Course Code : MCS-031
Course Title : Design and Analysis of Algorithms
Assignment Number : MCA(III)/031/Assign/2020-21
Maximum Marks : 100
Weightage : 25%
Last Dates for Submission : 31st October, 2020 (For July, 2020 Session)
: 15th April, 2021 (For January, 2021 Session)

Q1: Write Merge sort algorithm. Determine its complexity in Best, Average and Worst Case. Sort the following sequence in increasing order: 35, 37, 18, 15, 40, 12; Using Merge Sort.

Ans. Merge sort algorithm: Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

   middle m = (l+r)/2

2. Call mergeSort for first half:

   Call mergeSort(arr, l, m)

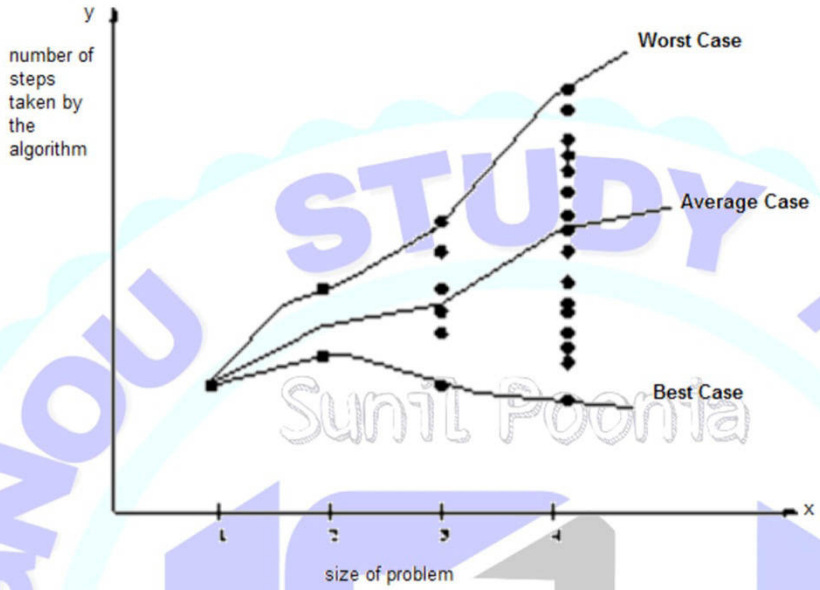
3. Call mergeSort for second half:

   Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

   Call merge(arr, l, m, r)

Determine its complexity in Best, Average and Worst Case: To understand the notions of the best, worst, and average-case complexity, one must think about running an algorithm on all possible instances of data that can be fed to it. For the problem of sorting, the set of possible input instances consists of all the possible arrangements of all the possible numbers of keys. We can represent every input instance as a point on a graph, where the x-axis is the size of the problem (for sorting, the number of items to sort) and the y-axis is the number of steps taken by the algorithm on this instance. Here we assume, quite reasonably, that it doesn't matter what the values of the keys are, just how many of them there are and how they are ordered. It should not take longer to sort 1,000 English names than it does to sort 1,000 French names, for example.



Best, worst, and average-case complexity

As shown in Figure , these points naturally align themselves into columns, because only integers represent possible input sizes. After all, it makes no sense to ask how long it takes to sort 10.57 items. Once we have these points, we can define three different functions over them:

- **The worst-case complexity** of the algorithm is the function defined by the maximum number of steps taken on any instance of size n. It represents the curve passing through the highest point of each column.
- **The best-case complexity** of the algorithm is the function defined by the minimum number of steps taken on any instance of size n. It represents the curve passing through the lowest point of each column.
- **The average-case complexity** of the algorithm is the function defined by the average number of steps taken on any instance of size n.

In practice, the most useful of these three measures proves to be the worst-case complexity, which many people find counterintuitive. To illustrate why worst-case analysis is important, consider trying to project what will happen to you if you bring n dollars to gamble in a casino. The best case, that you walk out owning the place, is possible but so unlikely that you should place no credence in it. The worst case, that you lose all n dollars, is easy to calculate and distressingly likely to happen. The average case, that the typical person loses 87.32% of the money that they bring to the casino, is difficult to compute and its meaning is subject to debate. What exactly does average mean? Stupid people lose more than smart people, so are you smarter or dumber than the average person, and by how much? People who play craps lose more money than those playing the nickel slots. Card counters at blackjack do better on average than customers who accept three or more free drinks. We avoid all these complexities and obtain a very useful result by just considering the worst case.

The important thing to realize is that each of these time complexities defines a numerical function, representing time versus problem size. These functions are as well-defined as any other numerical function, be it $y=x^2 - 2x+1$ or the price of General

Motors stock as a function of time. Time complexities are complicated functions, however. In order to simplify our work with such messy functions, we will need the big Oh notation.

Using Merge Sort 35, 37, 18, 15, 40, 12

we take an unsorted array as the following –

35	37	18	15	40	12

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 6 items is divided into two arrays of size 3.

35	37	18

15	40	12

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

35	37

18

15	40

12

We further divide these arrays and we achieve atomic value which can no more be divided.

35

37

18

15

40

12

Now, we combine them in exactly the same manner as they were broken down.
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 35 and 37 are in sorted positions. We compare 18 and 15 and in the target list of 2 values we put 15 first, followed by 18. We change the order of 40 and 12.

35	37

15	18

12	40

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

15	35	37

12	18	40

After the final merging, the list should look like this –

12	15	18	35	37	40

Q2: Explain how using dynamic programming reduces the complexity of a simple algorithm? Also explain the matrix chain multiplication algorithm in this context. Derive the principle of optimality for multiplication of matrix chain. Compute the optimal multiplications required following matrices.

A1 of order 30 x 35; A2 of order 35 x 15; A3 of order 15 x 5

Ans. Dynamic programming reduces the complexity of a simple algorithm: Dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of (it is hoped) a modest expenditure in storage space. (Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup.) The technique of storing solutions to subproblems instead of recomputing them is called "memoization".

Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. In comparison, a greedy algorithm treats the solution as some sequence of steps and picks the locally optimal choice at each step. Using a greedy algorithm does not always guarantee an optimal solution, whereas a dynamic programming algorithm does, because picking locally optimal choices may result in a bad global solution. One advantage of a greedy algorithm over a dynamic programming algorithm is that the greedy algorithm is often faster and simpler to calculate. Some greedy algorithms (such as Kruskal's or Prim's for minimum spanning trees) are known to lead to the optimal solution.

Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, the goal is to find the most efficient way to multiply these matrices. The problem is not actually to perform the multiplications, but merely to decide the sequence of the matrix multiplications involved.

There are many options because matrix multiplication is associative. In other words, no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices A, B, C, and D, we would have:

((AB)C)D = (A(BC))D = (AB)(CD) = A((BC)D) = A(B(CD)).

However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product, that is the computational complexity. For example, if A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix, then

computing (AB)C needs (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations, while

computing A(BC) needs (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

Clearly the first method is more efficient. With this information, the problem statement can be refined as "how to determine the optimal parenthesization of a product of n matrices?" Checking each possible parenthesization (brute force) would require a run-time that is exponential in the number of matrices, which is very slow and impractical for large n. A quicker solution to this problem can be achieved by breaking up the problem into a set of related subproblems. By solving subproblems once and reusing the solutions, the required run-time can be drastically reduced. This concept is known as dynamic programming.

Principle of optimality for multiplication of matrix chain: Multiplication of matrix chain is an optimization problem. Here the goal is find the most efficient way to multiply the matrices. Matrix multiplication is associative. The problem is to find the way of matrix multiplication which involves less and easy arithmetic operations. If each way of multiplication of matrix is verified, it would take a long time and the process will be very slow when there are many matrices involved.

The solution to this is breaking up the problem into smaller sets of related matrices, so that solutions of sub problems can be reused.

The dynamic programming algorithm is recursive and is as follows :

1. Split the given matrix expression in to two sub expressions.
2. Calculate the minimum cost of multiplying each sub sequence.
3. Add the costs together, to find it for the whole expression.
4. Repeat the same with the sub sequences and find the minimum cost of computation.

The results of each stage are stored so that they can be reused.

Multiplying of two matrices involves the minimum cost as there is only one way to do it. So, the sub sequences can be split, until a 2 are left.

Optimal multiplications required following matrices.

A1 of order 30 x 35; A2 of order 35 x 15; A3 of order 15 x 5

A_1 of order $30 * 35$

A_2 of order $35 * 15$

A_3 of order $15 * 5$

Because, if A_1 is 30×35, A_2 is 35×15, and A_3 is 15×5 matrix, then the number of scalar multiplications required for $(AB)C$ or $(A_1A_2)A_3$ is:

$30 \times 35 \times 15 = 15750$ (for (A_1A_2) which of order 30×15)

$(+) 30 \times 15 \times 5 = 2250$ (for product of (A_1A_2) with A_3)

Total = 18000 Scalar Multiplication

On the other Hand, no. of scalar multiplications for $A_1(A_2A_3)$ is:-

$35 \times 15 \times 5 = 2625$ (for (A_2A_3) which is of order 35×5)

$(+) 30 * 35 \times 5 = 5250$ (for product of with A_2A_3)

Total = 7875 Scalar multiplication.

Q3: Give a divide and conquer based algorithm (Write a pseudo-code) to perform following:

(i) find the is largest element in an array of size n. Derive the running time complexity of your algorithm.

(ii) finding the position of an element in an array of n numbers

Estimate the number of key comparisons made by your algorithms.

Ans. (i) Finding largest value in an array is a classic C array program. This program gives you an insight of iteration, array and conditional operators. We iteratively check each element of an array if it is the largest. See the below program.

Algorithm

Let's first see what should be the step-by-step procedure of this program −

START

Step 1 → Take an array A and define its values

Step 2 → Declare largest as integer

Step 3 → Set 'largest' to 0

Step 4 → Loop for each value of A

Step 5 → If A[n] > largest, Assign A[n] to largest

Step 6 → After loop finishes, Display largest as largest element of array

STOP

Pseudocode

Let's now see the pseudocode of this algorithm –

```
procedure largest_array(A)

  Declare largest as integer
  Set largest to 0
  FOR EACH value in A DO
    IF A[n] is greater than largest THEN
      largest ← A[n]
    ENDIF
  END FOR
  Display largest
end procedure
```

(ii) finding the position of an element in an array of n numbers:

Ans.

ALGORITHM *LargestElementPosition*(*Array*[*l*..*r*])

```
//Input of the function is an array with starting position l and ending position r
//The function returns the position of the largest element in the array
if l = r
    return l
else
    mid ← ⌊(l + r)/2⌋
    position1 ← LargestElementPosition(Array[l..mid])
    position2 ← LargestElementPosition(Array[(mid + 1)..r])
    if Array[position1] ≥ Array[position2]
        return position1
    else
        return position2
```

Q4: Write Quick Sort Algorithm. How is it Different from Randomized Quick Sort Algorithm? Prove that that Worst case of Quick Sort is Best case of Bubble Sort. Apply Quick sort Algorithm to sort the following list: Q U I C K S O R T, in alphabetical order. Find the element whose position is unchanged in the sorted list.

Ans.

Quick Sort Algorithm: In Quick sort algorithm, partitioning of the list is performed using following steps...

Step 1 - Consider the first element of the list as pivot (i.e., Element at first position in the list).

Step 2 - Define two variables i and j. Set i and j to first and last elements of the list respectively.

Step 3 - Increment i until list[i] > pivot then stop.

Step 4 - Decrement j until list[j] < pivot then stop.

Step 5 - If i < j then exchange list[i] and list[j].

Step 6 - Repeat steps 3,4 & 5 until i > j.

Step 7 - Exchange the pivot element with list[j] element.

Different from Randomized Quick Sort Algorithm: A random choice is one that can't be predicted in advance.
An arbitrary choice is one that can be. Let's imagine that you're using quicksort on the sorted sequence 1, 2, 3, 4, 5, 6, ..., n. If you choose the first element as a pivot, then you'll choose 1 as the pivot. All n - 1 other elements then go to the right and nothing goes to the left, and you'll recursively quicksort 2, 3, 4, 5, ..., n.

When you quicksort that range, you'll choose 2 as the pivot. Partitioning the elements then puts nothing on the left and the numbers 3, 4, 5, 6, ..., n on the right, so you'll recursively quicksort 3, 4, 5, 6, ..., n.

More generally, after k steps, you'll choose the number k as a pivot, put the numbers k+1, k+2, ..., n on the right, then recursively quicksort them.

The total work done here ends up being $\Theta(n^2)$, since on the first pass (to partition 2, 3, ..., n around 1) you have to look at n-1 elements, on the second pass (to partition 3, 4, 5, ..., n around 2), you have to look at n-2 elements, etc. This means that the work done is (n-1)+(n-2)+ ... +1 = Θ(n2), quite inefficient!

Now, contrast this with randomized quicksort. In randomized quicksort, you truly choose a random element as your pivot at each step. This means that while you technically could choose the same pivots as in the deterministic case, it's very unlikely (the probability would be roughly 2^{2-n}, which is quite low) that this will happen and trigger the worst-case behavior. You're more likely to choose pivots closer to the center of the array, and when that happens the recursion branches more evenly and thus terminates a lot faster.

The advantage of randomized quicksort is that there's no one input that will always cause it to run in time Θ(n log n) and the runtime is expected to be O(n log n).

Deterministic quicksort algorithms usually have the drawback that either (1) they run in worst-case time O(n log n), but with a high constant factor, or (2) they run in worst-case time $O(n^2)$ and the sort of input that triggers this case is deterministic.

Worst case of Quick Sort is Best case of Bubble Sort: Both Bubble Sort and Quick Sort are well known sorting algorithms in the in the CS community. To be blunt and to the point, Bubble Sort is inefficient compared to Quick Sort (as you might guess form Quick Sort's name).

Bubble Sort, on average, has a time complexity of O(n^2), that is, for 2 elements in the array, there are 4 operations that happen, for 3, 9 operations, for 4, 16 operations, and so on. For each element, the algorithm iterates through the whole array (minus an increasing number from the end of the array, when optimized). This becomes a problem when you have thousands of elements you're trying to sort.

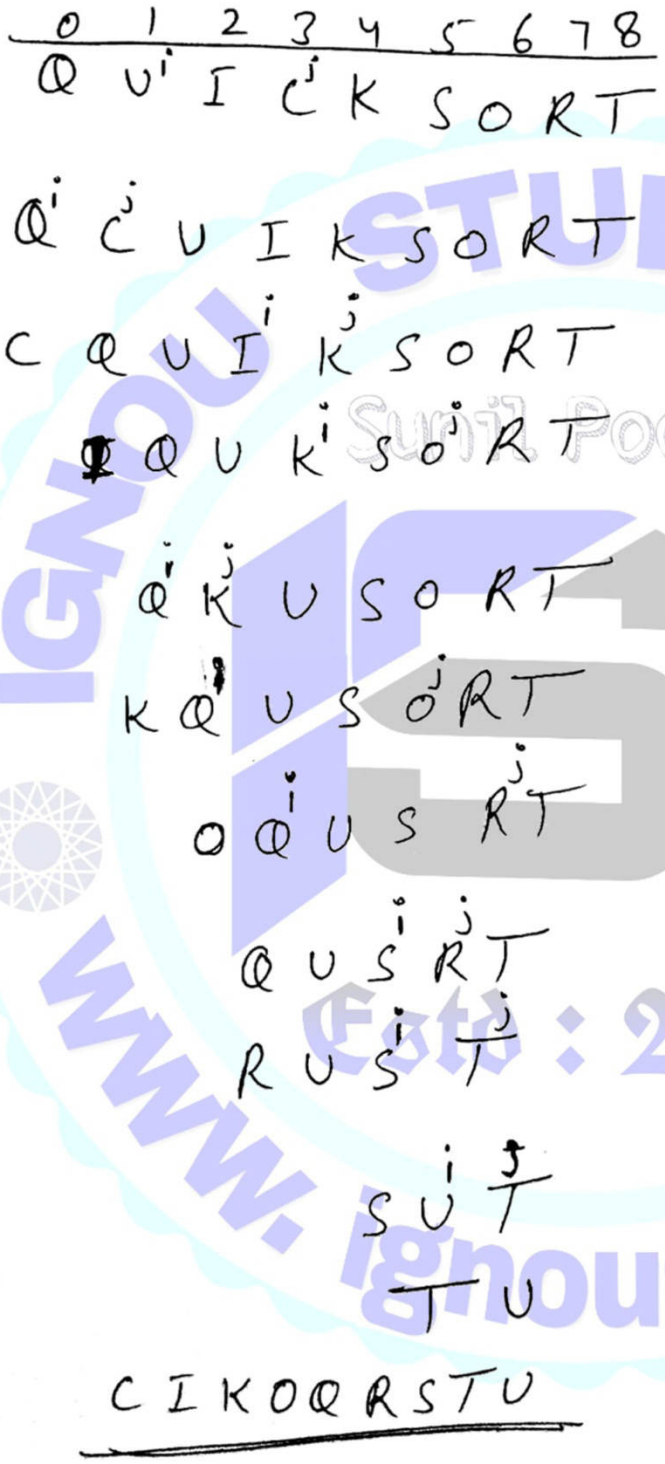
Quick Sort, on the other hand, has an average time complexity of O(n log n), meaning that it is less efficient than a linear algorithm, but each element added to the array is less expensive than the last. This property is invaluable compared to a quadratic algorithm.

Quick Sort recursively (or iteratively, based on implementation) splits the array, and subsequent parts, into left and right arrays, based on a pivot value. Worst case scenario, your pivot point is either the greatest or smallest element in the array, in which case, your time complexity increases to O(n^2), but if you choose a random pivot point, this shouldn't be something you have to worry about too much.
Bubble sort is commonly used by beginners in CS because it is logically easier to understand. After working with Tree data structures, Quick Sort's binary sort method becomes more clear.
Given that average case for Bubble Sort is the worst case for Quick Sort, it is safe to say that Quick Sort is the superior sorting algorithm. For short arrays (under 1,000 elements), the benefits of Quick Sort are minimal, and might be outweighed by it's complexity, if the goal is readability. But for production level projects, Quick Sort is the way to go.Some might argue that Quick Sort's name gives it away.

Quick sort Algorithm to sort the: Q U I C K S O R T, in alphabetical order:

Ans.

```
//Partitions a subarray by Hoare's algorithm, using the first element
                as a pivot
//Input: Subarray of array A[0..n − 1], defined by its left and right
                indices l and r (l < r)
//Output: Partition of A[l..r], with the split position returned as
                this function's value
p ← A[l]
i ← l; j ← r + 1
repeat
repeat i ← i + 1 until A[i] ≥ p repeat j ← j − 1 until A[j ] ≤ p swap(A[i], A[j ])
until i ≥ j
swap(A[i], A[j ]) //undo last swap when i ≥ j swap(A[l], A[j ])
return j
```



Q5: Answer the following:

(i) Explain the meaning of Big O notation with suitable example. How does it differ from Theta and Omega notations. Arrange the following growth rates in increasing order:

$O(3_n)$, $O(n^2)$, O (1), O (n log n)

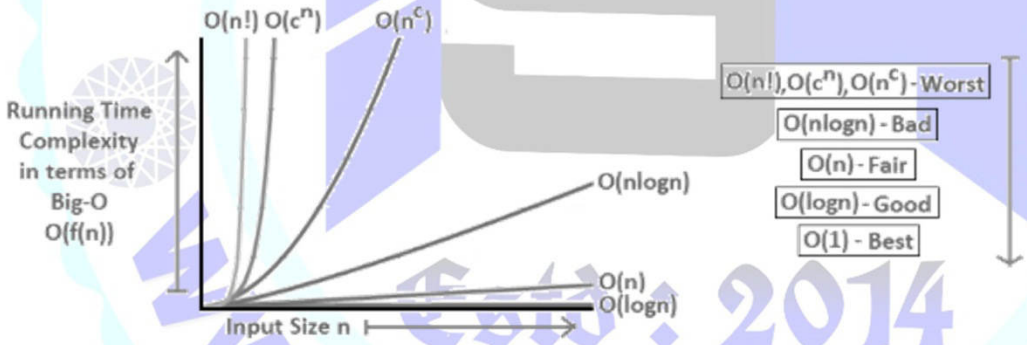
Ans. Big O notation: Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. Big O is a member of a family of notations invented by Paul Bachmann Edmund Landau and others, collectively called Bachmann–Landau notation or asymptotic notation.

In computer science big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows. In analytic number theory big O notation is often used to express a bound on the difference between an arithmetical function and a better understood approximation; a famous example of such a difference is the remainder term in the prime number theorem.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.

The letter O is used because the growth rate of a function is also referred to as the order of the function. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function. Associated with big O notation are several related notations, using the symbols o, Ω, ω, and Θ, to describe other kinds of bounds on asymptotic growth rates.

Big O notation is also used in many other fields to provide similar estimates.



Example:

$T(n) = 3n^3 + 2n + 7 \in \Theta(n^3)$

- If $n \geq 1$, then $T(n) = 3n^3 + 2n + 7 \leq 3n^3 + 2n^3 + 7n^3 = 12n^3$. Hence $T(n) \in O(n^3)$.
- On the other hand, $T(n) = 3n^3 + 2n + 7 > n^3$ for all positive n. Therefore $T(n) \in \Omega(n^3)$.
- And consequently $T(n) \in \Theta(n^3)$.

Differ from Theta and Omega notations:

Big O describes the upper bound of an algorithm. Using our Insertion Sort example, the rate of growth of our algorithm is at most quadratic, or O(n^2).
This is why, for us, as developers and practitioners, we are primarily concerned with Big O.
We want to know just how poorly an algorithm might perform.
To say that the Big O of Insertion Sort is O(n^2) doesn't mean that Insertion Sort will always be O(n^2). The actual runtime will vary based on the input. What we are saying is that, in a worst case scenario, this is the upper bound of the performance of Insertion Sort.

Big Omega describes the lower bound of an algorithm. Using our Insertion Sort example, if the input is already sorted, then the rate of growth of our algorithm is at least linear, or Ω(n).
If only life always handed us sorted arrays. We can also think of this as our best-case scenario.

Big Theta describes the tight bound of an algorithm, it's limit from above and below.
Using our Insertion Sort example, we know that the rate of growth is at most O(n^2) and at least Ω(n).
But Big Theta will change with our inputs. Returning to our Insertion Sort example, in a best case scenario, Big Theta is n, but in a worst case scenario, Big Theta is n^2. To say that Insertion Sort is Θ(n) or Θ(n^2) is incorrect as both of these imply that Insertion Sort will always run at either n or n^2.
So what use is Big Theta?
Big Theta is often used to describe the average, or expected, case for an algorithm. This isn't exactly true, but it's a useful shorthand.
So for Insertion Sort, the average case for Big Theta is n^2.
We can think of Big O, Big Omega, and Big Theta like conditional operators:

- **Big O** is like <=, meaning the rate of growth of an algorithm is less than or equal to a specific value, e.g: f(x) <= O(n^2)
- **Big Omega** is like >=, meaning the rate of growth is greater than or equal to a specified value, e.g: f(x) >= Ω(n).
- **Big Theta** is like ==, meaning the rate of growth is equal to a specified value, e.g: f(x) = Θ(n^2).

Arranging in Increasing Order :- O (3n), O (n2), O (1), O (n log n)

1) O(1) - bounded function
2) O(3n) = O(n) - linear function
3) O(n log n) - when comparing n log n and n^2, note that log n is not bounded. However, any exponent of n grows faster than log n, so for some constant C > 0 and large enough n, log n <= C * n, and so n log n <= C * n^2 for some C > 0 and n > N for some N
4) O(n2)

(ii) Explain the essential idea of Dynamic Programming. How Dynamic programming differs from Divide and conquers approach for solving problems?

Ans. Dynamic Programming is a technique to solving complex problems in programming. At its core, the concept is about remembering and making use of answers to smaller "subproblems" that are already solved.

Dynamic Programming (DP) is a powerful technique that makes it possible to solve certain problems considerably more efficiently than a recursive solution would usually be.

This method essentially trades space for time. Instead of calculating all the solution's different states (taking a lot of time but no space), we take up space for storing solutions of all the sub-problems to save time later. This is called "Memoization."

Dynamic programming differs from Divide and conquers:

Dynamic Programming is similar to Divide and Conquer when it comes to dividing a large problem into sub-problems. But here, each sub-problem is solved only once. There is no recursion. The key in dynamic programming is remembering. That is why we store the result of sub-problems in a table so that we don't have to compute the result of a same sub-problem again and again.

Some algorithms that are solved using Dynamic Programming are Matrix Chain Multiplication, Tower of Hanoi puzzle, etc..

Another difference between Dynamic Programming and Divide and Conquer approach is that -

In Divide and Conquer, the sub-problems are independent of each other while in case of Dynamic Programming, the sub-problems are not independent of each other (Solution of one sub-problem may be required to solve another sub-problem).

Divide and Conquer basically works in three steps.

1**. Divide** - It first divides the problem into small chunks or sub-problems.

2. **Conquer** - It then solve those sub-problems recursively so as to obtain a separate result for each sub-problem.

3. **Combine** - It then combine the results of those sub-problems to arrive at a final result of the main problem.

Some Divide and Conquer algorithms are Merge Sort, Binary Sort, etc.

(iii) Define Knapsack Problem and cite one instance of the problem. Compare Knapsack Problem with fractional Knapsack problem. Give a Greedy algorithm for fractional Knapsack Problem.

Ans. Knapsack Problem and cite one instance of the problem: The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, 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 and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

Compare Knapsack Problem with fractional Knapsack problem:

Knapsack Problem-

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

The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Problem Scenario

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack Problem-

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are n items in the store
- Weight of ith item $w_i>0w_i>0$
- Profit for i^{th} item $pi>0p_i>0$ and
- Capacity of the Knapsack is W

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of i^{th} item.

$$0⩽xi⩽10⩽xi⩽1$$

The i^{th} item contributes the weight $xi.w_ixi.w_i$ to the total weight in the knapsack and profit $x_i.p_ix_i.p_i$ to the total profit.

Hence, the objective of this algorithm is to

$$maximize \sum_{n=1}^{n}(x_i.pi)$$

subject to constraint,

$$\sum_{n=1}^{n}(x_i.wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{n=1}^{n}(x_i.wi) = W$$

In this context, first we need to sort those items according to the value of $\frac{p_i}{w_i}$, so that $\frac{p_i+1}{w_i+1} \leq \frac{p_i}{w_i}$. Here, ***x*** is an array to store the fraction of items.

Fractional Knapsack Problem Using Greedy Method-

Fractional knapsack problem is solved using greedy method in the following steps-

Step-01: For each item, compute its value / weight ratio.
Step-02: Arrange all the items in decreasing order of their value / weight ratio.
Step-03: Start putting the items into the knapsack beginning from the item with the highest ratio. Put as many items as you can into the knapsack.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
  do x[i] = 0
weight = 0
for i = 1 to n
  if weight + w[i] ≤ W then
    x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
    weight = W
    break
return x
```

(iv) Write pseudo code for DFS and calculate its time complexity. Explain briefly how DFS differs from BFS

Ans. pseudo code for DFS:

```
Procedure DFS(G, x)
        G->graph to be traversed
        x->start node
begin
  x.visited = true
  for each v ∈ G.Adj[x]
     if v.visited == false
        DFS(G,v)
end procedure
init() {
  For each x ∈ G
     x.visited = false
  For each x ∈ G
     DFS(G, x)
}
```

Time complexity

The time complexity of DFS if the entire tree is traversed is O(V)O(V) where V is the number of nodes. In the case of a graph, the time complexity is O(V + E)O(V+E) where V is the number of vertexes and E is the number of edges.

DFS differs from BFS:

S.NO	BFS	DFS
1.	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
3.	BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable when there are solutions away from source.
4.	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop.
5.	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.

(v) Write Prim's Algorithm. How Prim's algorithm differs from Kruskal's algorithm. Illustrate with the help of an example

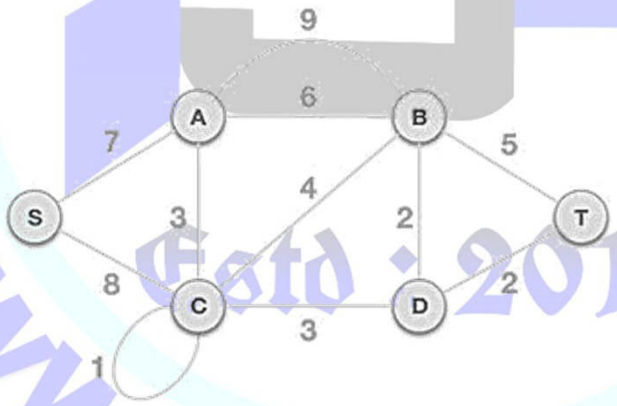
Ans. Prim's Algorithm: Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the shortest path first algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

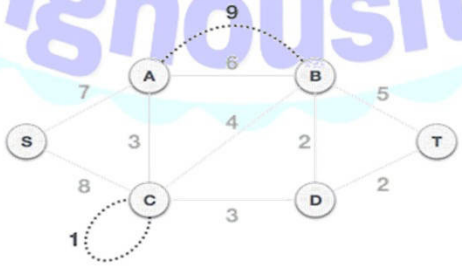
Prim's algorithm differs from Kruskal's algorithm:

PRIM'S ALGORITHM	KRUSKAL'S ALGORITHM
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of O(V^2), V being the number of vertices.	Kruskal's algorithm's time complexity is O(logV), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.

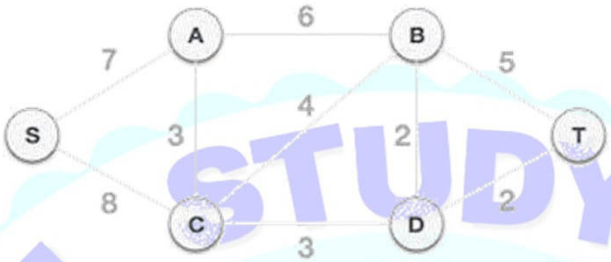
Example : To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

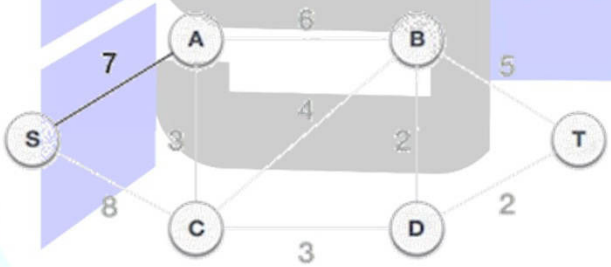


Step 2 - Choose any arbitrary node as root node

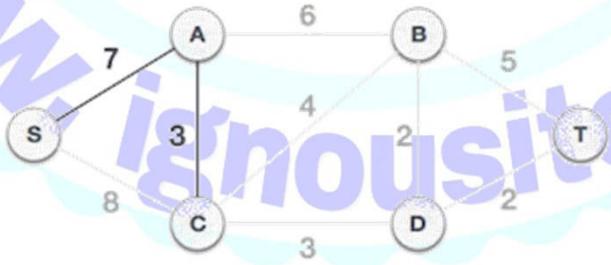
In this case, we choose S node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

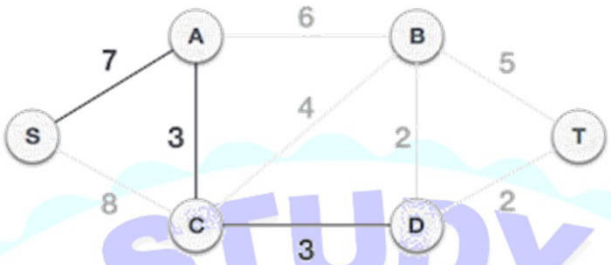
After choosing the root node S, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



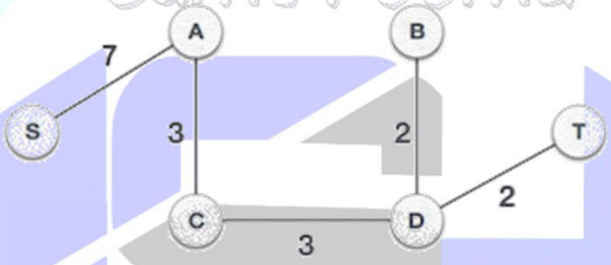
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node D to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

Q6: Answer the following:

(i) Write a context free grammar to generate palindromes of even length Over alphabet Σ= {a, b}.

Ans.

To read abba a from S ⇒ aSa | bSb | aSb | bSa |ε

S ⇒ aSa

S ⇒ abSba

S⇒ abεba

S⇒ abba

To read baab a from S ⇒ aSa | bSb | aSb | bSa |ε

S ⇒ bSb

S ⇒ baSab

S⇒ baεab

S⇒ baab

To read babb a from S ⇒ aSa | bSb | aSb | bSa |ε

S ⇒ bSb

S ⇒ baSbb

S⇒ baεbb

S⇒ babb

To read bbaa a from S ⇒ aSa | bSb | aSb | bSa |ε

S ⇒ bSa

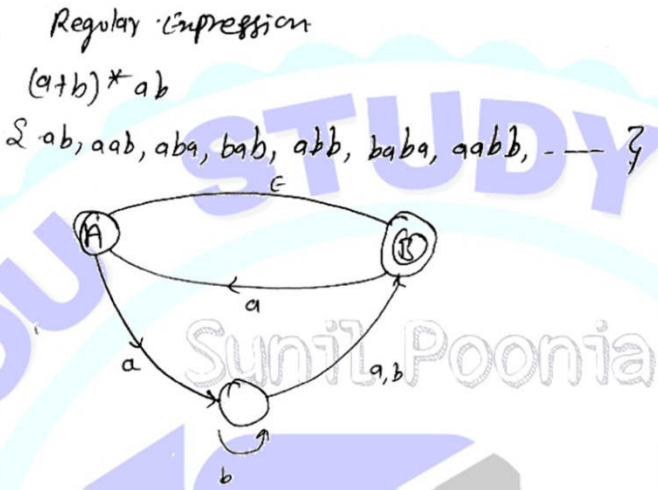
S ⇒ bbSaa

S⇒ bbεaa

S⇒ bbaa

(ii) Write the finite automata corresponding to the regular expression (a+b)*ab.

Ans.



(iii) Explain the Chomsky's Classification of grammars. What is an ambiguous grammar? How do you prove that a given grammar is ambiguous? Explain with an example.

Ans. Chomsky's Classification of grammars:

According to Noam Chomosky, there are four types of grammars – Type 0, Type 1, Type 2, and Type 3. The following table shows how they differ from each other –

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

Type - 0 Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.
The productions can be in the form of α → β where α is a string of terminals and nonterminals with at least one non-terminal and α cannot be null. β is a string of terminals and non-terminals.
Example
S → ACaB

Bc → acB
CB → DB
aD → Db

Type - 1 Grammar

Type-1 grammars generate context-sensitive languages. The productions must be in the form

α A β → α γ β

where A ∈ N (Non-terminal)

and α, β, γ ∈ (T ∪ N)* (Strings of terminals and non-terminals)

The strings α and β may be empty, but γ must be non-empty.

The rule S → ε is allowed if S does not appear on the right side of any rule. The languages generated by these grammars are recognized by a linear bounded automaton.

Example
AB → AbBc
A → bcA
B → b

Type - 2 Grammar

Type-2 grammars generate context-free languages.

The productions must be in the form A → γ

where A ∈ N (Non terminal)

and γ ∈ (T ∪ N)* (String of terminals and non-terminals).

These languages generated by these grammars are be recognized by a non-deterministic pushdown automaton.

Example
S → X a
X → a
X → aX
X → abc
X → ε

Type - 3 Grammar

Type-3 grammars generate regular languages. Type-3 grammars must have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a single non-terminal.

The productions must be in the form X → a or X → aY

where X, Y ∈ N (Non terminal)

and a ∈ T (Terminal)

The rule S → ε is allowed if S does not appear on the right side of any rule.

Example
X → ε
X → a | aY
Y → b

Ambiguous grammar: If a context free grammar G has more than one derivation tree for some string w ∈ L(G), it is called an ambiguous grammar. There exist multiple right-most or left-most derivations for some string generated from that grammar.

Prove that a given grammar is ambiguous:

Check whether the grammar G with production rules –

X → X+X | X*X |X| a

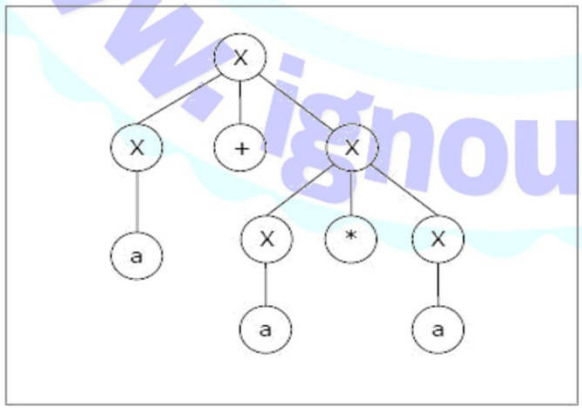
is ambiguous or not.

Solution

Let's find out the derivation tree for the string "a+a*a". It has two leftmost derivations.

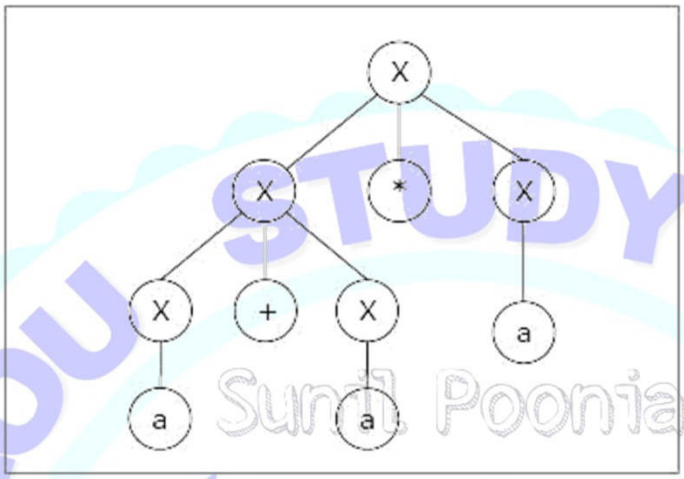
Derivation 1 – X → X+X → a +X → a+ X*X → a+a*X → a+a*a

Parse tree 1 –



Derivation 2 – X → X*X → X+X*X → a+ X*X → a+a*X → a+a*a

Parse tree 2 –



Since there are two parse trees for a single string "a+a*a", the grammar G is ambiguous.

(iv) If L1 and L2, are context free languages then, prove that L1 U L2 is a context free language.

Ans. In formal language theory, a context-free language is a language generated by some context-free grammar. The set of all context-free languages is identical to the set of languages accepted by pushdown automata.

Proof:

If L1 and L2 are context-free languages, then each of them has a context-free grammar; call the grammars G1 and G2.

Our proof requires that the grammars have no non-terminals in common. So we shall subscript all of G1's non-terminals with a 1 and subscript all of G2's non-terminals with a 2. Now, we combine the two grammars into one grammar that will generate the union of the two languages. To do this, we add one new non-terminal, S, and two new productions.

S -> S1 | S2

S is the starting non-terminal for the new union grammar and can be replaced either by the starting non-terminal for G1 or for G2, thereby generating either a string from L1 or from L2. Since the non-terminals of the two original languages are completely different, and once we begin using one of the original grammars, we must complete the derivation using only the rules from that original grammar. Note that there is no need for the alphabets of the two languages to be the same. Therefore, it is proved that if L1 and L2 are context – free languages, then L1 Union L2 is also a context – free language.

(v) Construct a turing machine that copies agiven string over {a, b}. Further find a computation of TM for the string 'aab'.

Ans. A Turing Machine (TM) is a mathematical model which consists of an infinite length tape divided into cells on which input is given. It consists of a head which reads the input tape. A state register stores the state of the Turing machine. After reading an input symbol, it is replaced with another symbol, its internal state is changed, and it moves from one cell to the right or left. If the TM reaches the final state, the input string is accepted, otherwise rejected.

Construct a Turing machine which will accept the set of strings over $\Sigma = \{a, b\}$ beginning with a 'a' and ending with a 'b'. Though this set can be accepted by a FSA, we shall give a TM accepting it.

M = (K, Σ, Γ, δ, q0, F) where
K = {q0, q1, q2, q3} F = {q3}
Σ = {a, b} Γ = {a, b, X, 6b} δ is defined as follows:
δ(q0, a) = (q1, X, R)
δ(q1, a) = (q1, X, R)
δ(q1, b) = (q2, X, R)
δ(q2, a) = (q1, X, R)
δ(q2, b) = (q2, X, R)
δ(q2, 6b) = (q3, halt)

Let us see how the machine accepts *aab*

The computations of aab in M are as follows:

State	String	Stack
q_0	aab	λ
q_1	aab	λ

State	String	Stack
q_0	aab	λ
q_0	ab	A
q_1	ab	A
q_1	b	λ

State	String	Stack
q_0	aab	λ
q_0	ab	A
q_0	b	AA
q_1	b	AA
q_1	b	A
q_1	b	λ

State	String	Stack
q_0	aab	λ
q_0	ab	A
q_0	b	AA
q_2	λ	A
q_2	λ	λ

The computations of abb in M are as follows:

State	String	Stack
q_0	abb	λ
q_1	abb	λ

State	String	Stack
q_0	abb	λ
q_0	bb	A
q_1	bb	A
q_1	bb	λ

State	String	Stack
q_0	abb	λ
q_0	bb	A
q_2	b	λ

The computations of aba in M are as follows:

State	String	Stack
q_0	aba	λ
q_1	aba	λ

State	String	Stack
q_0	aba	λ
q_0	ba	A
q_1	ba	A
q_1	ba	λ

State	String	Stack
q_0	aba	λ
q_0	ba	A
q_2	a	λ

Q7: Differentiate between :

(i) Strassen's Algorithm & Chain Matrix Multiplication algorithm

Ans. Strassen's Algorithm:

Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit.

Strassen's Matrix multiplication can be performed only on square matrices where n is a power of 2. Order of both of the matrices are n × n.

Divide X, Y and Z into four (n/2)×(n/2) matrices as represented below –

$$Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix} \quad X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Using Strassen's Algorithm compute the following –

$$M_1 := (A + C) \times (E + F)$$

$$M_2 := (B + D) \times (G + H)$$

$$M_3 := (A - D) \times (E + H)$$

$$M_4 := A \times (F - H)$$

$$M_5 := (C + D) \times (E)$$

$$M_6 := (A + B) \times (H)$$

$$M_7 := D \times (G - E)$$

Then,

$$I := M_2 + M_3 - M_6 - M_7$$

$$J := M_4 + M_6$$

$$K := M_5 + M_7$$

$$L := M_1 - M_3 - M_4 - M_5$$

Analysis

$$T(n) = \begin{cases} c & if\ n = 1 \\ 7\ x\ T(\frac{n}{2}) + d\ x\ n^2 & otherwise \end{cases}$$ where c and d are constants

Using this recurrence relation, we get $T(n) = O(n^{log7})$

Hence, the complexity of Strassen's matrix multiplication algorithm is $O(n^{log7})$.

Chain Matrix Multiplication algorithm: Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that can be solved using dynamic programming. Given a sequence of matrices, the goal is to find the most efficient way to multiply these matrices. The problem is not actually to perform the multiplications, but merely to decide the sequence of the matrix multiplications involved.

There are many options because matrix multiplication is associative. In other words, no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices A, B, C, and D, we would have:

((AB)C)D = (A(BC))D = (AB)(CD) = A((BC)D) = A(B(CD)).

However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product, that is the computational complexity. For example, if A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix, then

computing (AB)C needs (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations, while

computing A(BC) needs (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

(ii) Context free & Context sensitive Language

Ans: A context-free grammar (CFG) is a grammar where (as you noted) each production has the form A → w, where A is a nonterminal and w is a string of terminals and nonterminals. Informally, a CFG is a grammar where any nonterminal can be expanded out to any of its productions at any point. The language of a grammar is the set of strings of terminals that can be derived from the start symbol.

A context-sensitive grammar (CSG) is a grammar where each production has the form wAx → wyx, where w and x are strings of terminals and nonterminals and y is also a string of terminals. In other words, the productions give rules saying "if you see A in a given context, you may replace A by the string y." It's an unfortunate that these grammars are called "context-sensitive grammars" because it means that "context-free" and "context-sensitive" are not opposites, and it means that there are certain classes of grammars that arguably take a lot of contextual information into account but aren't formally considered to be context-sensitive.

(iii) NP-Complete & NP Hard Problem

Ans.

NP-HARD	NP-COMPLETE
NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) can be reducible into X in polynomial time.	NP-Complete problems can be solved by deterministic algorithm in polynomial time.
To solve this problem, it must be a NP problem.	To solve this problem, it must be both NP and NP-hard problem.
It is not a Decision problem.	It is exclusively Decision problem .
Example: Halting problem, Vertex cover problem, Circuit-satisfiability problem, etc.	Example: Determine whether a graph has a Hamiltonian cycle, Determine whether a Boolean formula is satisfiable or not, etc.

(iv) Greedy technique and Dynamic programming technique

Ans.

FEATURE	GREEDY TECHNIQUE	DYNAMIC PROGRAMMING
Feasibility	In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution.	In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution .
Optimality	In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.	It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.
Recursion	A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.	A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states.
Memorization	It is more efficient in terms of memory as it never look back or revise previous choices	It requires dp table for memorization and it increases it's memory complexity.
Time complexity	Greedy methods are generally faster. For example, Dijkstra's shortest path algorithm takes O(ELogV + VLogV) time.	Dynamic Programming is generally slower. For example, Bellman Ford algorithm takes O(VE) time.
Fashion	The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.	Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions.
Example	Fractional knapsack .	0/1 knapsack problem

(v) Decidable & Un-decidable problems

Ans.

Decidable Problems –

Semi-Decidable problems are those for which a Turing machine halts on the input accepted by it but it can either halt or loop forever on the input which is rejected by the Turing Machine. Such problems are termed as Turing Recognisable problems.

Examples – We will now consider few important Decidable problems:

- Are two regular languages L and M equivalent?
  We can easily check this by using Set Difference operation.
  L-M =Null and M-L =Null.
  Hence (L-M) U (M-L) = Null, then L,M are equivalent.
- Membership of a CFL?
  We can always find whether a string exists in a given CFL by using an algorithm based on dynamic programming.
- Emptiness of a CFL
  By checking the production rules of the CFL we can easily state whether the language generates any strings or not.

Undecidable Problems –

The problems for which we can't construct an algorithm that can answer the problem correctly in finite time are termed as Undecidable Problems. These problems may be partially decidable but they will never be decidable. That is there will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer at all.

We can understand Undecidable Problems intuitively by considering Fermat's Theorem, a popular Undecidable Problem which states that no three positive integers a, b and c for any n>2 can ever satisfy the equation: a^n + b^n = c^n.

If we feed this problem to a Turing machine to find such a solution which gives a contradiction then a Turing Machine might run forever, to find the suitable values of n, a, b and c. But we are always unsure whether a contradiction exists or not and hence we term this problem as an Undecidable Problem.

Examples – These are few important Undecidable Problems:

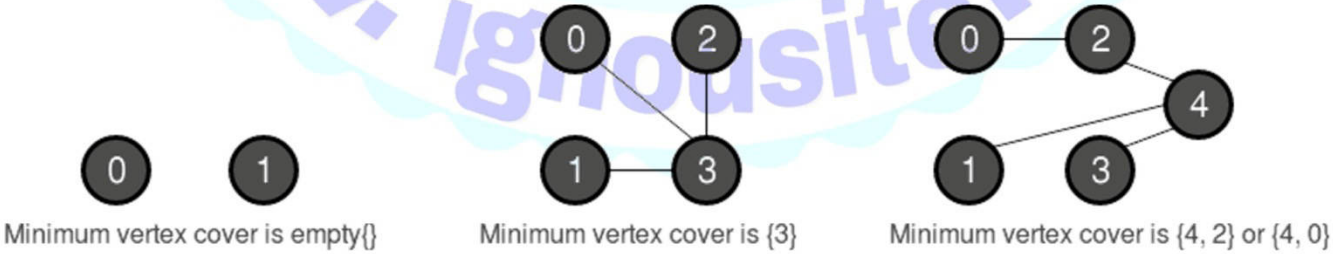
- Whether a CFG generates all the strings or not?
  As a CFG generates infinite strings, we can't ever reach up to the last string and hence it is Undecidable.
- Whether two CFG L and M equal?
  Since we cannot determine all the strings of any CFG, we can predict that two CFG are equal or not.
- Ambiguity of CFG?
  There exist no algorithm which can check whether for the ambiguity of a CFL. We can only check if any particular string of the CFL generates two different parse trees then the CFL is ambiguous.
- Is it possible to convert a given ambiguous CFG into corresponding non-ambiguous CFL?
  It is also an Undecidable Problem as there doesn't exist any algorithm for the conversion of an ambiguous CFL to non-ambiguous CFL.
- Is a language Learning which is a CFL, regular?
  This is an Undecidable Problem as we can not find from the production rules of the CFL whether it is regular or not.

Q8: Write note on each of the following:

(i) Verte Cover Problem

Ans. A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.

Following are some examples.



Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial time solution for this unless P = NP. There are approximate polynomial time algorithms to solve the problem though. Following is a simple approximate algorithm adapted from CLRS book.

(ii) Rice theorem

Ans. Rice theorem states that any non-trivial semantic property of a language which is recognized by a Turing machine is undecidable. A property, P, is the language of all Turing machines that satisfy that property.

Formal Definition

If P is a non-trivial property, and the language holding the property, Lp , is recognized by Turing machine M, then Lp = {<M> | L(M) ∈ P} is undecidable.

Description and Properties

- Property of languages, P, is simply a set of languages. If any language belongs to P (L ∈ P), it is said that L satisfies the property P.
- A property is called to be trivial if either it is not satisfied by any recursively enumerable languages, or if it is satisfied by all recursively enumerable languages.
- A non-trivial property is satisfied by some recursively enumerable languages and are not satisfied by others. Formally speaking, in a non-trivial property, where L ∈ P, both the following properties hold:
    - Property 1 − There exists Turing Machines, M1 and M2 that recognize the same language, i.e. either (<M1>, <M2> ∈ L) or (<M1>,<M2> ∉ L)
    - Property 2 − There exists Turing Machines M1 and M2, where M1 recognizes the language while M2 does not, i.e. <M1> ∈ L and <M2> ∉ L

(iii) Post correspondence problem

Ans. The Post Correspondence Problem (PCP), introduced by Emil Post in 1946, is an undecidable decision problem. The PCP problem over an alphabet ∑ is stated as follows −

Given the following two lists, M and N of non-empty strings over ∑ −

$M = (x_1, x_2, x_3, \ldots\ldots\ldots, x_n)$

$N = (y_1, y_2, y_3, \ldots\ldots\ldots, y_n)$

We can say that there is a Post Correspondence Solution, if for some $i_1, i_2, \ldots\ldots\ldots\ldots i_k$, where $1 \le i_j \le n$, the condition $x_{i1} \ldots\ldots\ldots x_{ik} = y_{i1} \ldots\ldots\ldots y_{ik}$ satisfies.

Example

Find whether the lists

M = (abb, aa, aaa) and N = (bba, aaa, aa)

have a Post Correspondence Solution?

Solution

	x1	x2	x3
M	Abb	aa	aaa
N	Bba	aaa	aa

Here,

$x_2x_1x_3$ = 'aaabbaaa'

and **$y_2y_1y_3$ = 'aaabbaaa'**

We can see that

$x_2x_1x_3 = y_2y_1y_3$

Hence, the solution is **i = 2, j = 1, and k = 3.**

(iv) Halting problem

Ans. The halting problem is a decision problem in computability theory. It asks, given a computer program and an input, will the program terminate or will it run forever? For example, consider the following Python program:

```
x = input()
while x:
   pass
```

It reads the input, and if it's not empty, the program will loop forever. Thus, if the input is empty, the program will terminate and the answer to this specific question is "yes, this program on the empty input will terminate", and if the input isn't empty, the program will loop forever and the answer is "no, this program on this input will not terminate".

Halting problem is perhaps the most well-known problem that has been proven to be undecidable; that is, there is no program that can solve the halting problem for general enough computer programs. It's important to specify what kind of computer programs we're talking about. In the above case, it's a Python program, but in computation theory, people often use Turing machines which are proven to be as strong as "usual computers". In 1936, Alan Turing proved that the halting problem over Turing machines is undecidable using a Turing machine; that is, no Turing machine can decide correctly (terminate and produce the correct answer) for all possible program/input pairs.

The decision problem HH, for Halting problem, is the set of all \{\langle p, x \rangle | \text{program \$p\$ halts on input \$x\$}\}{(p,x)|program p halts on input x}, for an appropriate definition of "program" (usually "Turing machine"), and where \langle \cdot \rangle⟨·⟩ denotes some kind of encoding. A Turing machine AA solves HH if, given any input \langle p, x \rangle⟨p,x⟩, it terminates in an accepting state if \langle p, x \rangle \in H⟨p,x⟩∈H, and terminates in a rejecting state otherwise. Note that it doesn't matter whether pp terminates because it accepts, or because it rejects, or because it gets some kind of error; as long as it terminates and doesn't loop forever, AA should accept it.

(v) K-colourability problem

Ans. The assignment of k colors (or any distinct marks) to the vertices of a graph. 2) The assignment of k colors to the edges of a graph . A coloring is a proper coloring if no two adjacent vertices or edges have the same color.

Vertex coloring is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using
same color. The other graph coloring problems like Edge Coloring (No vertex is incident to two edges of same color) and Face Coloring (Geographical Map Coloring) can be transformed into vertex coloring.
Chromatic Number: The smallest number of colors needed to color a graph G is called its chromatic number. For example, the following can be colored minimum 3 colors.

(v) Independent Set

Independent sets are represented in sets, in which

- there should not be any edges adjacent to each other. There should not be any common vertex between any two edges.
- there should not be any vertices adjacent to each other. There should not be any common edge etween any two vertices.