Solve of MID 20-21

1. a) **(Write answers with *single sentence* only)** 1x5

   i. Mention the characteristic(s) of algorithms that have a complexity of ***lgn*** ?
   ii. When does insertion sort perform better than merge sort?
   iii. Write down the general complexity equation of a divide-conquer algorithm.
   iv. Why does quick sort algorithm just have division (i.e. partition) cost but does not have any merging cost?
   v. Mention one application area where ***lglgn*** algorithm is taken as a significant improvement over ***lgn***?

(i) More efficient and reduce search space

Algorithms with O(log n) complexity typically exhibit **efficient**, divide-and-conquer strategies, commonly seen in binary search, balanced tree structures, and operations that progressively halve or **reduce the search space**.

(ii) for small size input

known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$.

and when n equals one million, $\lg n$ is approximately only 20.) 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 $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

(iii) Divide and Conquer Generic Method

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$\| $$

$$D(n) + C(n)$$

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

- a is the number of subproblems generated in the divide step,
- n/b is the size of each subproblem

(iv) Because of its in-place sorting, sorting and merging done in partitioning process-> pivot element is in its sorted position

```
Quicksort (A,p,r)
if p<r
  q = Partition (A,p,r)
  Quicksort (A,p,q-1)
  Quicksort (A,q+1,r)
```

p - start index

q - last index

} q dropped.

↳ Thats why merging is not necessary

q is in the right position.

$T(n) = T(q-1) + T(n-q) + D(n) + C(n)$

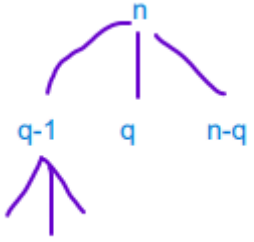
$q-1+n-q = n-1$

Divide Cost = Partition Cost

Conquer Cost = 0 (None)

Quicksort

$$T(n) = T(q-1) + T(n-q) + \Theta(n)$$



complexity: T(n)= T(n/2)+T(n/2)+n = 2*T(n/2)+n ==by solving using master method==>**O(nlogn)**
Quick sort algorithm: L-3.1: How Quick Sort Works | Performance of Quick Sort with E…

Worst case: **T(n)=T(n-1)+ n**==by solving using master method==>**O(n^2)**
0, pivot , n-1
Performance analysis: L-3.2: Performance of Quick Sort | Worst Case Time Complexit…

Divide (Partition): The array is partitioned into two subarrays based on a chosen pivot element. Elements smaller than the *pivot are moved to the left, and elements greater than the pivot are moved to the right*.

Conquer (Recursion): The algorithm recursively applies the same partitioning process to the subarrays created in the previous step.

No Merging Step: Unlike MergeSort, QuickSort does not require a separate merging step to combine the sorted subarrays. The combination or merging of elements is implicitly achieved during the partitioning process.

The partitioning step in QuickSort rearranges the elements in such a way that, after each partitioning, **the pivot element is in its final sorted position**. Therefore, the recursive application of partitioning inherently results in a sorted array.

This lack of a separate merging step is one reason why QuickSort can be more efficient in practice compared to algorithms like MergeSort, especially when it comes to space complexity. QuickSort is an in-place sorting algorithm, meaning it sorts the array without requiring additional memory for a separate output array, which can be advantageous in terms of memory usage.

(v) (found in chatgpt)-not sure about the answer

One application area where an algorithm with $O(\log(\log n))$ complexity might be considered a significant improvement over $O(\log n)$ is in the realm of distributed systems, particularly in the context of consensus algorithms.

In this context, an algorithm with $\mathrm{O}(\log(\log \mathrm{n}))$

$O(\log(\log n))$ complexity could potentially offer a substantial improvement over $\mathrm{O}(\log \mathrm{n})$ because it could lead to fewer communication rounds or steps needed to **reach consensus**. This efficiency gain is especially valuable in distributed systems where reducing the number of communication rounds can enhance overall system performance and responsiveness.

b) Write down the rationale of master method for its all three cases for determining the complexity of recursive algorithms. 6.66

Master method: must watch: The Master Method

Master Theorem for Solving Recurrence | In Hindi | Algorithm design and analysis

Master's Theorem | Solving Recurrences | The Master's Methods | GATECSE | DAA

The master theorem

26-01-23

The master method depends on the following theorem.

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

Its a cookbook for solving recusion

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

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 $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

$$T(n) = 4\,T\left(\frac{n}{2}\right) + n^3$$

$a = 4$

$b = 2$

$f(n) = n^3$

$n^{\log_b a} \Rightarrow n^{\log_2 4} \Rightarrow n^2$

Compare $f(n)$ and $n^{\log_b a}$:

$\Rightarrow f(n) > n^{\log_b a}$ so case 3 is applied. $[f(n) = \Omega(n^{\log_b a + \varepsilon})]$

$\Rightarrow T(n) = \Theta(f(n))$

$= \Theta(n^3)$

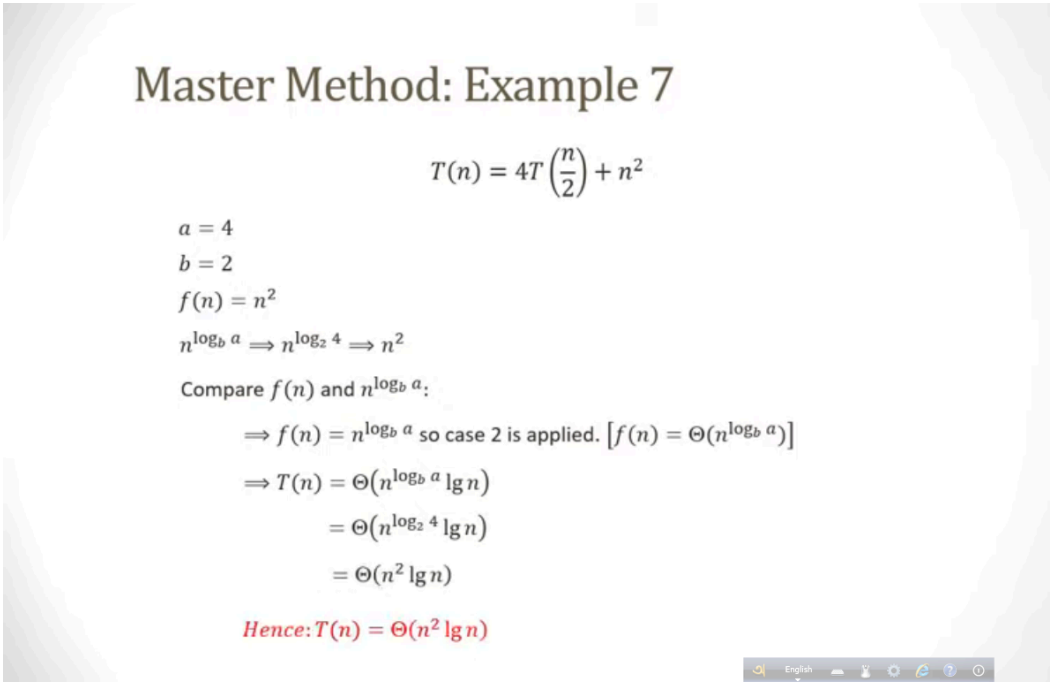
Verify Regularity Condition:

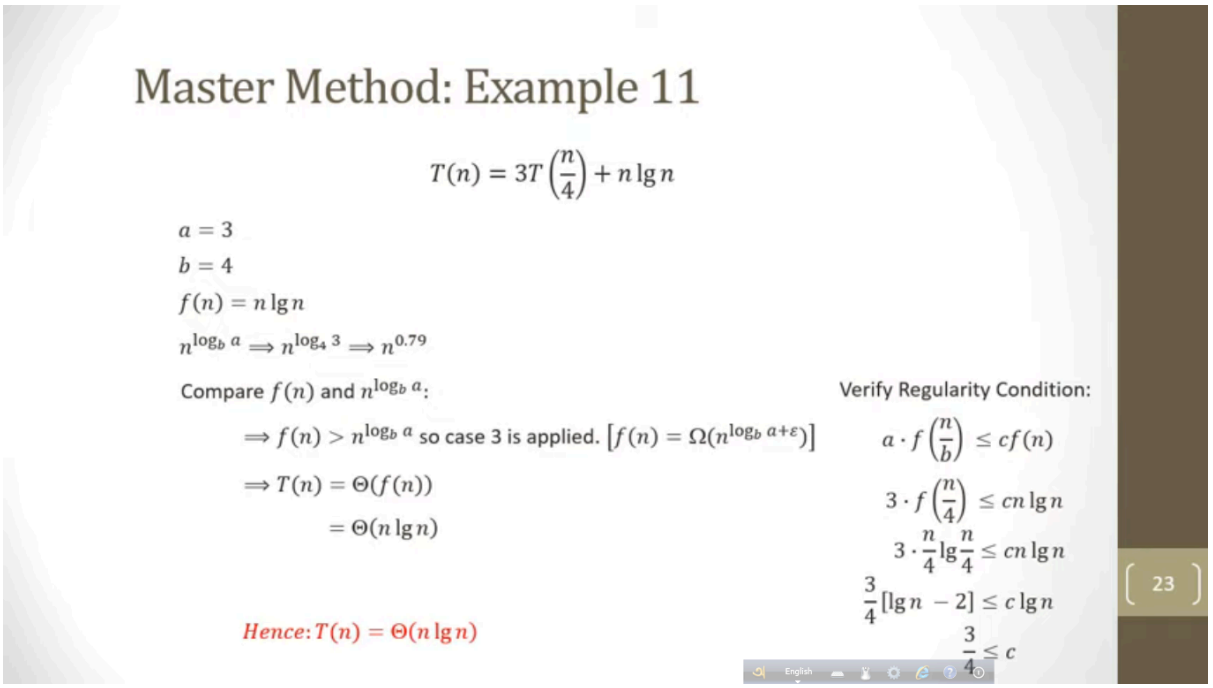
$$a \cdot f\left(\frac{n}{b}\right) \leq cf(n)$$

$$4 \cdot f\left(\frac{n}{2}\right) \leq cn^3$$

$$4 \cdot \frac{n^3}{8} \leq cn^2$$

$$\frac{1}{2} \leq c$$





c) Suppose in a merge-sort of n data points, the recursive division is stopped when any sub array size becomes k. Then each of the (n/k) sub-arrays is sorted by insertion sort. After the sorting of the (n/k) sub arrays, each of size k, the typical merge-sort procedure is followed. Comment on the complexity of the combined algorithm and justify your answer. 5

- How Merge Sort Works?? Full explanation with example
- L-3.5: Insertion Sort | Time Complexity Analysis | Stable Sort | Inplace Sorting

Let's analyze the complexity of the combined algorithm, which involves using merge-sort up to a certain subarray size (k), and then switching to insertion sort for smaller subarrays before proceeding with the regular merge-sort procedure.

1. **Recursive Division:**
   - The recursion stops when the size of each subarray becomes k.
   - The number of recursive levels is $\log_k n$ because we are repeatedly dividing the array into subarrays of size k until reaching individual elements.
2. **Insertion Sort:**
   - After the recursion stops, each of the $\frac{n}{k}$ subarrays of size k is sorted using insertion sort.
   - The time complexity of insertion sort for an array of size k is $O(k^2)$.
3. **Merging:**
   - After the insertion sort step, the algorithm proceeds with the typical merge-sort procedure, which has a time complexity of $O(n \log n)$.

Now, let's analyze the overall time complexity:

$$\text{Total Time Complexity} = \text{Time for Recursive Division} + \text{Time for Insertion Sort} + \text{Time for Merging}$$

1. **Time for Recursive Division:**
   - $O(\log_k n)$ levels, and at each level, there are $\frac{n}{k}$ subarrays.
2. **Time for Insertion Sort:**
   - $\frac{n}{k}$ subarrays, each of size k, results in a time complexity of $O\left(\frac{n}{k} \cdot k^2\right) = O(nk)$.
3. **Time for Merging:**
   - $O(n \log n)$ for the standard merge-sort procedure.

Combining these components, the total time complexity is:

$$O(\log_k n) + O(nk) + O(n \log n)$$

Now, let's consider the dominant term:

- If $k = O(\log n)$, then the term $O(\log_k n)$ dominates, and the overall complexity is $O(\log n)$.
- If k is a constant or grows slower than $\log n$, then the term $O(nk)$ dominates, and the overall complexity is $O(nk)$.

In practice, the choice of k depends on factors such as the constant factors involved in different algorithms, hardware characteristics, and the nature of the input data. It's common to choose k based on empirical testing to find an optimal balance.

2. a) Define NP. Establish the relationships among P, NP and NPC problems. 5
b) What is quasi polynomial time algorithm? Outline the solution for 0-1 Knapsack problem and justify its complexity as quasi polynomial. Mention at least one more algorithm that is also quasi polynomial. 6.66
c) What is *reduction*? How is it helpful in deciding a problem to be NPC? 5

2(a) problem is called NP (nondeterministic polynomial) if its solution can be guessed and verified in polynomial time; nondeterministic means that no particular rule is followed to make the guess. If a problem is NP and all other NP problems are polynomial-time reducible to it, the problem is NP-complete.

The relationships between **P, NP, and NPC (NP-Complete) problems** are foundational to understanding computational complexity:

1. **P (Polynomial Time):** This class contains all decision problems that can be solved by a deterministic Turing machine in polynomial time. In other words, there exists an algorithm that can solve any problem in P efficiently (in terms of the size of the input).

2. **NP (Nondeterministic Polynomial Time):** As mentioned, NP includes problems that, given a candidate solution, can have their solution verified in polynomial time by a deterministic Turing machine. It's important to note that while all problems in P are also in NP (since any problem that can be solved quickly can also have its solution verified quickly), it is not known whether all problems in NP can be solved quickly (i.e., whether P=NP).

3. **NPC (NP-Complete):** This subset of NP comprises problems to which any other NP problem can be reduced in polynomial time and whose solution can be verified in polynomial time. In essence, an NP-Complete problem is as hard as the hardest problems in NP. If a polynomial-time algorithm can be found for any NP-Complete problem, then P=NP, and all problems in NP can also be solved in polynomial time.

The relation between these classes can be visualized as concentric circles, with P inside NP, indicating that every problem that can be solved in polynomial time can also have its solution verified in polynomial time. NP-Complete problems are at the boundary of NP, representing the hardest problems in NP, to which any other NP problem can be reduced. There's also a class called NP-hard, which includes problems at least as hard as the hardest problems in NP, but they don't have to be decision problems and may not belong to NP.

The question of whether P equals NP is one of the seven Millennium Prize Problems, for which a solution has yet to be found. If P=NP, it would mean that every problem whose solution can be quickly verified can also be quickly solved.

2(b)

A quasi-polynomial time algorithm is an algorithm whose running time is bounded by a polynomial in the logarithm of the input size. Specifically, if n is the input size, a quasi-polynomial time algorithm has a time complexity of the form $O(\text{poly}(\log n))$, where poly is a polynomial function.

0-1 Knapsack Problem:

The 0-1 Knapsack problem is a classic optimization problem where you have a set of items, each with a weight and a value, and the goal is to determine the maximum value that can be obtained by selecting a subset of items without exceeding a given weight capacity.

1. **Dynamic Programming Approach:**
   - Create a 2D array dp with dimensions $(n+1) \times (W+1)$, where n is the number of items and W is the weight capacity.
   - Initialize the array with base cases: $dp[i][0] = dp[0][w] = 0$ for all i and w.
   - Fill in the array using the recurrence relation:
     $dp[i][w] = \max(dp[i-1][w], \text{value}[i] + dp[i-1][w - \text{weight}[i]])$
   - The final answer is stored in $dp[n][W]$.
2. **Time Complexity:**
   - The time complexity of this dynamic programming solution is $O(n \cdot W)$, where n is the number of items and W is the weight capacity.
3. **Justification as Quasi-polynomial:**
   - The time complexity is quasi-polynomial because it is $O(\log W)$ when W is exponential in the input size.

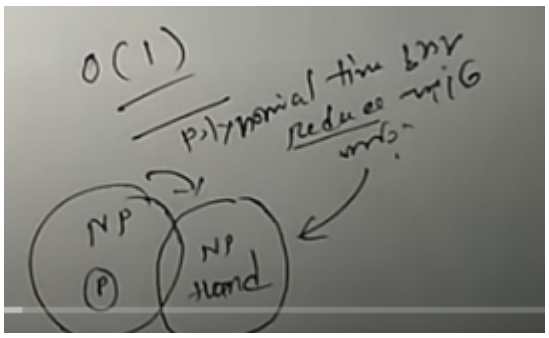
Another Quasi-polynomial Algorithm:

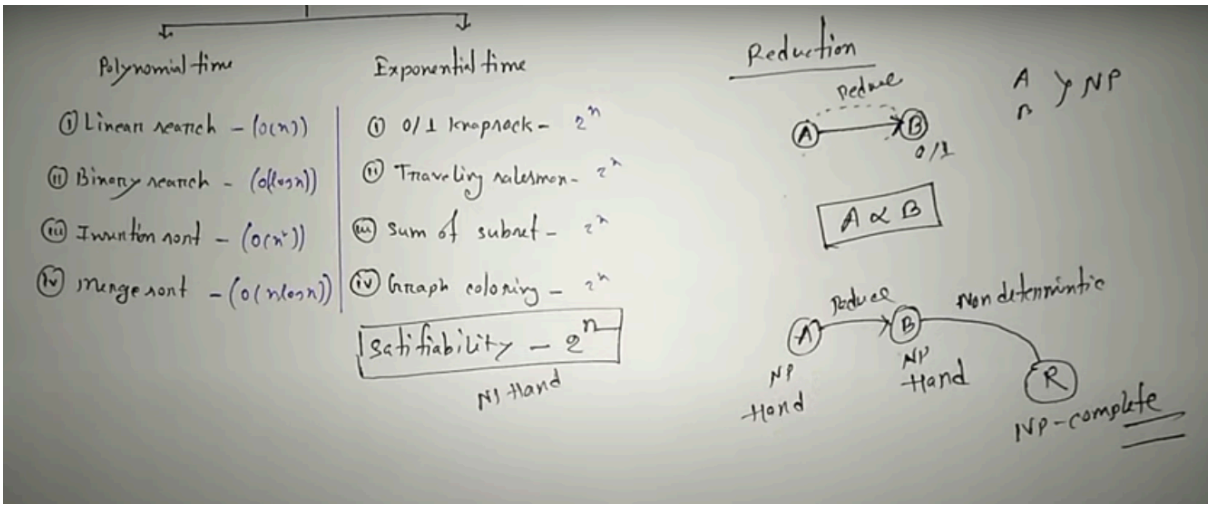
Elliptic Curve Factorization Algorithm:

- The elliptic curve factorization algorithm is another example of a quasi-polynomial time algorithm. It is used for integer factorization.
- The algorithm's complexity is $O((\log N)^{O(1)})$, where N is the number to be factored.

Quasi-polynomial time algorithms strike a balance between the efficiency of polynomial time algorithms and the exponential time complexity of problems that are known to be NP-hard. They provide a more nuanced understanding of the growth rates of certain problems, acknowledging that their complexity is polynomial in the logarithm of the input size.

2(c)





Reduction in the context of computational complexity theory is a method of solving one problem in terms of another. It is a way of transforming instances of one problem into instances of another problem in such a way that the solution to the transformed problem (the second problem) gives us a solution to the original problem.

Reduction is helpful in deciding if a problem is NP-Complete (NPC) by demonstrating that a known NP-Complete problem can be transformed into the problem in question in polynomial time. If this transformation is possible and the problem is in NP, then the problem is also NP-Complete. This shows that the problem is at least as hard as the hardest problems in NP, establishing its NP-Completeness.

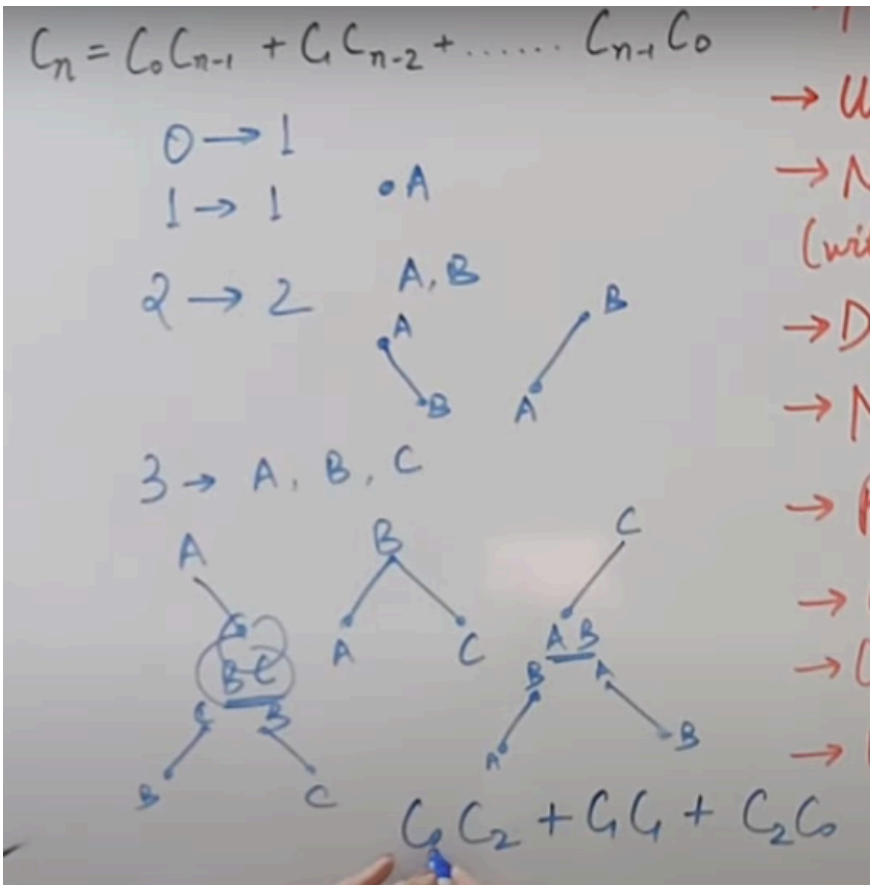
3. a) **(Write answers with *single sentence* only)** 1x5

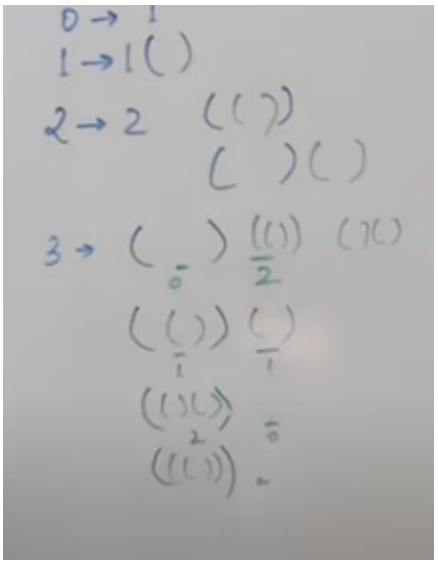
i. What is Catalan number? How is it related to optimal parenthesis problem?
ii. Every problem that has an optimal greedy algorithm should also have a dynamic programming solution- why or how?
iii. What is optimal substructure equation?
iv. Why longest simple path finding problem does not have an optimal substructure property?
v. Can dynamic programming solve multi-objective optimization problem? Justify your answer.

3(a)(i) 7:00:00 to 10:22:00

10 Questions, 1 concept - Catalan Numbers Applications | Dynamic Programming! ✌️💪

The Catalan number is a sequence of natural numbers that occur in various counting problems, representing the number of ways to parenthesize expressions; it is related to the optimal parenthesis problem by providing the **total number of distinct ways to arrange parentheses in an expression.**





(ii)Not necessarily; while some problems with optimal greedy algorithms also have dynamic programming solutions, the existence of a greedy algorithm does not guarantee a dynamic programming solution, as the two paradigms address optimization problems **differently and have distinct criteria for applicability**.

- There are problems for which greedy algorithms work well but DP is not applicable. For instance:
    - Fractional knapsack problem (greedy works, DP is not needed).
    - Activity selection problem (greedy works, DP is not needed).
- Conversely, there are problems where DP is essential, but greedy algorithms fail:
    - Longest increasing subsequence (requires DP).
    - Shortest common supersequence (requires DP).

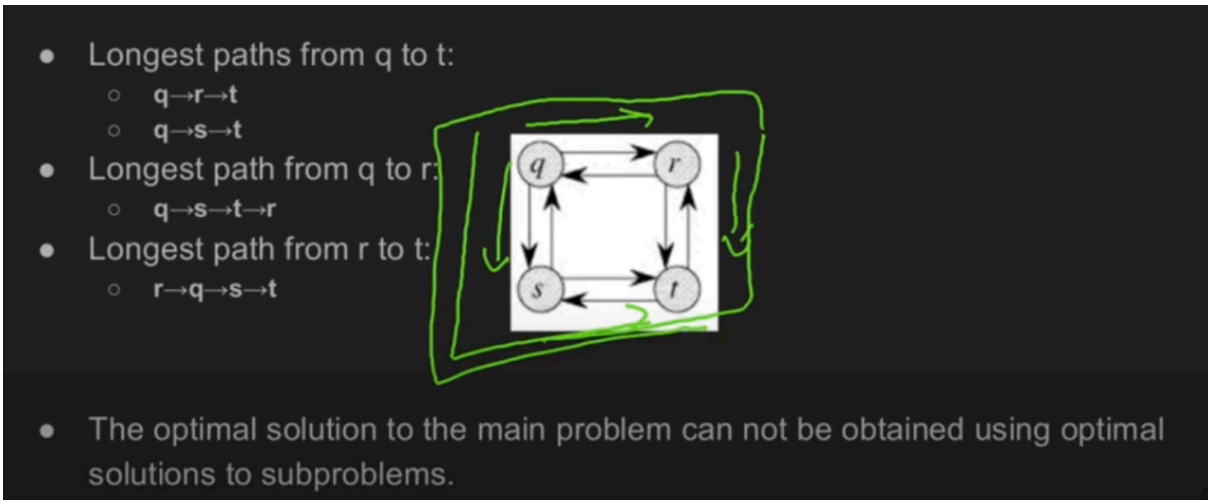
iii) just an equation that relates to the solution of larger problem to a solution of a smaller problem

Introduction to Dynamic Programming: Understanding Optimal Substructures

In mathematical terms, for a given problem P, if we can break it down into subproblems $S_1, S_2, ..., S_n$, and if $OPT(P)$ denotes the optimal solution for problem P, then under the assumption of optimal substructure, $OPT(P)$ can be derived from the optimal solutions of its subproblems, which could be represented generically as:

$$OPT(P) = \text{Some function}(OPT(S_1), OPT(S_2), ..., OPT(S_n))$$

iv) Dynamic Programming | Set 2 (Optimal Substructure Property) | GeeksforGeeks



The longest simple path finding problem lacks optimal substructure because the longest path between two vertices **may not be formed by concatenating longest paths between intermediate vertices**; the length of the longest path is not determined by optimal solutions to smaller subproblems, making it unsuitable for a dynamic programming approach.

v) Dynamic programming is typically designed for solving single-objective optimization problems, where a single criterion or goal is optimized. Multi-objective optimization involves optimizing multiple conflicting objectives simultaneously, and

dynamic programming, as traditionally formulated, is not directly applicable to such problems. Dynamic programming relies on the principle of optimal substructure, which is challenging to extend to scenarios with multiple conflicting objectives.In summary, dynamic programming is not directly suited for multi-objective optimization **due to its reliance on optimal substructure**

b) Write down the optimal substructure equation for DTW after briefly describing the optimization problem the algorithm attempts to solve. Comment on the complexity of the DTW algorithm. Briefly describe two ways to improve the time complexity of the algorithm. 6.66

One of the reasons DTW was initially developed was for speech recognition.

- How DTW (Dynamic Time Warping) algorithm works
- Dynamic time warping 2: Algorithm

Optimal Substructure Equation for DTW:

The DTW algorithm has an optimal substructure characterized by the following recurrence relation:

$$DTW[i,j] = d(i,j) + \min(DTW[i-1,j], DTW[i,j-1], DTW[i-1,j-1])$$

Here, $DTW[i,j]$ represents the accumulated cost up to point (i,j), and $d(i,j)$ is the local cost or distance between elements i and j in the sequences.

Complexity of DTW:

The time complexity of the standard DTW algorithm is $O(n^2)$, where n is the length of the input sequences. This is because the algorithm builds a matrix of size $n \times n$ to store the accumulated costs.

Ways to Improve Time Complexity:

- **Fast DTW:** one of the two signal will be scaled down to smaller one , after matching the points will be interpolated, complexity is O(m)
- **Constant dtw:** In this users some skip distance beyond diagonally,no interpolation needed. Progressive sequence followed only . as matching faster so constant.

c) The time complexity of Viterbi algorithm is S^2O, where S is the number of hidden states and O is the number of observations. Validate the time complexity by describing the problem, assumption(s) and the solution briefly. 5

The Viterbi algorithm utilizes dynamic programming to efficiently compute the most likely sequence of hidden states. It does this by iteratively calculating the probability of the most likely path ending in each state for each observation. The algorithm maintains a dynamic programming table where each entry represents the probability of the most likely path ending in a particular state for a given observation.

The time complexity is analyzed as follows:

- For each observation (O), we iterate through each hidden state (S).
- At each iteration, we calculate the probability of transitioning to the current state from every possible previous state, leading to the S^2 factor in the time complexity.
- Overall, we iterate through O observations, resulting in the O factor in the time complexity.

Therefore, the time complexity of the Viterbi algorithm is O(S^2 * O), capturing the nested iteration over both hidden states and observations. This analysis helps in understanding the computational efficiency of the algorithm, especially when dealing with a large number of hidden states and observations.

Solve of MID 19-20

b) DTW algorithm can be used to solve LCS problem? Explain how.

Dynamic Programming | Set 4 (Longest Common Subsequence) | GeeksforGeeks

3b)ans :

Example: Consider the input strings L_1 with length m and L_2 with length n such that L_1 = "AGGTAB" and L_2 = "GXTXAYB"

1. There can be two cases. The last characters match or the last characters do not match.
2. If the last characters match: Increment the length of LCS by 1 and process L_1[m-1] and L_2[n-1].
3. If the last characters do not match: Find max(L_1[m-1]L_2[n], L_1[m]L_2[n-1]).

LCS	Ø	A
Ø	0	0
G	0	0

LCS	Ø	A	G	G	T	A	B
Ø	0	0	0	0	0	0	0
G	0	0	1				
X	0						
T	0						
X	0						

*** DTW dynamically adjust the alignment by stretching the sequence in the time domain to find the optimal path.
On the other hand LCS is specifically designed for finding the longest sub-sequence between two sequence.
to adapt DTW to solve the LCS problem by constraining the warping path to only allow horizontal and vertical steps (no diagonal steps)

c) How does markovian propertiy helps to derive the optimal substructure equation of viterbi algorithm?

3c)Ans : The Markovian property, which states that the future state depends only on the current state and not on the past states, allows the Viterbi algorithm to efficiently compute the most likely sequence of hidden states in a Hidden Markov Model (HMM). This property enables the derivation of the optimal substructure equation for the Viterbi algorithm because it allows the algorithm to recursively compute the probabilities of reaching each state along the sequence, making it possible to efficiently find the most likely path through the model.

3. a) **(Write answers with *single sentence* only)**
   i. How does dynamic programming save computation of a combinatorial optimization problem?
   ii. Every problem that has an optimal greedy algorithm should also have a dynamic programming solution- why or how?
   iii. Can dynamic programming algorithm be used in a path finding problem where the problem is to find the list of paths within a ratio of the optimal paths?
   iv. There can be some algorithm possible to devise for 3.iii). What will be complexitiy of such an algoirthm in general?
   v. Why do we use bottom up solution for a dynamic programming algorithm rather than using top- down approach?

Ans :

i) Dynamic programming saves computation in combinatorial optimization problems by storing the results of **subproblems and reusing** them when needed, rather than recalculating them. This approach reduces redundant calculations and improves efficiency by avoiding the need to solve the same subproblems multiple times. By breaking down the problem into smaller, overlapping subproblems and solving them iteratively, dynamic programming can significantly reduce the overall computational complexity of the problem.

ii) Greedy algorithms only consider one subproblem at a time and do not compare their results to other options. Dynamic programming guarantees the optimal solution if the problem has overlapping subproblems and optimal substructure.

iii) Yes, dynamic programming can be used in path finding problems where the objective is to find a list of paths within a certain ratio of the optimal paths. One way to approach this is by using dynamic programming to find the optimal path first, and then adjusting the algorithm to return a list of paths that are within the desired ratio of the optimal path.

Here's a basic approach:

1. Use dynamic programming to find the optimal path from the start to the destination.
2. Once the optimal path is found, calculate a threshold distance based on the ratio you want to achieve. For example, if you want paths within 1.5 times the length of the optimal path, you would calculate a threshold distance of 1.5 times the length of the optimal path.

3. Modify the dynamic programming algorithm to continue searching for paths even after finding the optimal path, but stop searching when the length of the path exceeds the threshold distance.
4. Store all paths found within the threshold distance of the optimal path.

iv) Yes, it's possible to devise an algorithm for finding a list of paths within a ratio of the optimal paths in a path finding problem using dynamic programming. One approach could be to modify the dynamic programming algorithm to explore multiple paths simultaneously, keeping track of the optimal path length and exploring other paths until they exceed a certain ratio of the optimal path length.

The **complexity of such an algorithm would depend on the specific details of the problem, such as the size of the graph, the number of paths to explore, and the desired ratio of the paths to the optimal path.** In general, the complexity of this algorithm would likely be higher than that of a traditional dynamic programming algorithm for finding a single optimal path, as it involves exploring multiple paths and potentially backtracking to explore different branches of the search space.

v)

The second approach is the ***bottom-up method***. This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems. We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems. The bottom-up approach often has much better constant factors, since it has less overhead for procedure calls.

Here is the the pseudocode for the top-down CUT-ROD procedure, with memoization added:

1. a) **(Write answers with *single sentence* only)**
   i. Why are asymptotic notations important?
   ii. How does divide and conquer help merge sort algorithm in sorting?
   iii. Why does quick sort algorithm just have division (i.e. partition) cost but does not have any merging cost?
   iv. What do you understand by polynomial time algorithm?
   v. Why are we not interested in exponential time algorithms for solving a problem?

Ans :

i) Asymptotic notations, such as **Big O, Omega, and Theta**, are crucial in analyzing the efficiency and performance of algorithms. They provide a concise way to describe how the **runtime or space requirements** of an algorithm grow as the size of the input increases. This helps in comparing algorithms, predicting their behavior on large inputs, and making informed decisions about which algorithm to use in different scenarios.

The order of growth of the running time of an algorithm, defined in Chapter 2, gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms. Once the input size n becomes large enough, merge sort, with its $\Theta(n \lg n)$ worst-case running time, beats insertion sort, whose worst-case running time is $\Theta(n^2)$. Although we can sometimes determine the exact running time of an algorithm, as we did for insertion sort in Chapter 2, the extra precision is not usually worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the ***asymptotic*** efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

This chapter gives several standard methods for simplifying the asymptotic analysis of algorithms. The next section begins by defining several types of "asymptotic notation," of which we have already seen an example in Θ-notation. We then present several notational conventions used throughout this book, and finally we review the behavior of functions that commonly arise in the analysis of algorithms.

ii)

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 2. Each divide step then yields two subsequences of size exactly $n/2$. In Chapter 4, we shall see that this assumption does not affect the order of growth of the solution to the recurrence.

We reason as follows to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers. 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 subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.

Conquer: We recursively solve two subproblems, 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 subarray takes time $\Theta(n)$, and so $C(n) = \Theta(n)$.

iii) Quick sort algorithm divides the array into smaller subarrays based on a chosen pivot element, which is the partitioning step. This partitioning step has a cost, as it involves rearranging elements around the pivot. However, quick sort does not have a separate merging step because the merging of subarrays is inherently done in place during the partitioning process. Each partitioning step effectively places one element (the pivot) into its correct final position, which implicitly merges the smaller subarrays. Therefore, quick sort avoids the explicit merging step that other sorting algorithms like merge sort have, resulting in a more efficient average-case time complexity of **O(n log n).**

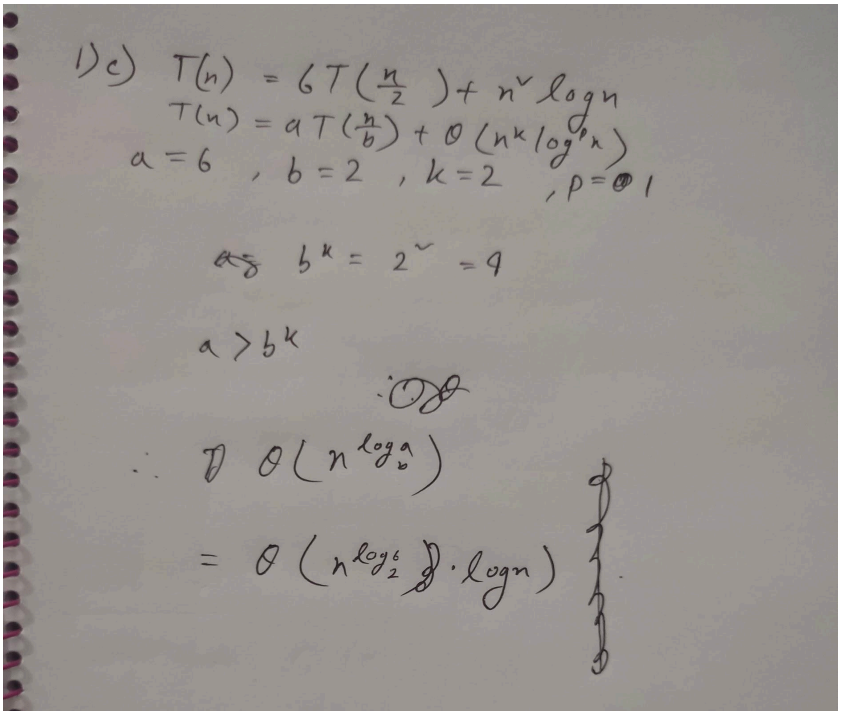
iv) A polynomial-time algorithm is one whose running time grows as a polynomial function of the size of its input.

v) Exponential time algorithms are generally not preferred because they become **impractical or infeasible for large problem sizes**. As the input grows, the time required by these algorithms grows exponentially, leading to extremely long computation times. In contrast, polynomial time algorithms, such as those with quadratic or linear time complexity, offer more manageable and scalable solutions for larger problem instances.

c) Find the solution to the resursion $T(n) = 6T(n/2) + n^2 \lg n$ using master methd.

Doesn't fall any of the 3 cases directly

ans:



b) Write an algorithm to find the median of a data array in linear time.

Ans : To find the median of a data array in linear time, you can use the QuickSelect algorithm, which is an efficient variation of the QuickSort algorithm. Here's how you can implement it:

1. Choose a pivot element from the array.
2. Partition the array into two sub-arrays: one with elements less than the pivot and one with elements greater than the pivot.
3. Recursively select the sub-array that contains the median until the pivot is the median element.

2. NOT IN syllabus

2)(iv) A problem is called NP (nondeterministic polynomial) if its solution can be guessed and verified in polynomial time; nondeterministic means that no particular rule is followed to make the guess. If a problem is NP and all other NP problems are polynomial-time reducible to it, the problem is NP-complete.

2021-22

B.Sc. Engg. CSE 8th Semester — 22 February 2023 (Morning)

ISLAMIC UNIVERSITY OF TECHNOLOGY (IUT)
ORGANISATION OF ISLAMIC COOPERATION (OIC)
Department of Computer Science and Engineering (CSE)

MID SEMESTER EXAMINATION — **SUMMER SEMESTER, 2021-2022**
DURATION: 1 HOUR 30 MINUTES — **FULL MARKS: 50**

CSE 4809: Algorithm Engineering

Programmable calculators are not allowed. Do not write anything on the question paper.
Answer **all 3 (three)** questions. Figures in the right margin indicate full marks of questions whereas corresponding CO and PO are written within parentheses.

1. a) Answer the following questions with single sentence only: — 1×3 (CO2) (PO1)
   i. Write names of two applications that use memory based balanced tree.
   ii. Why are memory-based balanced trees not suitable for disc based search and indexing?
   iii. Mention one application area where $\log_2 \log_2(n)$ algorithm is taken as a significant improvement over $\log_2(n)$.

   b) Given a regular recursion: $T(n) = aT(n/b) + f(n)$
   If $f(n) = O(n^{\log_b a - \epsilon})$, prove that $T(n) = \theta(n^{\log_b a})$. — 7.67 (CO1) (PO1)

   c) With three example instances, show the key insertion mechanism of B+ Tree. Assume $t = 3$. — 6 (CO2) (PO1)

i), ii) not in syllabus

iii) previously came in mid ques 20-21

1)b) **The Master Method**

Master Method: Case 1 → $n^{\log_b a} > f(n)$

$$\left(\frac{n^{\log_b a}}{f(n)}\right) = \Omega(n^{\varepsilon}) \Rightarrow \left(\frac{n^{\log_b a}}{f(n)}\right) \geq cn^{\varepsilon} \Rightarrow f(n) \leq \left(\frac{n^{\log_b a}}{cn^{\varepsilon}}\right) \Rightarrow f(n) \leq cn^{\log_b a - \varepsilon} \Rightarrow f(n) = O(n^{\log_b a - \varepsilon})$$

$$T(n) = n^{\log_b a} + \sum_{k=0}^{\log_b n - 1} a^k f\left(\frac{n}{b^k}\right)$$

Suppose $f(n) = n^{\delta}$, where $\delta < \log_b a$ or $b^{\delta} < a$. Then:

$$T(n) = n^{\log_b a} + \sum_{k=0}^{\log_b n - 1} \left(\frac{a}{b^{\delta}}\right)^k n^{\delta} \Rightarrow n^{\log_b a} + n^{\delta}\left[\frac{\left(\frac{a}{b^{\delta}}\right)^{\log_b n} - 1}{\frac{a}{b^{\delta}} - 1}\right] \Rightarrow n^{\log_b a} + n^{\delta}\left[\frac{\frac{n^{\log_b a}}{n^{\delta}} - 1}{c}\right]$$

$$\Rightarrow n^{\log_b a} + \frac{n^{\log_b a} - n^{\delta}}{c} \Rightarrow \Theta(n^{\log_b a})$$

Case 1: If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

1)c) not in syllabus

2.	a)	What is quasi polynomial time algorithm? Outline the solution for 0-1 Knapsack problem and justify its complexity as quasi polynomial. Mention at least one more algorithm that is also quasi polynomial.	5 (CO1) (PO1)
	b)	How is GPU parallelism used to speed up the training in Deep Learning setup?	6.67 (CO2) (PO1)
	c)	Briefly describe one divide and conquer algorithm that is used in recent research related to you.	5 (CO4) (PO1)

2a) solve in 20-21

2b) GPU parallelism speeds up deep learning training by **allowing simultaneous computations of multiple operation**s, especially **matrix multiplication**s and other linear algebra operations that are fundamental to neural network processing. This is due to the GPU's architecture, which consists of many cores capable of handling numerous calculations concurrently, making them ideal for the batch processing and high computational demands of deep learning algorithms.

2)c) Divide and conquer is a strategy that breaks a problem down into smaller sub-problems, solves each sub-problem independently, and then combines the solutions to solve the original problem. In the context of NLP (Natural Language Processing) and sentiment analysis, divide and conquer can be used in various ways:

1. **Preprocessing**: Text data can be vast and complex. A divide and conquer approach might involve breaking the text into smaller parts (such as sentences or paragraphs), processing these parts independently (like tokenization, stemming, or lemmatization), and then reassembling them for further analysis.

2. **Feature Extraction**: Large texts can be divided into chunks, and linguistic features (like n-grams, part-of-speech tags, or syntactic structures) can be extracted from each chunk in parallel. This can be particularly useful when working with big data sets where computational efficiency is a concern.

3. **Parallel Processing of Data**: In sentiment analysis, the dataset (corpus) can be enormous. By dividing the dataset into smaller, more manageable pieces, the sentiment analysis can be performed concurrently on multiple processors or machines, which can drastically reduce the computation time.

4. **Algorithm Application**: Some sentiment analysis algorithms, particularly those based on machine learning, can be trained on different subsets of the data simultaneously. For instance, an ensemble method like Random Forest inherently uses a divide and conquer approach by building multiple decision trees on different subsets of the data and then aggregating their outputs.

5. **Hierarchical Sentiment Analysis**: For complex documents that express multiple sentiments about different aspects, a divide and conquer approach can be taken to break down the document into aspects and analyze the sentiment for each aspect separately.

6. **Distributed Word Vectors**: Techniques like Word2Vec or GloVe can be used in a divide and conquer manner by segmenting the corpus into smaller parts, training word vectors on these parts in parallel, and then merging the vectors, which can improve performance on very large datasets.

7. **Recursive Deep Models**: Some deep learning models, such as Recursive Neural Tensor Networks, inherently use a divide and conquer strategy to analyze sentence structures, breaking sentences into sub-phrases, analyzing these, and combining them to understand the sentiment of the whole sentence.

8. **Handling Imbalanced Data**: When faced with imbalanced classes in sentiment analysis (e.g., more positive than negative examples), a divide and conquer approach can be used to divide the dataset into more balanced subsets and train classifiers on each subset.

The key benefit of using divide and conquer in sentiment analysis is to improve the scalability and efficiency of the processing, which is crucial given the large volume of data typically involved in NLP tasks. However, it is essential to ensure that the division of the problem does not overlook the context or the nuances of language, which are critical in sentiment analysis.

3. a) Answer the following questions with single sentence only — 1 × 4 (CO2) (PO1)

i. How is Catalan number related to matrix chain multiplication?

ii. "Every problem that has an optimal greedy algorithm should also have a dynamic programming solution"- why?

iii. Why does the longest simple path finding problem not have an optimal substructure property?

iv. Can dynamic programming solve multi-objective optimization problem? Justify your answer.

i) The number of ways to multiply a matrix chain is related to the Catalan numbers, which is a famous function in combinatorics. The Catalan numbers are related to the **number of different binary trees on n nodes**. The solution to the recurrence relation for matrix chain multiplication is the sequence of Catalan numbers. In particular, P (n) = C (n − 1), where C (n) is the nth Catalan number

- When you write the **recurrence relation** for the matrix chain multiplication problem, it turns out to be the **same as that of Catalan numbers**.
- Therefore, Catalan numbers can be used to solve the matrix chain multiplication problem efficiently.

The nth Catalan number can be calculated using the following recurrence relation:

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$

ii) came in previous mid 20-21,19-20 both

iii)Previously came in mid 20-21

iv)Previously came in mid 20-21

b) Write down the optimal substructure equation for DTW after briefly describing the optimization problem the algorithm attempts to solve. Comment on the complexity of the DTW algorithm. — 5.66 (CO2) (PO1)

Previously answered in 20-21 mid ques

c) Given a grid of $(m \times n)$ dimension containing cells filled with reward (positive or negative), the reward of an area will be the sum of rewards of all the cells withing the area. 7 (CO2) (PO1)

If a problem is defined as to find the maximum possible reward from a minimum square:

i. Can you use Dynamic Programming algorithm for the problem defined above? If so, what will be optimal substructure for the problem?

ii. If Dynamic Programming cannot be used in this context or the benefit of using the algorithm is hindered for some reason, point out why?

i) Yes, we can use dynamic programming (DP) for this problem. The problem has an optimal substructure, meaning the solution to the problem can be composed of optimal solutions to its subproblems. For example, knowing the maximum rewards for smaller squares can help in calculating the reward for a larger square.
The optimal substructure is that the maximum reward of a square ending at a cell (i, j) can be calculated from the maximum rewards of squares ending at cells (i-1, j), (i, j-1), and (i-1, j-1). By storing the maximum rewards of squares ending at each cell, you can build up to the solution for the entire grid, ensuring that each subproblem is solved only once, and solutions are reused, which is the essence of dynamic programming.

ii) If dynamic programming cannot be used, it might be due to the ***lack of overlapping subproblems*** or **excessive memory requirements**. For instance, if each cell's reward depends on non-local properties or the rewards are not additive, dynamic programming may not be applicable. Additionally, if memory usage is a concern, the space complexity of maintaining a DP table might be prohibitive, especially for very large grids. In such cases, alternative algorithms that use less memory or handle non-local interactions might be necessary.

From Final questions

a) Construct the dynamic programming algorithm for DTW and Viterbi algorithm. 8

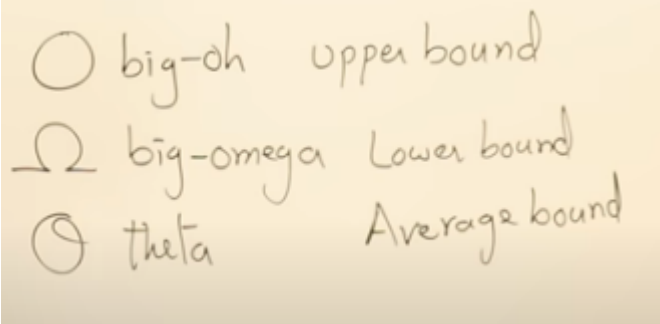
b) Solve the following recurrence using master method and using recursion tree

$$T(n) \leq T(2n/3) + \Theta(1)$$

Syllabus:

Chapters

- 1, 2 = Introduction, C2 has some analysis as well
- **3 = asymptotic**



- **4 =div and conquer very important**
- **7 = Quicksort**

Best case theta(n logn)

Worst case theta(n^2)

- insertion sort

theta(n) best case

Theta (n^2) worst case

-mergesort

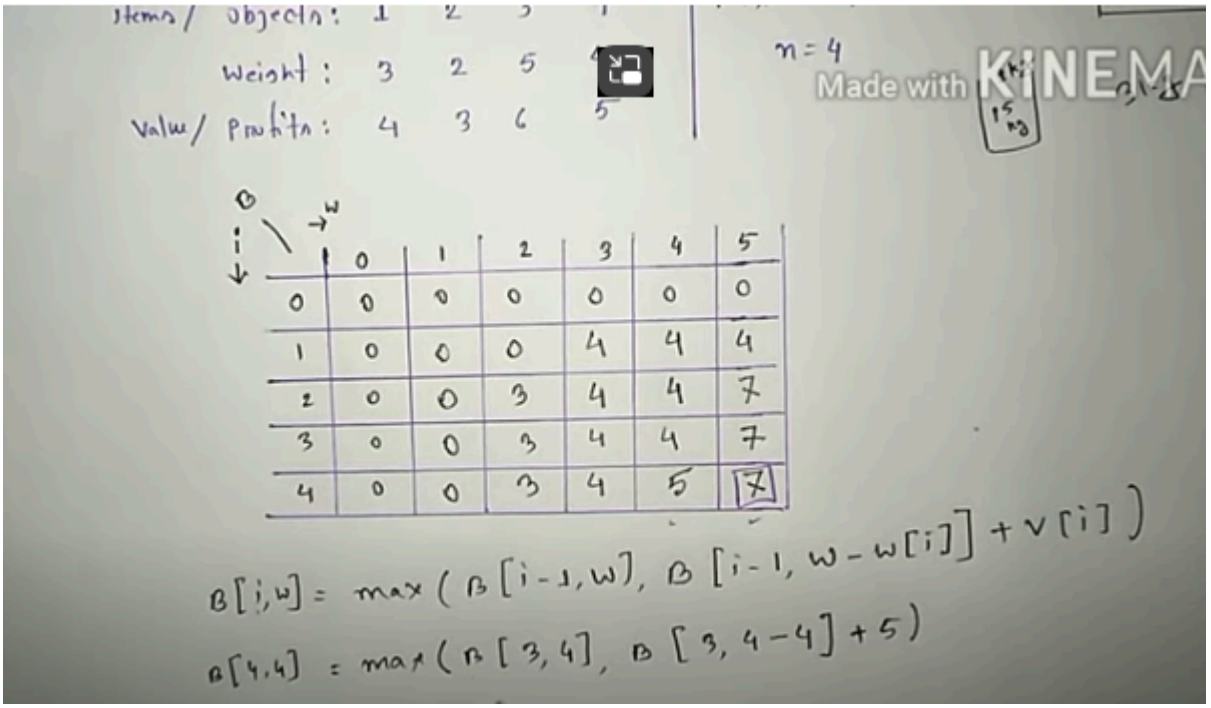
In all case theta(n log n)

- **9 = Medians and Order Statistics**

Same as quick sort

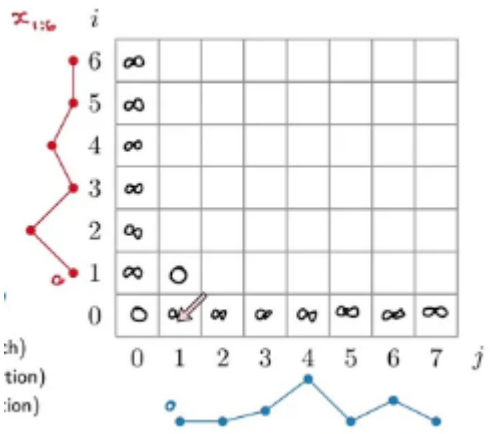
- 15 = dynamic programming very important

- Knapsack from web



- **DTW algo from medium:**

Steps:



-LCS-

Steps:

1 if matches then diagonal +1

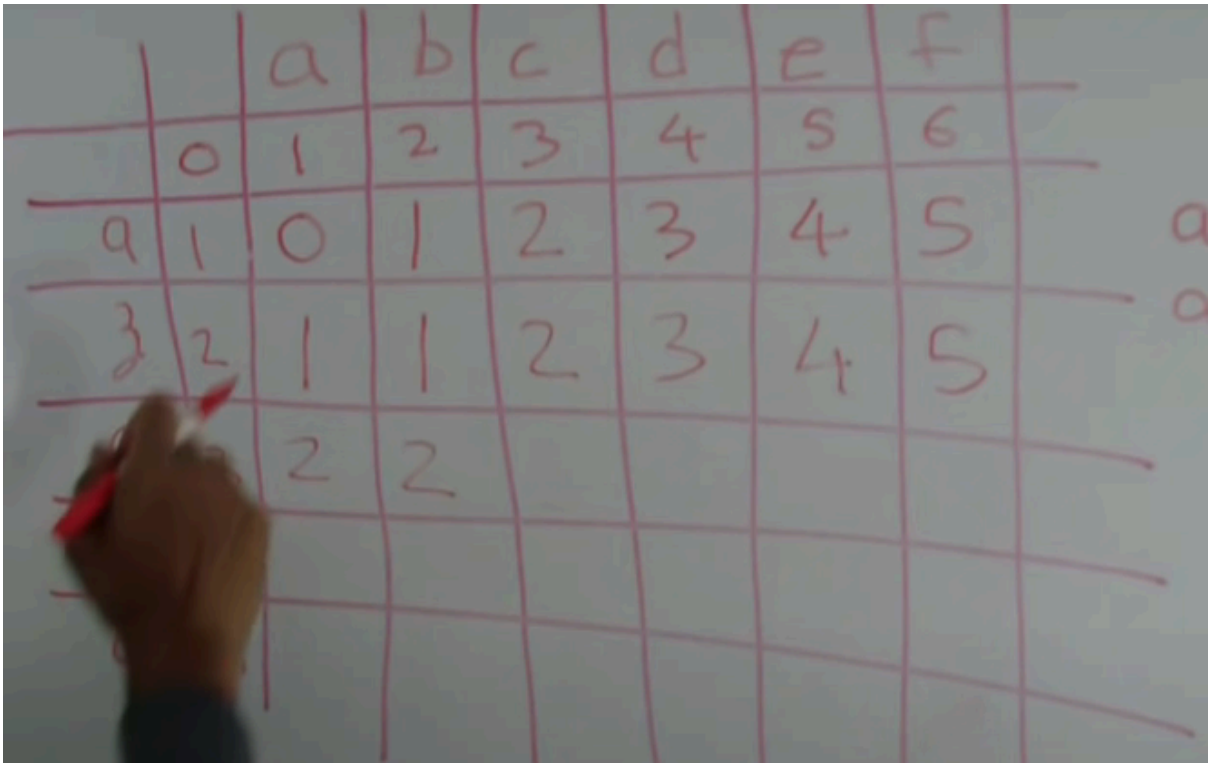
2. If doesn't match then minimum(left, top)

LCS	Ø	A	G	G	T	A	B
Ø	0	0	0	0	0	0	0
G	0	0	1	1	1		
X	0						
T	0						

- **Edit distance algorithm**, (read from online, after LCS)

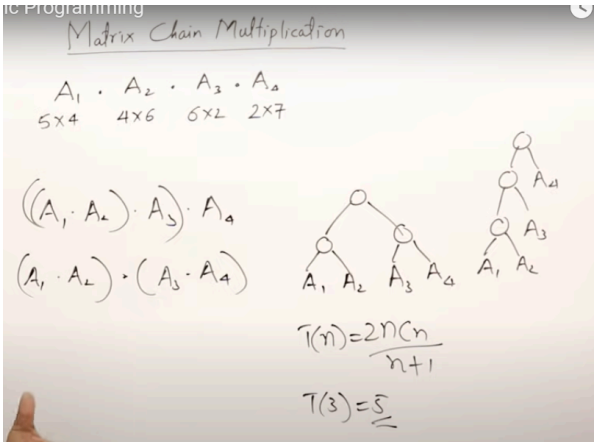
Steps:

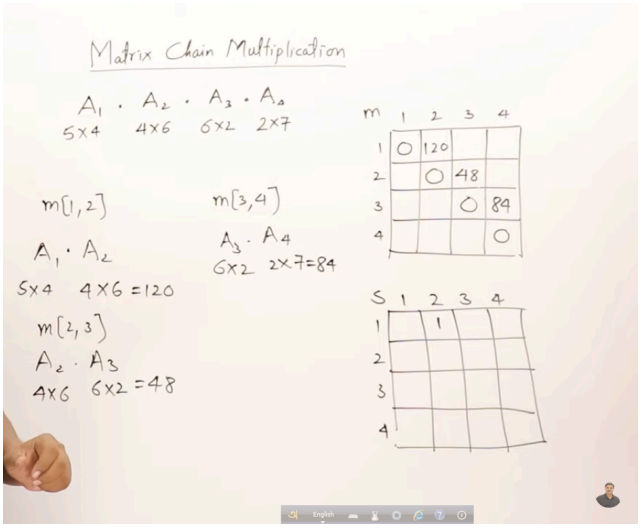
1. If matches write the diagonal one
2. If doesn't matches then write the minimum of the rest 3 +1

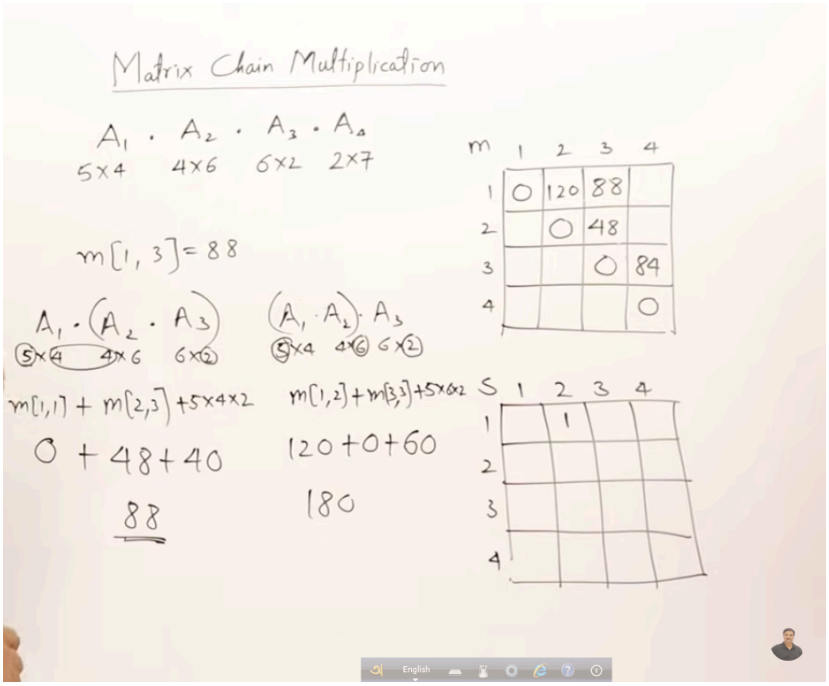


- Exercise not needed
- We need logical answer/explanation, diagram, table etc if needed
- **NP Hard**
- Dijkstra, Floyd, Bellman(just brute force bellman, not the one we know)

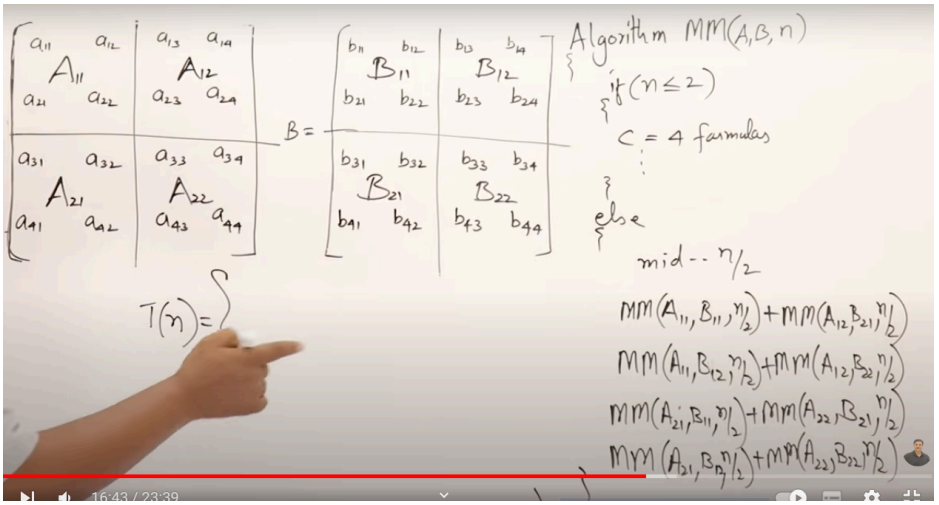
- **Matrix chain**

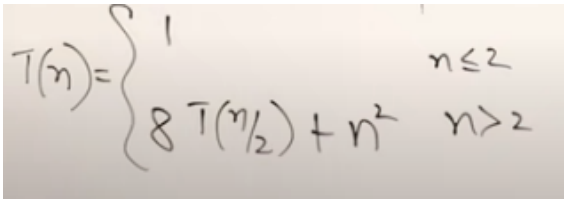






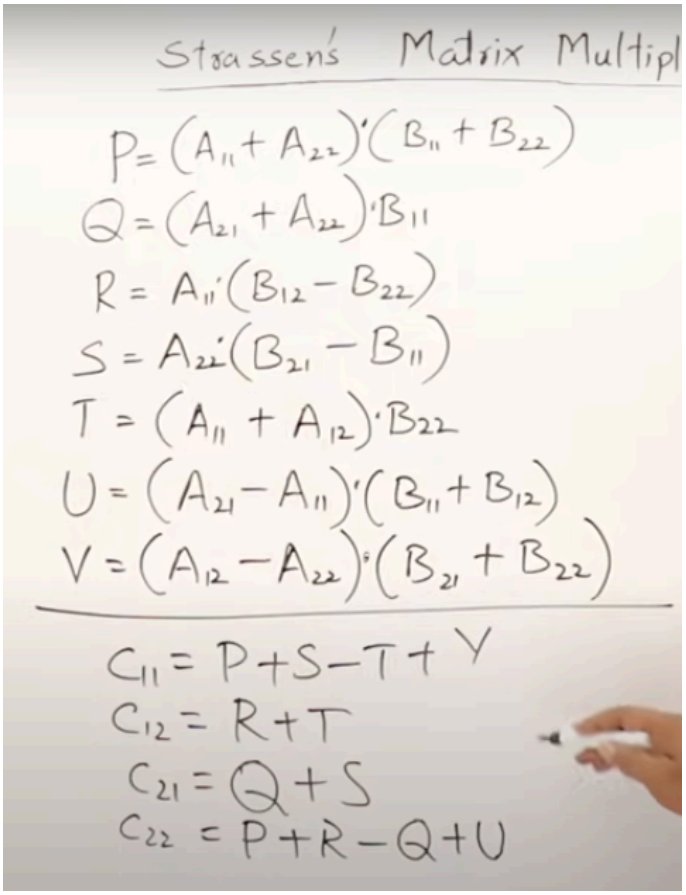
-Matrix multiplication (theta(n^3))

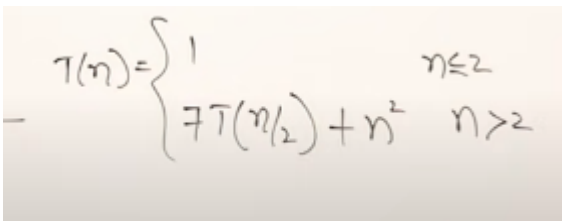


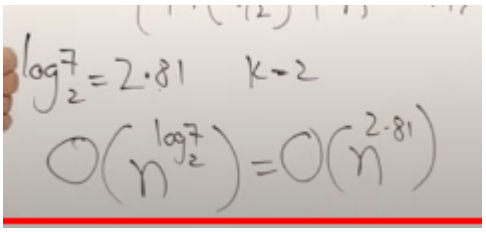


- strassen

Has done 7 multiplications


Strassen's Matrix Multipl
$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
$Q = (A_{21} + A_{22}) \cdot B_{11}$
$R = A_{11} \cdot (B_{12} - B_{22})$
$S = A_{22} \cdot (B_{21} - B_{11})$
$T = (A_{11} + A_{12}) \cdot B_{22}$
$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$
$V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$
$C_{11} = P + S - T + V$
$C_{12} = R + T$
$C_{21} = Q + S$
$C_{22} = P + R - Q + U$


$T(n) = \begin{cases} 1 & n \leq 2 \\ 7T(n/2) + n^2 & n > 2 \end{cases}$


$\log_2^7 = 2.81$ $k = 2$
$O(n^{\log_2^7}) = O(n^{2.81})$