash table
A collision between two keys suppose K & K' occurs when both have to be stored in the table & both hash to the same address in the table.
Let us consider the following elements

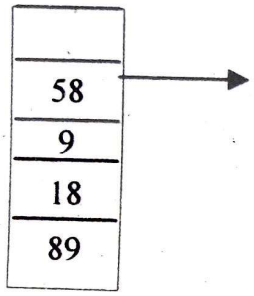
89, 18, 49, 58, 9, 7,

H(89) = 5 (Using Division – Remainder Method)
H(18) = 4 (Using Division – Remainder Method)
H(49) = 0 (Using Division – Remainder Method)
H(58) = 2 (Using Division – Remainder Method)
H (9) = 2 (Using Division – Remainder Method)

Now here, already there is one element in the position 2, which is 58 in our example. But 9 is also hashed to position 2, which is occupied by 58 in our example. When this kind of situation occurs we say that a collision has taken place.

3rd Part:

Linear Probing: It is one method for dealing with collisions. If a data element hashes to a location in the table that is already occupied, the table is searched sequentially from that location until an open location is found. Taking the collision in our case



Here 9 also wanted this location. But 58 have been allocated here. Thus the next free space is calculated to adjust this collision. Thus by sequential search we find that 3 is the location so we place 9 in address 3.

This is linear probing.

➲ **4. What is hashing? Discuss some popular hash functions.** **[Model Question]**

OR,

What is hashing? Explain how it helps in faster accessing of information. Discuss on any one hashing technique.

What are the advantages and disadvantages of the various collision resolution strategies?

Answer: **[Model Question]**

Hashing:

It is a technique whereby item or items are placed into a structure based on a key to-address transformation. We use Hashing for

1) Performing optimal searches & retrieval, insertion and deletion of data at a average constant time complexity of O (1)

2) It increases speed, betters ease of transfer, improves retrieval, optimizes searching of data and reduces overhead.

Hash Function: A hash function is a mapping between a set of input values and a set of integers, known as hash values. It is usually denoted by H.

The different types of hashing functions are:

1) Division Remainder Method:

According to this method, the key value is divided by an appropriate number, generally a prime number, and the division of remainder is used as the address for the record.

H (K) = K Mod M (Definition)

K → key

M→ size of the table (**usually taken as prime number**)

H (K)→Hash Function

2) Mid-Square Method:

Definition: H (K) = K^2 or some digits taken from K^2

Let us consider the element 7 in our previous example

H (7) = 7^2 = 49

We consider the first digit i.e. 4 from 49 to indicate that the element 7 will be in the address 4 of the array.

There are several collision resolution technique:

(1) Closed Hashing [Open Addressing], (2) Open Hashing [chaining]

Advantages and disadvantages of Closed Hashing:-

Advantages:

Closed Hashing (Open Addressing) is more complex but can be more efficient, especially for small data records.

Disadvantages:

The major drawback of closed hashing is that, as half of the hash table is filled, there is a tendency towards clustering; that is key values are clustered in large groups and as a result sequential search becomes slower and slower. This kind of clustering typically known as primary clustering.

Advantages and disadvantages of chaining:

There are several advantages of chaining method:

1. Open Hashing (chaining) is simpler to implement, and more efficient for large records or sparse tables.
2. Overflow situation never arises. Hash table maintains lists, which can contain any number of key values.
3. Collision resolution can be achieved very efficiently if the lists maintain an ordering of keys, so that keys can be searched quickly.

Drawback of this method is that it is not much efficient technique for small data size.

2) It increases speed, betters ease of transfer, improves retrieval, optimizes searching of data and reduces overhead.

Hash Function: A hash function is a mapping between a set of input values and a set of integers, known as hash values. It is usually denoted by H.

The different types of hashing functions are:

1) Division Remainder Method:

According to this method, the key value is divided by an appropriate number, generally a prime number, and the division of remainder is used as the address for the record.

H (K) = K Mod M (Definition)

K → key

M→ size of the table (**usually taken as prime number**)

H (K)→Hash Function

2) Mid-Square Method:

Definition: H (K) = K^2 or some digits taken from K^2

Let us consider the element 7 in our previous example

H (7) = 7^2 = 49

We consider the first digit i.e. 4 from 49 to indicate that the element 7 will be in the address 4 of the array.

There are several collision resolution technique:

(1) Closed Hashing [Open Addressing], (2) Open Hashing [chaining]

Advantages and disadvantages of Closed Hashing:-

Advantages:

Closed Hashing (Open Addressing) is more complex but can be more efficient, especially for small data records.

Disadvantages:

The major drawback of closed hashing is that, as half of the hash table is filled, there is a tendency towards clustering; that is key values are clustered in large groups and as a result sequential search becomes slower and slower. This kind of clustering typically known as primary clustering.

Advantages and disadvantages of chaining:

There are several advantages of chaining method:

1. Open Hashing (chaining) is simpler to implement, and more efficient for large records or sparse tables.
2. Overflow situation never arises. Hash table maintains lists, which can contain any number of key values.
3. Collision resolution can be achieved very efficiently if the lists maintain an ordering of keys, so that keys can be searched quickly.

Drawback of this method is that it is not much efficient technique for small data size.

➲ **5. a) Describe the separate chaining method or collision resolution during hash table construction.**
b) Differentiate between Coalesced chaining and open addressing techniques. [Model Question]

Answer:

a) Collision Resolution using separate chaining: If two keys hash to the same index, the corresponding records cannot be stored in the same location. We must find a place to store the new record if its ideal location is already occupied. One method of doing this is via separate chaining.

In the simplest chained hash table technique, each slot in the array references a linked list of inserted records that collide to the same slot. Insertion requires finding the correct slot, and appending to either end of the list in that slot; deletion requires searching the list and removal.

Separate chaining hash tables have advantages over open addressed hash tables in that the removal operation is simple and resizing the table can be postponed for a much longer time because performance degrades more gracefully even when every slot is used. Indeed, many chaining hash tables may not require resizing at all since performance degradation is linear as the table fills. For example, a chaining hash table containing twice its recommended capacity of data would only be about twice as slow on average as the same table at its recommended capacity.

b) Coalesced chaining: A hybrid of chaining and open addressing coalesced chaining links together chains of nodes within the table itself. Like open addressing, it achieves space usage and (somewhat diminished) cache advantages over chaining. Like chaining, it does not exhibit clustering effects; in fact, the table can be efficiently filled to a high density. Unlike chaining, it cannot have more elements than table slots.

Open addressing: It is a general collision resolution scheme for a hash table. In case of collision, other positions of the hash table are checked (**a probe sequence**) until an empty position is found.

The different types of Open addressing scheme includes

a) Linear Probing (Sequential Probing)
b) Quadratic Probing
c) Double Hashing (Re Hashing)

➲ **6. Write an algorithm to insert a node in AVL tree.** [Model Question]

Answer:

Following steps are followed for insertion in a AVL tree:
Let the newly inserted node be w
1) Perform standard BST insert for w.
2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

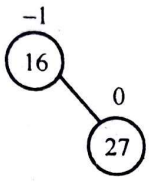
a) y is left child of z and x is left child of y (Left Left Case)
b) y is left child of z and x is right child of y (Left Right Case)
c) y is right child of z and x is right child of y (Right Right Case)
d) y is right child of z and x is left child of y (Right Left Case)

➲ **7. Create an AVL tree using the following sequence of data: 16, 27, 9, 11, 2, 15, 99, 36, 54, 81, 63, 72.** **[Model Question]**

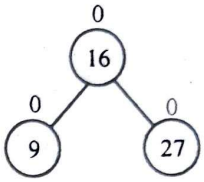
Answer:

i)

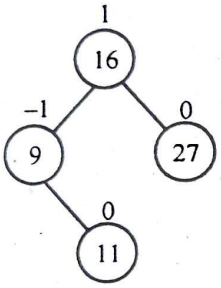
ii)



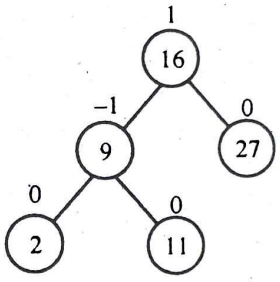
iii)

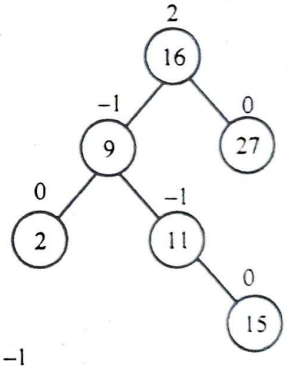


iv)

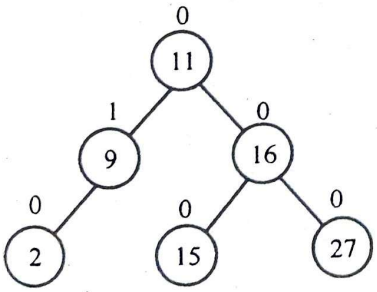


v)

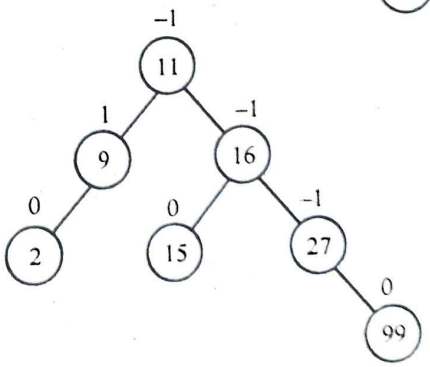




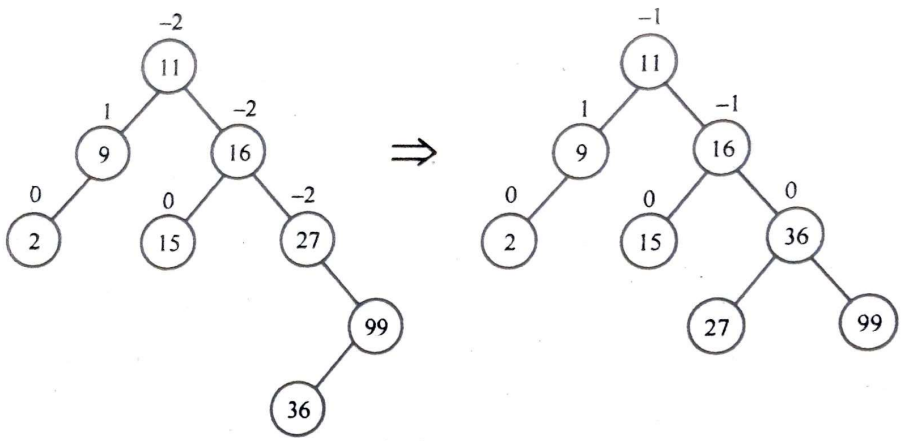
vi)



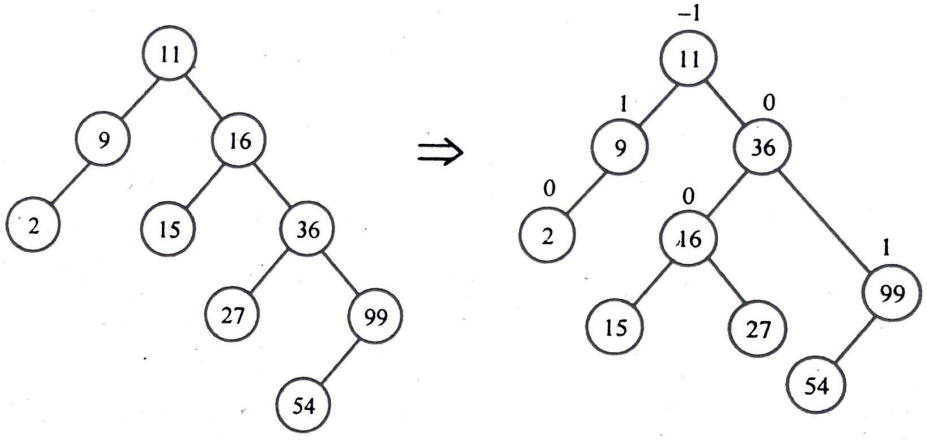
vii)



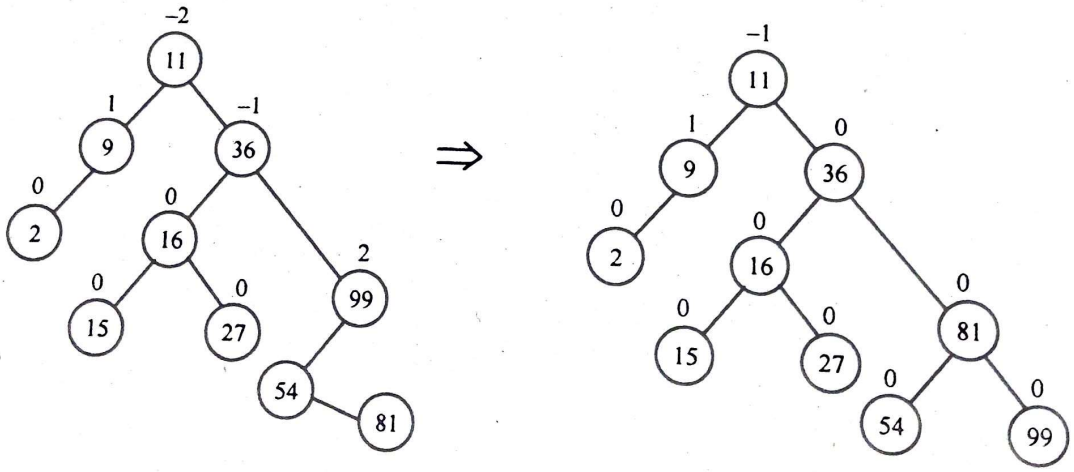
viii)


-2
11
1
9
-2
16
0
2
0
15
-2
27
99
36
⇒
-1
11
1
9
-1
16
0
2
0
15
0
36
27
99

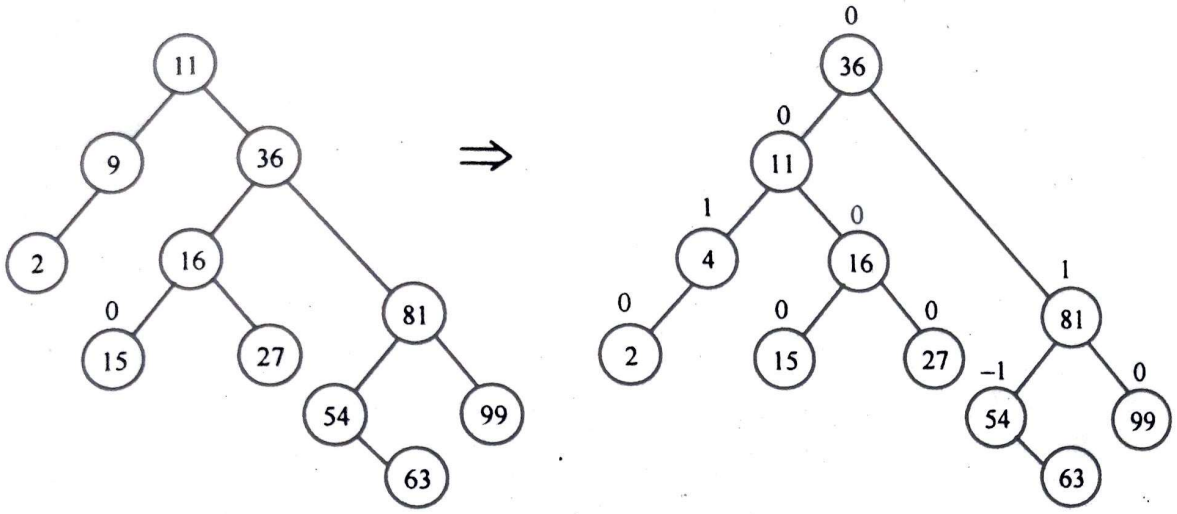
ix)


11
9
16
2
15
36
27
99
54
⇒
-1
11
1
9
0
36
0
2
0
16
1
99
15
27
54

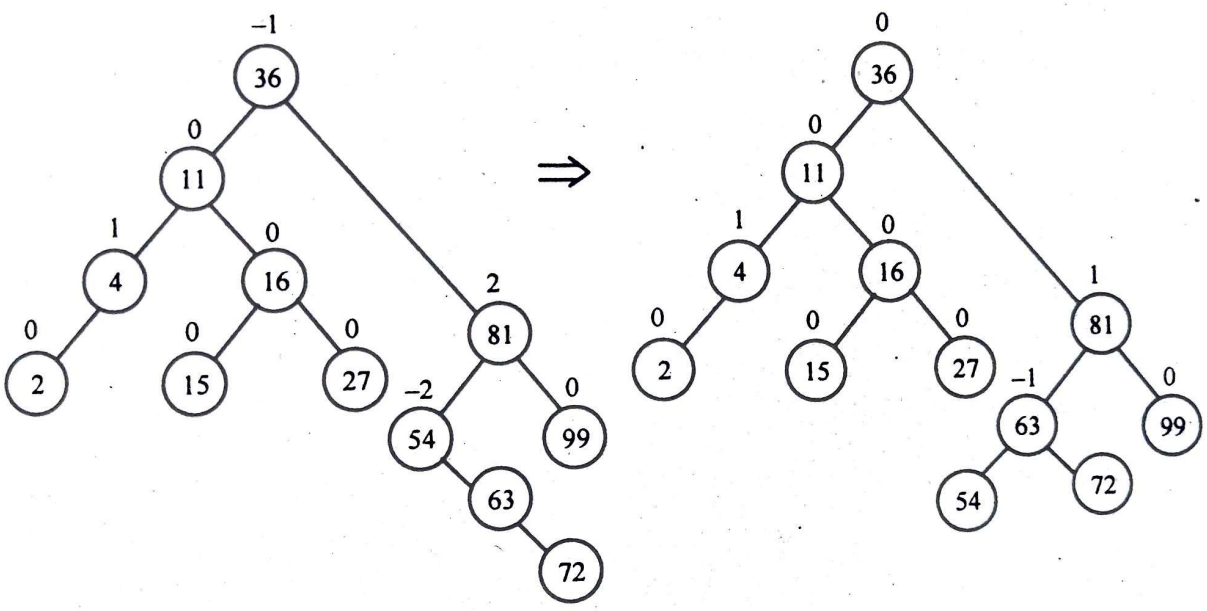
x)


-2
11
1
9
-1
36
0
2
0
16
2
0
15
0
27
99
54
81
⇒
-1
11
1
9
0
36
0
2
0
16
0
0
15
27
0
81
0
54
0
99

xi)


11
9
36
2
16
81
0
15
27
54
99
63
⇒
0
36
0
11
1
4
0
16
1
0
2
0
15
0
27
81
–1
0
54
99
63

xii)

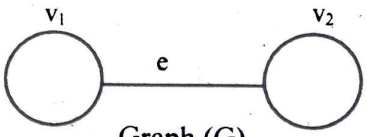

–1
36
0
11
1
4
0
16
2
0
2
0
15
0
27
81
–2
0
54
99
63
72
⇒
0
36
0
11
1
4
0
16
1
0
2
0
15
0
27
81
–1
0
63
99
54
72

Unit 4: Graph

Unit at a glance:

4.1 Graph

A graph is a mathematical tool used to represent a physical problem. It is also used to model networks, data structures, scheduling, computation and a variety of other systems where the relationship between the objects in the system plays a dominant role. Mathematically a graph consists of two sets V and E where, V is a finite, non-empty set of vertices and E is a set of pairs of vertices. The line joining a pair of vertices is called an edge.



Graph (G)

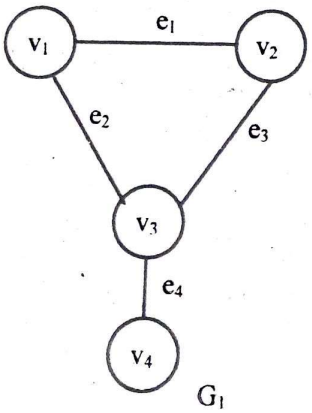
The above figure represents a graph with v_1, v_2 as the set of vertices and e, an edge, joining the two vertices v_1 and v_2, v_1, v_2 and e together form graph . The notation $v_1 - v_2$ means that the edge e has v_1 and v_2 as end points. In other words e connects vertices v_1 and v_2 and that v_1 and v_2 are adjacent.

4.2 Types of Graph

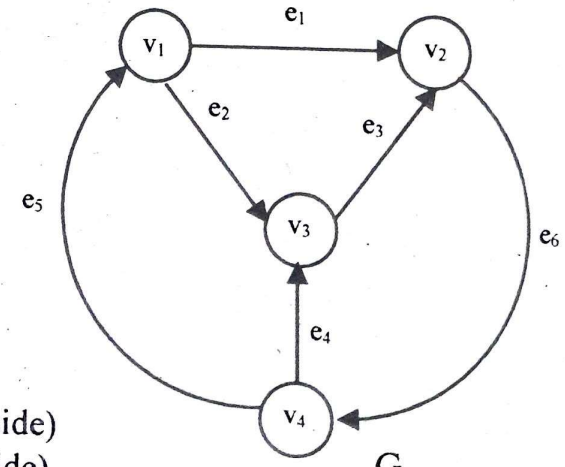
A graph often symbolized as G can be of two types:

- **Undirected graph:** where a pair of vertices representing an edge is unordered.
- **Directed graph:** where a pair of vertices representing an edge is ordered.

The figures shown below represent both types.



G_1



G_2

Undirected Graph (left side)
Directed Graph (right side)

4.3 Some important definitions

- **Weighted/ Un-weighted Graph:** A *weighted graph* is a graph for which each edge has an associated *weight*, usually given by a *weight function w*: $E \rightarrow R$
- An *un-weighted graph* is a graph for which each edge has no associated *weight*.
- **Sub-graph:** A sub graph of G=G(V,E) is a subset W of the vertex set V together with all of the edges that connect pairs of vertices in W.
- **Degree:** ***Degree*** of a vertex in an undirected graph is the number of edges incident on it. In a directed graph, the ***out degree*** of a vertex is the number of edges leaving it and the ***in degree*** is the number of edges entering it.
- **Cut Vertex/ Articulation Point:** In graph theory, a bi-connected component (or 2-connected component) is a maximal bi-connected sub-graph. Any connected graph decomposes into a tree of bi-connected components called the block tree of the graph. The blocks are attached to each other at shared vertices called cut vertices or articulation points.
- **Pendant node:** A leaf vertex is also known pendent vertex, is a vertex with degree one.
- **Clique:** A clique in an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge.
- **Complete Graph:** A complete graph is an undirected/directed graph in which every pair of vertices is adjacent. If (u, v) is an edge in a graph G, we say that vertex v is adjacent to vertex u.
- **Connected Components:** A connected component of an undirected graph is a sub graph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices.
- **Strongly Connected Component:** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex.
- **Weakly Connected Component:** A weakly connected graph can be thought of as a digraph in which every vertex is "reachable" from every other but not necessarily following the directions of the arcs.
- **Path:** A simple path is a walk with no repeated nodes
- **Shortest Path:** The shortest path is a path between two vertices(or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.
- **Isomorphism:** n graph theory, an isomorphism of graphs G and H is a bijection between the vertex sets of G and H

$$f : V(G) \rightarrow V(H)$$

- such that any two vertices u and v of G are adjacent in G if and only if f(u) and f(v) are adjacent in H. This kind of bijection is commonly called "edge-preserving bijection", in accordance with the general notion of isomorphism being a structure-preserving bijection.

4.4 Graph Traversals

A graph can be traversed in two ways:

- ***Depth first search traversal:*** often analogous to preorder traversal of an ordered tree.
- ***Breadth first search traversal:*** often analogous to level-by-level traversal of an ordered tree.

Apart from the two types of graph i.e. the undirected type and the directed type, a graph can also have other forms.

Empty graph: A graph G with no edges is called an empty graph.

Null graph: A graph G with no vertices (and hence no edge) is called a null graph.

Connected graph: A graph G is connected if there exists a path between every pair of vertices. In other words, we can also say that if we can traverse every other node from one node by means of any traversal algorithm, then also we can say that the graph is connected.

Simple graph: **A** graph G is simple if it has no parallel edge or self loops.

4.5 Spanning Tree

- Given a connected, undirected graph, a spanning tree of that graph is a sub graph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree.
- A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of minimum spanning trees for its connected components.

4.6 Shortest path

- In graph theory, the shortest path problem is the problem of finding a path between two vertices in a weighted graph such that the sum of the weights of its constituent edges is minimized. An example is finding the quickest way to get from one location to another on a road map; in this case, the vertices represent locations and the edges represent segments of road and are weighted by the cost needed to travel that segment.
- There are several variations according to whether the given graph is undirected, directed, or others. For undirected graphs, the shortest path problem can be formally defined as follows. Given a weighted graph (that is, a set V of vertices, a set E of edges, and a real-valued weight function $f : E \rightarrow R$), and elements v and v' of V, find a path P from v to a v' of V so that

$$\sum_{p \in P} f(p)$$

is minimal among all paths connecting v to v' .

The problem is also sometimes called the single-pair shortest path problem.

There are several other variations of this problem.

- **The single-source shortest path problem:** In this problem we have to find shortest paths from a source vertex v to all other vertices in the graph.
- **The single-destination shortest path problem**: Here we have to find shortest paths from all vertices in the directed graph to a single destination vertex v. This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.
- **The all-pairs shortest path problem**: In this problem, we have to find shortest paths between every pair of vertices v, v' in the graph.

The most important algorithms for solving this problem are:

- **Dijkstra's algorithm** solves the single-source shortest path problems.
- **Bellman-Ford algorithm** solves the single source problem if edge weights may be negative.
- **A* search algorithm** solves for single pair shortest path using heuristics to try to speed up the search.
- **Floyd-Warshall** algorithm solves all pairs shortest paths.
- **Johnson's algorithm** solves all pairs shortest paths, and may be faster than Floyd-Warshall on sparse graphs.

Perturbation theory finds (at worst) the locally shortest path.

A. Choose the correct answer from the given alternatives in each of the following:

[WBSCTE 2022]

1. Which following statement is true?
   (a) All graphs are trees
   (b) All trees are graphs
   (c) Some trees are graphs
   (d) No tree is a graph

Answer: (b)

[WBSCTE 2022]

2. Which following statement is true?
   (a) All graphs are trees
   (b) All trees are graphs
   (c) Some trees are graphs
   (d) No tree is a graph

Answer: (b)

3. Which of the following is true? [WBSCTE 2022]
   (a) Prim's algorithm initializes with a vertex
   (b) Prim's algorithm initializes with an edge
   (c) Prim's algorithm initializes with a vertex which has smallest edge
   (d) Prim's algorithm initializes with a forest

Answer: (a)

4. Worst case is the worst case time complexity of Prim's algorithm if adjacency matrix is used? [WBSCTE 2022]
   (a) $O(\log V)$ (b) $O(V2)$ (c) $O(E2)$ (d) $O(V \log E)$

Answer: (b)

5. Floyd Warshall Algorithm used to solve the shortest path problem has a time complexity of [WBSCTE 2022]
   (a) O(V*V) (b) O(V*V*V) (c) O(E*V) (d) O(E*E)

Answer: (b)

6. A path is [Model Question]
   (a) a closed walk with no vertex repetition
   (b) an open walk with no vertex repetition
   (c) an open walk with no edge repetition
   (d) a closed walk with no edge repetition

Answer: (c)

7. Maximum number of edges in a n-node undirected graph without self loop is [Model Question]
   (a) n^2 (b) $\frac{n(n-1)}{2}$ (c) $n-2$ (d) $\frac{(n+1)(n)}{2}$

Answer: (b)

8. A complete directed graph of 5 nodes has..... [Model Question]
   (a) 5 (b) 10 (c) 20 (d) 25

Answer: (b)

B. Fill in the blanks in the following statements:

9. In the ***Shortest Path*** Problem when a node is visited, it is deleted from the edge list. [Model Question]

10. In ***Dijkstra's*** algorithm, a sorted array of edges is required in order to construct a minimal spanning tree. [Model Question]

C. Answer the following questions:

11. Write one difference between path and cycle. [WBSCTE 2022]

Answer:

Path: a trail that connects all vertices.

Cycle: a trail that finishes on the vertex it begins on. No vertex is repeated.

12. What is cut vertex? [WBSCTE 2022]

Answer:

A vertex whose deletion along with incident edges results in a graph with more components than the original graph.

13. What is sink in a graph? [WBSCTE 2022]

Answer:

A local sink is a node of a directed graph with no exiting edges, also called a terminal. A global sink (often simply called a sink) is a node in a directed graph which is reached by all directed edges.

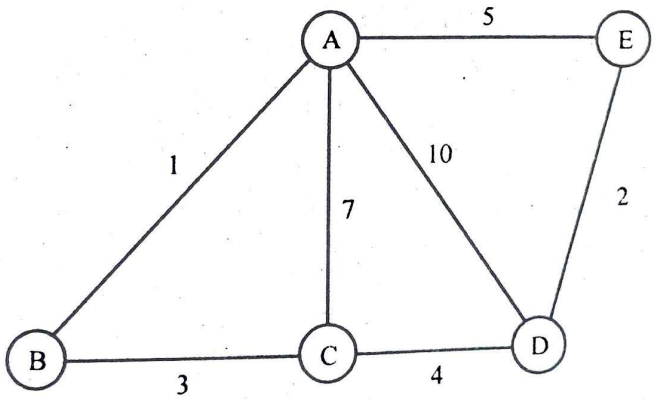
14. Define in-degree of a graph. [WBSCTE 2022]

Answer:

Indegree of vertex V is the number of edges which are coming into the vertex V.

Long Answer Type Questions

➲ **1. a) Explain path, cycle and directed acyclic graph with proper examples.**
b) Use Kruskal's algorithm to find out the minimum spanning tree for the graph. [WBSCTE 2022]

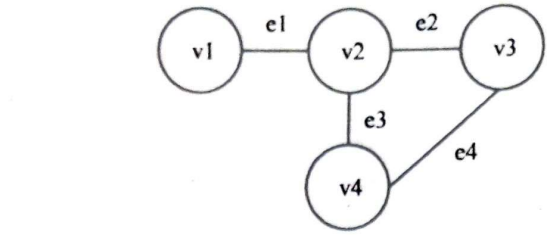


Answer:

a) Path: In practical terms, a path is a sequence of non-repeated nodes connected through edges present in a graph. We can understand a path as a graph where the first and the last nodes have a degree one, and the other nodes have a degree two.

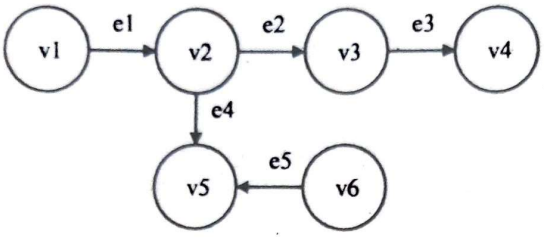
If the graph contains directed edges, a path is often called dipath. Thus, besides the previously cited properties, a dipath must have all the edges in the same direction.
The following image shows particular examples of paths and dipaths:

PATH EXAMPLES



v1 - (e1) - v2 - (e2) - v3

v1 - (e1) - v2 - (e3) - v4 - (e4) - v3



v1 - (e1) - v2 - (e2) - v3 - (e3) - v4

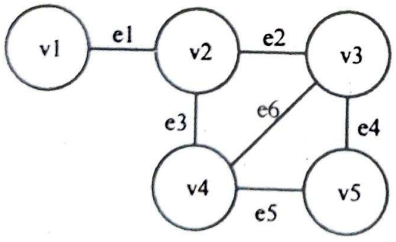
v1 - (e1) - v2 - (e4) - v5

v6 - (e5) - v5

Cycle:

A cycle consists of a sequence of adjacent and distinct nodes in a graph. The only exception is that the first and last nodes of the cycle sequence must be the same node.
In this way, we can conclude that every cycle is a circuit, but the contrary is not true. Furthermore, another inferring is that every Hamiltonian circuit is also a cycle.
So, we call a graph with cycles of cyclic graphs. Oppositely, we call a graph without cycles of acyclic graphs. Finally, if a connected graph does not have cycles, we call it a tree.
The image next presents an example of a cyclic graph, acyclic graph, and tree:

CYCLIC GRAPH



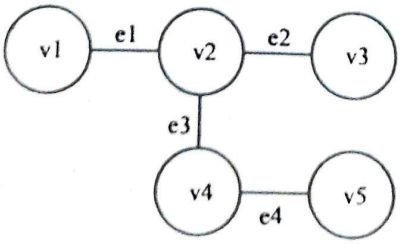
CYCLIC EXAMPLE

v2 - (e2) - v3 - (e4) - v5 - (e5) - v4 - (e3) - v2

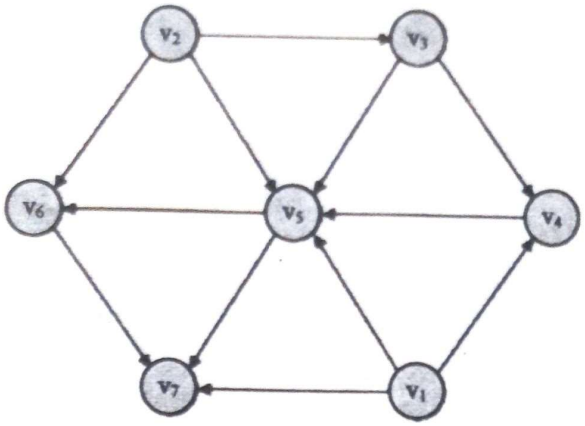
v2 - (e2) - v3 - (e6) - v4 - (e3) - v2

v3 - (e4) - v5 - (e5) - v4 - (e6) - v3

ACYCLIC AND TREE GRAPH



Directed Acyclic Graph:
A DAG is a directed graph that contains no directed cycles.



a DAG

b) ***Refer to Question No. 4 and 6 of Long Answer Type Questions.***

➲ **2. What is a complete graph? Show that the sum of degree of all the vertices in a graph is always even.** **[Model Question]**

Answer:
A graph is said to be complete if there exist an edge between every pair of vertices.
At first we have to know, the degree of a vertex. Degree of a vertex is defined as no. of edges incident on a particular vertex with the self loop counted twice. Now, let us take the sum of degree of all the vertices. Now, this summation is an even number, because each edge contributes twice when we are calculate degree of different vertices. Now, if we take the summation of all the degrees that is nothing but the twice the number of edges $\sum_{i=1}^{n} d(v_i) = 2e \Rightarrow$ even number

➲ **3. Write a note on Sequential and linked representation of graphs in memory of computer.** **[Model Question]**

OR,

Discuss the two different ways of representing a graph. **[Model Question]**

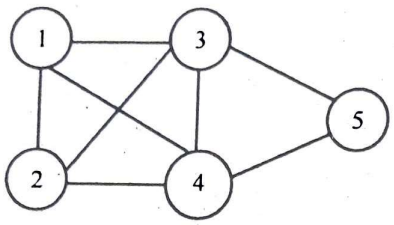
Answer:
There are three ways by which graphs can be represented in memory.

i) Adjacency matrices.
ii) Adjacency list.
iii) Adjacency multilists.

i) ***Adjacency matrices:*** For a graph G with n vertices the adjacency matrix is a 2D array with n×n element.

For undirected graphs
A [i, j] = 1, if there is an edge between i & j
A[i, j]=0, if there is no edge between i & j.

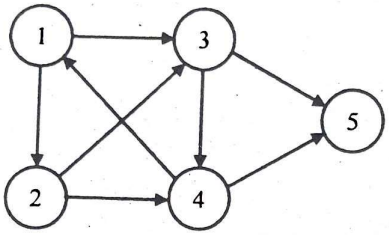
For directed graphs
A[i, j]=1, if there is an edge directed i & j
A[i, j]=0, otherwise



Undirected graph

Adjacency Matrix

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	1	0
3	1	1	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0



Directed graph

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	1	1	0
3	0	0	0	1	1
4	1	0	0	0	1
5	0	0	0	0	0

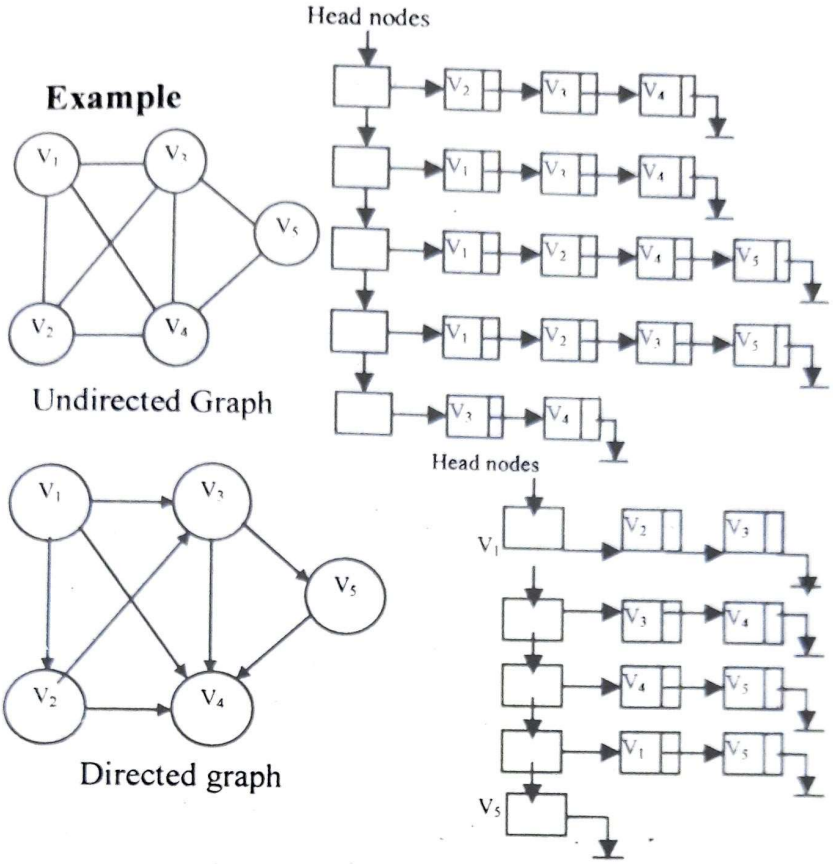
The adjacency matrix for an undirected graph is symmetrical i.e., diagonal elements are zero.

Advantage: It is possible to determine whether an edge is connecting two vertices or not.

Disadvantage:

- It requires n x n memory locations. If the graph is not complete, there will be a number of zero entries & this will waste the memory area. To avoid this, the adjacency list method is used.
- There will be two entries for the same edge. This information is redundant.

ii) ***Adjacency List:*** It is a linked list representation of a graph where each vertex represents a head node. Each head node has a list associated with it which represents the edges. The nodes of the list form a path between the head node and all other reachable vertices in the graph from that head node.



Drawbacks: When a graph is large, it is necessary to handle to many numbers of pointers, which can be complex.

iii) ***Adjacency Multilist:*** From the adjacency list representation of an undirected graph, it is clear that each edge (Vi, Vj) is represented by two entries, one is list Vi and another is list Vj. This is unnecessary wastage of the memory as the information present is same at both the nodes (1,2)= (2,1).

To avoid this, the adjacency multilist is used. Using this representation, the nodes are shared amongst the several lists.

In this method for each edge of the graph, there will be exactly one node. But this node will provide the information about two more nodes to which it is incident.

The node structure with this representation will be:

m	V_1	V_2	P_1	P_2

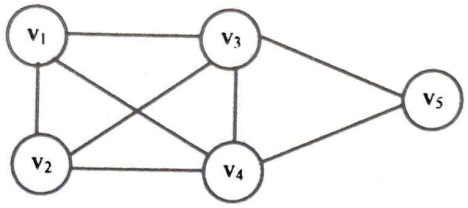
m --- tagfield

V_1, V_2 ....Vertex

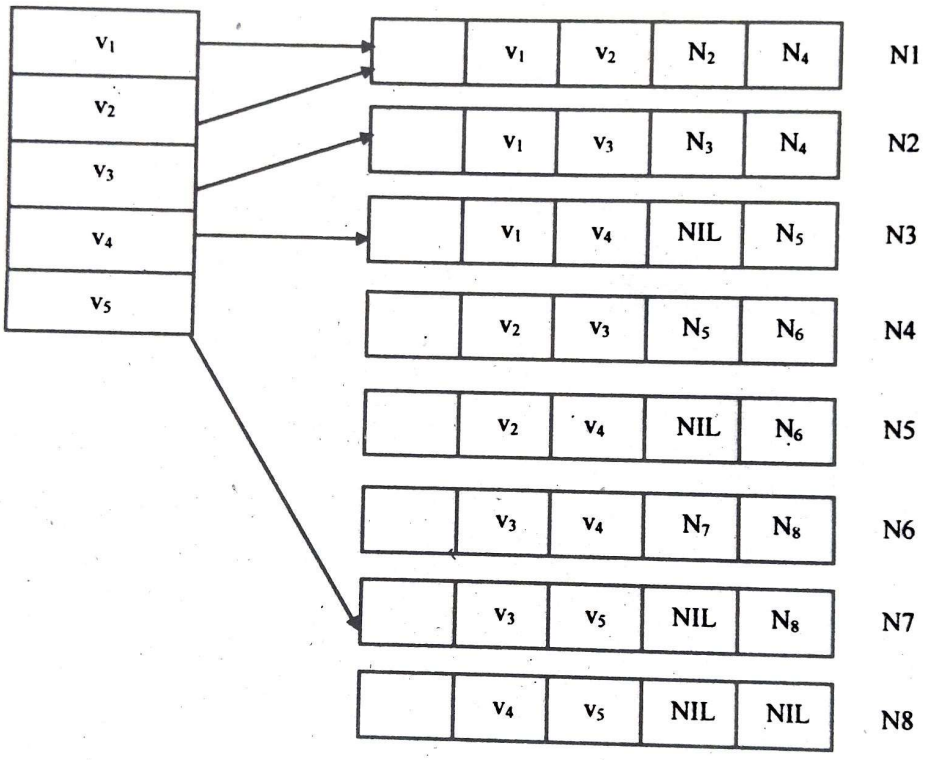
P1, P_2----- Incident paths

m is a one bit mark field that is used to indicate whether or not the edge has been examined.

Consider the following graph.



Total distinct edges are:
$\{v_1, v_2\}$, $\{v_1, v_3\}$, $\{v_1, v_4\}$, $\{v_2, v_3\}$, $\{v_2, v_4\}$, $\{v_3, v_4\}$, $\{v_3, v_5\}$ & $\{v_4, v_5\}$. All other edges represent the same information, because this is an undirected graph.
The adjacency multilist representation is as follows



The lists are: -
Vertex 1 : N1 → N2 → N3
Vertex 2 : N1 → N4 → N5
Vertex 3 : N2→ N4 → N6 → N7
Vertex 4 : N3 → N5 → N6 → N8
Vertex 5 : N7 → N8

➲ **4. Define spanning tree. Write a method or algorithm which will generate a minimum cost spanning tree. Explain your method with an example.**

[Model Question]

Answer:

Spanning Tree: Given a connected, undirected graph, a spanning tree of that graph is a sub graph, which is a tree and connects all the vertices together. A single graph can have many different spanning trees.

A **minimum spanning tree** (MST) or **minimum weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree.

Kruskal's algorithm:

- create a forest F (a set of trees), where each vertex in the graph is a separate tree
- create a set S containing all the edges in the graph
- while S is nonempty
    - remove an edge with minimum weight from S
    - if that edge connects two different trees, then add it to the forest, combining two trees into a single tree
    - Otherwise discard that edge.
    - End

Example

This is our original graph. The numbers near the arcs indicate their weight. None of the arcs is highlighted.

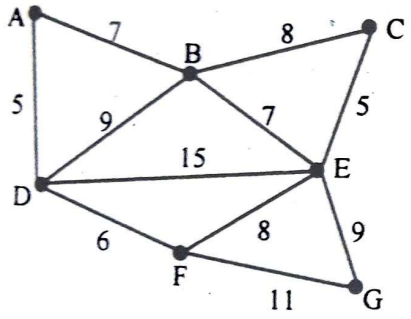


Fig: 1

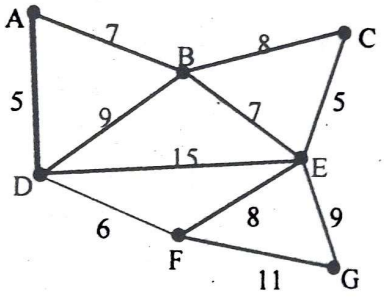


Fig: 2

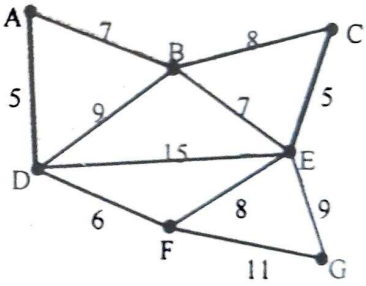


Fig: 3

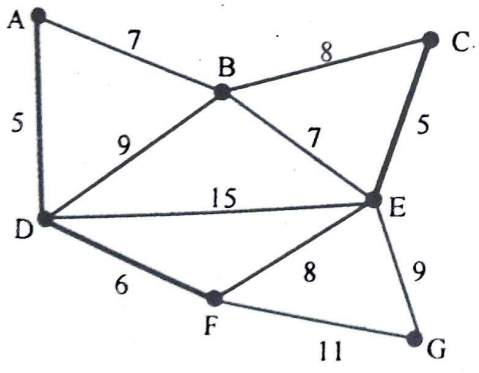


Fig: 4

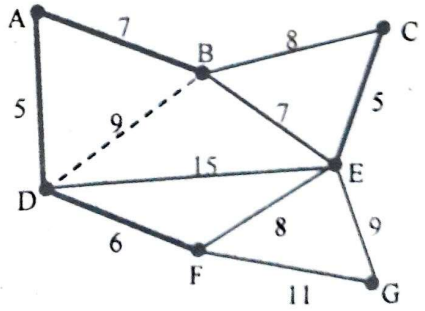


Fig: 5

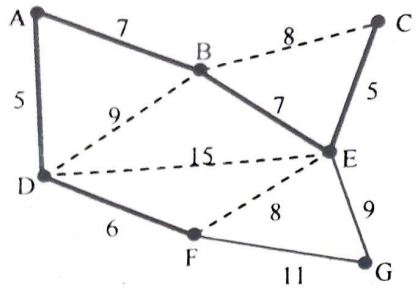
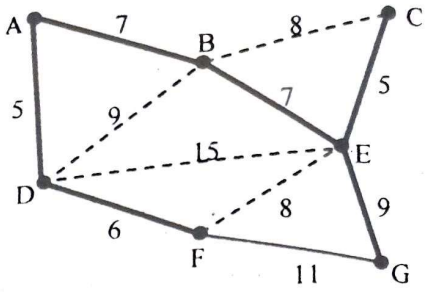


Fig: 6



Final figure

➲ **5. Define minimal spanning tree.** **[Model Question]**
Discuss Prim's algorithm with example. **[Model Question]**

Answer:

A **minimal spanning tree** (MST) or **minimal weight spanning tree** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a **minimal spanning forest**, which is a union of minimum spanning trees for its connected components.

Prim's algorithm:

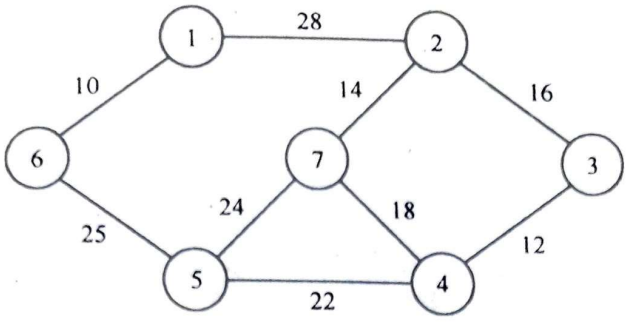
Prim's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it will only find a minimum spanning tree for one of the connected components. It works as follows:

- create a tree containing a single vertex, chosen arbitrarily from the graph
- create a set containing all the edges in the graph
- loop until every edge in the set connects two vertices in the tree
  - remove from the set an edge with minimum weight that connects a vertex in the tree with a vertex not in the tree
  - add that edge to the tree

Prim's algorithm can be shown to run in time, which is O (m + n log n) where m is the number of edges and n is the number of vertices.

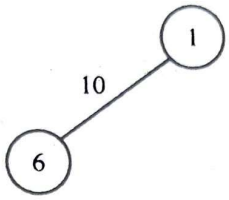
For example,

Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-

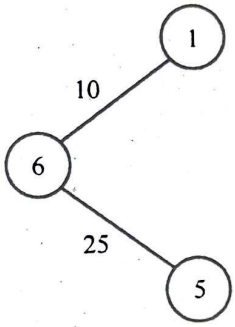


The above discussed steps are followed to find the minimum cost spanning tree using Prim's Algorithm-

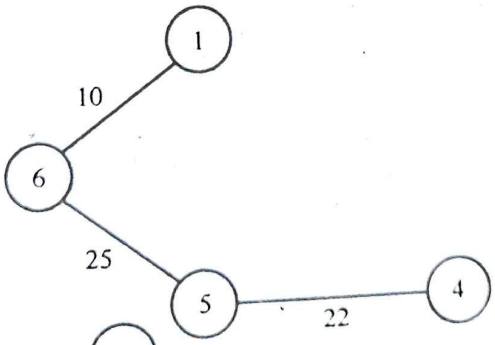
Step-01:



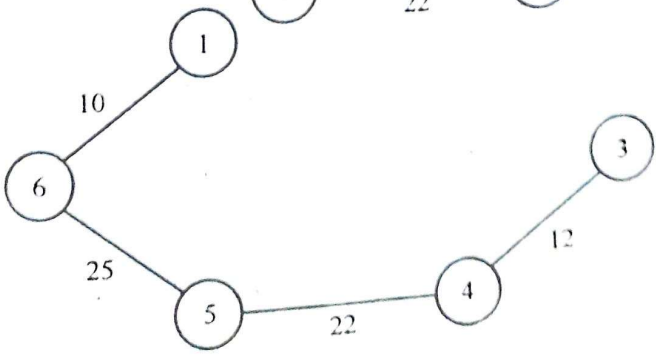
Step-02:



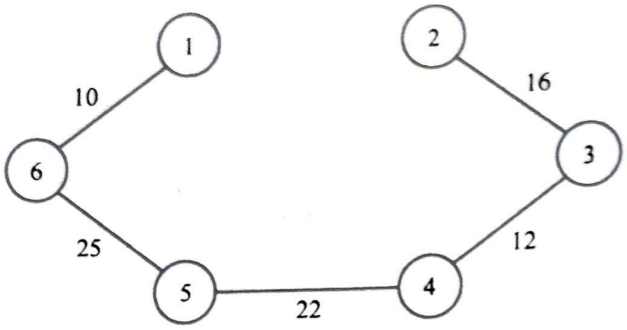
Step-03:



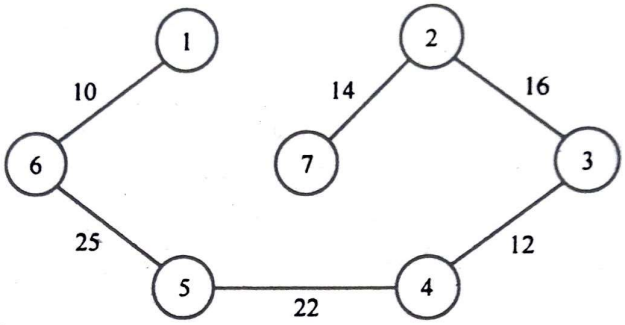
Step-04:



Step-05:



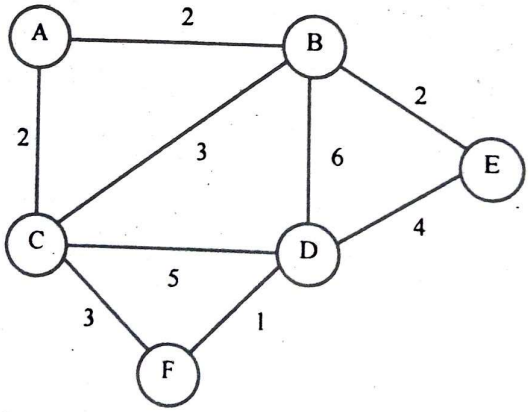
Step-06:



Since all the vertices have been included in the MST, so we stop.
Now, Cost of Minimum Spanning Tree
= Sum of all edge weights
= 10 + 25 + 22 + 12 + 16 + 14
= 99 units

➲ **6. Consider the graph given below. Find the minimum spanning tree of this graph using Kruskal's algorithm.** **[Model Question]**

Answer:

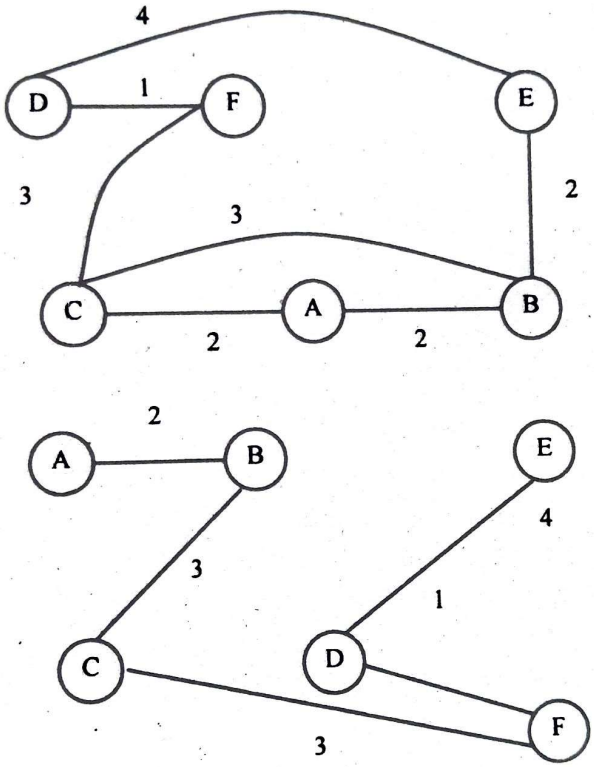


Applying Kruskal's algorithm:
Total vertices = 6
Total edges = 9

∴ MST will have = 5 edges

Weight	Src	Dest
1	D	F
2	A	B
2	A	C
2	B	E
3	B	C
3	C	F
4	D	E
5	C	D
	B	D

6



It is the MST of the graph given.

[Model Question]

➲ **7. Discuss about the following terminology**

a) Indegree
b) Sink
c) Cycle
d) Network

Answer:

a) In a directed graph the **indegree** of a vertex v is the number of edges terminating at v.

b) A local sink is a node of a directed graph with no exiting edges – it is also called a terminal. A global sink (often simply called a sink) is a node in a directed graph which is reached by all directed edges.

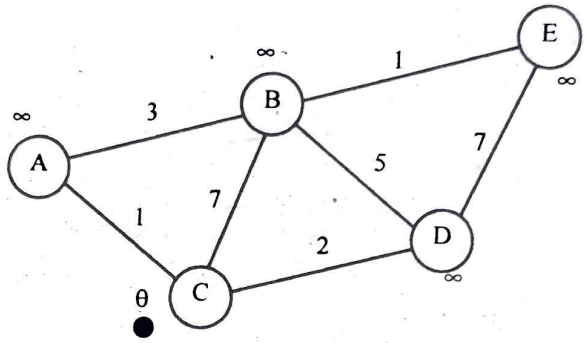
c) In a graph, a cycle is a sequence $v_0, e_1, v_1, \ldots, e_k, v_k (k \geq 1)$ of alternately vertices and edges (where e_i is an edge joining v_{i-1} and v_i), with all the edges different and all the vertices different, except that $v_0 = v_k$.

d) A network is a set of items (nodes or vertices) connected by edges or links. A network is represented by a graph using adjacency matrix ususally.

➲ **8. Explain Dijkstra's algorithm for finding the shortest distance between two given nodes. Consider a graph of your choice having at least five nodes and specific weight assigned to each edge.** **[Model Question]**

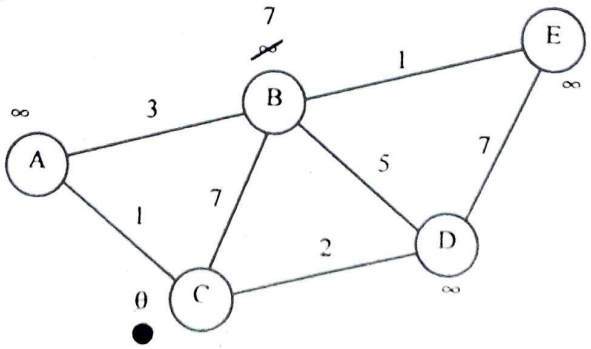
Answer:

During the algorithm execution, we'll mark every node with its minimum distance to node C (our selected node). For node C, this distance is 0. For the rest of nodes, as we still don't know that minimum distance, it starts being infinity (∞):

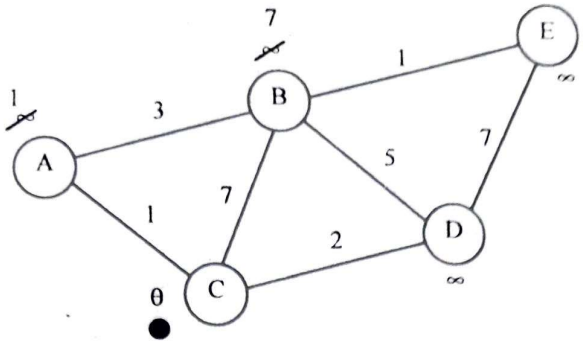


We'll also have a current node. Initially, we set it to C (our selected node). In the image, we mark the current node with a red dot.

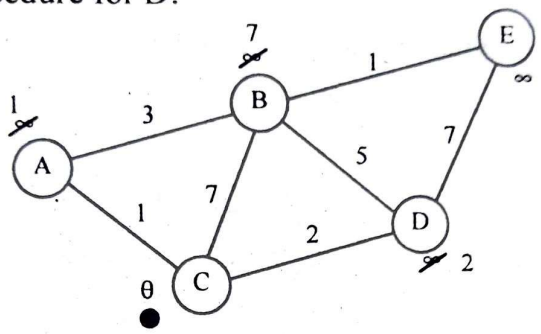
Now, we check the neighbours of our current node (A, B and D) in no specific order. Let's begin with B. We add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects our current node with B (in this case, 7), and we obtain 0 + 7 = 7. We compare that value with the minimum distance of B (infinity); the lowest value is the one that remains as the minimum distance of B (in this case, 7 is less than infinity):



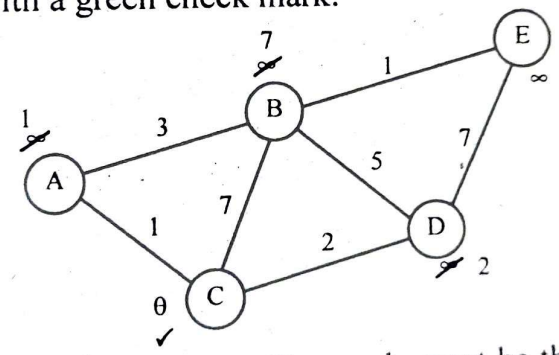
Now, let's check neighbour A. We add 0 (the minimum distance of C, our current node) with 1 (the weight of the edge connecting our current node with A) to obtain 1. We compare that 1 with the minimum distance of A (infinity), and leave the smallest value:



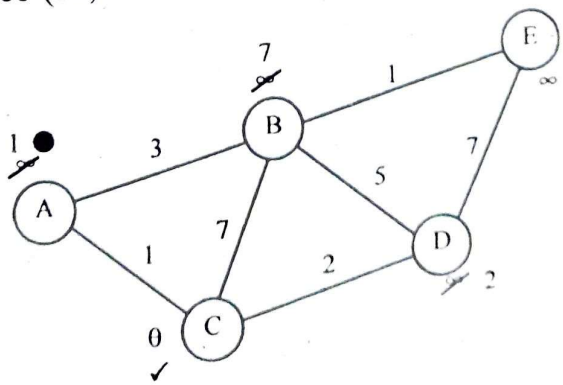
OK. Repeat the same procedure for D:



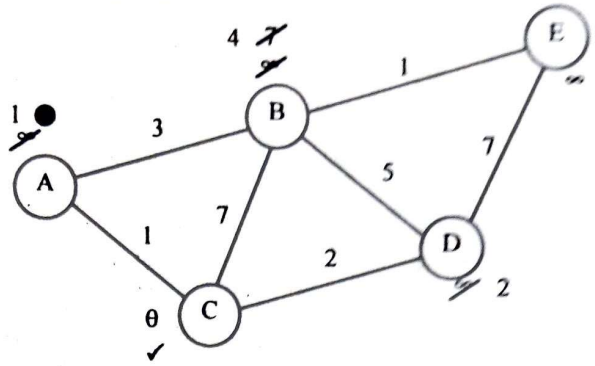
We have checked all the neighbours of C. Because of that, we mark it as *visited*. Let's represent visited nodes with a green check mark:



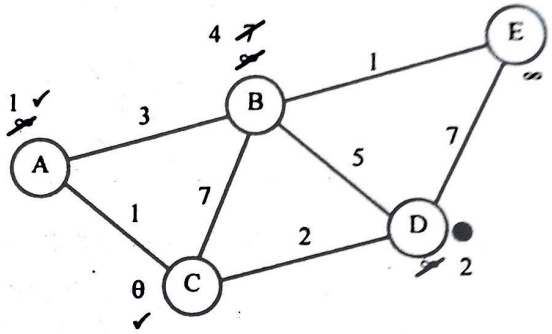
We now need to pick a new current node. That node must be the unvisited node with the smallest minimum distance (so, the node with the smallest number and no check mark). That's A.



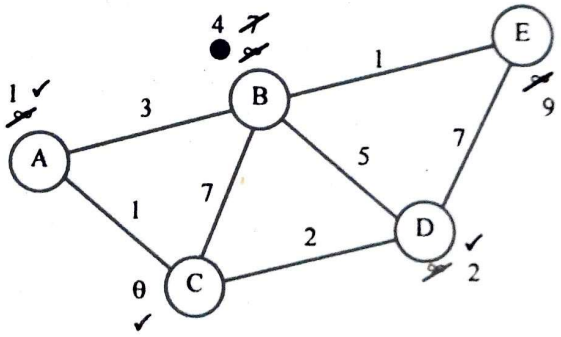
And now we repeat the algorithm. We check the neighbours of our current node, ignoring the visited nodes. This means we only check B.
For B, we add 1 (the minimum distance of A, our current node) with 3 (the weight of the edge connecting A and B) to obtain 4. We compare that 4 with the minimum distance of B (7) and leave the smallest value: 4.

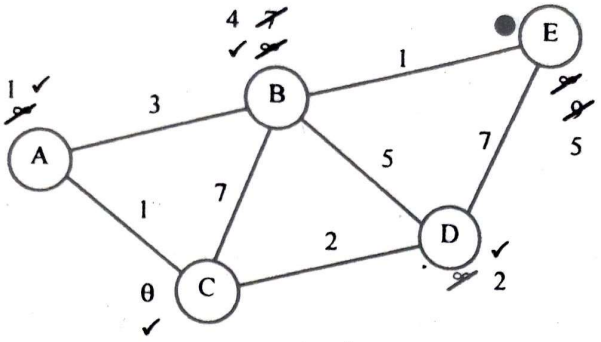


Afterwards, we mark A as visited and pick a new current node: D, which is the non-visited node with the smallest current distance.

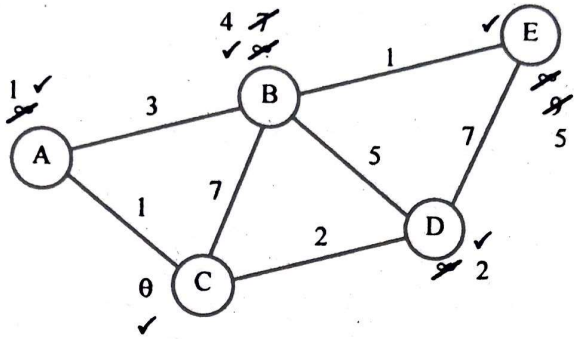


We repeat the algorithm again. This time, we check B and E.
For B, we obtain 2 + 5 = 7. We compare that value with B's minimum distance (4) and leave the smallest value (4). For E, we obtain 2 + 7 = 9, compare it with the minimum distance of E (infinity) and leave the smallest one (9).
We mark D as visited and set our current node to B.
We only need to check E. 4 + 1 = 5, which is less than E's minimum distance (9), so we leave the 5. Then, we mark B as visited and set E as the current node.





E doesn't have any non-visited neighbours, so we don't need to check anything. We mark it as visited.



As there are not univisited nodes, we are done! The minimum distance of each node now actually represents the minimum distance from that node to node C.

➲ 9. Explain Bellman Ford algorithm for finding single source shortest path.

[Model Question]

Answer:

Bellman Ford algorithm is used to find the shortest path from the source vertex to remaining all other vertices in the weighted graph. We can find an optimal solution to this problem using dynamic programming.

- It is slower compared to Dijkstra's algorithm but it can handle ***negative weights*** also.
- If the graph contains negative –weight cycle (edge sum of the cycle is negative), it is not possible to find the minimum path, because on every iteration of cycle gives a better result.
- Bellman Ford algorithm can detect a negative cycle in the graph but it cannot find the solution for such graphs.
- Like Dijkstra, this algorithm is also based on the principle of relaxation. The pathis updated with better values until it reaches the optimal value.
- Dijkstra uses apriority queue to greedily select the closest vertex and perform relaxation on all its adjacent outgoing edges. By contrast, Bellman Ford relaxes all the edges |V|- 1 time. Bellman Ford runs in O(|V|.|E|) time.

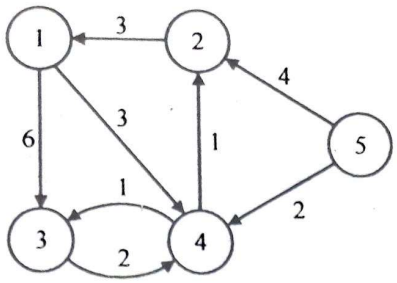
Algorithm

Algorithm BELLMAN-FORD (G)

```
// Description : Find the shortest path from source vertex to all
other vertices using Bellman-Ford algorithm

// Inputs:
Weighted graph G = (V, E, W)
w(u, v): Weight of edge (u, v)
d[u] : distance of vertex u from source
π[u]: Successor of vertex u

// Output: Shortest distance of each vertex from given source
vertex.

// Initialization
for v ∈ V do
    d[v] ← ∞
    π[v] ← NULL
end

d[s] ← 0

// Relaxation
for i ← 1 to |V| - 1 do
    for each edge (u, v) ∈ E do
        if d[u] + w (u, v) < d[v] then
            d[v] ← d[u] + w(u, v)
            π[v] ← u
        end
    end
end

// Check for negative cycle
for each edge (u, v) ∈ E do
    if d[u] + w(u, v) < d[v] then
        error "Graph contains negative cycle"
    end
end
return d, π
```

Examples of Bellman Ford Algorithm

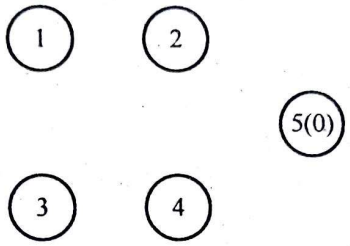
Example: Find the shortest path from source vertex 5 for a given graph



Solution:

Initialize the distance: Set distance of source vertex to 0 and every other vertex to ∞.

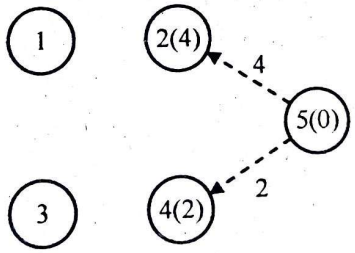
V	1	2	3	4	5
d[v]	∞	∞	∞	∞	0
p[v]	/	/	/	/	–



Iteration 1:

Edges <5, 2> and <5, 4> will be relaxed as their distance will be updated to 4 and 2 respectively.

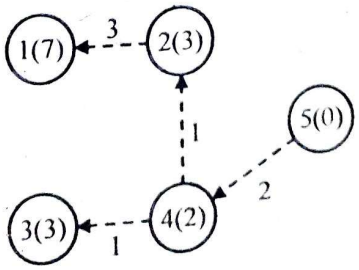
V	1	2	3	4	5
d[v]	∞	4	∞	2	0
p[v]	/	5	/	5	–



Iteration 2:

Edges <2, 1>, <4, 3> and <2, 4> will be relaxed as their distance will be updated to 7, 3 and 3 respectively.

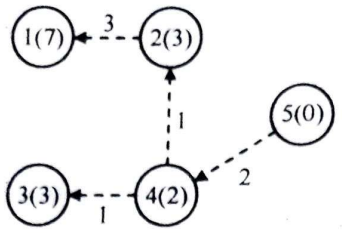
V	1	2	3	4	5
d[v]	7	3	3	2	0
p[v]	2	4	4	5	–



Iteration 3:

Edge <2, 1>will be relaxed as its distance will be updated to 6.

V	1	2	3	4	5
d[v]	6	3	3	2	0
p[v]	2	4	4	5	–

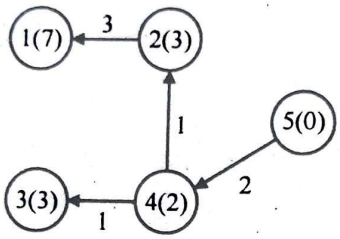


Iteration 4:

No edge relaxed

So the final shortest path from source vertex 5 is:

V	1	2	3	4	5
d[v]	6	3	3	2	0
p[v]	2	4	4	5	–



The shortest path from any vertex can be found by starting at the vertex and following its predecessor π's back to the source. For example, starting at vertex 1, $u_1.\pi = 2$, $u_2.\pi = 4$, $u_4.\pi = 5 \Rightarrow$ the shortest path to vertex 1 is $\{5, 4, 2, 1\}$.

Negative cycle check:

We have to test following condition for each edge.

```
if
d[u] + w(u, v) < d[v]
then
    error "Graph contains negative cycle"
end
```

$$u_1.d + w(1,3) < v_3.d \Rightarrow 6+6 \geq 4$$

$$u_1.d + w(1,4) < v_4.d \Rightarrow 6+3 \geq 2$$

$$u_2.d + w(2,1) < v_1.d \Rightarrow 3+3 \geq 6$$

$$u_3.d + w(3,4) < v_4.d \Rightarrow 3+2 \geq 2$$

$$u_4.d + w(4,2) < v_2.d \Rightarrow 2+1 \geq 3$$

$$u_4.d + w(4,3) < v_3.d \Rightarrow 2+1 \geq 3$$

$$u_5.d + w(5,2) < v_2.d \Rightarrow 0+4 \geq 3$$

$$u_5.d + w(5,4) < v_4.d \Rightarrow 0+2 \geq 2$$

None of the above conditions are satisfied, so the graph does not contain any negative cycle.

➲ 11. Explain Floyd-Warshall all pairs shortest path algorithm. [Model Question]

Answer:

The Floyd-Warshall algorithm is a popular algorithm for finding the shortest path for each vertex pair in a weighted directed graph.

In all pair shortest path problem, we need to find out all the shortest paths from each vertex to all other vertices in the graph.

Now, let's jump into the algorithm:

Algorithm 1: Pseudocode of Floyd-Warshall Algorithm
Data: A directed weighted graph G(V, E)
Result: Shortest path between each pair of vertices in G

```
for each d∈V do
 | distance[d][d]←0;
end
for each edge (s, p)∈ E do
 | distance[s][p]←weight(s, p);
end
n=cardinality(V);
for k=1 to n do
 |  for i=1 to n do
 |   | for j=1 to n do
 |   |  | if distance[i][j]>distance[i][k]+distance[k][j] then
 |   |  |  | distance[i][j]←distance[i][k]+distance[k][j];
 |   |  |  end
 |   |  end
 |  end
end
```

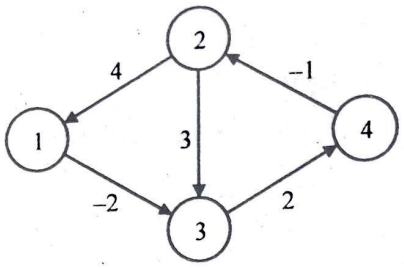
We're taking a directed weighted graph *G(V, E)* as an input. **And first, we construct a graph matrix from the given graph.** This matrix includes the edge weights in the graph. **Next, we insert 0 in the diagonal positions in the matrix.** The rest of the positions are filled with the respective weight from the input graph.

Then, we need to find the distance between two vertices. While finding the distance, we also check if there's any intermediate vertex between two picked vertices. If there exists an intermediate vertex then we check the distance between the selected pair of vertices which goes through this intermediate vertex.

If this distance when traversing through the intermediate vertex is less then the distance between two picked vertices without going through the intermediate vertex, we update the shortest distance value in the matrix.

The number of iterations is equal to the cardinality of the vertex set. The algorithm returns the shortest distance from each vertex to another in the given graph.

Algorithm 1: Pseudocode of Floyd-Warshall Algorithm
Data: A directed weighted graph G(V, E)
Result: Shortest path between each pair of vertices in G
Let's run the Floyd-Warshall algorithm on a weighted directed graph:



At first, we construct a graph matrix from the input graph. Next, we insert 0 to the diagonal positions in the matrix, and the rest of the positions will be filled with the edge weights from the input graph:

$$\begin{bmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 3 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{bmatrix}$$

Now, we're ready to start the iteration. The cardinality of the vertex set is 4. We'll iterate the loops 4 times. Let's start with the first loop. For the first loop $k=1, i=1, j=1$ we'll check if we should update the matrix:

distance[i][j] > distance[i][k] + distance[k][j] $\Rightarrow$ distance[1][1] > distance [1][1] + distance[1][1] $\Rightarrow 0 > 0 + 0 \Rightarrow$ FALSE

As the loop values don't satisfy the condition, there will be no update in the matrix. Let's continue, now for the values $k=1, i=1, j=2$ and check again:

distance[i][j] > distance[i][k] + distance[k][j] $\Rightarrow$ distance[1][2] > distance [1][1] + distance[1][2] $\Rightarrow 0 > 0 + \infty \Rightarrow$ FALSE

Thus, there will be no changes in the matrix. In this way, we'll continue and check pair of vertices.

Let's fast-forward to some values that will satisfy the distance condition.

For the loop values $k = 1, i = 2, j = 3$ we'll see that the condition is satisfied:

distance[i][j] > distance[i][k] + distance[k][j] $\Rightarrow$ distance[2][3] > distance [2][1] + distance[1][3] $\Rightarrow$ 3 > 4 + -2 $\Rightarrow$ 3 >2 $\Rightarrow$ TRUE

Because of that, we'll compute a new distance:

distance[i][j] > distance[i][k] + distance[k][j] $\Rightarrow$ distance[2][3] = distance [2][1] + distance[1][3] = 2

Hence, the condition satisfies for the vertex pair (2, 3). At first, the distance between the vertex 2 to 3 was 3. However, we found a new shortest distance 2 here. Because of that, we update the matrix with this new shortest path distance:

$$\begin{bmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 2 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{bmatrix}$$

Let's take another set of values for the three for the three nested loops such that the loop values satisfy the distance condition given in the algorithm: $k = 2, i = 4, j = 1$:

distance[i][j] > distance[i][k] + distance[k][j] $\Rightarrow$ distance[4][1] > distance [4][2] + distance[2][1] $\Rightarrow \infty >$ -1+4 $\Rightarrow \infty > 3 \Rightarrow$ TRUE

As the condition satisfies, we'll calculate a new distance calculation:

distance[i][j] > distance[i][k] + distance[k][j] $\Rightarrow$ distance[4][1] = distance [4][2] + distance[2][1] = 3

Therefore, we update the matrix now with this new value:

$$\begin{bmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 2 & \infty \\ \infty & \infty & 0 & 2 \\ 3 & -1 & \infty & 0 \end{bmatrix}$$

Similarly, we continues and checks for different loop values. **Finally, after the algorithm terminates, we'll get the output matrix containing all pair shortest distances:**

$$\begin{bmatrix} 0 & -1 & -2 & 0 \\ 4 & 0 & 2 & 4 \\ 5 & 1 & 0 & 2 \\ 3 & -1 & 1 & 0 \end{bmatrix}$$

Time Complexity Analysis

First, we inserted the edge weight into the matrix. We do this using a for loop that visits all the vertices of the graph. This can be performed in *O(n)* time.

Next, we've got three nested loops, each of which goes from one to the total number of vertices in the graph. **Hence, the total time complexity of this algorithm is $O(n^3)$.**

➲ 12. Write a short note on Topological sorting. [Model Question]

Answer:

The topological sort of a DAG is a linear ordering of the vertices such that if there exists a path from vertex x to y, then x appears before y in the topological sort.

Formally, for a directed acyclic graph $G=(V,\ddot{E})$, where $V=\{V_1,V_2,V_3,....,V_n\}$, if there exists a path from any V_i and V_j, then V_i appears before V_j in the topological sort. An acyclic directed graph can have more than one topological sorts. For example, two different topological sorts for the graph illustrated in Fig, 1 are (1, 4, 2, 3) and (1, 2, 4, 3)

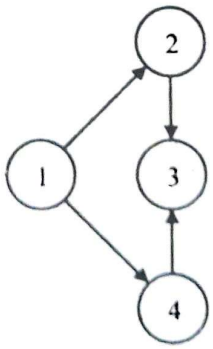


Fig: 1 Directed Acyclic Graph

Clearly, if a directed graph contains a cycle, the topological ordering of vertices is not possible. It is because for any two vertices V_i and V_j in the cycle, V_i precedes V_j as well as V_j precedes V_i. To exemplify this, consider a simple cyclic directed graph shown in Fig. 2. The topological sort for this graph is (1, 2, 3,4) (assuming the vertex 1 as starting vertex). Now, since there exists a path from the vertex 4 to 1, then according to the definition of topological sort, the vertex 4 must appear before the vertex 1, which

contradicts the topological sort generated for this graph. Hence, topological sort can exists only for the acyclic graph.

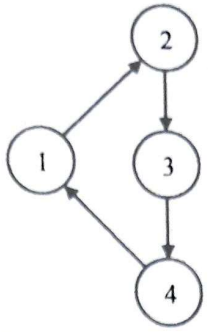


Fig: 2 Cyclic Directed Graph

In an algorithm to find the topological sort of an acyclic directed graph, the indegree of the vertices is considered. The following are the steps that are repeated until the graph is empty.

1. Select any vertex V_i with 0 indegree.
2. Add vertex V_i to the topological sort (initially the topological sort is empty).
3. Remove the vertex V_i along with its edges from the graph and decrease the indegree of each adjacent vertex of V_i by one.

To illustrate this algorithm, consider an acyclic directed graph shown in Fig 3.

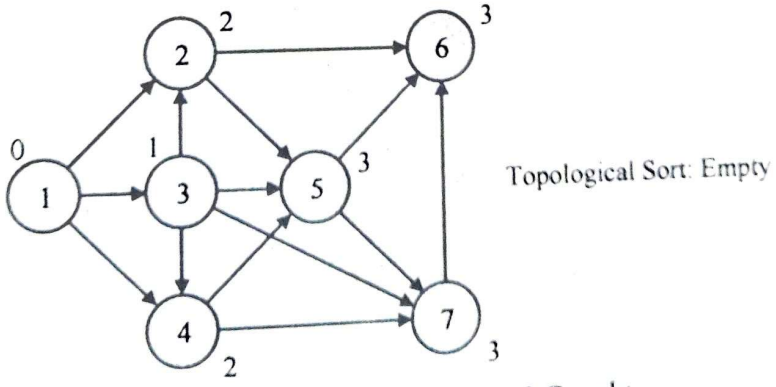
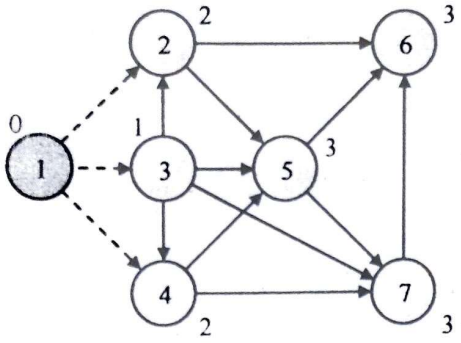
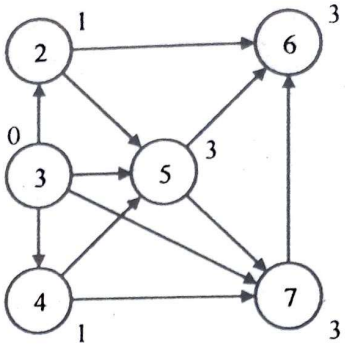


Fig: 3 Acyclic Directed Graph

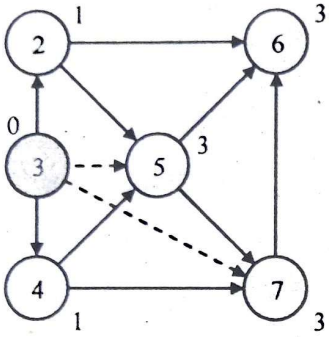
The steps for finding topological sort for this graph are shown in Fig. 4.



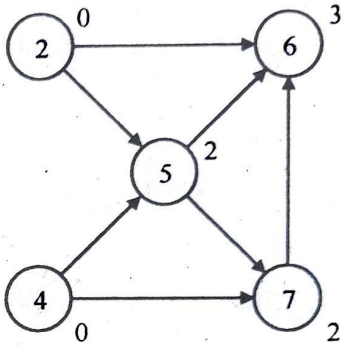
Topological Sort: 1



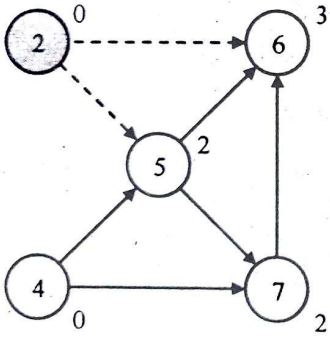
(a) Removing vertex 1 with 0 indegree



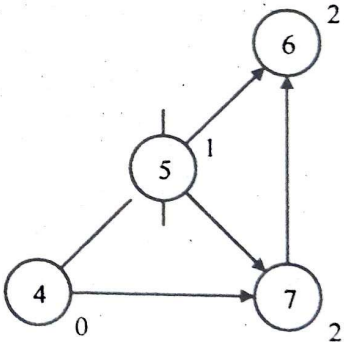
Topological Sort: 1, 3



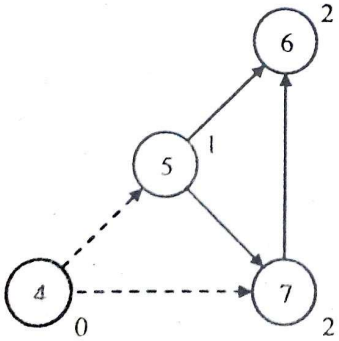
(b) Removing vertex 3 with 0 indegree



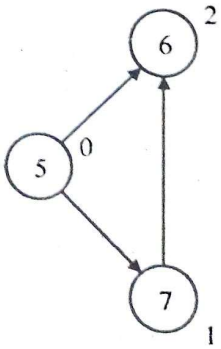
Topological Sort: 1, 3, 2



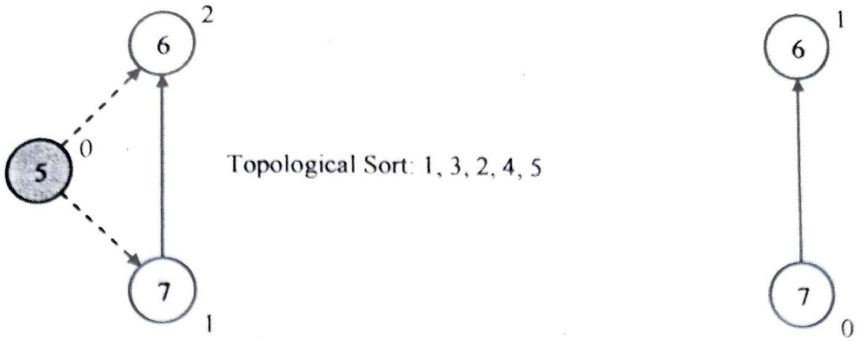
(c) Removing vertex 2 with 0 indegree



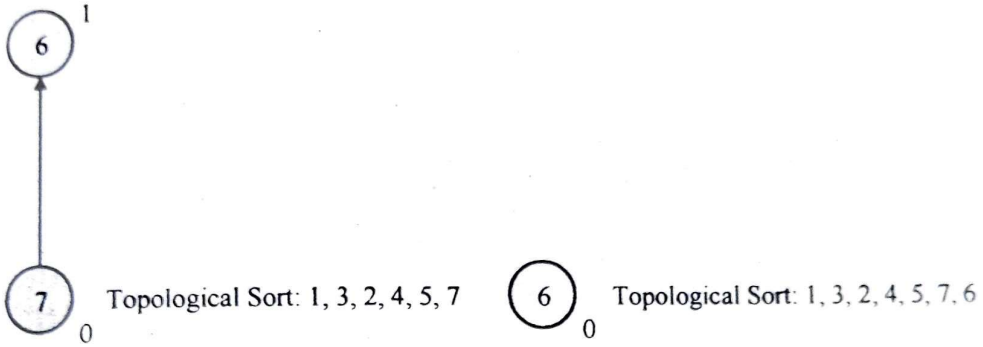
Topological Sort: 1, 3, 2, 4



(d) Removing vertex 4 with 0 indegree



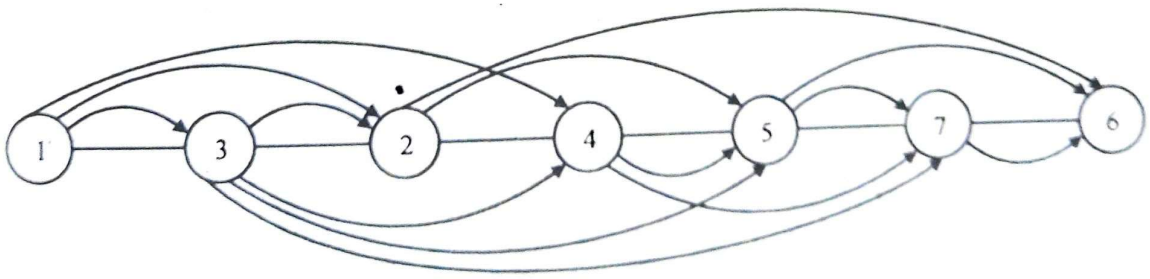
(e) Removing vertex 5 with 0 indegree



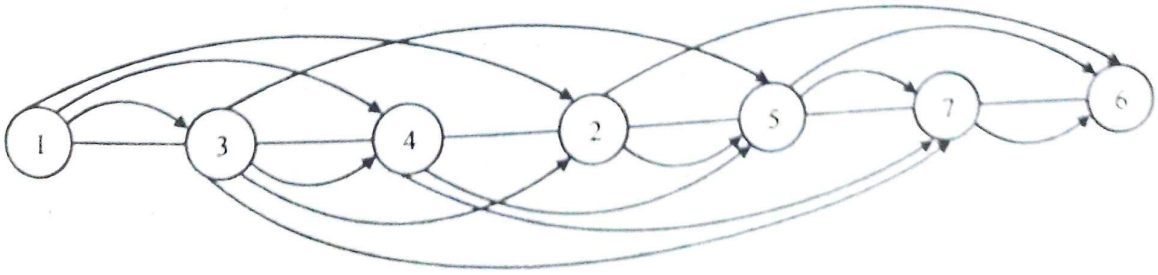
(f) Removing vertex 7 with 0 indegree

Fig: 4 Steps for Finding Topological Sort

Another possible topological sort for this graph is (1, 3, 4, 2, 5, 7, 6). Hence, it can be concluded that the topological sort for an acyclic graph is not unique. Topological ordering can be represented graphically. In this representation, edges are also included to justify the ordering of vertices (Fig. 5).



(a)



(b)

Fig: 5 Graphical Representation of Topological Sort

Topological sort is useful for the proper scheduling of various sub tasks to be executed for completing a particular task. In the computer field, it is used for scheduling the instructions. For example, consider a task in which smaller number is to be subtracted from the larger one. The set of instructions for this task is as follows:

```
1. If A>B then goto step 2, else goto step 3
2. C = A-B, goto step 4
3. C = B-A, goto step 4
4. Print C
5. End
```

The two possible scheduling orders to accomplish this task are (1, 2, 4, 5) and (1, 3, 4, 5). From this, it can be concluded that the instruction 2 cannot be executed unless the instruction 1 is executed before it. Moreover, these instructions are non-repetitive; hence, acyclic in nature.

Unit 5: Strings

Unit at a glance:

5.1 Strings

We communicate by exchanging strings of characters. We consider classic algorithms for addressing the underlying computational challenges surrounding applications such as the following:

- **String Sorts** includes LSD radix sort, MSD radix sort, and 3-way radix quicksort for sorting arrays of strings.
- **Tries** describes R-way tries and ternary search tries for implementing symbol tables with string keys.
- **Substring Search** describes algorithms for searching for a substring in a large piece of text, including the classic Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp algorithms.
- **Regular Expressions** introduces a quintessential search tool known as grep that we use to search for incompletely specified substrings.
- **Data Compression** introduces data compression, where we try to reduce the size of a string to the minimum possible. We present the classic Huffman and LZW algorithms.

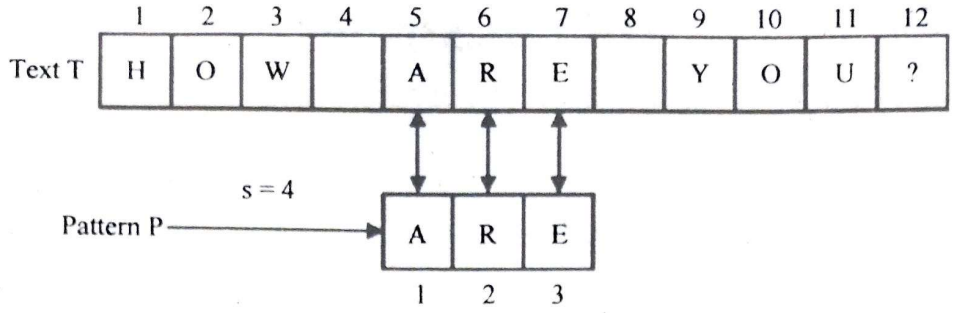
5.2 String Matching Algorithm

String matching operation is a core part in many text processing applications. The objective of this algorithm is to find pattern P from given text T. Typically $|P| << |T|$. In the design of compilers and text editors, string matching operation is crucial. So locating P in T efficiently is very important.

The problem is defined as follows: "Given some text string T[1....n] of size n, find all occurrences of pattern P[1...m] of size m in T."

We say that P occurs in text T with number of shifts s, if $0 \le s \le n-m$ and $T[(s+1)...(s+m)]=P[1...m]$.

Consider the following example



- In this example, pattern P = ARE is found in text T after four shifts.
- The classical application of such algorithms are to find particular protein pattern in DNA sequence.
- Strings may be encoded using set of character alphabets {a, b, ..., z}, binary alphabets {0, 1}, decimal alphabets {0, 1, 2, ..., 9}, DNA alphabets {A, C, G, T}. The encoding of the string directly affects the efficiency of searching.

Short Answer Type Questions

A. Choose the correct answer from the given alternatives in each of the following:

1. What is a Rabin and Karp Algorithm? **[WBSCTE 2022]**

(a) String Matching Algorithm
(b) Shortest Path Algorithm
(c) Minimum spanning tree Algorithm
(d) Approximation Algorithm

Answer: (a)

2. Which of the following is the fastest algorithm in string matching field?

[Model Question]

(a) Boyer-Moore's algorithm
(b) String matching algorithm
(c) Quick search algorithm
(d) Linear search algorithm

Answer: (c)

3. Which of the following algorithms formed the basis for Quick search algorithm?

[Model Question]

(a) Boyer-Moore's algorithm
(b) Parallel string matching algorithm
(c) Binary search algorithm
(d) Linear search algorithm

Answer: (a)

4. What is the pre-processing time of Rabin and Karp Algorithm? [Model Question]

(a) Theta(m^2) (b) Theta$(m \log n)$ (c) Theta(m) (d) Big-Oh(n)

Answer: (c)

B. Answer the following questions:

5. What is data compression? [Model Question]

Answer:

Compression is performed by a program that uses a formula or algorithm to determine how to shrink the size of the data. For instance, an algorithm may represent a string of bits -- or 0s and 1s -- with a smaller string of 0s and 1s by using a dictionary for the conversion between them.

6. What is trie? [Model Question]

Answer:

In computer science, a trie, also called digital tree or prefix tree, is a type of k-ary search tree, a tree data structure used for locating specific keys from within a set. These keys are most often strings, with links between nodes defined not by the entire key, but by individual characters.

Long Answer Type Questions

➲ 1. Explain Naïve string matching algorithm. [Model Question]

Answer:

The naïve approach tests all the possible placement of Pattern P[1.....m] relative to text T[1.....n]. We try shift s=0, 1.....n–m, successively and for each shift s. Compare T [s+1.....s+m] to P [1.....m].

The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.....m] = T [s+1.....s+m] for each of the n–m+1 possible value of s.

NAIVE-STRING-MATCHER (T, P)

1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n–m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s

Analysis: This for loop from 3 to 5 executes for n-m + 1(we need at least m characters at the end) times and in iteration we are doing m comparisons. So the total complexity is O(n–m+1).

➲ **2. What is Rabin-Karp algorithm? Explain it with an example.** **[Model Question]**

Answer:

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

RABIN-KARP-MATCHER (T, P, d, q)

1. n ← length [T]
2. m ← length [P]
3. h ← d^{m-1} mod q
4. p ← 0
5. t_0 ← 0
6. for i ← 1 to m
7. do p ← (dp+P[i]) mod q
8. t_0 ← (dt_0+T [i]) mod q
9. for s ← 0 to n–m
10. do if p = t_s
11. then if P [1.....m] = T [s+1.....s+m]
12. then "Pattern occurs with shift" s
13. If s<n–m
14. then t_{s+1} ← (d (t_s–T[s+1]h)+T [s+m+1])mod q

Example: For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounters in Text T=31415926535.....

1. T=31415926535.....
2. P=26
3. Here T.Length=11 so Q=11
4. And P mod Q=26 mod 11=4
5. Now find the exact match of P mod Q...

Solution:

T = 

P = 

S = 0 →

31 mod 11 = 9 not equal to 4

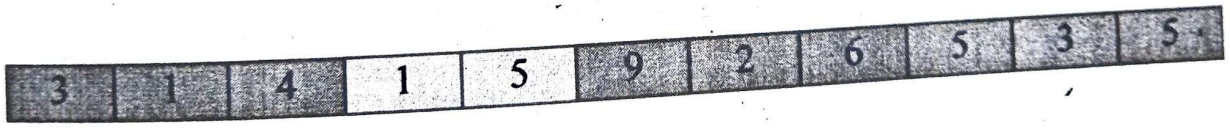
S = 1 →

14 mod 11 = 3 not equal to 4

S = 2 →

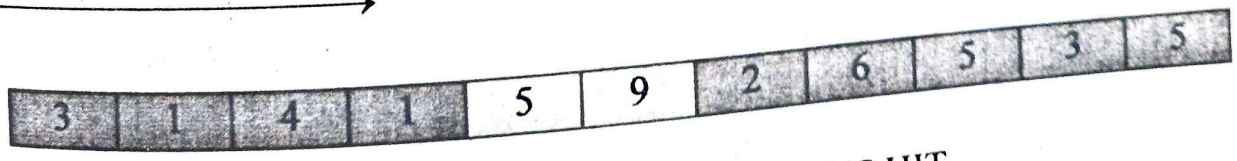
41 mod 11 = 8 not equal to 4

S = 3 →



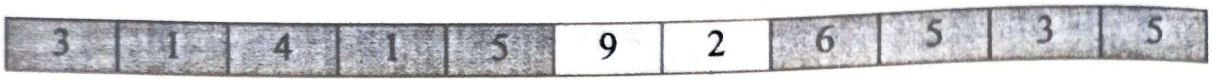
15 mod 11 = 4 equal to 4 SPURIOUS HIT

S = 4 →



59 mod 11 = 4 equal to 4 SPURIOUS HIT

S = 5 →



92 mod 11 = 4 equal to 4 SPURIOUS HIT

S = 6 →

26 mod 11 = 4 EXACT MATCH

S = 7 →

S = 7 →

65 mod 11 = 10 not equal to 4

S = 8 →

53 mod 11 = 9 not equal to 4

S = 9 →

35 mod 11 = 2 not equal to 4

The Pattern occurs with shift 6.

The running time of **RABIN-KARP-MATCHER** in the worst case scenario **O((n–m+1)m** but it has a good average case running time. If the expected number of strong shifts is small **O(1)** and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time **O(n+m)** plus the time to require to process spurious hits.

➲ 3. What are the components of Knuth Morris Pratt algorithm? [Model Question]

Answer:

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

Components of KMP Algorithm:

1) The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2) The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

The Prefix Function (Π)

Following pseudo code compute the prefix function, Π:

COMPUTE- PREFIX- FUNCTION (P)

1. m ←length [P] //'p' pattern to be matched
2. Π [1] ← 0
3. k ← 0
4. for q ← 2 to m
5. do while k > 0 and P [k + 1] ≠ P [q]
6. do k ← Π [k]
7. If P [k + 1] = P [q]
8. then k← k + 1
9. Π [q] ← k
10. Return Π

Running Time Analysis:

In the above pseudo code for calculating the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step1 to Step3 take constant time. Hence the running time of computing prefix function is O (m).

The KMP Matcher:

The KMP Matcher with the pattern 'p,' the string 'S' and prefix function 'Π' as input, finds a match of p in S. Following pseudo code compute the matching component of KMP algorithm:

KMP-MATCHER (T, P)

```
1. n ← length [T]
2. m ← length [P]
3. Π ← COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0                    // numbers of characters matched
5. for i ← 1 to n           // scan S from left to right
6. do while q>0 and P[q+1]≠T[i]
7. do q ← Π[q]              // next character does not match
8. If P[q+1]=T[i]
9. then q ← q+1             // next character matches
10. If q=m                  // is all of p matched?
11. then print "Pattern occurs with shift" i–m
12. q ← Π[q]                // look for the next match
```

Running Time Analysis:

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S.' Since step 1 to step 4 take constant times, the running time is dominated by this for the loop. Thus running time of the matching function is O(n).

➲ 4. Explain Boyer Moore string matching algorithm. [Model Question]

Answer:

It is considered as the most efficient string matching algorithm. A simplified version of it or the entire algorithm is used in text editors for **search** and **substitute** commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two pre-computed functions to shift the window to the right.

The two shifts functions are

- **good suffix shift or matching shift:** aligns only matching pattern characters against target characters already successfully matched.
- **bad character shift or occurrence shift:** avoids repeating unsuccessful comparisons against a target character.

Assume that a mismatch occurs between the character x[i]=a of the pattern and the character y[i+j]=b of the text during an attempt at position j. Then, x[i+1...m–1]=y[i+j+1.. j+m–1]=u and x[i] not equals to y[i+j]. The good-suffix shift consists in aligning the segment y[i+j+1...j+m–1]=x[i+1...m–1] with its rightmost occurrence in x that is preceded by a character different from x[i] as shown in the Fig 1.

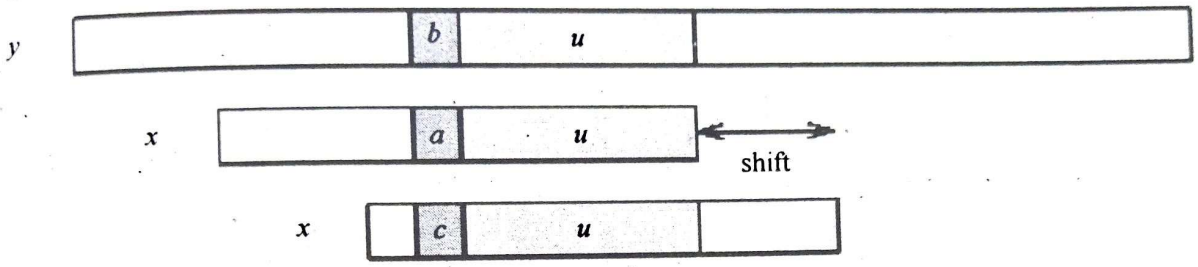


Fig: 1 The good-suffix shift, *u* re-occurs preceded by *a* character *c* different from *a*

If there exists no such segment, the shift consists in aligning the longest suffix v of y[i+j+1...j+m–1] with a matching prefix of x as shown in Fig 2.

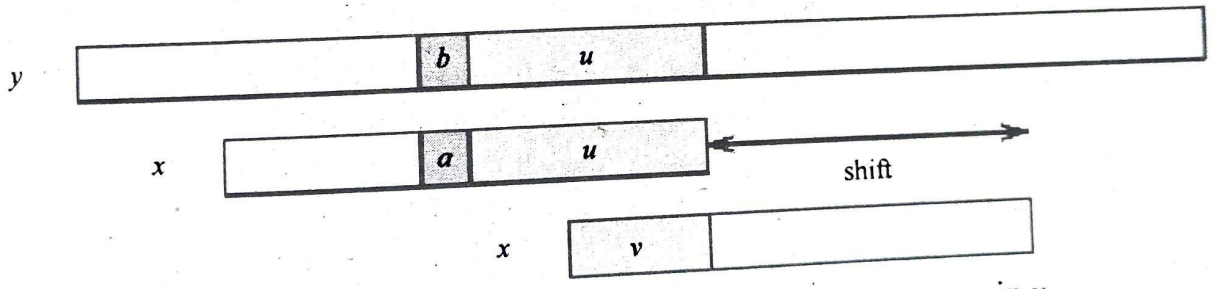


Fig: 2 The good-suffix shift, only *a* suffix of *u* re-occurs in *x*

The bad-character shift consists in aligning the text character y[i+j] with its rightmost occurrence in x[0.. m–2] as shown in Fig. 3.

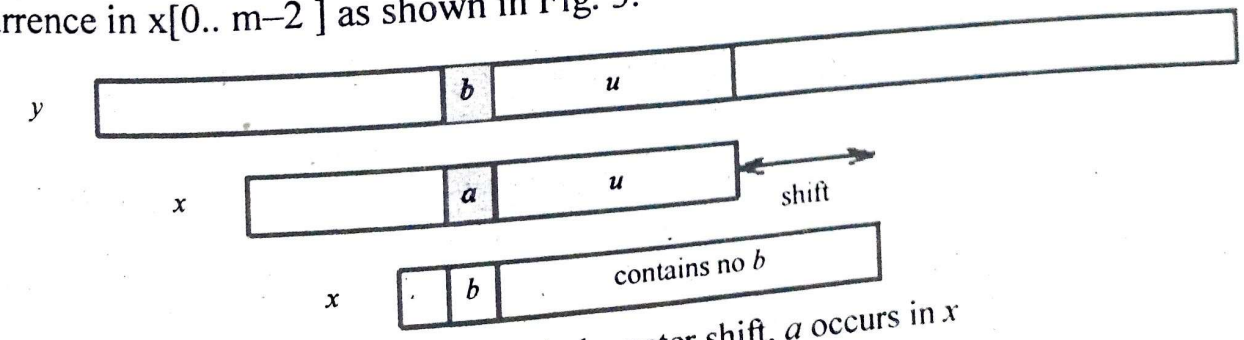


Fig: 3 The bad-character shift, *a* occurs in *x*

If y[i+j] does not occur in the pattern x, no occurrence of x in y can include y[i+j] and the left end of the window is aligned with the character immediately after y[i+j], namely y[i+j+1] as shown in Fig 4.

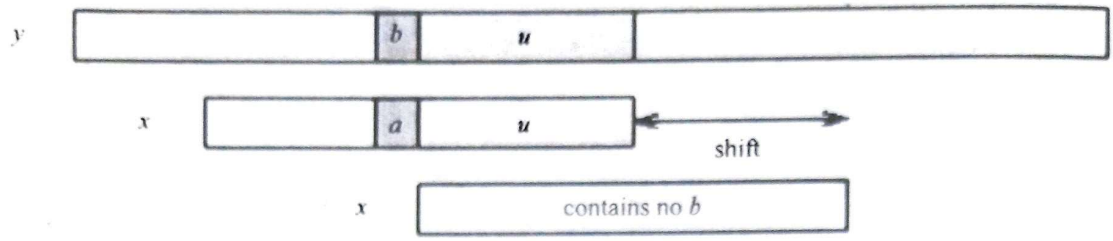


Fig: 4 The bad-character shift, *b* does not occur in *x*

Note that the bad-character shift can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the the good-suffix shift and bad-character shift. More formally the two shift functions are defined as follows.
The good-suffix shift function is stored in a table **bmGs** of size m+1.

Lets define two conditions:

1. Cs (i, s): for each k such that i<k<m, s>=k or x[k–s] = x[k] and
2. Co (i, s): if s<i then x[i–s] not equals to x[i]

Then, for 0<=i<m: **bmGs** [i+1]=min{s>0 : Cs (i, s) and Co(i, s) hold}
And we define bmGs[0] as the length of the period of x. The computation of the table **bmGs** use a table **suff** defined as follows: 1<= i< m, suff[i]=max{k:x[i–k + 1.....i]=x[m–k.....m–1]}
The bad-character shift function is stored in a table **bmBc** of size σ.

For c in Σ: **bmBc** [c]=min{i:1<=i<m–1 and x[m–1–i]=c} if c occurs in x, m otherwise.
Tables **bmBc** and **bmGs** can be precomputed in time O(m+σ) before the searching phase and require an extra-space in O(m+σ). The searching phase time complexity is quadratic but at most 3n text character comparisons are performed when searching for a non periodic pattern. On large alphabets (relatively to the length of the pattern) the algorithm is extremely fast.

➲ 5. Which string-sorting algorithm should I use?

[Model Question]

Answer:

The following table summarizes the important characteristics of the string-sort algorithms.

Algorithm	Stable?	Inplace?	Order of growth of typical number calls to charAt() to sort N strings from an R-character alphabet (average length w, max length W)		Sweet spot
			Running time	**Extra space**	
Insertion sort for strings	Yes	Yes	Between N and N^2	1	Small arrays, arrays in order
Quicksort	No	Yes	$N \log^2 N$	$\log N$	General-purpose when space is tight
Mergesort	Yes	No	$N \log^2 N$	N	General-purpose stable sort
3-way quicksort	No	Yes	Between N and $N \log^2 N$	$\log N$	Large numbers of equal keys
LSD string sort	Yes	No	Nw	N	Short fixed-length strings
MSD string sort	Yes	No	Between N and Nw	$N + WR$	Random strings
3-way string quicksort	No	Yes	Between N and $Nw \log R$	$W + \log N$	General-purpose strings with long prefix matches

➲ 6. What is LSD string sort?

[Model Question]

Answer:

LSD (Least-Significant-Digit) string sort: assume that the N keys to be sorted are strings, all of the same length W, and that the alphabet has size R. (The default we are accustomed to is $R = 256$, strings are extended ascii characters.) Note that we can sort a collection of characters using counting sort.

The general idea for LSD string sort is to apply counting sort on the characters at each position, beginning with the "least significant" (highest index = W-1) character and

working up to the "most significant" (smallest index = 0) character. Because counting sort is stable, the final result will be a sort of all of the keys.

This is exactly what we did implementing byte_sort, except that we were sorting actual numbers, which consisted of at most 8 bytes, so our byte_sort looked like this:

mask = 255 = 0x00000000000000FF;
for i = 0 ... 7
apply counting sort to (keys & mask);
mask = mask << 8;

In other words we just apply counting sort to the right-most byte, then shift over 8 bits, and repeat until we have moved the mask all the way to the left-most byte.

If we think of a string as a "base *R*" number, LSD sort does the same thing to string keys, except that we have much bigger "numbers" that cannot be represented numerically in the machine. Therefore we must maintain the symbolic representation as strings of "digits".

LSD can be adapted with almost no extra effort when the String class used to house strings has a built-in null-terminator, a la C-strings. in fact, all that is needed is an Element(index) method that returns the null character whenever there is no character at that index.

➲ 7. Write a note on Horspool algorithm. [Model Question]

Answer:

Horspool's algorithm shifts the pattern by looking up shift value in the character of the text aligned with the last character of the pattern in table made during the initialization of the algorithm. The pattern is check with the text from right to left and progresses left to right through the text.

Let *c* be the character in the text that aligns with the last character of the pattern. If the pattern does not match there are 4 cases to consider.

The **mismatch occurs at the last character** of the pattern:

Case 1: *c* does not exist in the pattern (Not the mismatch occurred here) then shift pattern right the size of the pattern.

T[0] ... S ... *T*[*n*-1]
|
LEADER

LEADER

Case 2: The mismatch happens at the last character of the pattern and ***c* does exist** in the pattern then the shift should be to the **right most *c*** in the *m*-1 remaining characters of the pattern.

T[0] ... A ... *T*[*n*-1]

|

LEADER

LEADER

The **mismatch happens in the middle** of the pattern:

Case 3: The mismatch happens in the middle (therefore *c* is in pattern) and there are **no other *c* in the pattern** then the shift should be the pattern length.

T[0] ... MER ... *T*[*n*-1]

|

LEADER

LEADER

Case 4: The mismatch happens in the middle of the pattern but **there is other *c* in pattern** then the shift should be the **right most *c*** in the *m*-1 remaining characters of the pattern.

T[0] ... EDER ... *T*[*n*-1]

|

LEADER

LEADER

The table of shift values, *table*(*c*), is a table of the entire alphabet of the text and should give

t(*c*) = *m* if *c* is not in the first *m*-1 characters of the pattern

t(*c*) = distance of the right most *c* in the first *m*-1 characters of the pattern

To make the shift table we initialize the table to *m* and then scan the pattern left to right writing *m*-1-*j* in for the *j* character in the pattern.

Algorithm:

ShiftTable(P[0...m-1])

```
    // output Table the size of the alphabet
    // and gives the amount to shift the table.
    initialize all entries of Table to m
    for j ← 0 to m-2 do Table[P[j]] ← m-1-j //  character
    position from end of pattern
    return Table
```

The line

for *j* ← 0 to *m*-2 **do** *Table*[*P*[*j*]] ← *m*-1-*j*

Finds the right most location in the pattern of the character

What is the size of the extra storage? What is the cost of making the shift table? Not *m*, but rather the size of the alphabet.

Now we can use *Table* in Horspol's algorithm.

Algorithm:

HorspoolMatching(P[0...m–1], T[0...n–1])

Table ← ShiftTable(P[0...m–1])

i ← *m*-1 // index for text aligned with the last character of the pattern

while *i* ≤ *n*–1 **do**

k ← 0 // matching index

while *k* ≤ *m*-1 and *P*[*m*-1-*k*]==*T*[*i*-*k*] **do** *k*++ // check matching

if *k* = *m* **then return** *i*-*m*+1 // Match

else *i* ← *i* + *Table*[*T*[i]] // Shift

return–1