Data Structure Analysis and Algorithm

Unit – 1

Q1. What is algorithm? What does an algorithm do?

- An algorithm is a set of rules for carrying out calculation either by hand or on a machine.
- An algorithm is a sequence of computational steps that transform the input into the output.
- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- A finite set of instruction that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems is called an algorithm.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of computation).

Q2. What are the criteria all algorithm must follow?

An algorithm must have the following properties :

- **Finiteness.** Algorithm must complete after a finite number of instructions have been executed.
- **Absence of ambiguity.** Each step must be clearly defined, having only one interpretation.
- **Definition of sequence.** Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted.
- **Input/output.** Number and types of required inputs and results must be specified.
- **Feasibility.** It must be possible to perform each instruction.

Q3. What are characteristics of an algorithm?

Characteristics of an Algorithm

Every algorithm should have the following *five* characteristic features :

1. Input 2. Output 3. Definiteness 4. Effectiveness 5. Termination

Q4. What are some Algorithm Design Techniques?

For a given problem, there are many ways to design algorithms for it, (*e.g.*, Insertion sort is an incremental approach). The following is a list of several popular design approaches.

- Divide-and-Conquer (*D* and *C*)
- Greedy Approach
- Dynamic Programming
- Branch-and-Bound
- Randomized Algorithms
- Backtracking Algorithms

Q5. What is analysis of algorithm?

The purpose of analysis of algorithms is not to give a formula that will tell us exactly how many seconds or computer cycles a particular algorithm will take. This is not useful information because we would then need to talk about the type of computer, whether it has one or many users at a time, what processor it has, how fast its clock is, whether it has a complex or reduced instruction set processor chip, and how well the compiler optimizes the executable code.

The analysis of an algorithm is to evaluate the performance of the algorithm based on the given models and metrics.

- **Input size**
- **Running time (worst-case and average-case) :** The running time of an algorithm on a particular input is the number of primitive operations or steps executed. Unless otherwise specified, we shall concentrate on finding only the worst case running time.
- **Order of growth.** To simplify the analysis of algorithms, we are interested in the growth rate of the running time, *i.e.*, we only consider the leading terms of a time formula. *e.g.*, the leading term is n^2 in the expression $n^2 + 100n + 50000$.

Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or complexity of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations.

Q6. Name Some Computational Models?

- **RAM.** time and space (traditional serial computers)
- **PRAM.** parallel time, number of processors, and read-and-write restrictions (SIMD type of parallel computers)
- **Message Passing Model.** communication cost (number of message), and computational cost (usually the cost for local computation is ignored) (Distributed computing, peer-to-peer networking, MIMD type of Machine)
- **Turing Machine.** time and space (abstract theoretical machine)

Q7. What is Formal and Informal Algorithm?

Formal Algorithm Analysis involves the use of mathematical methods and rigorous proofs to analyze the time and space complexity of an algorithm. It focuses on worst-case, average-case, and best-case scenarios using Big-O, Big-Ω, and Big-Θ notations. This method is theoretical and abstract, providing precise complexity bounds. Formal analysis is essential for proving the efficiency and correctness of algorithms under various conditions, but it may require a deep understanding of mathematical principles.

Informal Algorithm Analysis is more intuitive and practical. It involves estimating an algorithm's performance based on experience, experimentation, and general observation without detailed mathematical proof. Informal analysis often includes testing the algorithm on different input sizes and scenarios to determine its behavior in real-world settings. It provides a quick understanding of an algorithm’s efficiency, but the results are less precise compared to formal methods.

Q8. What is the difference between an algorithm and a program?

The difference between an algorithm and a program lies in their definitions and roles in problem-solving:

- **Algorithm**: It is a step-by-step, well-defined set of instructions to solve a problem. Algorithms are abstract concepts and independent of any programming language or implementation. They focus on logic and approach, not how the solution is carried out in a machine.
- **Program**: A program is the concrete implementation of an algorithm in a specific programming language that a computer can execute. It includes detailed instructions that a machine can follow, considering real-world constraints like memory, performance, and syntax.

In short, an algorithm is a plan, while a program is its translation into executable code.

Q9. What is RAM Machine?

A RAM (Random Access Machine) is a theoretical model used to design and analyze algorithms. It simulates a real computer by assuming an idealized machine with an infinite memory and a central processing unit (CPU) capable of executing basic instructions in constant time. The key features of the RAM model include:

1. **Instructions**: The machine can perform basic operations like arithmetic, comparison, and data movement between memory and registers.
2. **Memory**: It has an infinite amount of memory, where each memory location can be accessed directly in constant time (hence "random access").
3. **Time Complexity**: Each instruction takes a fixed amount of time, simplifying the analysis of algorithms.

The RAM machine is used in algorithm analysis to focus on logical steps without worrying about hardware limitations, making it useful for theoretical studies.

Q10. What is Order of Growth of Function?

Order of Growth describes how the runtime or space requirements of an algorithm increase as the input size grows. It focuses on the dominant factor, ignoring constants and lower-order terms. This helps classify algorithms by efficiency, such as O(1), O(n), O(n²), etc. Understanding this is crucial in analyzing the performance of algorithms for large datasets, especially in tech-driven countries like India, where optimizing software for scalability is vital.

Order of Growth	Notation	Description	Example
Constant	O(1)	Time remains constant, regardless of input size	Accessing an array element
Logarithmic	O(log n)	Grows slowly as input size increases	Binary search
Linear	O(n)	Grows directly with the input size	Traversing an array
Linearithmic	O(n log n)	Grows faster than linear but slower than quadratic	Merge sort, Quick sort
Quadratic	$O(n^2)$	Grows proportionally to the square of input size	Nested loops (e.g., Bubble sort)
Cubic	$O(n^3)$	Grows proportionally to the cube of input size	Matrix multiplication
Exponential	$O(2^n)$	Grows exponentially with input size	Solving the Tower of Hanoi
Factorial	O(n!)	Grows very rapidly with input size	Permutation generation

Q11. What is Growth Rate of Function?

Resources for an algorithm are usually expressed as a function of input. Often this function is messy and difficult to work. To study function growth easily, we reduce the function down to the important part.

Let $f(n) = an^2 + bn + c$

In this function, the n^2 term dominates the function, that is when n gets sufficiently large, the other terms bare factor into the result.

Dominate terms are what we are interested in to reduce a function, in this we ignore all constants and coefficients and look at the highest order term in relation to n.

Q12. What are Asymptotic notation? Write down its 5 types.

Asymptotic notation is a shorthand way to write down and talk about 'fastest possible' and 'slowest possible' running times for an algorithm, using high and low bounds on speed. These are also referred to as 'best case' and 'worst case' scenarios respectively.

2.5.1 Why are Asymptotic Notations Important ?

1. They give a simple characterization of an algorithm's efficiency.
2. They allow the comparison of the performances of various algorithms.

Q13. Explain Big-Oh Notation with graph and example.

2.5.2 Asymptotic Notations

1. Big-oh notation. Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time it could possibly take for the algorithm to complete. More formally, for non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$.

$$f(n) \leq cg(n)$$

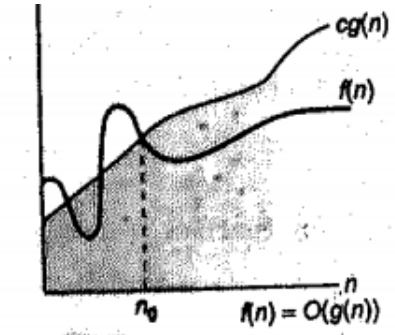


Figure 2.1

Then $f(n)$ is Big-oh of $g(n)$. This is denoted as

$$"f(n) \in O(g(n))"$$

i.e., the set of functions which, as n gets large, grow no faster than a constant times $f(n)$.

Q14. Explain Big-Omega Notation with graph and example.

2. Big-omega notation. For non-negative functions, $f(n)$ and $g(n)$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n > n_0$, $f(n) \geq cg(n)$ then $f(n)$ is big omega of $g(n)$. This is denoted as

$$"f(n) \in \Omega(g(n))"$$

This is almost the same definition as Big-oh, except that "$f(n) \geq (g(n))$", this makes $g(n)$ a lower bound function instead of an upper bound function. It describes the best that can happen for a given data size.

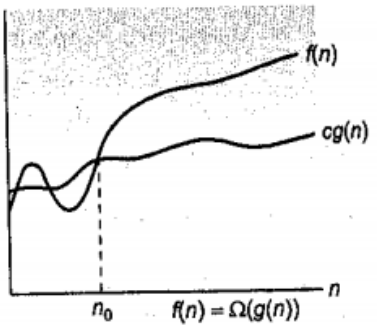


Figure 2.2

Q15. Explain Big-Theta Notation with graph and example.

3) Big theta ('Θ')

This will provide - Avg. case behaviour
It is also known as - tight bound.

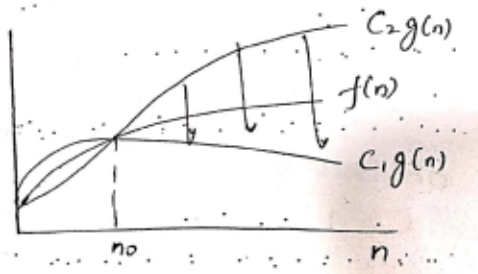
$f(n) = \Theta(g(n))$

$\Rightarrow$ $f(n)$ & $g(n)$ are asymptotically equal.

then there exists 3 +ve constants C_1, C_2 & n_0 such that

$$C_1 g(n) \leq f(n) \leq C_2 g(n) \quad \forall \quad n \geq n_0.$$

$n_0 \geq 1$.



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

$C_1(g(n)) \leq f(n) \leq C_2(g(n))$

Lower bound
'Ω'

upper bound.
'O'

"Θ"

Q16. Explain Little-Oh Notation with graph and example.

4. Little-oh notation (o). Asymptotic upper bound provided by O-notation may not be asymptotically tight. So o-notation is used to denote an upper bound that is asymptotically tight.

$$o(g(n)) = \{ f(n): \text{ for any +ve constant } c>0, \text{ if a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \ \forall \ n \geq n_0 \}$$

The function $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity, *i.e.*,

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

For example, $2n = o(n^2)$

$2n^2 \neq o(n^2)$

Q17. Explain Little-Omega Notation with graph and example.

5. Little-Omega Notation. As o-notation is to O-notation, we have ω-notation as Ω-notation. Little-omega (ω) is used to denote an lower bound that is asymptotically tight,

$$\omega(g(n)) = \{ f(n): \quad \text{for any +ve constant } c > 0, \text{ if a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \}$$

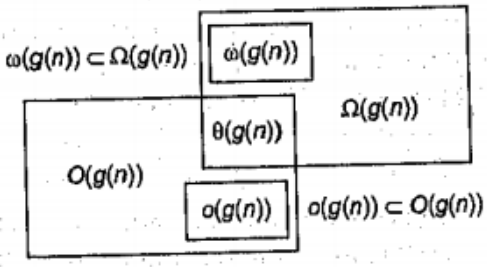
Here $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$.

Q18. Give Relation Between Asymptotic notation.

Thus asymptotic notation gives us a way to express their relationship.

- If $f(n)$ is $O(g(n))$ *i.e.*, $f(n)$ grows no faster than $g(n)$
- If $f(n)$ is $o(g(n))$ *i.e.*, $f(n)$ grows slower than $g(n)$
- If $f(n)$ is $\omega(g(n))$ *i.e.*, $f(n)$ grows faster than $g(n)$.
- If $f(n)$ is $\Omega(g(n))$ *i.e.*, $f(n)$ grows no slower than $g(n)$.
- If $f(n)$ is $\theta(g(n))$ *i.e.*, $f(n)$ and $g(n)$ grow at the same rate.

Relationships between O, o, θ, Ω, ω notations



θ is a subset of Ω and of O.

In other words

$$f(n) = \theta(g(n)) \text{ iff } f(n) = o(g(n)) \text{ and } f(n) = \Omega(g(n))$$

Q19. What is a recurrence relation?

A recurrence relation is an equation that recursively defines a sequence of values, where each term is defined in terms of one or more preceding terms. It is often used in algorithm analysis to describe the time complexity of recursive algorithms.

For example, the recurrence relation for the Fibonacci sequence can be defined as:

- $F(n) = F(n-1) + F(n-2)$ for $n > 1$

Q20. Explain Substitute Method.

It involves guessing the form of the solution and then using mathematical induction to find the constants and show that the solution works. This method is powerful, but it can be applied only in cases when it is easy to guess the form of the answer. The substitution method can be used to establish either upper or lower bounds on a recurrence.

Example. *Consider the recurrence* $T(n)=T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)+1$

we have to show that it is asymptotically bound by $O(\log n)$.

Solution. For $T(n)=O(\log n)$

we have to show that for some constant c.

$$T(n)\le c\log n.$$

Put this in the given recurrence equation.

$$T(n)\le c\log\left(\left\lfloor\frac{n}{2}\right\rfloor\right)+1$$

$$\le c\log\left(\frac{n}{2}\right)+1=c\log n-c\log_2 2+1$$

$$\le c\log n \text{ for } c\ge 1$$

Thus $T(n)=O(\log n)$

Q21. Explain Internation Method.

In iteration method the basic idea is to expend the recurrence and express it as a summation of terms dependent only on 'n' and the initial conditions.

Q22.

Example. *Consider the recurrence :* $T(n)=3T\left(\left\lfloor\frac{n}{4}\right\rfloor\right)+n$

Solution. We iterate it as follows :

$$T(n)=n+3T\left(\left\lfloor\frac{n}{4}\right\rfloor\right)$$

$$= n + 3\left(\left\lfloor \frac{n}{4} \right\rfloor + 3T\left(\left\lfloor \frac{n}{16} \right\rfloor\right)\right)$$

$$= n + 3\left(\left\lfloor \frac{n}{4} \right\rfloor + 3\left(\left\lfloor \frac{n}{16} \right\rfloor + 3T\left(\left\lfloor \frac{n}{64} \right\rfloor\right)\right)\right)$$

$$= n + 3\left\lfloor \frac{n}{4} \right\rfloor + 9\left\lfloor \frac{n}{16} \right\rfloor + 27T\left(\left\lfloor \frac{n}{64} \right\rfloor\right)$$

$$\leq n + \frac{3n}{4} + \frac{9n}{16} + \ldots 3^i T\left(\frac{n}{4^i}\right)$$

The series terminates when $\frac{n}{4^i} = 1 \Rightarrow n = 4^i$ or $i = \log_4 n$.

$$T(n) \leq n + \frac{3n}{4} + \frac{9n}{16} + \frac{27n}{64} + \ldots + 3^{\log_4 n} T(1)$$

$$\leq n + \frac{3n}{4} + \frac{9n}{16} + \frac{27n}{64} + \ldots + 3^{\log_4 n} \theta(1)$$

$$\leq n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + \theta(n^{\log_4 3}) \text{ as } 3^{\log_4 n} = n^{\log_4 3}$$

$$\leq n \cdot \frac{1}{1 - \frac{3}{4}} + o(n) \text{ as } \log_4 3 < 1 \text{ i.e., } \theta(n^{\log_4 3}) = o(n)$$

$$= 4n + o(n) = O(n)$$

Q23.

Example. *Consider the recurrence*

$$T(n) = T(n-1) + 1 \text{ and } T(1) = \theta(1). \text{ Solve it}$$

Solution. $T(n) = T(n-1) + 1$

$$= (T(n-2) + 1) + 1$$

$$= (T(n-3) + 1) + 1 + 1$$

$$= T(n-4) + 4 = T(n-5) + 1 + 4$$

$$= T(n-5) + 5$$

$$= T(n-k) + k$$

where $k = n - 1$

i.e., $T(n-k) = T(1) = \theta(1)$

i.e., $T(n) = \theta(1) + (n-1) = 1 + n - 1 = n = \theta(n)$.

Q24. What is Recursion Tree?

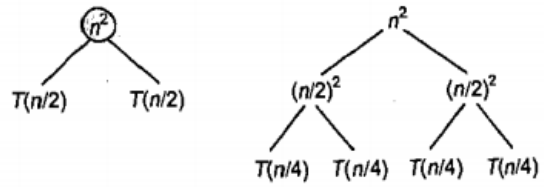
3.3.1 Recursion Tree

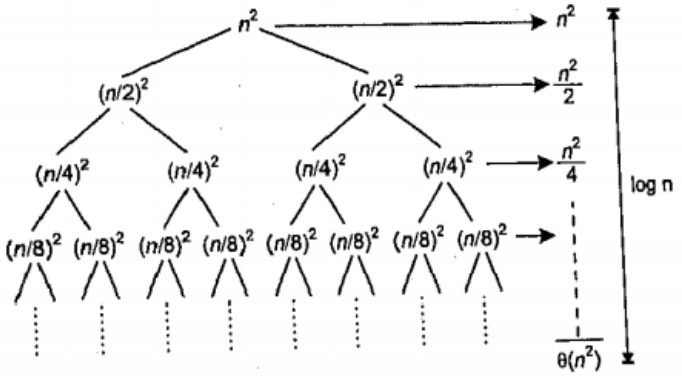
Recursion tree method is a pictorial representation of an iteration method, which is in the form of a tree, where at each level nodes are expanded. In general, we consider second term in recurrence as root. It is useful when divide and conquer algorithm is used.

Example. *Consider* $T(n) = 2T\left(\frac{n}{2}\right) + n^2$.

We have to obtain the asymptotic bound using recursion tree method.

Solution. The recursion tree for the above recurrence is





Hence solution is $\theta(n^2)$, because the values decrease geometrically, the total is at most a constant factor more than largest (first) term, and hence the solution is $\theta(n^2)$.

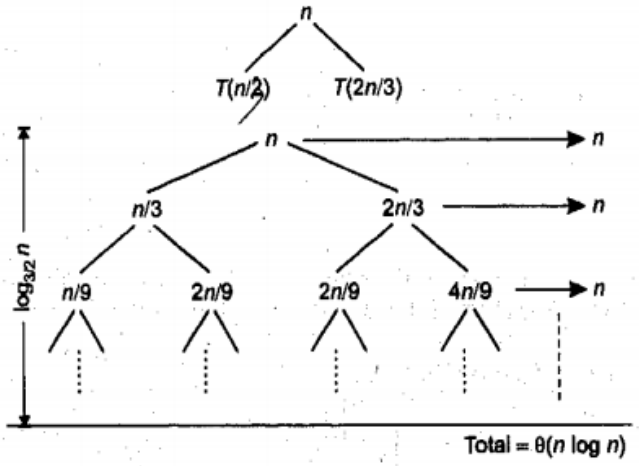
Q25.

Example. *Consider the following recurrence*

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

Obtain the asymptotic bound using recursion tree method.

Solution. The given recurrence has the following recursion tree.



When we add the values across the levels of the recursion tree, we get a value of n for every level. The longest path from the root to a leaf is

$$n \to \frac{2}{3}n \to \left(\frac{2}{3}\right)^2 n \to \ldots 1$$

Since $\left(\frac{2}{3}\right)^i n = 1$ when $i = \log_{3/2} n$.

Thus the height of the tree is $\log_{3/2} n$.

$\therefore \quad T(n) = n + n + n + \ldots + \log_{3/2} n$ times.

$= \theta(n \log n)$

Q26. What is Master Theorem?

Master Theorem

Let $T(n)$ be defined on the non-negative integers by the recurrence.

$T(n)=aT\left(\frac{n}{b}\right)+f(n)$ where $a\geq 1$ and $b>1$ be constants and $f(n)$ be a function and $\frac{n}{b}$ can be interpreted as $\left\lfloor\frac{n}{b}\right\rfloor$ or $\left\lceil\frac{n}{b}\right\rceil$.

Then $T(n)$ can be bound asymptotically as follows :

1. If $f(n)=O(n^{\log_b a-\epsilon})$ for some constant $\epsilon>0$, then $T(n)=\theta(n^{\log_b a})$
2. If $f(n)=\theta(n^{\log_b a})$, then $T(n)=\theta(n^{\log_b a}\lg n)$
3. If $f(n)=\Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon>0$ and if a $af\left(\frac{n}{b}\right)\leq cf(n)$ for some $c<1$ and all sufficiently large n, then $T(n)=\theta(f(n))$, a $af\left(\frac{n}{b}\right)\leq cf(n)$ is called the "regularity" condition.

Example. *Solve the recurrence* $T(n)=3T\left(\frac{n}{4}\right)+n\lg n$ *by master method.*

Solution. Here $a=3,\quad b=4\quad f(n)=n\lg n$.

$$n^{\log_b a}=n^{\log_4 3}=O(n^{0.793})$$

$$f(n)=\Omega(n^{\log_4 3+\epsilon})\quad \text{where } \epsilon=0.2$$

Case 3 applies, now for regularity condition *i.e.*,

$$af\left(\frac{n}{b}\right)\leq cf(n)$$

$$3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right)\leq\left(\frac{3}{4}\right)n\lg n=cf(n)\text{ for } c=\frac{3}{4}.$$

Therefore, the solution is

$$T(n)=\theta(n\lg n)$$

Example. *Solve the recurrence using master method.*

$$T(n)=9T\left(\frac{n}{3}\right)+n$$

Solution. Compare, the given recurrence with the $T(n)=aT\left(\frac{n}{b}\right)+f(n)$

We get $a=9,\quad b=3,\quad f(n)=n$.

Now, $n^{\log_b a}=n^{\log_b a}=n^2=\theta(n^2)$

$$f(n)=n=n^{\log_3 9-1}=n^{2-1}$$

Hence we can apply case 1 and the solution is $\theta(n^{\log_3 9})=\theta(n^2)$.

Q27. What Bubble Sort?

Bubble Sort is a simple and intuitive sorting algorithm used to arrange elements in a list or array in a specific order (usually ascending or descending). It works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. This process is repeated until the list is sorted.

```
Bubble Sort (A)
1.    for i ← 1 to length [A]
2.        for j ← length [A] downto i + 1
3.            if A[j] < A[j − 1]
4.                exchange (A[j], A[j − 1])
```

Q28 What is Selection Sort?

Selection Sort is a straightforward and intuitive comparison-based sorting algorithm. It works by repeatedly selecting the smallest (or largest, depending on the sorting order) element from the unsorted portion of the list and moving it to the beginning (or end) of the sorted portion. This process is continued until the entire list is sorted.

```
Selection Sort (A)
1.    n ← length [A]
2.        for j ← 1 to n − 1
3.            smallest ← j
4.            for i ← j + 1 to n
5.                if A[i] <A [smallest]
6.                    then smallest ← i
7.            exchange (A[j], A [smallest])
```

Q29. What is Insertion Sort?

Insertion Sort is a simple comparison-based sorting algorithm that builds the sorted list one element at a time. It works by taking elements from the unsorted portion of the list and inserting them into their correct position in the sorted portion.

```
Insertion-Sort (A)
1.    for j ← 2 to length [A]
2.        do key ← A[j]
3.            ▷Insert A[j] into the sorted sequence A[1..j − 1]
4.            i ← j − 1
5.            while i > 0 and A[i] > key
6.                do A[i + 1] ← A[i]
7.                    i ← i − 1
8.            A[i + 1] ← key
```

Calculating time Complexity

The running time of an algo. is the sum of running times for each statement executed.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Best case - already sorted array. $\therefore t_j = 1$.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Linear function $\therefore$ $(an+b)$-form

Worst case $(t_j = j)$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$
$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) - (c_2 + c_4 + c_5 + c_8)$$
$$= O(n^2)$$

$(an^2 + bn + c)$-form

Q30. Explain Merge Sort.

This algorithm was invented by John von Neumann in 1945. It closely follows the divide-and-conquer paradigm.

Conceptually, it works as follows :

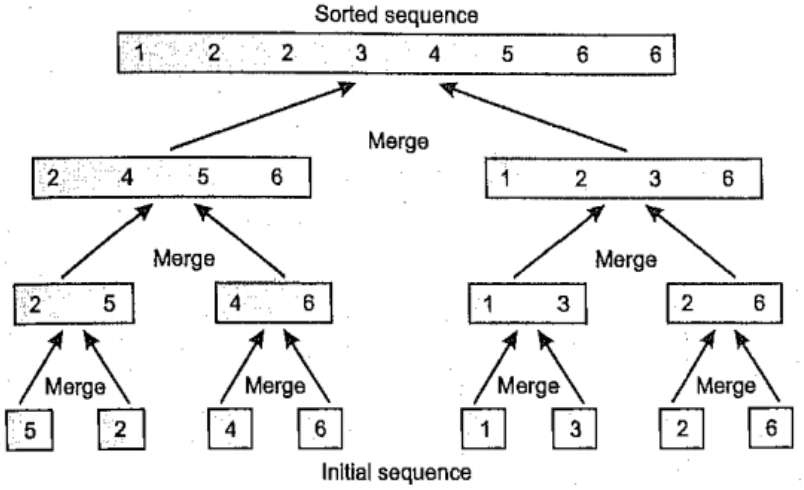
1. **Divide.** Divide the unsorted list into two sub lists of about half the size
2. **Conquer.** Sort each of the two sub lists recursively until we have list sizes of length 1, in which case the list itself is returned
3. **Combine.** Merge the two-sorted sub lists back into one sorted list.

```
MERGE (A, p, q, r)

1.   n1 ← q − p + 1
2.   n2 ← r − q
3.   create arrays L[1.. n1 + 1] and R[1.. n2 + 1]
4.   for i ← 1 to n1
5.       do L[i] ← A[p + i − 1]
6.   for j ← 1 to n2
7.       do R[j] ← A[q + j]
8.   L[n1 + 1] ← ∞
9.   R[n2 + 1] ← ∞
10.  i ← 1
11.  j ← 1
12.  for k ← p to r
13.      do if L[i] ≤ R[j]
14.           then A[k] ← L[i]
15.                i ← i + 1
16.           else A[k] ← R[j]
17.                j ← j + 1
```

```
MERGE-SORT (A, p, r)

1.   if p < r
2.       then q ← ⌊(p + r) / 2⌋
3.            MERGE-SORT (A, p, q)
4.            MERGE-SORT (A, q + 1, r)
5.            MERGE (A, p, q, r)
```



Q31. Format of Analysis of Algorithm.

The analysis of an algorithm is to evaluate the performance of the algorithm based on the given models and metrics.

- **Input size**
- **Running time (worst-case and average-case)** : The running time of an algorithm on a particular input is the number of primitive operations or steps executed. Unless otherwise specified, we shall concentrate on finding only the worst case running time.
- **Order of growth.** To simplify the analysis of algorithms, we are interested in the growth rate of the running time, *i.e.*, we only consider the leading terms of a time formula. *e.g.*, the leading term is n^2 in the expression $n^2 + 100n + 50000$.

Q32. Algorithm as a technology. Comment.

Different algo. used to solve the same problem often differ dramatically in their efficiency.

Let us consider 2 algorithms insertion sort and merge sort for sorting a sequence of elements.

eg. when $n = 1000$, $\log n \simeq 10$

Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\log n$ vs n will more than compensate for diff. in constant factors.

Q33 Do Analysis of Merge Sort.

Analysis.

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

$D(n) \rightarrow$ time to divide the problem into subproblems

$C(n) \rightarrow$ time to combine the solutions to subproblems into the soln to original problem

We add the costs across each level of the tree.

Top level → cn

next level → $c\left(\frac{n}{2}\right) + c\left(\frac{n}{2}\right) = 2\frac{cn}{2} = cn.$

next level → $c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) + c\left(\frac{n}{4}\right) = 4\frac{cn}{4} = cn.$ & so on.

In general

~~level i^{th} level below the top has total cost $2^i\, c\left(\frac{n}{2^i}\right) = cn.$~~

level i^{th} level below the top has ~~total cost~~ 2^i nodes, each contributing a cost of $c\left(\frac{n}{2^i}\right)$

total cost = $2^i \times \frac{cn}{2^i} = cn.$

The bottom level has 'n' nodes, each contributing a cost for a total cost of $\underline{\underline{cn}}$.

Total no. of levels in recursion tree = $\log_2 n + 1$.

$n \rightarrow$ no. of leaves, corresponding to input size.

Best case, occurs when $n=1$, in which case the tree has only 1 level.

Since $\log 1 = 0$, we have that $\log n + 1$ gives correct no. of levels.

Computation

Total level = $\log n + 1$.

each costing = cn

$\therefore$ total cost = $cn(\log n + 1)$

$= cn \log n + cn$

ignoring low-order term & constant c gives the desired result of $= \Theta(n \log n)$

For merge sort (Analysis)

Although the pseudocode for MERGE-SORT works correctly when the no of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of 2.

Each divide step then yields 2 subsequences of size exactly $n/2$.

→ Merge sort on just 1 element → constant time

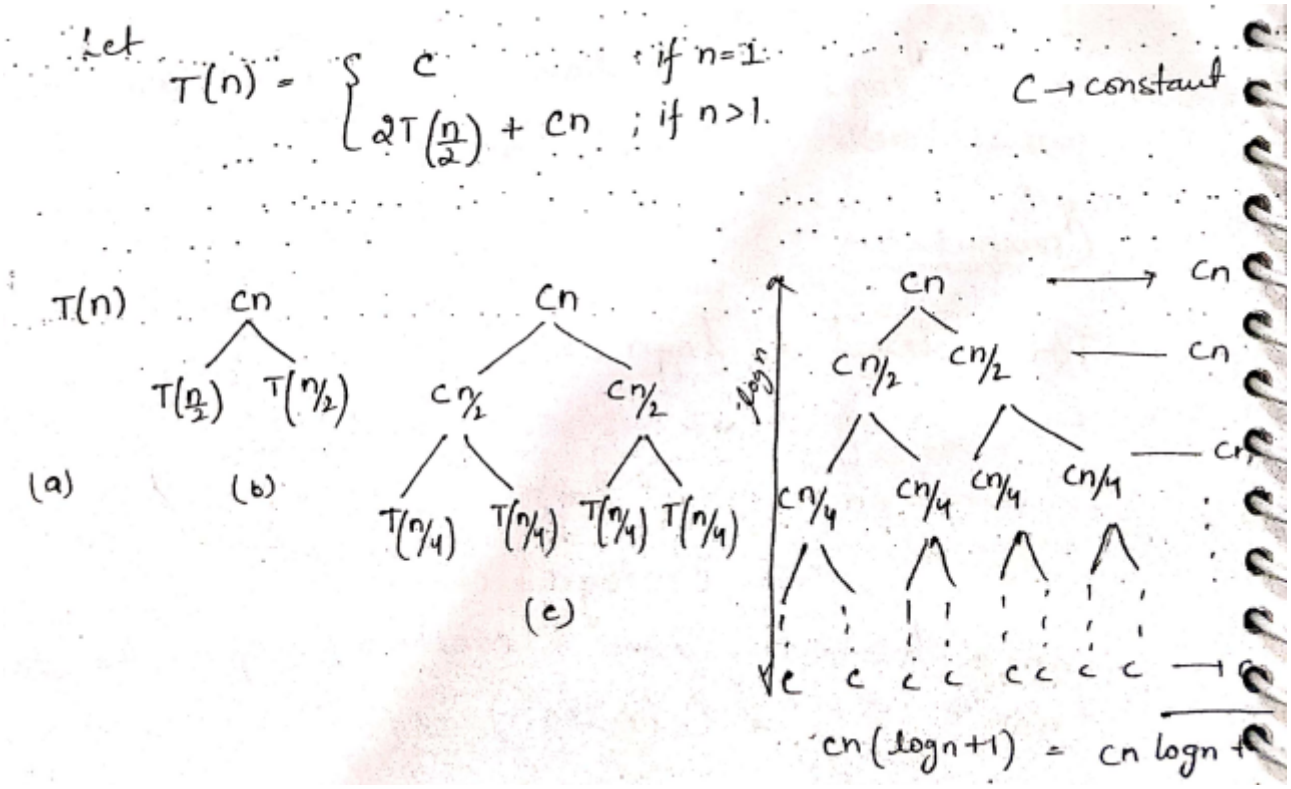
when $n > 1$ elements, we break down the running time as follows :-

Divide → divide step just computes the middle of subarray which takes const. time. $D(n) = \Theta(1)$

Conquer → we recursively solve 2 subproblems, each of size $n/2$, which contributes $2T(n/2)$ time.

Combine → Merge procedure on n-element : $C(n) = \Theta(n)$

$\therefore$ $$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$



or

5.2 Analysis of Merge Sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of two. Each divide step then yields two subsequence of size exactly $n/2$. This assumption does not affect the order of growth of the solution to the recurrence.

Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows.

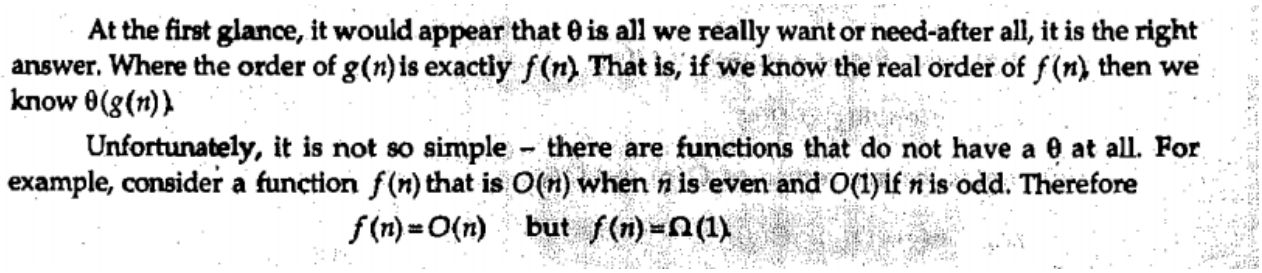
- **Divide.** The divide step just computes the middle of the sub array, which takes constant time. Thus, $D(n) = \theta(1)$.
- **Conquer.** We recursively solve two sub problems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- **Combine.** We have already noted that the MERGE procedure on an n-element sub array takes time $\theta(n)$, so $C(n) = \theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\theta(n)$ and a function that is $\theta(1)$. This sum is a linear function of n, that is, $\theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort :

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

By Master Theorem , we shall show that $T(n)$ is $\theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. For large enough inputs, merge sort, with its $\theta(n \lg n)$ running time, outperforms insertion sort, whose running time is $\theta(n^2)$, in the worst case

Q34. Why not use Theta all the time? Why use big Oh as default?



At the first glance, it would appear that θ is all we really want or need-after all, it is the right answer. Where the order of $g(n)$ is exactly $f(n)$. That is, if we know the real order of $f(n)$, then we know $\theta(g(n))$.

Unfortunately, it is not so simple – there are functions that do not have a θ at all. For example, consider a function $f(n)$ that is $O(n)$ when n is even and $O(1)$ if n is odd. Therefore

$$f(n) = O(n) \quad \text{but} \quad f(n) = \Omega(1).$$

On the other hand, **Big-O (O)** notation is used by default because it only focuses on the upper bound, describing the worst-case scenario, which is more practical in many cases. It ensures that no matter the input, the algorithm will not perform worse than a certain limit. This makes Big-O more flexible and useful for comparing the maximum potential growth of algorithms, which is crucial in understanding performance under the worst possible conditions.

Q35. Explain Graph and all its basics.

A graph is a mathematical and data structure representation of a set of vertices (nodes) connected by edges (lines). Graphs are used to model relationships, networks, and structures in domains like social networks, transportation systems, and computer networks.

Types of Graphs:

1. **Undirected Graph**: Edges have no direction. If there's an edge from vertex A to B, there's also an edge from B to A.
2. **Directed Graph (Digraph)**: Edges have direction. An edge from vertex A to B doesn't imply an edge from B to A.
3. **Weighted Graph**: Assigns a weight or cost to each edge, representing some value associated with the connection between vertices.
4. **Cyclic Graph**: Contains at least one cycle (a closed path that starts and ends at the same vertex).
5. **Acyclic Graph**: Has no cycles. A special case is a DAG (Directed Acyclic Graph), used for topological sorting and dynamic programming.

Graph Terminology:

- **Vertex (Plural: Vertices)**: The fundamental building blocks of a graph.
- **Edge**: Represents a connection between two vertices.
- **Adjacent**: Two vertices are adjacent if there's an edge connecting them.
- **Path**: A sequence of vertices where each adjacent pair is connected by an edge.
- **Cycle**: A path that starts and ends at the same vertex.
- **Degree**: The degree of a vertex is the number of edges connected to it.
- **Connected Graph**: In an undirected graph, there is a path between any pair of vertices.

Graph Representations:

- **Adjacency Matrix**: A 2D array where matrix[i][j] is 1 if there's an edge between vertex iii and jjj, and 0 otherwise. Suitable for dense graphs.
- **Adjacency List**: A collection of lists, one for each vertex, where each list contains the vertices adjacent to that vertex. Suitable for sparse graphs.

Graph Algorithms:

- **Breadth-First Search (BFS)**: Explores all vertices reachable from a starting vertex in shortest-path order.
- **Depth-First Search (DFS)**: Explores as far as possible along each branch before backtracking.
- **Shortest Path Algorithms**: Such as Dijkstra's and Bellman-Ford, used to find the shortest path between vertices in weighted graphs.
- **Minimum Spanning Tree**: Algorithms like Kruskal's and Prim's are used to find the minimum spanning tree in a weighted graph.

Q36. Explain Binary Search. Analyse Binary Search.

Binary search is an efficient algorithm used to find the position of a target value within a **sorted** array or list. The key advantage of binary search over linear search is its logarithmic time complexity, making it much faster for large datasets.

```
FUNCTION binary_search(arr, x):
    SET low = 0
    SET high = LENGTH(arr) - 1

    WHILE low <= high:
        SET mid = (low + high) // 2

        IF arr[mid] == x THEN:
            RETURN mid  // Target found

        ELSE IF arr[mid] < x THEN:
            SET low = mid + 1  // Search in right half

        ELSE:
            SET high = mid - 1  // Search in left half

    RETURN -1  // Target not found
```

Analysis of Binary Search using the Master Theorem

To apply the **Master Theorem**, we need to express the recurrence relation for binary search.

1. **Recurrence Relation**:
   - In each iteration, the problem size is reduced by half.
   - The time to search in one half is $T(n/2)$, and the work done at each level (comparing middle element) is constant, i.e., $O(1)$.
   - So, the recurrence relation becomes:

$$T(n) = T(n/2) + O(1)$$

2. **Master Theorem Format**: The recurrence is of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

For binary search:

- $a = 1$ (one subproblem)
- $b = 2$ (array is divided into two parts)
- $d = 0$ (since the work done outside the recursive call is constant $O(1)$)

3. **Apply the Master Theorem**:
   - Compare $\log_b a = \log_2 1 = 0$ with $d = 0$.
   - According to the Master Theorem, if $\log_b a = d$, the time complexity is $O(n^d \log n)$.
   - Substituting $d = 0$, we get the time complexity of $O(\log n)$.

Q37. Explain Linear Search. Analyse Binary Search.

Linear search is a straightforward algorithm used to find a target value within an array or list. Unlike binary search, linear search examines each element of the array sequentially until it finds the target or reaches the end of the array.

```
FUNCTION linear_search(arr, x):
    FOR i FROM 0 TO LENGTH(arr) - 1:
        IF arr[i] == x THEN:
            RETURN i  // Target found
    RETURN -1  // Target not found
```

Analysis of Linear Search using the Master Theorem

Recurrence Relation:

- In linear search, for each element, a comparison is made with the target value. If the target is not found, the entire array must be checked, leading to a time complexity that is proportional to the number of elements.
- The recurrence can be expressed as:

$$T(n) = T(n-1) + O(1)$$

Master Theorem Format:

- The recurrence is of the form:

$$T(n) = aT\left(n-b\right) + O(n^d)$$

  Where:

  - $a = 1$ (one subproblem)
  - $b = 1$ (the size of the problem decreases by one)

- $d = 0$ (the work done outside the recursive call is constant $O(1)$)

Applying the Master Theorem:

- The case for linear search does not fit the standard form of the Master Theorem because it doesn't divide the problem into subproblems of the same size. Instead, it just reduces the problem size by 1.
- Instead, we can analyze it directly:
  - The recurrence leads to:

$$T(n) = O(n)$$

  - The linear search iterates through each element in the worst case.

Q38. What is disjoined set? What are some operations of disjoined set? Also mention Applications of disjoined sets.

An important application of the tree is the representation of sets, where "n" distinct elements are needed to be grouped into a number of disjoint sets.

A *disjoint-set data structure* maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. Each set is identified by a *representative*, which is some member of the set. If we have two sets S_x and S_y, $x \neq y$, such that $S_x = \langle 3,4,5,6,7 \rangle$ and $S_y = \langle 1,2 \rangle$ then these sets are called disjoint sets as there is no element which is common in both sets.

20.2 Disjoint-Set Operations

In the dynamic-set implementations, an object represents each element of a set. Letting x denote an object, we wish to support the following operations.

- **MAKE-SET(x)** creates a new set whose only member (and thus representative) is pointed to by x. Since the sets are disjoint, we require that x not already be in a set.

- **UNION(x, y)** unites the dynamic sets that contain x and y, say S_x and S_y into a new set that is the union of these two sets. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is some member of $S_x \cup S_y$, although many implementations of UNION choose the representative of either S_x or S_y, as the new representative. Since we require the sets in the collection to be disjoint, we "destroy" sets S_x and S_y removing them from the collection S.
- **FIND-SET(x)** returns a pointer to the representative of the (unique) set containing x.

20.4 Applications of Disjoint-set Data Structures

One of the many applications of disjoint-set data structures arises in determining the connected components of an undirected graph.

The procedure CONNECTED-COMPONENTS that follows uses the disjoint-set operations to compute the connected components of a graph. Once CONNECTED-COMPONENTS has been run as a preprocessing step, the procedure SAME-COMPONENT answers queries about whether two vertices are in the same connected component. The set of vertices of a graph G is denoted by $V[G]$, and the set of edges is denoted by $E[G]$.

Q39. What is UNION-FIND Algorithm?

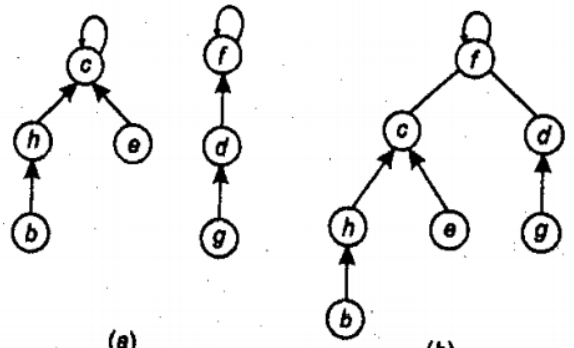
A **union-find algorithm** is an algorithm that performs two useful operations on such a data structure :

- **Find.** Determine which set a particular element is in. Also useful for determining if two elements are in the same set.
- **Union.** Combine or merge two sets into a single set.

Because it supports these two operations, a disjoint-set data structure is sometimes called a ***merge-find set.***

Q40. What is Disjoined Set Forest?

In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set. In a ***disjoint-set forest,*** each member points only to its parent. The root of each tree contains the representative and is its own parent.



We perform the three disjoint-set operations as follows. A MAKE-SET operation simply creates a tree with just one node. We perform a FIND-SET operation by chasing parent pointers until we find the root of the tree. The nodes visited on this path toward the root constitute the ***find path.*** A UNION operation causes the root of one tree to point to the root of the other.

Q41. What is quick sort?

http://www.btechsmartclass.com/data_structures/quick-sort.html

Quick sort works by partitioning a given array $A[p..r]$ into two non-empty sub arrays $A[p..q]$ and $A[q+1..r]$ such that every key in $A[p..q]$ is less than or equal to every key in $A[q+1..r]$. Then the two sub arrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

```
QUICK-SORT (A, p, r)
1.  If p < r then
2.      q ←PARTITION (A, p, r)
3.      QUICK SORT (A, p, q − 1)
4.      QUICK SORT (A, q + 1, r)
```

```
PARTITION (A, p, r)
1.  x ← A[r]
2.  i ← p − 1
3.  for j ← p to r − 1
4.      do if A[j] ≤ x
5.          then i ← i + 1
6.              exchange A[i] ↔ A[j]
7.  exchange A[i + 1] ↔ A[r]
8.  return i + 1
```

Q42. Analyse Quick Sort.

The running time of quick sort depends on whether partition is balanced or unbalanced, which in turn depends on which elements of an array to be sorted are used for partitioning.

A very good partition splits an array up into two equal sized arrays. A bad partition, on other hand, splits an array up into two arrays of very different sizes. The worst partition puts only one element in one array and all other elements in the other array. If the partitioning is balanced, the Quick sort runs asymptotically as fast as merge sort. On the other hand, if partitioning is unbalanced, the Quick sort runs asymptotically as slow as insertion sort.

Best Case

The best thing that could happen in Quick sort would be that each partitioning stage divides the array exactly in half. In other words, the best to be a median of the keys in $A[p..r]$ every time procedure 'Partition' is called. The procedure 'Partition' always splits the array to be sorted into two equal sized arrays. If the procedure 'Partition' produces two regions of size $n/2$, the recurrence relation is then

$$T(n) = T(n/2) + T(n/2) + \theta(n)$$
$$= 2T(n/2) + \theta(n)$$

And from case 2 of Master theorem

$$T(n) = \theta(n \lg n)$$

Worst case Partitioning

The worst-case occurs if given array $A[1..n]$ is already sorted. The PARTITION (A, p, r) call always returns p so successive calls to partition will split arrays of length n, $n-1$, $n-2$,...,2 and running time proportional to $n+(n-1)+(n-2)+...+2 = [(n+2)(n-1)/2] = \theta(n^2)$. The worst-case also occurs if $A[1..n]$ starts out in reverse order.

Q43. Analyse Selection Sort.

Analysis of Selection Sort using the Master Theorem

Recurrence Relation:

- Selection sort operates with two nested loops:
  - The outer loop runs n times.
  - The inner loop runs $n - i - 1$ times for each iteration of the outer loop, leading to the following calculation:

$$T(n) = T(n-1) + O(n)$$

Total Comparisons:

- For the first pass, there are $n - 1$ comparisons.
- For the second pass, there are $n - 2$ comparisons, and so on, until there is 1 comparison for the last element.
- This leads to a total of:

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

Master Theorem Format:

- The recurrence is of the form:

$$T(n) = aT\left(n - b\right) + O(n^d)$$

Where:

- $a = 1$ (one subproblem)
- $b = 1$ (the size of the problem decreases by one)
- $d = 2$ (the work done outside the recursive call is linear)

Applying the Master Theorem:

- The recurrence does not fit the standard form of the Master Theorem due to the nature of selection sort.
- Instead, we analyze the total number of comparisons directly.
- Since the dominant term is $O(n^2)$, we conclude that:

Q44. What is Strassen's Matrix?

Strassen's Matrix Multiplication Algorithm is an efficient algorithm for multiplying two matrices. It was developed by Volker Strassen in 1969 and reduces the time complexity of matrix multiplication compared to the standard algorithm. The time complexity taken by this approach is **$O(n^3)$**, since it takes two loops to multiply. Strassen's method was introduced to reduce the time complexity from **$O(n^3)$** to **$O(n^{\log 7})$**.

Q45. Analysis of Strassen's Matrix?

Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

Recurrence Relation

The recursive structure of Strassen's algorithm leads to a recurrence relation for its time complexity. The algorithm breaks the original $n \times n$ matrix multiplication into 7 multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and some additional additions and subtractions.

- The number of multiplications is 7, and each multiplication involves matrices of size $\frac{n}{2}$.
- The additional operations (additions and subtractions) can be done in $O(n^2)$ time since they involve matrix addition or subtraction of size n.

Therefore, the recurrence relation can be expressed as:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

Solving the Recurrence

Using the **Master Theorem** to analyze the recurrence relation:

- Here, $a = 7$, $b = 2$, and $f(n) = O(n^2)$.
- We need to compare $f(n)$ with $n^{\log_b a}$:
  - $\log_b a = \log_2 7 \approx 2.81$
- Since $O(n^2)$ is polynomially smaller than $n^{\log_b a}$, we apply case 1 of the Master Theorem:

$$T(n) = \Theta\left(n^{\log_2 7}\right) \approx \Theta(n^{2.81})$$

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Data Structure Analysis and Algorithm

Unit – 2

Q1. Explain Greedy Algorithm.

A **Greedy Algorithm** is an approach used in optimization problems that builds a solution piece by piece, always choosing the next piece that offers the most immediate benefit or is the most optimal at that moment. The fundamental idea behind greedy algorithms is to make a series of choices, each of which looks best at the moment, with the hope that these local optimum choices will lead to a global optimum solution.

Example Problems Solved by Greedy Algorithms:

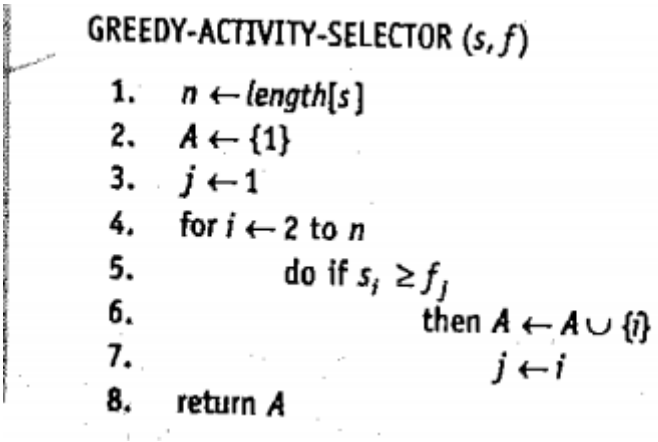
1. **Activity Selection Problem**: Selecting the maximum number of activities that do not overlap in time.
2. **Fractional Knapsack Problem**: Maximizing the total value of items in a knapsack, allowing fractional amounts of items.
3. **Prim's and Kruskal's Algorithms**: Finding the Minimum Spanning Tree (MST) of a graph.
4. **Huffman Coding**: Creating an optimal prefix code for data compression.

Q2. What is Activity Selector Problem?

The **Activity Selection Problem** is a classic optimization problem that can be efficiently solved using a greedy algorithm. The objective of the problem is to select the maximum number of activities that do not overlap in time, given a set of activities with their start and finish times.

Problem Statement

Given a set of activities, each with a start time sis_isi and a finish time fif_ifi, the task is to select the maximum number of activities that can be performed by a single person, assuming that a person can only work on one activity at a time.



Time Complexity

The time complexity of the activity selection problem using the greedy approach is $O(n \log n)$ due to the sorting step, followed by a linear scan $O(n)$, leading to an overall complexity of $O(n \log n)$.

Q3 Calculate Time Complexity of Activity Selection Problem.

To calculate the time complexity of the Activity Selection Problem using the greedy approach, we can break down the steps involved in the algorithm:

1. **Sorting the Activities**:
   - The first step involves sorting the activities based on their finish times. If we have n activities, the time complexity for sorting is $O(n \log n)$. This is typically done using efficient sorting algorithms like Merge Sort or Quick Sort.
2. **Selecting Activities**:
   - After sorting, we need to iterate through the list of activities to select the maximum number of non-overlapping activities. This involves a single pass through the sorted list of activities. The time complexity for this step is $O(n)$.

Overall Time Complexity

Combining both steps, we get the overall time complexity of the greedy algorithm for the Activity Selection Problem:

$$O(n \log n) + O(n) = O(n \log n)$$

Thus, the dominant factor in the time complexity is the sorting step, leading to a final time complexity of $O(n \log n)$.

Q4.

Example. *Given 10 activities along with their start and finish time as*

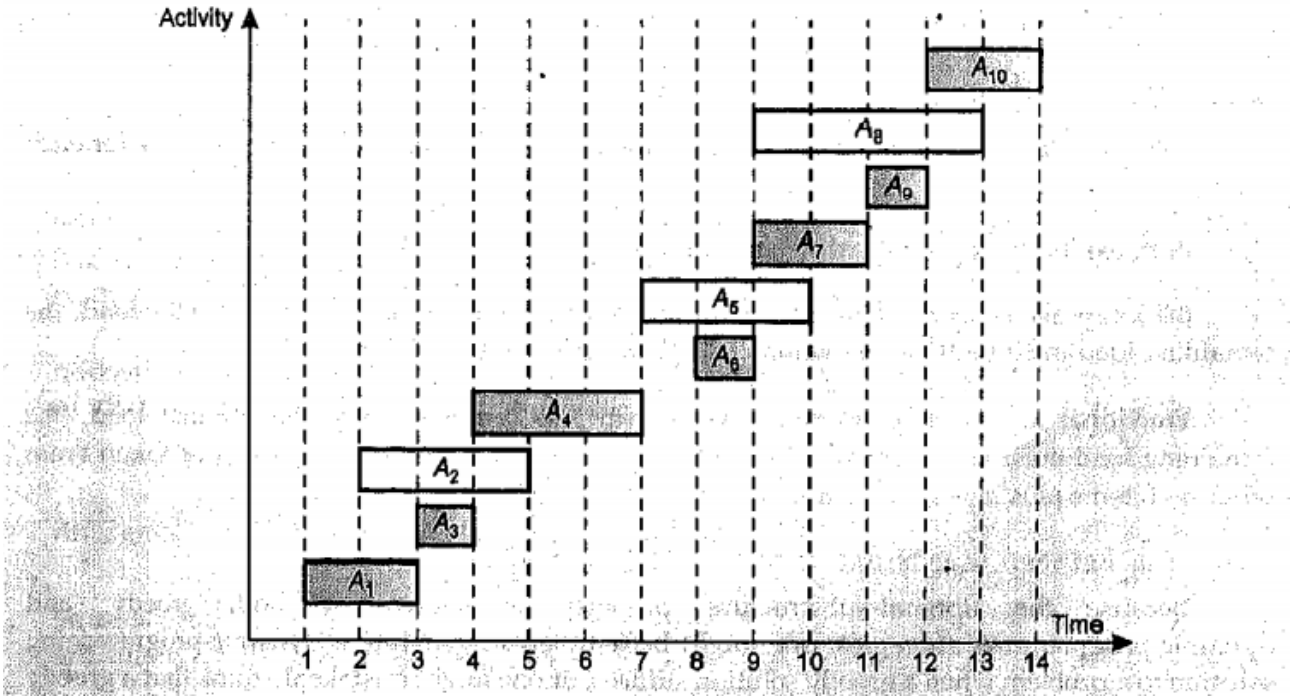
$$S = \langle A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10} \rangle$$
$$S_i = \langle 1,2,3,4,7,8,9,9,11,12 \rangle$$
$$f_i = \langle 3,5,4,7,10,9,11,13,12,14 \rangle$$

Compute a schedule where the largest number of activities takes place.

Solution. The solution for the above activity scheduling problem using greedy strategy is illustrated below.

Arranging the activities in increasing order of finish time.

Activity	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14



Now, schedule A_1

Next, schedule A_3, as A_1 and A_3 are non-interfering

Next, Skip A_2 as it is interfering.

Next, schedule A_4 as A_1, A_3 and A_4 are non-interfering, then next, schedule A_6 as A_1, A_3, A_4 and A_6 are non-interfering.

Skip A_5 as it is interfering.

Next, schedule A_7 as A_1, A_3, A_4, A_6 and A_7 are non-interfering.

Next, schedule A_9 as A_1, A_3, A_4, A_6, A_7 and A_9 are non-interfering.

Skip A_8, as it is interfering.

Next, schedule A_{10} as A_1, A_3, A_4, A_6, A_7, A_9 and A_{10} are non-interfering.

Thus, the final activity schedule is

$$\langle A_1, A_3, A_4, A_6, A_7, A_9, A_{10} \rangle$$

Q5. Knapsack Problems

The **Knapsack Problem** is a classic optimization problem that can be solved using various approaches, including dynamic programming and greedy algorithms. The problem involves selecting items with given weights and values to maximize the total value in a knapsack of limited capacity.

Problem Statement

Given a set of items, each with a weight w_i and a value v_i, the goal is to determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit W and the total value is as large as possible.

Fractional Knapsack Problem

- fractions of items can be taken rather than having to make a binary (0-1) choice for each item

Both exhibit the optimal-substructure property

0-1 knapsack problem. Consider a optimal solution. If item j is removed from the load, the remaining load must be the most valuable load weighing at most $W - w_j$.

Fractional knapsack problem. If w of item j is removed from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that can be taken from other $n-1$ items plus $w_j - w$ of item j.

```
Fractional Knapsack (Array v, Array w, int W)
 1.  for i = 1 to Size(v)
 2.      do p[i] = v[i] / w[i]
 3.  Sort-Descending(p)
 4.  i ← 1
 5.  while (W > 0)
 6.      do amount = min(W, w[i])
 7.      solution[i] = amount
 8.      W = W-amount
 9.      i ← i + 1
10.  return solution
```

Q6.

Example. *Consider 5 items along their respective weights and values*

$$I = \langle I_1, I_2, I_3, I_4, I_5 \rangle$$
$$w = \langle 5, 10, 20, 30, 40 \rangle$$
$$v = \langle 30, 20, 100, 90, 160 \rangle$$

The capacity of knapsack $W = 60$. Find the solution to the fractional knapsack problem.

Solution. Initially,

Item	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio *i.e.*, $p_i = v_i / w_i$

Item	w_i	v_i	$p_i = v_i / w_i$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

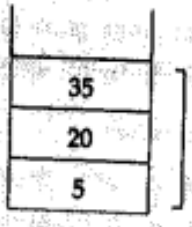
Now, arrange the value of p_i in decreasing order.

Item	w_i	v_i	$p_i = v_i / w_i$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Now, fill the knapsack according to the decreasing value of p_i.

First we choose item I_1 whose weight is 5, then choose item I_3 whose weight is 20. Now the total weight in knapsack is $5+20=25$.

Now, the next item is I_5 and its weight is 40, but we want only 35. So we choose fractional part of if *i.e.*,



The value of fractional part of I_5 is

$$\frac{160}{40} \times 35 = 140$$

Thus the maximum value is

$$= 30 + 100 + 140 = 270$$

Q7. Time Complexity of fractional knapsack problem.

To calculate the time complexity of the **Fractional Knapsack Problem**, let's break down the steps involved in the greedy approach:

1. **Calculate Value-to-Weight Ratios**:
    - For each item, you need to compute the value-to-weight ratio. This requires a single pass through the list of items, which takes O(n), where n is the number of items.
2. **Sort Items by Value-to-Weight Ratio**:
    - After calculating the ratios, you need to sort the items based on their value-to-weight ratios in descending order. The time complexity for sorting is O(nlogn). This is typically done using efficient sorting algorithms like Merge Sort or Quick Sort.
3. **Greedily Select Items**:
    - Iterate through the sorted list and add items to the knapsack. For each item, if the remaining capacity allows, you take the entire item; otherwise, you take the fraction that fits. This step requires a single pass through the sorted list, which takes O(n).

Combining these steps, we get the overall time complexity of the greedy algorithm for the Fractional Knapsack Problem:

$$O(n) + O(n \log n) + O(n) = O(n \log n)$$

Thus, the dominant factor in the time complexity is the sorting step, leading to a final time complexity of $O(n \log n)$.

Q8. Huffman Code

Data can be encoded efficiently using Huffman codes. It is a widely used and very effective technique for compressing data ; savings of 20% to 90% are typical, depending on the characteristics of the file being compressed. Huffman's greedy algorithm uses a table of the frequencies of occurrence of each character to build up an optimal way of representing each character as a binary string.

Suppose we have 10^5 characters in a data file. Normal storage: 8 bits per character (ASCII)- 8×10^5 bits in file. But, we want to compress the file and store it compactly. Suppose only 6 characters appear in the file :

	a	*b*	*c*	*d*	*e*	*f*	Total
Frequency	45	13	12	16	9	5	100

How can we represent the data in a compact way ?

(i) **Fixed Length code.** Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character :

For example :

a	000
b	001
c	010
d	011
e	100
f	101

(ii) A **variable-length code** can do considerably better than a fixed-length code, by giving frequent characters short code words and infrequent characters long code words.

For example,

a	0
b	101
c	100
d	111
e	1101
f	1100

Number of bits $= (45*1+13*3+12*3+16*3+9*4+5*4)*1000$

$= 2.24 \times 10^5$ bits.

Thus, 224,000 bits to represent the file, a saving of approximately 25%. In fact, this is an optimal character code for this file.

Let us denote the characters by $C_1, C_2, \ldots, C_n$ and denote their frequencies by $f_1, f_2, \ldots, f_n$. Suppose there is an encoding E in which a bit string S_i of length s_i represents C_i, the length of the file compressed by using encoding E is

$$L(E, F) = \sum_{i=1}^{n} s_i \cdot f_i$$

Q9. Huffman Code Analysis

Steps in Huffman Coding

1. **Count Frequencies**: Count the frequency of each character in the input text.
2. **Build a Priority Queue**: Insert all characters with their frequencies into a priority queue (or min-heap).
3. **Build the Huffman Tree**: Continuously extract the two nodes with the lowest frequencies, combine them into a new node, and insert this node back into the priority queue until only one node remains (the root of the Huffman tree).
4. **Assign Codes**: Traverse the tree to assign binary codes to each character.

Operation	Time Complexity	Space Complexity
Building Huffman Tree	O(N log N)	O(N)
Encoding	O(N)	O(1)
Decoding	O(N)	O(1)

Q10. Task Scheduling Problems

Task scheduling problems involve organizing and allocating tasks to resources over time, with the goal of optimizing various criteria such as completion time, resource utilization, and overall efficiency. Here's an overview of some common task scheduling problems:

1. Definition

Task scheduling is the process of assigning tasks to resources (e.g., machines, processors, or workers) while satisfying specific constraints and optimizing an objective function. This can include minimizing completion time, maximizing resource usage, or adhering to deadlines.

This is the problem of optimally scheduling unit-time tasks on a single processor, where each task has a dead line and a penalty that must be paid if the dead line is missed.

A **unit time task** is a job such as a program to be run on a computer that requires exactly one unit of time to complete. Given a finite set S of unit-time tasks, a schedule for S is a permutation of S specifying the order in which these tasks are to be performed. The first task in the schedule begins at time 0 and finishes at time 1, the second task begins at time 1 and finishes at time 2 and so on.

The problem of scheduling unit time tasks with dead lines and penalties for a single processor has the following inputs :

- a set $S = \{1, 2, 3, \ldots n\}$ of n-unit-time tasks.
- a set of n integer dead lines $d_1, d_2, d_3, \ldots d_n$ such that each d_i satisfies $1 \le d_i \le n$ and task i is supposed to finish by time d_i and
- a set of n non-negative weights or penalties $w_1, w_2, \ldots w_n$ such that a penalty w_i is incurred if task i is not finished by time d_i and no penalty is incurred if a task finishes by its dead line.

Here we find a schedule for S that minimizes the total penalty incurred for missed dead lines.

A task is **late** in this schedule if it finished after its dead line. Otherwise, the task is **early** in the schedule. An arbitrary schedule can always be put into **early-first form**, in which the early tasks precede the late tasks, *i.e.*, if some early task x follows some late task y, then we can switch the positions of x and y without affecting x being early or y being late.

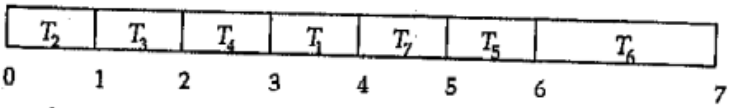
Example. *Find the optimal schedule for the following task with given weights (penalties) and dead lines.*

	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Solution. According to Greedy algorithm we sort the jobs in decreasing order of their penalties so that minimum of penalties will be charged.

In this problem, we can see that maximum time for which uniprocessor machine will run is 6 units because it is the maximum dead line.

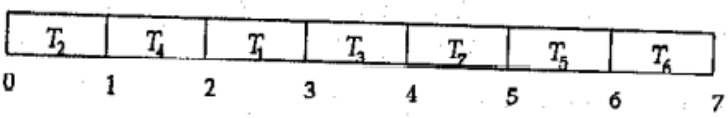
Let T_i represents the tasks where $i = 1$ to 7



T_5 and T_6 cannot be accepted after T_7 so penalty is

$$w_5 + w_6 = 30 + 20 = 50. \qquad \langle 2\ 3\ 4\ 1\ 7\ 5\ 6 \rangle$$

other schedule is



$\langle 2\ 4\ 1\ 3\ 7\ 5\ 6 \rangle$

There can be many other schedules but $\langle 2\ 4\ 1\ 3\ 7\ 5\ 6 \rangle$ is optimal.

Note *In this smaller dead line come first and complete the task earlier than larger dead line tasks.*

Q11. Travelling Sales man problem-solving

22.8 Travelling Sales Person Problem

In this a salesman needs to visit 'n' cities in such a manner that all cities must be visited at once and in the end he returns to the city from where he started with minimum cost.

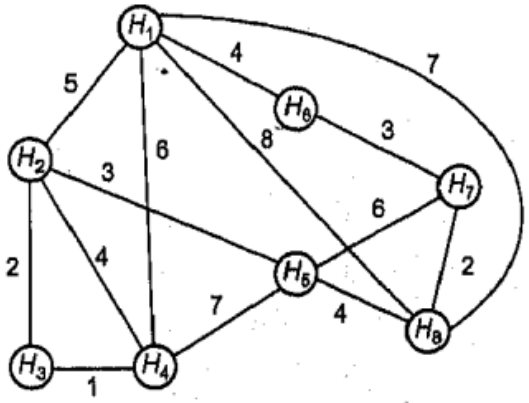
Suppose the cities are $x_1, x_2, \ldots x_n$ where cost c_{ij} denotes the cost of travelling from city x_i to x_j. The travelling salesman problem is to find a route starting and ending at x_1 that will take in all the cities with the minimum cost.

Greedy Strategy for Travelling Salesman Problem

Start with any arbitrary city called x_1 then choose the minimum cost city from x_1. The method is represented until all the cities are visited and at every step we have to select the city with the minimum weight.

Example. *A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.*

The area assigned to the agent where he has to drop the newspaper is shown in the Fig.



Solution. The cost-adjacency matrix of graph G is as follows :

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	(4)	0	7
H_2	5	0	(2)	4	3	0	0	0
H_3	0	2	0	(1)	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	(3)	0	7	0	0	6	4
H_6	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	(2)
H_8	7	0	0	0	(4)	0	2	0

Mark area H_4 and select minimum cost area reachable from H_4 it is H_1.

So, using the greedy strategy we get the following

$$H_1 \overset{4}{\rightarrow} H_6 \overset{3}{\rightarrow} H_7 \overset{2}{\rightarrow} H_8 \overset{4}{\rightarrow} H_5 \overset{3}{\rightarrow} H_2 \overset{2}{\rightarrow} H_3 \overset{1}{\rightarrow} H_4 \overset{6}{\rightarrow} H_1$$

Thus, the minimum travel cost

$$= 4+3+2+4+3+2+1+6 = 25$$

Q12. Huffman Algorithm

```
HUFFMAN(C)
1.  n ← |C|
2.  Q ← C
3.  for i ← 1 to n − 1
4.      do z ← ALLOCATE-NODE()
5.         x ← left[z] ← EXTRACT-MIN(Q)
6.         y ← right[z] ← EXTRACT-MIN(Q)
7.         f[z] ← f[x] + f[y]
8.         INSERT(Q, z)
9.  return EXTRACT-MIN(Q)
```

Q13. Minimum Spanning Tree

A Minimum Spanning Tree (MST) is a subset of the edges of a connected, weighted, undirected graph that connects all the vertices together without any cycles and with the minimum possible total edge weight. Here's a detailed explanation of MSTs:

Definition

A Minimum Spanning Tree of a graph is a spanning tree with the smallest sum of edge weights. A graph may have multiple spanning trees, but only one of them will be the minimum spanning tree.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both algorithms differ in their methodology, but both eventually end up with the MST. ***Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST.*** Both algorithms are greedy algorithms that run in polynomial time. At each step of an algorithm, one of several possible choices must be made.

Q14. Kruskal's Algorithms

27.2 Kruskal's Algorithm

Kruskal's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted graph. It finds a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge (u,v) of least weight. 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 finds a minimum spanning forest (a minimum spanning tree for each connected component).**

```
MST-KRUSKAL (G, w)
1.  A ← ϕ
2.  for each vertex v ∈ V[G]
3.      do MAKE-SET(v)
4.  sort the edges of E into nondecreasing order by weight w
5.  for each edge (u,v) ∈ E, taken in nondecreasing order by weight
6.      do if FIND-SET(u) ≠ FIND-SET(v)
7.          then A ← A ∪ {(u,v)}
8.               UNION (u,v)
9.  return A
```

Analysis

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, all with simple data structures. These running times are equivalent because :

- E is at most V^2 and $\log V^2 = 2 \times \log V$ is $O(\log V)$.
- If we ignore isolated vertices, which will each be their own component of the minimum spanning tree anyway, $V \leq 2E$, so $\log V$ is $O(\log E)$.

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time ; Next, we use a disjoint-set data structure to keep track of which vertices are in which components. We need to perform $O(E)$ operations, two find operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform $O(E)$ operations in $O(E \log V)$ time. Thus the total time is

$$O(E \log E) = O(E \log V).$$

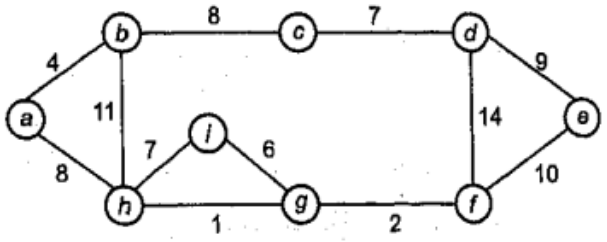
Q15.

Example. *Let (u,v) be a minimum weight edge in a graph G. Show that (u,v) belongs to some minimum spanning tree of G.*

Solution. Proof by contradiction : assume that (u, v) doesn't belong to some MST of G. If we would add (u,v) to the MST we would get a cycle because MST is a tree. But if we then removed another edge in the cycle we would end up with a MST that is at least as cheap as the original.

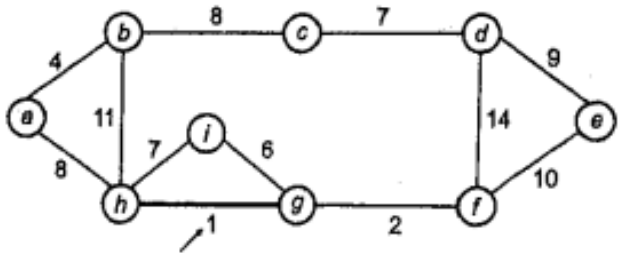
Hence, (u,v) does belong to some minimum spanning tree of G.

Example. *Find the minimum spanning tree of the following graph using kruskals algorithm.*

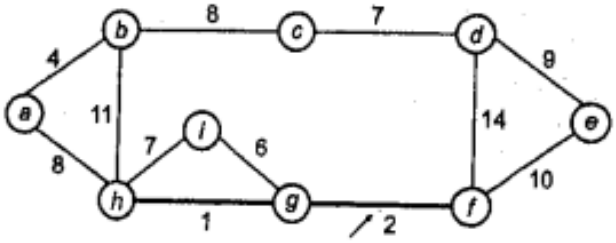


Solution. First we initialize the set A to the empty set and create $|v|$ trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight, *i.e.*,

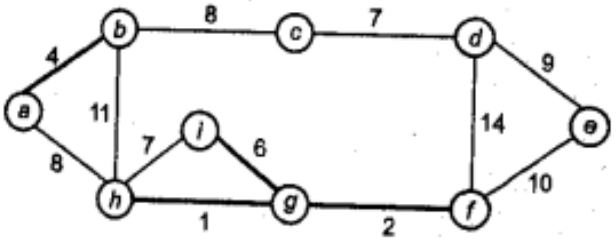
So, first take (h, g) edge



the (g, f) edge.

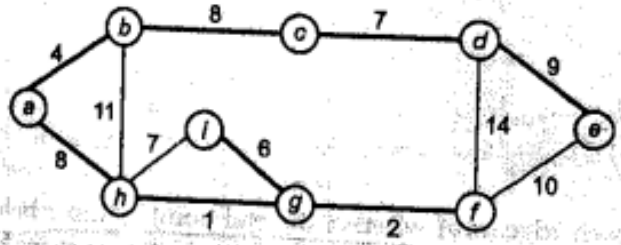


then (a, b) and (i, g) edges are considered and forest becomes.



Now, edge (h, i). Both h and i vertices are in same set, thus it creates a cycle. So this edge is discarded.

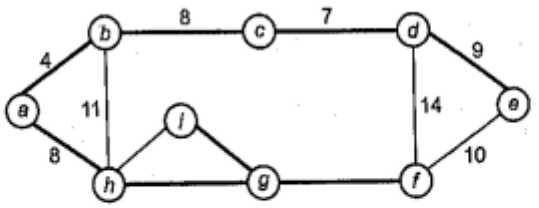
Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered and forest becomes.



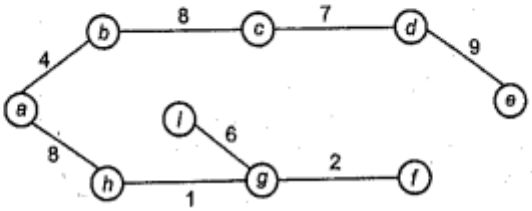
In (e, f) edge both end points e and f exist in same tree so discarded this edge.

Then (b, h) edge, it also creates a cycle.

After that edge (d, f) and the final spanning tree is shown as in dark lines.



or



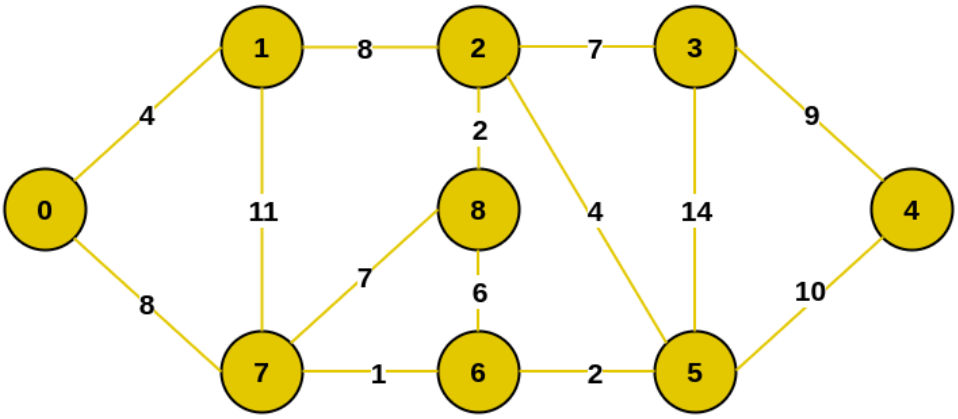
Minimum cost MST

Q16. Prim's Algorithm

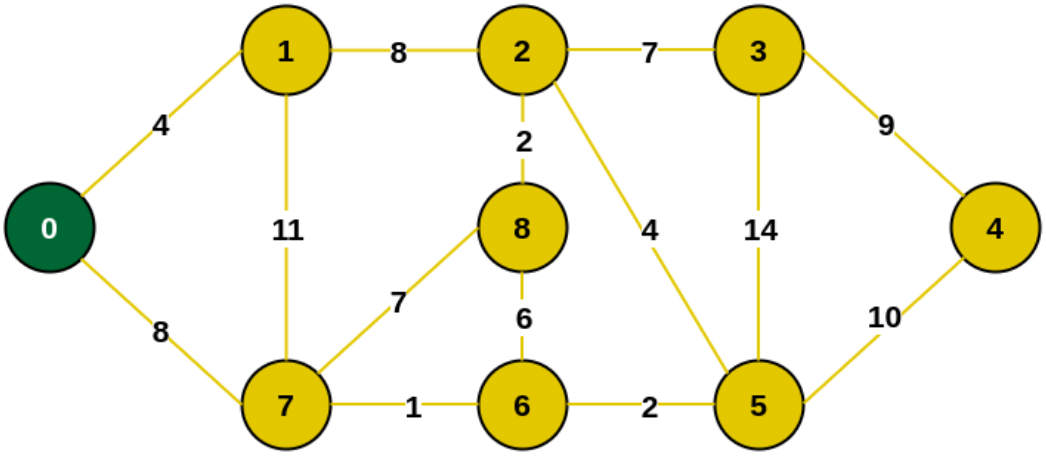
27.3 Prim's Algorithm

The main idea of Prim's algorithm is similar to that of Dijkstra's algorithm (discussed later) for finding shortest path in a given graph. Prim's algorithm has the property that the edges in the set A always form a single tree. We begin with some vertex v in a given graph $G=(V, E)$, defining the initial set of vertices A. Then, in each iteration, we choose a minimum-weight edge (u,v), connecting a vertex v in the set A to the vertex u outside of set A. Then vertex u is brought in to A. This process is repeated until a spanning tree is formed. Like Kruskal's algorithm, here too, the important fact about MSTs is we always choose the smallest-weight edge joining a vertex inside set A to the one outside the set A. The implication of this fact is that it adds only edges that are safe for A; therefore when the algorithm terminates, the edges in set A form a MST.

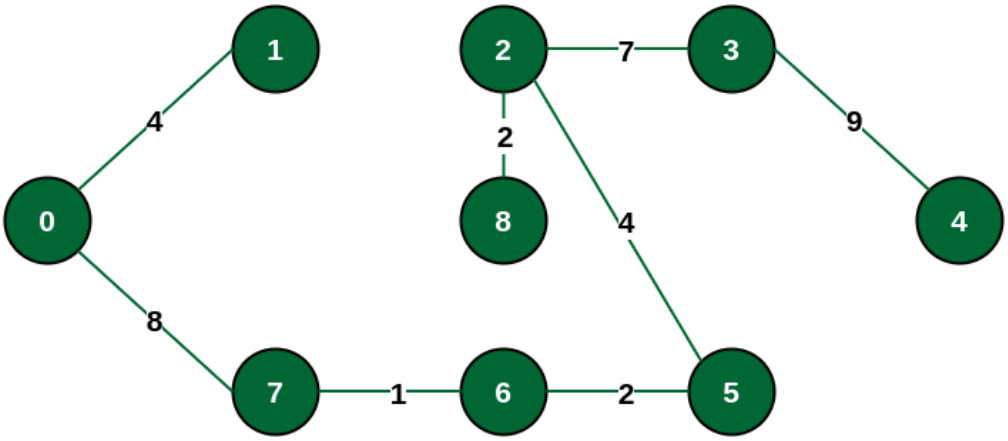
```
MST-PRIM(G, w, r)
1.  for each u ∈ V[G]
2.      do key[u] ← ∞
3.          π[u] ← NIL
4.  key[r] ← 0
5.  Q ← V[G]
6.  while Q ≠ ϕ
7.      do u ← EXTRACT-MIN(Q)
8.          for each v ∈ Adj[u]
9.              do if v ∈ Q and w(u,v) < key[v]
10.                 then π[v] ← u
11.                     key[v] ← w(u,v)
```



Example of a Graph



Select an arbitrary starting vertex. Here we have selected 0



The final structure of MST

Q17. Single source shortest path

The **Single Source Shortest Path** (SSSP) problem involves finding the shortest paths from a given source vertex to all other vertices in a weighted graph. The graph can be either directed or undirected, and the weights of the edges can be positive or negative, depending on the algorithm used. SSSP is fundamental in many areas, such as network routing, navigation systems, and various optimization problems.

```
INITIALIZE-SINGLE-SOURCE (G, s)
1.  for each vertex v ∈ V[G]
2.      do d[v] ← ∞
3.      π[v] ← NIL
4.  d[s] ← 0
```

Q18. Dijkstra's Algorithm

Dijkstra's algorithm, named after its discoverer, Dutch computer scientist **Edsger Dijkstra**, is a greedy algorithm that solves the single-source shortest path problem for a directed graph $G=(V, E)$ with nonnegative edge weights *i.e.* we assume that $w(u,v) \geq 0$ for each edge $(u,v) \in E$.

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. That is, for all vertices $v \in S$, we have $d[v] = \delta(s,v)$. The algorithm repeatedly selects the vertex $u \in V-S$ with the minimum shortest-path estimate, inserts u into S, and relaxes all edges leaving u. We maintain a priority queue Q that contains all the vertices in $v-s$, keyed by their d values. Graph G is represented by adjacency lists.

Dijkstra's algorithm finds the shortest path from the source vertex to all other vertices in a graph with non-negative edge weights. It is a greedy algorithm that repeatedly selects the vertex with the smallest known distance to the source and updates the distances to its neighboring vertices.

```
DIJKSTRA(G, w, s)
1.  INITIALIZE-SINGLE-SOURCE (G, s)
2.  S ← ϕ
3.  Q ← V[G]
4.  while Q ≠ ϕ
5.      do u ← EXTRACT-MIN(Q)
6.          S ← S ∪ {u}
7.          for each vertex v ∈ Adj[u]
8.           do RELAX (u, v, w)
```

Q19. Dijkstra's Analysis

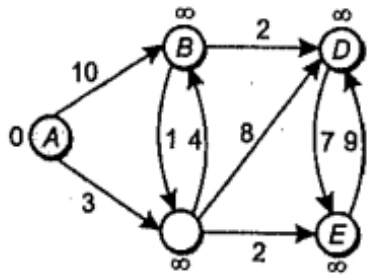
Analysis

The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of $|E|$ and $|V|$ using the Big-O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract-Min (Q) is simply a linear search through all vertices in Q. In this case, the running time is $O(|V|^2 + |E| = O(V^2)$.

For sparse graphs, that is, graphs with many fewer than $|V|^2$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a binary heap or Fibonacci heap as a priority queue to implement the Extract-Min function. With a binary heap, the algorithm requires $O((|E|+|V|)\log|V|)$ time (which is dominated by $O(|E|\log|V|)$ assuming every vertex is connected), and the Fibonacci heap improves this to $O(|E|+|V|\log|V|)$.

Q20.

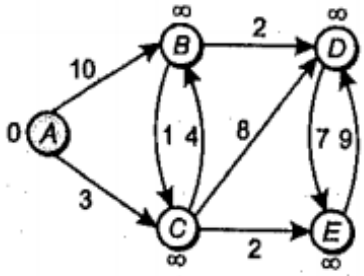
Initialize :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: { }

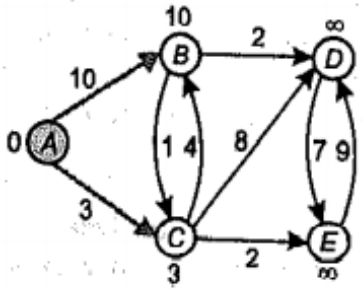
"A" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {A}

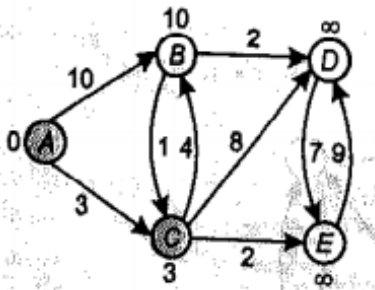
Relax all edges leaving A:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

S: {A}

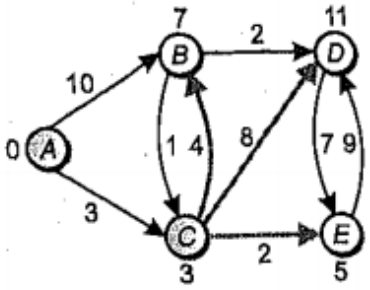
"C" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

S: {A, C}

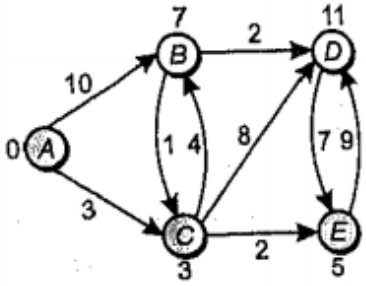
Relax all edges leaving C:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

$S: \{A, C\}$

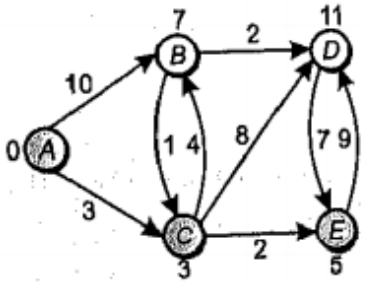
"E" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

$S: \{A, C, E\}$

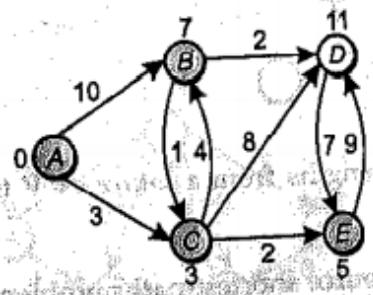
Relax all edges leaving E:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{A, C, E\}$

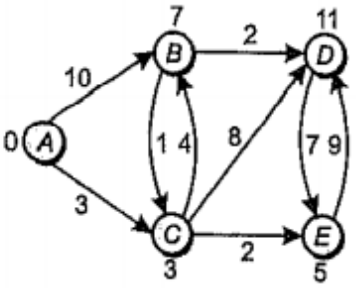
"B" ← EXTRACT-MIN(Q) :



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{A, C, E, B\}$

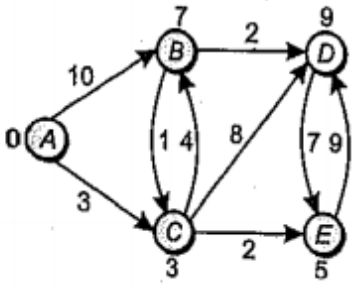
Relax all edges leaving B:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: {A, C, E, B}

"D" ← EXTRACT-MIN(Q):



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: {A, C, E, B, D}

Q21. The Bellman-Ford Algorithm

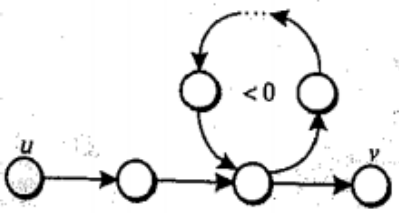
The **Bellman-Ford Algorithm** is a single-source shortest path algorithm designed to handle graphs where edges can have negative weights. It can detect negative-weight cycles (cycles in which the total weight of the edges is negative), making it more versatile than Dijkstra's algorithm, which cannot process graphs with negative weights.

Bellman-Ford uses a dynamic programming approach and repeatedly relaxes all the edges in the graph to guarantee that the shortest path from the source vertex to any other vertex is calculated correctly.

28.6 The Bellman-Ford Algorithm

If a graph $G=(V, E)$ contains a negative weight cycle, then some shortest paths may not exist.

Example :



Bellman-Ford algorithm finds all shortest-path lengths from a *source* $s \in V$ to all $v \in V$ or determines that a negative-weight cycle exists.

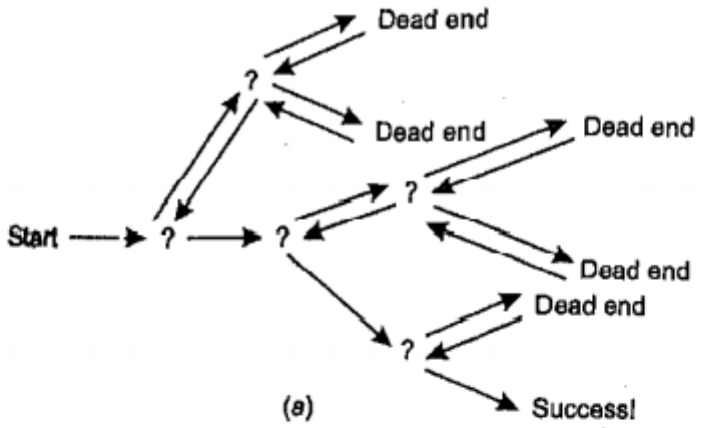
Thus, ***Bellman-Ford algorithm*** solves the single-source shortest-paths problem in the more general case in which edge weights can be **negative**. Given a weighted, directed graph $G=(V, E)$ with source s and weight function $w : E \rightarrow R$, the Bellman-Ford algorithm returns a **boolean value**

Q22. What is backtracking?

Backtracking is a problem-solving algorithmic technique used to solve complex problems by breaking them down into smaller sub-problems. It explores possible solutions incrementally, abandoning ("backtracking" from) a path as soon as it determines that the path cannot lead to a valid solution

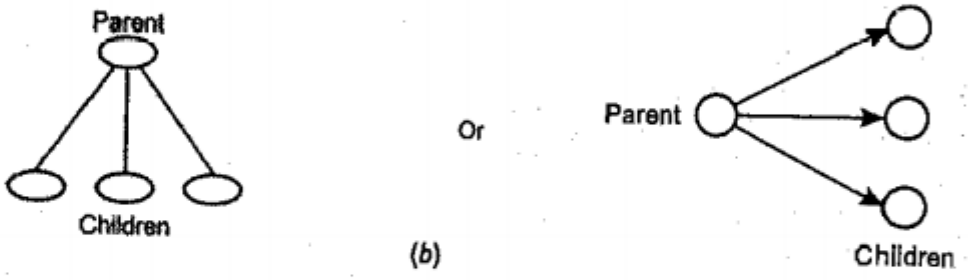
Key Points:

- **Recursive Approach**: Backtracking is often implemented as a recursive algorithm.
- **Trial and Error**: It tries to build a solution piece by piece, checking whether the current solution satisfies the problem's conditions.
- **Backtracking Step**: When a partial solution is found to be invalid or no longer leads to a complete solution, the algorithm backtracks to the previous step and tries a different option.



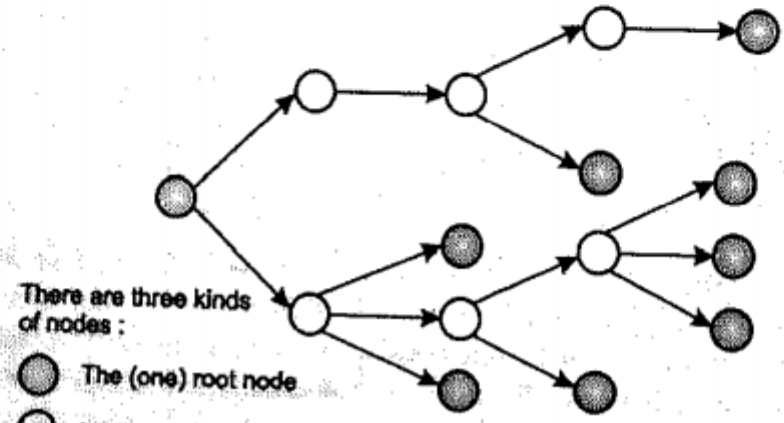
(a)

Usually, however, we draw our trees *downward*, with the root at the top.



(b)

A tree is composed of nodes

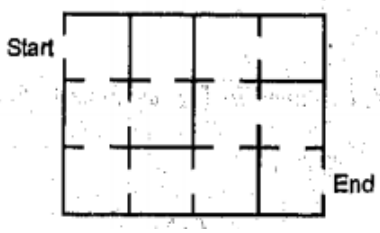


Q23. Recursive Maze Algorithms

Recursive maze algorithm is one of the good examples for backtracking algorithms. In fact Recursive maze algorithm is one of the most available solutions for solving maze.

Maze

Maze is an area surrounded by walls; in between we have a path from starting position to ending position. We have to start from the starting point and travel towards the ending point.



Principle of Maze

As explained above, in maze we have to travel from the starting point to ending point. The problem is to choose the path. If we find any dead-end before ending point, we have to backtrack and change the direction. The direction for traversing is North, East, West and South. We have to continue "move and backtrack" until we reach the ending point.

Assume that we are having a two-dimensional maze cell [WIDTH][HEIGHT]. Here cell[x][y]=1 denotes wall and cell[x][y]=0 denotes free cell in the particular location x, y in the maze. The directions we can move in the array are North, East, West and South. The first step is to make the boundary of the two-dimensional array as 1 so that we won't go out of the maze, and always reside inside the maze at any time.

Example Maze

1	1	1	1	1	1	1
1	0	0	0	1	1	1
1	1	1	0	1	1	1
1	1	1	0	0	0	1
1	1	1	1	1	0	1
1	1	1	1	1	1	1

Now start moving from the starting position (since the boundary is filled by 1) and find the next free cell, then move to the next free cell and so on. If we reach a dead-end, we have to backtrack and make the cells in the path as 1(wall). Continue the same process till the ending point is reached.

Q24. Hamiltonian Circuit Problems

A **Hamiltonian Circuit** (or **Hamiltonian Cycle**) is a problem in graph theory where the goal is to determine whether there exists a path in a graph that visits each vertex exactly once and returns to the starting vertex.

Problem Definition:

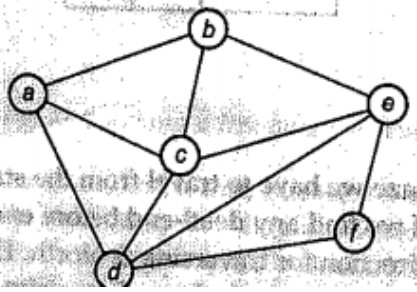
- **Input**: A graph G=(V,E), where V is the set of vertices and E is the set of edges.
- **Output**: Determine if there exists a Hamiltonian circuit, i.e., a cycle that visits every vertex exactly once and returns to the starting vertex.

23.3 Hamiltonian Circuit Problem

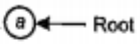
Given a graph $G=(V, E)$, we have to find the ***Hamiltonian Circuit*** using Backtracking approach. We start our search from any arbitrary vertex, say '*a*'. This vertex '*a*' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed. The next adjacent vertex is selected on the basis of alphabetical (or numerical) order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex '*a*' then we say that **dead end** is reached. In this case we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial solution must be removed. The search using backtracking is successful if a Hamiltonian cycle is obtained.

Q25.

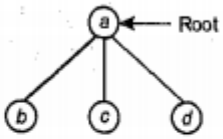
Example. *Consider a graph G=(V, E) shown in Fig. we have to find a Hamiltonian circuit using Back tracking method.*



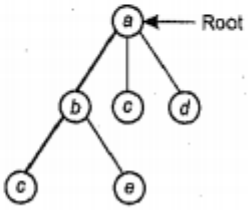
Solution. Firstly, we start our search with vertex '*a*', this vertex '*a*' becomes the root of our implicit tree.



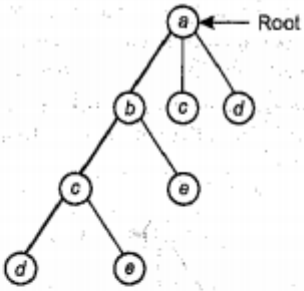
Next, we choose vertex '*b*' adjacent to '*a*' as it comes first in lexicographical order (*b, c, d*).



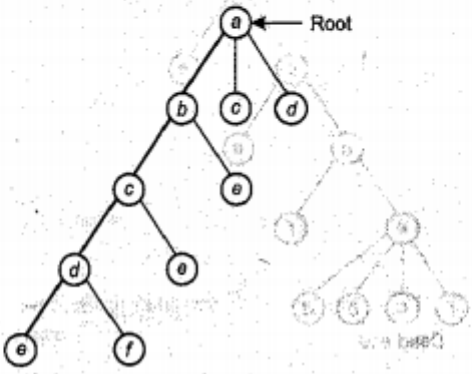
Next, we select '*c*' adjacent to '*b*'

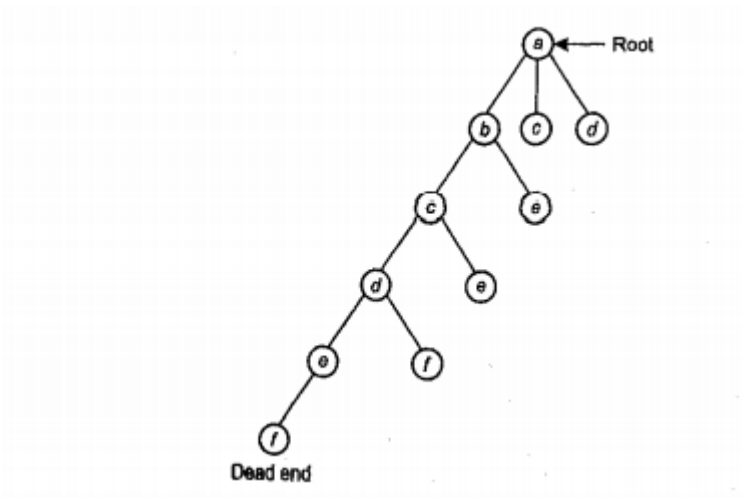


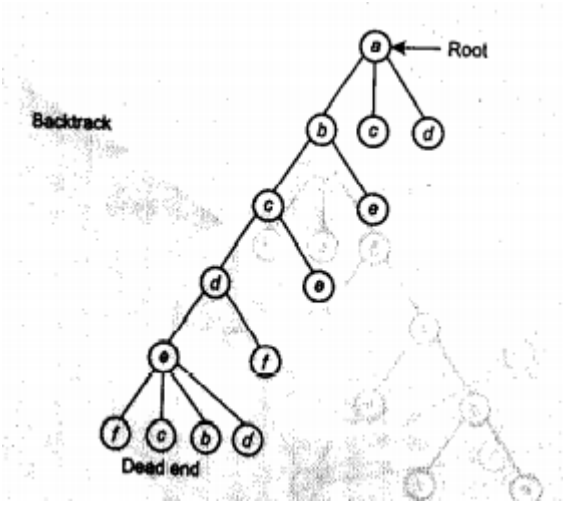
Next, we select '*d*' adjacent to '*c*'

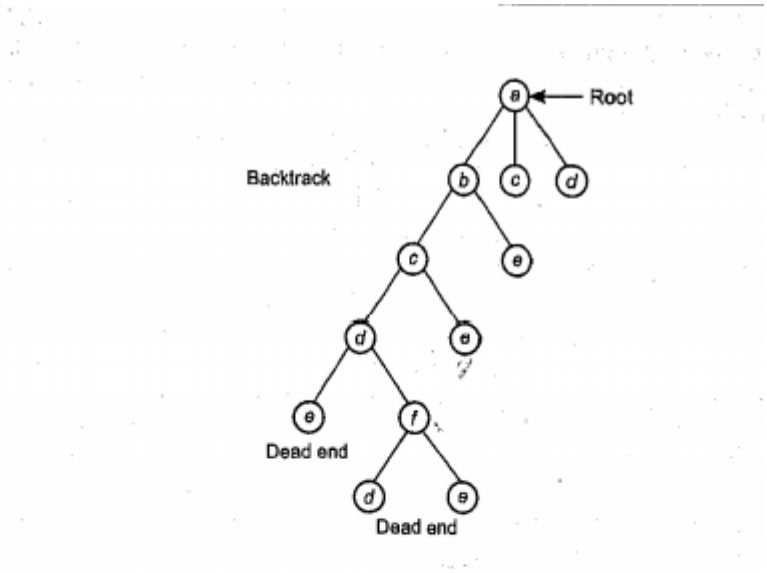


Next, we select '*e*' adjacent to '*d*'

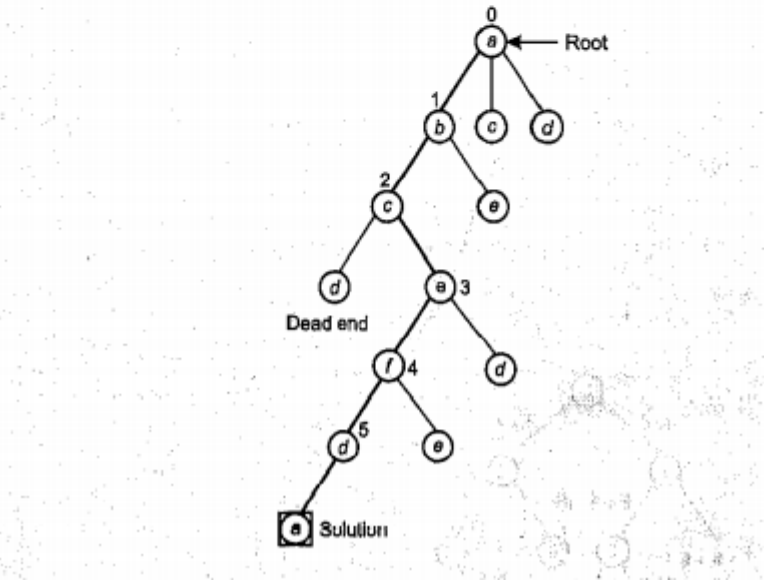








Again back track



Q26. Subset-sum Problem

The **Subset Sum Problem** is a classic decision problem in computer science where, given a set of integers, the goal is to determine whether there is a subset of those integers that sums up to a given target value.

In the subset-sum problem we have to find a subset s' of the given set $S=\langle S_1, S_2, S_3, \ldots S_n\rangle$ where the elements of the set S are n positive integers in such a manner that $s' \in S$ and sum of the elements of subset 's' ' is equal to some positive integer 'X'.

The subset-sum problem can be solved by using the backtracking approach. In this implicit tree is a binary tree. The root of the tree is selected in such a way that it represents that no decision is yet taken on any input. We assume that the elements of the given set are arranged in an increasing order :

$$S_1 \le S_2 \le S_3 \ldots \le S_n$$

The left child of the root node indicates that we have to include 'S_1' from the set 'S' and the right child of the root node indicates that we have to exclude 'S_1' . Each node stores the sum of the partial solution elements. If at any stage the number equals to 'X' then the search is successful and terminates.

The dead end in the tree occurs only when either of the two inequalities exist :

- The sum of s' is too large *i.e.*

$$s' + S_i + 1 > X$$

- The sum of s' is too small *i.e.*

$$s' + \sum_{j=i+1}^{n} S_j < X$$

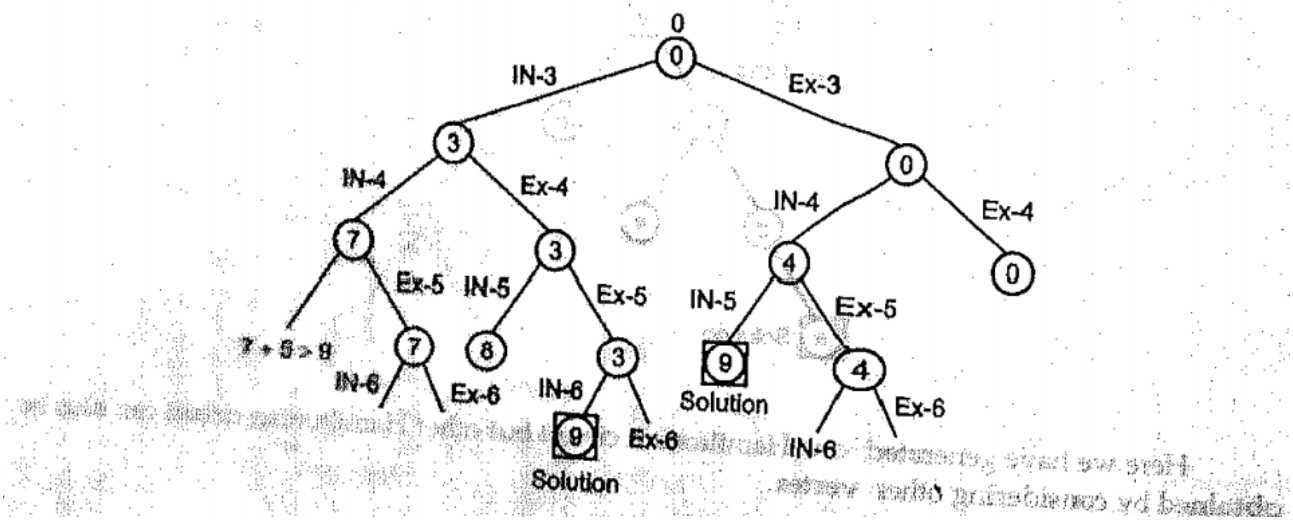
Q27.

Example. *Given a set* $S=\langle 3,4,5,6\rangle$ *and* $X=9$. *Obtain the subset sum using Back tracking approach.*

Solution. Initially $S=\langle 3,4,5,6\rangle$ and $X=9$

$$S' = \langle \phi \rangle$$

The implicit binary tree for subset sum problem is shown in Fig.



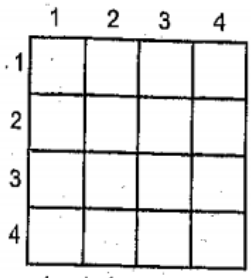
Q28. N-Queen Problems

23.5 *N*-Queens Problem

N-queens problem is to place n-queens in such a manner on an $n \times n$ chessboard that no two queens attack each other by being in the same row, column or diagonal.

It can be seen that for $n=1$, the problem has a trivial solution, and no solution exists for $n=2$ and $n=3$. So first we will consider the 4-queens problem and then generalise it to n-queens problem.

Given, a 4×4 chessboard and number the rows and column of the chessboard 1 through 4.



4 × 4 chessboard

Figure 23.3

This is one possible solution for 4-queens problem. For other possible solution the whole method is repeated for all partial solutions. The other solution for 4-queens problem is $\langle 3,1,4,2 \rangle$ *i.e.*,

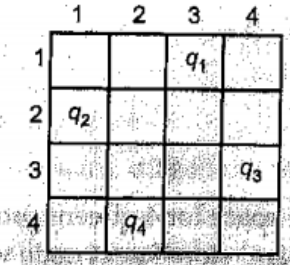
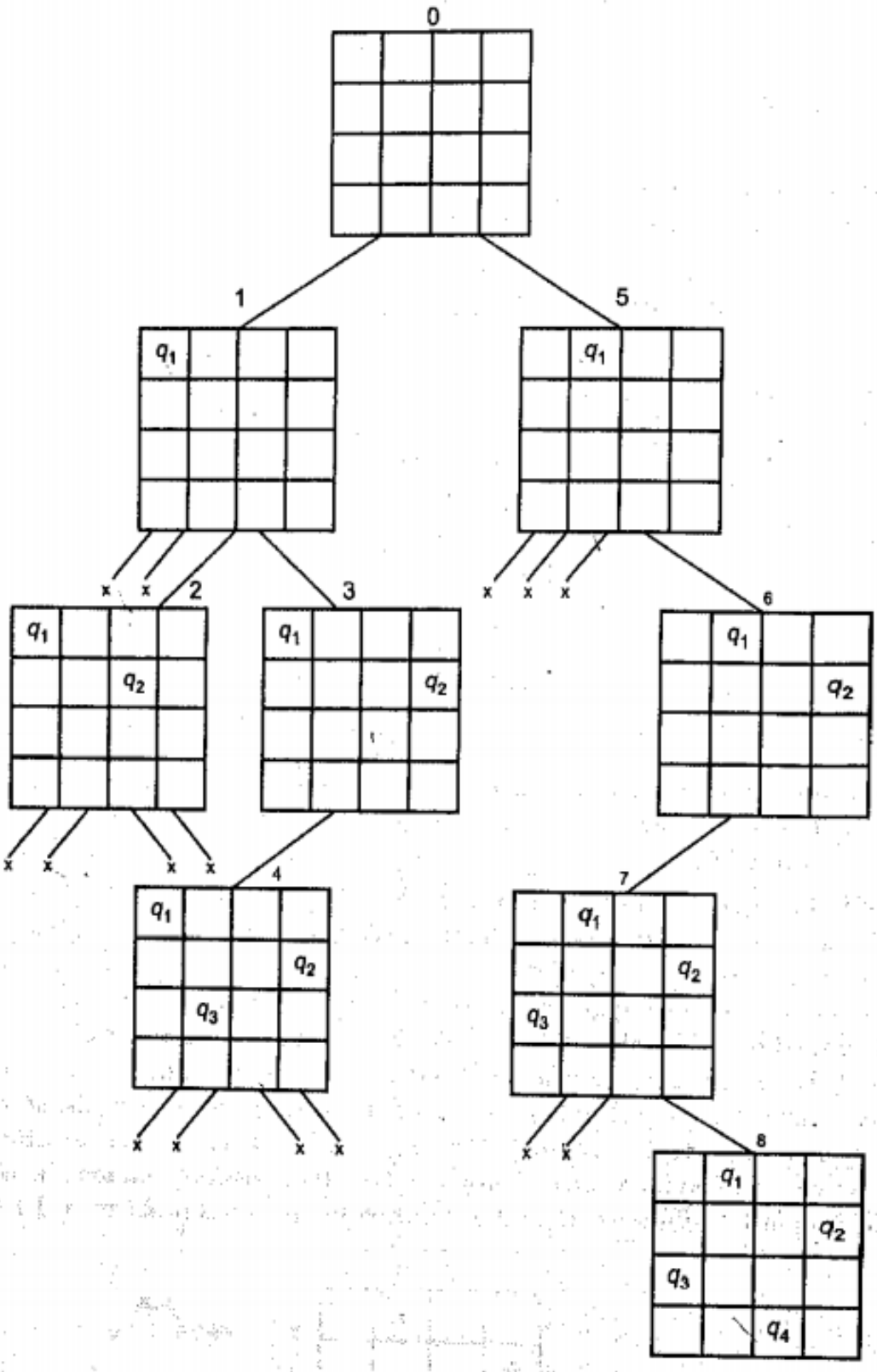


Figure 23.4

The implicit tree for 4-queen problem for solution $\langle 2,4,1,3 \rangle$ is as follows :



re 23.5

It can be seen that all the solutions to the 4-queens problem can be represented as 4-tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in the Fig. 23.7

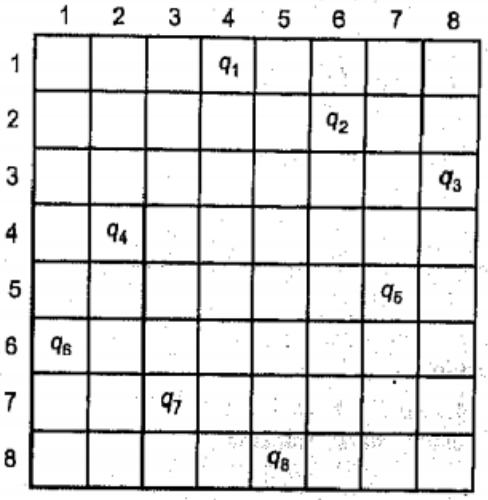


Figure 23.7

Thus, solution for 8-queen problem for $\langle 4,6,8,2,7,1,3,5 \rangle$.

If two queens are placed at positions (i, j) and (k, l).

Then, they are on the same diagonal only if $(i-j)=k-l$ or $i+j=k+l$.

Then *first* equation implies that $j-l=i-k$

The *second* equation implies that $j-l=k-i$

Therefore, two queens lie on the same diagonal if and only if $|j-l|=|i-k|$

Place (k, i) returns a boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous values $x_1, x_2 \ldots x_{k-1}$ and whether there is no other queen on the same diagonal.

Analysis

Time Complexity: O(N!)
Auxiliary Space: O(N)

Q29. Analysis of Hamilton circuit

Time Complexity : O(N!), where N is number of vertices.
Auxiliary Space : O(1), since no extra space used.

Q30. Analysis of prim's algorithm

Complexity Analysis of Prim's Algorithm:

Time Complexity: O(E*log(E)) where E is the number of edges
Auxiliary Space: O(V^2) where V is the number of vertex

Prim's algorithm for finding the minimum spanning tree (MST):

Q31. Graph Coloring Problem

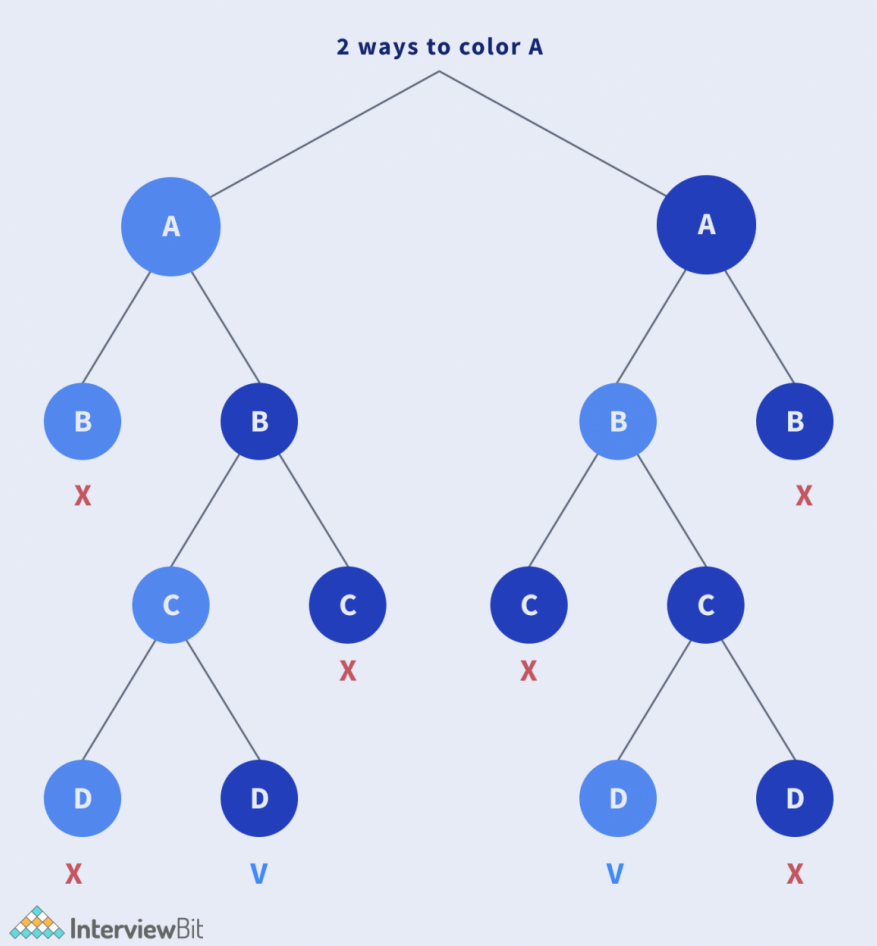
Graph coloring refers to the problem of **coloring vertices** of a graph in such a way that **no two adjacent** vertices have the **same color**. This is also called the **vertex coloring** problem. If coloring is done using at most m colors, it is called m-coloring.

Chromatic Number:

The **minimum** number of **colors** needed to **color** a graph is called its **chromatic** number. For example, the following can be colored a minimum of 2 colors.

Assign colors one by one to different vertices, starting from vertex **0**. Before assigning a color, check if the adjacent vertices have the same color or not. If there is any color assignment that does not violate the conditions, mark the color assignment as part of the solution. If no assignment of color is possible then backtrack and return false.

Time Complexity: O(m^n), where n is the number of vertices and m is the number of colors.



PYQ

Q1. Differentiate between Big Oh and Big Omega

Aspect	Big Oh (O)	Big Omega (Ω)
Definition	Represents the upper bound (worst-case)	Represents the lower bound (best-case)
Usage	Describes the maximum time or space complexity	Describes the minimum time or space complexity
Formal Definition	$f(n) \leq c \cdot g(n)$ for large n	$f(n) \geq c \cdot g(n)$ for large n
Focus	Worst-case scenario	Best-case scenario
Purpose	Gives an upper bound for growth rate	Gives a lower bound for growth rate
Example (Binary Search)	$O(\log n)$	$\Omega(1)$
Perspective	How bad the performance can be (maximum)	How good the performance can be (minimum)

Q2. Show that backtracking is better than brute force approach in terms of time complexity.

Backtracking is generally better than the brute force approach in terms of time complexity due to the following reasons:

1. **Pruning the Search Space**: Backtracking systematically explores potential solutions and abandons paths that are determined to be invalid early on. This reduces the number of configurations that need to be examined compared to brute force, which evaluates every possible solution.
2. **Efficiency in Finding Solutions**: Backtracking narrows down the search by only considering feasible solutions, leading to a quicker resolution for many problems. Brute force, on the other hand, often involves checking all combinations, making it less efficient.
3. **Reduced Time Complexity**: While both approaches may exhibit exponential time complexity in the worst case, backtracking typically has a lower average-case time complexity due to its ability to skip invalid paths. This means that, in practical scenarios, backtracking usually performs better than brute force.
4. **Scalability**: Backtracking algorithms can handle larger inputs more effectively because they do not exhaustively search all possibilities, allowing them to scale better with increased problem size.

Q3. Define greedy approach. For what find of problems it can be applied? What are its advantage and disadvantage.

he greedy approach is an algorithmic strategy that makes a series of choices, each of which looks best at the moment, with the hope of finding a global optimum. Rather than considering all possible solutions, it makes the locally optimal choice at each step with the intention that these local solutions will lead to a globally optimal solution.

Problems Suitable for the Greedy Approach

The greedy approach can be effectively applied to problems where:

1. **Optimal Substructure**: The optimal solution to the problem can be constructed from optimal solutions of its subproblems. This means that the problem can be broken down into smaller, manageable subproblems that are solved independently.
2. **Greedy Choice Property**: A globally optimal solution can be reached by selecting the local optimal choices. In other words, a local optimal choice will lead to a global optimal solution.

Common problems that can be solved using the greedy approach include:

- **Activity Selection Problem**
- **Fractional Knapsack Problem**
- **Prim's and Kruskal's algorithms for Minimum Spanning Tree**
- **Dijkstra's Algorithm for Shortest Path**
- **Huffman Coding for data compression**

Advantages of the Greedy Approach

1. **Simplicity**: Greedy algorithms are often easier to understand and implement compared to other approaches like dynamic programming.
2. **Efficiency**: They typically have lower time complexity, as they make decisions without exploring all possible configurations, leading to faster execution.
3. **Space Efficiency**: Greedy algorithms usually require less memory, as they do not store multiple solutions or states.

Disadvantages of the Greedy Approach

1. **Not Always Optimal**: The greedy approach does not guarantee a globally optimal solution for all problems. It can fail to find the best solution in cases where local optimal choices lead to suboptimal results overall.
2. **Requires Specific Problem Characteristics**: The greedy method is applicable only to specific problems that exhibit the optimal substructure and greedy choice property, limiting its usability.

Q4. The frequency of character A=5, B=45,C=30, D=15, E=3,F=2. Generate an efficient coding using huffman encoding. also calculate the numbers of bits saved for a 100 character text in comparison to fix length encoding

To generate efficient coding using Huffman encoding for the given character frequencies, we follow these steps:

Step 1: Create a Huffman Tree

1. **Character Frequencies**:
    - A: 5
    - B: 45
    - C: 30
    - D: 15
    - E: 3
    - F: 2
2. **Build the Tree**:
    - Create leaf nodes for each character and insert them into a priority queue based on their frequencies.
    - Extract the two nodes with the lowest frequencies, combine them into a new node with a frequency equal to the sum of the two, and insert it back into the queue.
    - Repeat this process until only one node remains, which becomes the root of the Huffman tree.

Step 2: Generate Huffman Codes

By traversing the Huffman tree, we can assign binary codes to each character:

- Assign '0' for the left branch and '1' for the right branch.

Step 3: Huffman Encoding

Following the Huffman tree construction, we get the following codes:

- A: 001
- B: 0
- C: 10
- D: 011
- E: 0001
- F: 0000

Step 4: Calculate Bits Saved Compared to Fixed-Length Encoding

1. **Fixed-Length Encoding**:
    - Since there are 6 characters (A, B, C, D, E, F), we need ⌈log2(6)⌉=3\lceil \log_2(6) \rceil = 3⌈log2(6)⌉=3 bits per character.
    - For a 100-character text: 100×3=300 bits100 \times 3 = 300 \text{ bits}100×3=300 bits
2. **Huffman Encoding Calculation**:
    - Calculate the total number of bits used for Huffman encoding based on character frequencies and their corresponding codes:
    - Total bits:
        - A: 5×3=155 \times 3 = 155×3=15
        - B: 45×1=4545 \times 1 = 4545×1=45

- C: 30×2=6030 \times 2 = 6030×2=60
- D: 15×3=4515 \times 3 = 4515×3=45
- E: 3×4=123 \times 4 = 123×4=12
- F: 2×4=82 \times 4 = 82×4=8

- Total bits used in Huffman Encoding:

  15+45+60+45+12+8=185 bits15 + 45 + 60 + 45 + 12 + 8 = 185 \ text{ bits}15+45+60+45+12+8=185 bits

3. **Bits Saved**:
   - Calculate the number of bits saved:

300−185=115 bits saved300 - 185 = 115 \text{ bits saved}300−185=115 bits saved

Algorithm	Time Complexity
Merge Sort	O(n log n)
Binary Search	O(log n)
Linear Search	O(n)
Selection Sort	O(n^2)
Quick Sort	O(n log n) (avg), O(n^2) (worst)
Huffman Encoding	O(n log n)
Dijkstra's Algorithm	O((V + E) log V) (using priority queue)
Bellman-Ford Algorithm	O(VE)
Graph Coloring (Greedy)	O(V^2) (adjacency matrix), O(V + E) (adjacency list)
Hamiltonian Circuit	O(n!)
Backtracking (Subset Sum)	O(2^n)
N-Queens Problem	O(n!)
Activity Selection Problem	O(n log n)
Fractional Knapsack Problem	O(n log n)
Knapsack Problem	O(nW) (dynamic programming)
Minimum Spanning Tree (Kruskal)	O(E log E)
Minimum Spanning Tree (Prim)	O((V + E) log V)