DESIGN AND ANALYSIS OF ALGORITHM

INTRODUCTION TO ALGORITHM ANALYSIS

There are five design strategies or approaches that can help in devising algorithms: Divide and Conquer, Greedy Method, Dynamic Programming, Backtracking, and Branch and Bound. These strategies provide different techniques for solving problems and designing algorithms effectively.

1. **Divide and Conquer:** This approach involves breaking down a complex problem into smaller subproblems, solving each subproblem independently, and then combining the solutions to obtain the final solution. It follows a recursive approach and is often used in algorithms like Merge Sort and Quick Sort.

2. **Greedy Method:** The greedy method involves making locally optimal choices at each step, hoping that the overall solution will be optimal. It focuses on selecting the best choice at each stage without considering future consequences. Greedy algorithms are easy to implement but may not always yield globally optimal solutions. Examples include the Knapsack problem and Dijkstra's algorithm.

3. **Dynamic Programming** :- Dynamic programming is a problem-solving technique that helps us efficiently solve complex problems by breaking them down into smaller, simpler subproblems. It follows a bottom-up approach, where we solve the subproblems first and store their solutions in a table or memoization array. This allows us to avoid redundant computations and reuse the solutions to subproblems as needed.

Here's a step-by-step breakdown of dynamic programming:

- Identifying Overlapping Subproblems: Dynamic programming focuses on problems that can be divided into smaller subproblems. These subproblems often overlap or share common computations.
- Solving Subproblems: We start by solving the smallest subproblems and gradually build up to solve larger ones. The solutions to subproblems are stored in a table or array for efficient retrieval.
- Memoization: To avoid redundant computations, we use memoization. This means that once we solve a subproblem, we store its solution so that we can reuse it whenever that subproblem occurs again in the computation.
- Constructing the Solution: By solving the sub problems and reusing their solutions, we can construct the solution to the original problem efficiently. The final solution is often obtained by combining the solutions to the sub problems.

To illustrate this concept, consider the Fibonacci sequence. Instead of naively calculating Fibonacci numbers from scratch, dynamic programming allows us to store the solutions to smaller Fibonacci numbers as we calculate them. We reuse these stored solutions to compute larger Fibonacci numbers without repeating the same calculations, resulting in significant efficiency improvements.

4. **Backtracking**: Backtracking is a systematic approach for exploring all possible solutions to a problem by incrementally building candidates and abandoning those that do not satisfy the problem constraints. It is often used for problems involving choices or decisions. Backtracking explores the search space and prunes branches that cannot lead to a valid solution. Examples include the N-Queens problem and Sudoku solving.

Imagine you have a puzzle to solve, but you're not sure how to find the right pieces. Backtracking is like trying different puzzle pieces, one by one, until you find the right one that fits.

Here's how it works:

1. Start with the first puzzle piece and try it in a certain spot. If it fits and satisfies the rules of the puzzle, you keep it there.
2. If the puzzle piece doesn't fit or breaks the rules, you backtrack or take it out and try a different piece in that spot.
3. Keep trying different puzzle pieces in each spot, following the rules, until you find the right combination that solves the puzzle.

Backtracking is useful for problems where you have to make choices or decisions to find the solution. It helps you explore all the possible options step by step. If you realize a certain choice doesn't work, you go back and try a different choice instead.

For example, think of a game where you have to place queens on a chessboard in a way that they can't attack each other. Backtracking helps you try different positions for each queen, checking if they are safe from attacks. If you find a position that doesn't work, you go back and try a different position until you find a solution where all the queens are safe.

Similarly, backtracking can be used to solve puzzles like Sudoku. You fill in numbers one by one, checking if they follow the Sudoku rules. If you reach a point where a number doesn't fit or breaks the rules, you go back and try a different number until you solve the entire puzzle.

So, backtracking is like trying different options and making choices until you find the right solution for a problem. It helps you explore and find the right pieces for your puzzle!

HOW TO ANALYZE AN ALGORITHM

When we analyze an algorithm, we are trying to figure out how efficient it is in terms of the time it takes to run and the amount of space it requires. This process is called algorithm analysis or performance analysis.

Algorithm analysis helps us make comparisons between different algorithms and determine which one is more efficient. It allows us to make quantitative judgments about the value and effectiveness of one algorithm compared to another.

Here's a more detailed explanation:

1. Efficiency Estimation: When we analyze an algorithm, we estimate how quickly it can solve a problem and how much memory or space it needs to do so. This estimation helps us understand how efficient the algorithm is in terms of time and space usage.
2. Comparing Algorithms: By analyzing multiple algorithms, we can compare their efficiencies. We can determine which algorithm performs better in terms of time and space requirements. This helps us choose the most suitable algorithm for a specific problem or situation.
3. Efficiency Constraints: Algorithm analysis allows us to predict whether an algorithm will meet any efficiency constraints or limitations that may exist. For example, if a program needs to process a large amount of data within a specific time frame, algorithm analysis helps determine if the algorithm can handle the task efficiently.
4. Quantitative Judgments: Through algorithm analysis, we can make objective and measurable judgments about the value and effectiveness of different algorithms. It provides us with numeric data and metrics that allow us to compare and evaluate algorithms based on their performance.

By analyzing algorithms, we gain insights into their efficiency and performance characteristics. This information helps us make informed decisions about selecting the most appropriate algorithm for a given problem or scenario. It also allows us to predict whether an algorithm will meet the efficiency requirements or constraints of a particular software or system. Ultimately, algorithm analysis enables us to make quantitative assessments and optimize the performance of our algorithms.

HOW TO VALIDATE AN ALGORITHM

When we validate an algorithm, we are making sure that it can produce the correct answer for a given input. This process is crucial to ensure that the algorithm works as intended and provides accurate results.

Here's a more detailed explanation:

1. Algorithm Design: The first step is to design the algorithm, which involves coming up with a set of step-by-step instructions to solve a problem. During this stage, it is essential to carefully plan and define the algorithm's logic and operations.

2. Validating Correctness: Once the algorithm is designed, we need to validate its correctness. This means ensuring that the algorithm can produce the correct answer for all possible legal inputs or scenarios. We want to make sure that the algorithm solves the problem accurately, without any errors or mistakes.

3. Testing and Input Coverage: To validate the algorithm, we perform testing using different inputs and scenarios. We want to cover a wide range of possible input values to ensure that the algorithm handles all cases correctly. By testing with various inputs, we can verify that the algorithm consistently produces the correct results.

4. Program Verification: Once the algorithm's validity has been demonstrated, the next phase begins, known as program verification. This phase involves implementing the algorithm in a programming language to create a program. The program is then thoroughly tested to ensure that it behaves correctly according to the validated algorithm.

5. Error Handling: During validation and program verification, it's important to consider error handling. This involves anticipating and addressing potential errors, edge cases, and exceptional situations that the algorithm may encounter. By incorporating appropriate error handling mechanisms, we can enhance the robustness and reliability of the algorithm.

The validation process ensures that the algorithm is capable of producing the correct results for various inputs and scenarios. By rigorously testing and verifying the algorithm's correctness, we can have confidence in its ability to solve the problem accurately. This ultimately leads to the development of reliable programs that implement the validated algorithm.

ASYMPTOTIC NOTATIONS

Asymptotic notation is a mathematical tool used to describe and analyze the efficiency or performance of an algorithm or function as the input size increases. It helps us understand how the behavior of an algorithm changes when we have more data or a larger problem to solve.

When we talk about the behavior or growth rate of an algorithm, we're interested in how the algorithm's time or space requirements change relative to the size of the input. Asymptotic notation allows us to focus on the most significant factors that affect the algorithm's performance and ignore less significant details.

Three commonly used symbols in asymptotic notation are:

Big O notation (O):- This symbol represents the upper bound or worst-case scenario of an algorithm's time or space complexity. It tells us how the algorithm's performance scales as the input size increases. For example, if we say an algorithm has a time complexity of O(n∧2), it means that the algorithm's running time will not exceed a quadratic growth rate as the input size increases. The "O" notation allows us to describe the maximum amount of resources an algorithm may require.

- Represents the upper bound or worst-case scenario of an algorithm's time or space complexity.
- It describes the maximum rate at which an algorithm's performance grows as the input size increases.
- When we say an algorithm has a time complexity of O(f(n)), it means the algorithm's running time will not exceed a certain multiple of the function f(n) as the input size grows.
- It provides an upper limit on the growth rate of the algorithm.

Big Omega notation (Ω):- This symbol represents the lower bound or best-case scenario of an algorithm's time or space complexity. It gives us an idea of the minimum amount of resources an algorithm will require to solve a problem as the input size increases. For example, if an algorithm has a time complexity of Ω(n), it means the algorithm's running time will grow at least linearly with the input size. The "Ω" notation provides information about the minimum efficiency of an algorithm.

- Represents the lower bound or best-case scenario of an algorithm's time or space complexity.
- It describes the minimum rate at which an algorithm's performance grows as the input size increases.
- When we say an algorithm has a time complexity of Ω(g(n)), it means the algorithm's running time will grow at least as fast as the function g(n) as the input size increases.
- It provides a lower limit on the growth rate of the algorithm.

Big Theta notation (Θ):- This symbol represents both the upper and lower bounds, providing a tight range of possible behaviors for an algorithm's time or space complexity. It describes the average-case scenario when the best and worst cases are similar. For example, if we say an algorithm has a time complexity of Θ(n), it means that the algorithm's running time grows linearly with the input size, neither faster nor slower. The "Θ" notation allows us to describe the exact growth rate or efficiency of an algorithm within a specific range.

- Represents both the upper and lower bounds, providing a tight range of possible growth rates.
- It describes the average-case scenario when the best and worst cases are similar.
- When we say an algorithm has a time complexity of Θ(h(n)), it means the algorithm's running time grows at the same rate as the function h(n) as the input size increases.
- It provides an exact description of the growth rate of the algorithm within a specific range.

BIG – O THEOREMS

The Big-O theorems and properties help us understand the behavior and relationships between different functions in terms of their growth rates. They provide guidelines and rules for analyzing and comparing functions using Big-O notation.

Detailed Explanation:

Theorem 1: The theorem states that a constant, represented by k, is O(1). It means that regardless of the value of k, it is considered to have a constant complexity or growth rate. In Big-O notation, constants can be ignored because they do not affect the scalability or efficiency of an algorithm.

Theorem 2: This theorem focuses on polynomials and states that the growth rate of a polynomial is determined by the term with the highest power of n. For example, if we have a polynomial f(n) of degree d, then it is O(n∧d). The dominant term in the polynomial determines the overall growth rate of the function.

Theorem 3: The theorem states that a constant factor multiplied by a function does not change its Big-O complexity. The constant factor can be ignored when analyzing the growth rate of the function. For example, if we have a function f(n) = 7n∧4 + 3n∧2 + 5n + 1000, it is O(n∧4). The constant factor 7 can be disregarded.

Theorem 4 (Transitivity): This theorem states that if function f(n) is O(g(n)) and g(n) is O(h(n)), then f(n) is O(h(n)). It means that if one function has a certain growth rate and another function has a higher growth rate, the first function is also guaranteed to have a growth rate no higher than the second function.

This theorem states that if function f(n) is O(g(n)) and g(n) is O(h(n)), then f(n) is O(h(n)). In simpler terms, it means that if one function grows no faster than another function, and that other function grows no faster than a third function, then the first function also grows no faster than the third function.

To understand this theorem, let's break it down further:

1. **Function f(n) is O(g(n)):** This means that the growth rate of function f(n) is no greater than the growth rate of function g(n). It indicates that as the input size increases, the resources (time or space) required by f(n) will not exceed the resources required by g(n).

2. **Function g(n) is O(h(n)):** This means that the growth rate of function g(n) is no greater than the growth rate of function h(n). It indicates that as the input size increases, the resources required by g(n) will not exceed the resources required by h(n).

The conclusion from these two statements is that:

3. **Function f(n) is O(h(n)):** Combining the information from statements 1 and 2, we can say that the growth rate of function f(n) is no greater than the growth rate of function h(n). It implies that as the input size increases, the resources required by f(n) will not exceed the resources required by h(n).

This theorem allows us to chain together multiple comparisons of functions' growth rates using Big-O notation. If we have evidence that one function grows no faster than another, and that other function grows no faster than a third function, we can confidently say that the first function also grows no faster than the third function.

The transitivity property is helpful when comparing and analyzing the efficiency of algorithms. It allows us to make logical deductions about the scalability and resource requirements of algorithms based on their growth rates.

Theorem 5: The theorem states that for any base b, logarithm base b of n (logb(n)) is O(log(n)). It means that logarithmic functions with different bases grow at the same rate. The base of the logarithm does not significantly affect the growth rate.

Theorem 6: This theorem provides a hierarchy of growth rates for various types of functions. It states that each function in the list is O of its successor. Functions such as constant (k), logarithmic (logn), linear (n), linearithmic (nlogn), quadratic (n∧2), exponential (2∧n, 3∧n), larger constants to the nth power, factorial (n!), and n to higher powers all have different growth rates. As we move up the list, the growth rate increases.

PROPERTIES OF BIG-O

Higher powers grow faster: When we compare functions with different powers of n, the one with a higher power grows faster. For example, if we have a function n∧r, where r is a positive number, and another function n∧s, where s is a positive number and greater than r, then the function n∧s grows faster as the input size increases.

Example: Consider two functions, f(n) = n∧2 and g(n) = n∧3. As the input size increases, the function g(n) grows faster than f(n) because it has a higher power of n. For instance, when n = 10, f(n) = 100 and g(n) = 1000. Thus, g(n) grows faster than f(n).

Fastest growing term dominates a sum: When we have a sum of functions, the one with the fastest growing term determines the overall growth rate. For example, if we have a sum f(n) + g(n), and the growth rate of g(n) is faster than that of f(n), then the sum is dominated by g(n) in terms of its growth rate.

Example: Let's say we have two functions, f(n) = n∧2 and g(n) = n∧3 + 1000. In this case, the fastest growing term in g(n) is n∧3, which dominates the overall growth rate. Even though f(n) has a lower power term and an additional constant, as the input size increases, the term with the highest power, n∧3, will dominate the sum. Thus, the overall growth rate is determined by the fastest growing term.

Exponential functions grow faster than powers: Exponential functions, which are represented as b∧n, where b is greater than 1, grow faster than functions with powers of n. This means that as the input size increases, the growth rate of an exponential function outpaces the growth rate of a function with a power of n.

Example: Consider two functions, f(n) = 2∧n and g(n) = n∧3. As the input size increases, the function f(n) with an exponential growth rate grows significantly faster than g(n) with a polynomial growth rate. For example, when n = 10, f(n) = 1024 and g(n) = 1000. The exponential growth of f(n) surpasses the polynomial growth of g(n) as the input size increases

Logarithms grow more slowly than powers: Logarithmic functions, represented as log_b(n), where b is greater than 1, grow more slowly than functions with powers of n. As the input size increases, the growth rate of logarithmic functions is lower compared to functions with powers of n.

Example: Let's compare two functions, f(n) = log_2(n) and g(n) = n∧2. As the input size increases, the function f(n) with a logarithmic growth rate grows much more slowly than g(n) with a quadratic growth rate. For instance, when n = 100, f(n) = 6.64 (approximately) and g(n) = 10,000. The logarithmic growth of f(n) is significantly lower compared to the quadratic growth of g(n) as the input size increases.

These properties help us understand and compare the growth rates of functions using Big-O notation. They provide insights into how different types of functions behave as the input size increases. By analyzing and comparing functions based on their growth rates, we can make predictions about the efficiency and scalability of algorithms.

FORMAL DEFINITION OF BIG-O NOTATION

Big O notation is a way to compare and analyze algorithms based on their efficiency and performance. It helps us understand how the runtime or resource requirements of an algorithm change as the input size increases. Big O notation focuses on the largest or most significant term in the algorithm's expression, as it becomes the dominant factor for larger input sizes.

Detailed Explanation: Big O notation provides a standardized way to express the complexity or growth rate of an algorithm. It allows us to make comparisons and predictions about how the algorithm will perform for larger values of input.

Formal Definition: In Big O notation, we say that f(n) = O(g(n)) if there exist positive constants "c" and "k" such that for all values of n greater than or equal to "k", the function f(n) is less than or equal to "c" multiplied by the function g(n). This definition helps us establish an upper bound on the growth rate of f(n) relative to g(n).

Examples: The following points are facts that you can use for Big-Oh problems:

- $1 <= n$ for all $n >= 1$
- $n <= n^2$ for all $n >= 1$
- $2^n <= n!$ for all $n >= 4$
- $\log_2 n <= n$ for all $n >= 2$
- $n <= n\log_2 n$ for all $n >= 2$

Describing the above

- For any value of n greater than or equal to 1, we can say that 1 is less than or equal to n. This is true for all positive integers.

- Similarly, for any value of n greater than or equal to 1, we can say that n is less than or equal to n∧2. This inequality holds for all positive integers.
- As the value of n increases, the growth rate of 2∧n becomes larger than the growth rate of n! (n factorial). This is true for all values of n greater than or equal to 4.
- For values of n greater than or equal to 2, we can observe that the logarithm base 2 of n is less than or equal to n. The logarithmic growth rate is smaller compared to linear growth (n).
- Additionally, for values of n greater than or equal to 2, we can see that n is less than or equal to n log base 2 of n. The linear growth rate is smaller compared to linearithmic growth (n log n).

Example :- f(n)=10n+5 and g(n)=n. Show that f(n) is O(g(n)).

We want to show that the function f(n) = 10n + 5 is "smaller" or "equal to" the function g(n) = n when n gets bigger. We'll do this by finding some numbers that help us compare the two functions.

Step 1: To show that f(n) is O(g(n)), we need to find some special numbers called constants c and k. These numbers will help us prove that for any value of n that is greater than or equal to k, the function f(n) is less than or equal to c multiplied by g(n).

Step 2: Let's try a value for c. We'll choose c = 15. Now we need to check if 10n + 5 is less than or equal to 15n for all values of n that are greater than or equal to some number k.

Step 3: To find the value of k, we need to solve the inequality 10n + 5 ≤ 15n. This inequality helps us determine the smallest value of n for which the inequality is true. By solving the inequality, we can find the value of k.

10n + 5 ≤ 15n Subtracting 10n from both sides: 5 ≤ 5n Dividing both sides by 5: 1 ≤ n

From this, we can see that for any value of n that is greater than or equal to 1, the inequality 10n + 5 ≤ 15n holds true.

So, in this case, we have found that k = 1. This means that for all values of n that are 1 or larger (n ≥ 1), the function f(n) = 10n + 5 is less than or equal to 15 multiplied by g(n) = n.

Step 4: Since the inequality is true for n = 1, we can say that for all values of n that are 1 or larger (n >= 1), the function f(n) = 10n + 5 is less than or equal to 15 multiplied by g(n).

Conclusion: So we have found our constants: c = 15 and k = 1. This means that for all values of n that are 1 or larger, the function f(n) = 10n + 5 is "smaller" or "equal to" 15 multiplied by g(n) = n.

In simpler terms, we have shown that as n gets bigger, the function f(n) = 10n + 5 is not much larger than g(n) = n. This comparison helps us understand how the two functions grow and how their values relate to each other.

THE CONSTANTS K AND C

The constants "k" and "c" are used to establish the relationship between two functions when analyzing their growth rates using Big O notation. Here's what each constant represents:

"k": The constant "k" represents the starting point or threshold value from which the relationship between the functions holds true. It signifies the input size from which we can say that one function's growth rate is no greater than the other.

"c": The constant "c" represents a scaling factor or a value that helps determine the upper bound of the growth rate. It allows us to establish an upper limit on the resources (time or space) required by an algorithm.

When using Big O notation to compare functions, we want to find values for "k" and "c" that satisfy the condition f(n) ≤ c * g(n) for all n ≥ k. In other words, for any input size n greater than or equal to "k", the function f(n) will be less than or equal to "c" multiplied by the function g(n).

By finding suitable values for "k" and "c" and proving this inequality, we can determine the relationship between the growth rates of the two functions and analyze their efficiency or scalability. These constants help us quantify and compare the performance of algorithms or functions based on their growth rates.

HOW TO FIND THE VALUE OF C ?

To find the value of the constant "c" in Big O notation, you typically need to analyze the behavior of the functions and determine an upper bound on the growth rate. Here are a few general steps to help find the value of "c":

Identify the relevant functions: Determine which functions you want to compare and analyze in terms of their growth rates.

Write down the inequality: Write the inequality $f(n) \leq c * g(n)$ for all $n \geq k$, where f(n) and g(n) represent the functions you are comparing, and k is a threshold value.

Simplify the inequality: Simplify the inequality to isolate the terms involving the functions. This step usually involves canceling out common factors or rearranging terms to make the inequality easier to work with.

Determine an upper bound: Look for an upper bound on the growth rate by finding the highest power of n or the term that dominates the growth of the function. This term will determine the upper limit of the growth rate.

Choose a value for c: Once you have determined the upper bound, you can choose a suitable value for "c" that satisfies the inequality. This value should be greater than or equal to the upper bound to ensure that the inequality holds for all values of n greater than or equal to k.

Test the inequality: Substitute the chosen value of "c" back into the inequality and verify that it holds true for all values of n greater than or equal to k. If the inequality is satisfied, you have found a valid value for "c".

Remember that finding the exact value of "c" is not always necessary in Big O notation. The focus is on establishing an upper bound and showing that a constant "c" exists for which the inequality holds.

Example :- f(n) = 3n2 +4n+1. Show that f(n)=O(n∧2).

Introduction: We want to show that the function f(n) = 3n∧2 + 4n + 1 is "smaller" or "equal to" the function g(n) = n∧2 when n gets bigger. We'll do this by finding some numbers that help us compare the two functions.

Step 1: To show that f(n) is O(n∧2), we need to establish that for any value of n greater than or equal to a certain number k, the function f(n) is less than or equal to a constant c multiplied by g(n).

Step 2: Let's break down the inequality and simplify it step by step. We start by comparing the individual terms and inequalities involved.

4n ≤ 4n∧2 for all n ≥ 1: This step shows that for any value of n greater than or equal to 1, the inequality 4n ≤ 4n∧2 holds true. In other words, the term 4n is always smaller or equal to 4n∧2 when n is 1 or larger.

1 ≤ n∧2 for all n ≥ 1: This step establishes that for any value of n greater than or equal to 1, the inequality 1 ≤ n∧2 holds true. This inequality tells us that n∧2 is always greater than or equal to 1 when n is 1 or larger.

Step 3: Now that we have these two inequalities, we can combine them to form a new inequality:

3n∧2 + 4n + 1 ≤ 3n∧2 + 4n∧2 + n∧2 for all n ≥ 1.

Simplifying the inequality, we get:

3n∧2 + 4n + 1 ≤ 8n∧2 for all n ≥ 1.

This shows that for any value of n greater than or equal to 1, the function f(n) = 3n∧2 + 4n + 1 is less than or equal to 8 multiplied by g(n) = n∧2.

Step 4: By choosing c = 8 and k = 1, we have shown that for all values of n greater than or equal to 1, the function f(n) = 3n∧2 + 4n + 1 is "smaller" or "equal to" 8 multiplied by g(n) = n∧2.

In simpler terms, we have demonstrated that as n gets larger, the function f(n) grows no faster than g(n). This comparison helps us understand the growth rates of the two functions and how they relate to each other.

Example :- F(n) = 5n + 6 and g(n) = n , show that f(n) is O(g(n)) , to show that f(n) is O(g(n)) **?**

Introduction: We want to show that the function f(n) = 5n + 6 is "smaller" or "equal to" the function g(n) = n when n gets larger. We'll do this by finding some numbers that help us compare the two functions.

Step 1: To show that f(n) is O(g(n)), we need to establish that for any value of n greater than or equal to a certain number k, the function f(n) is less than or equal to a constant c multiplied by g(n).

Step 2: Let's compare the individual terms and inequalities involved.

5n ≤ 5n for all n ≥ 1: This inequality tells us that for any value of n greater than or equal to 1, the term 5n is always equal to 5n. It does not change in relation to the value of n.

6 ≤ n for all n ≥ 6: This inequality states that for any value of n greater than or equal to 6, the term 6 is less than or equal to n. This means that once n becomes 6 or larger, the value of 6 is always smaller than or equal to n.

Step 3: Combining these inequalities, we can form the following inequality:

5n + 6 ≤ 5n + n for all n ≥ 6.

Simplifying the inequality, we get:

5n + 6 ≤ 6n for all n ≥ 6.

This shows that for any value of n greater than or equal to 6, the function f(n) = 5n + 6 is less than or equal to 6 multiplied by g(n) = n.

Step 4: By choosing c = 6 and k = 6, we have shown that for all values of n greater than or equal to 6, the function f(n) = 5n + 6 is "smaller" or "equal to" 6 multiplied by g(n) = n.

In simpler terms, we have demonstrated that as n gets larger, the function f(n) grows no faster than g(n). This comparison helps us understand the growth rates of the two functions and how they relate to each other.

BUT WE CAN ALSO DO THIS

Introduction: We want to show that the function f(n) = 5n + 6 is "smaller" or "equal to" the function g(n) = n when n gets larger. In other words, we want to demonstrate that the growth rate of f(n) is no faster than the growth rate of g(n) for sufficiently large values of n.

Detailed Explanation:

Step 1: To show that f(n) is O(g(n)), we need to establish that for any value of n greater than or equal to a certain number k, the function f(n) is less than or equal to a constant c multiplied by g(n).

Step 2: Let's start by comparing the individual terms and inequalities involved.

5n + 6 ≤ 11n for all n ≥ 1: This inequality tells us that for any value of n greater than or equal to 1, the term 5n + 6 is less than or equal to 11n. We want to find a value of n (which we will call k) from which this inequality holds true.

Step 3: Simplifying the inequality, we get:

6 ≤ 6n for all n ≥ 1.

This inequality indicates that for any value of n greater than or equal to 1, the constant term 6 is less than or equal to 6n.

Step 4: By examining this inequality, we can see that it is true for n = 1. When n = 1, we have 6

$\leq$ 6. Therefore, we can say that for any value of n greater than or equal to 1, the function f(n) = 5n + 6 is less than or equal to 11n.

Step 5: By choosing c = 11 and k = 1, we have shown that for all values of n greater than or equal to 1, the function f(n) = 5n + 6 is "smaller" or "equal to" 11 multiplied by g(n) = n.

In simpler terms, we have demonstrated that as n gets larger, the function f(n) grows no faster than g(n). This comparison helps us understand the growth rates of the two functions and how they relate to each other.

CHAPTER TWO

DIVIDE AND CONQUER

The Divide and Conquer approach is a problem-solving paradigm that involves breaking down a complex problem into smaller and more manageable subproblems, solving them independently, and then combining the solutions to obtain the final solution.

The approach consists of three steps:

1. **Divide**: The problem is divided into smaller subproblems of the same type or structure. By breaking down the problem into smaller parts, it becomes easier to solve them individually. This division is typically done recursively until the sub problems become simple enough to be solved directly.

2. **Conquer**: Each subproblem is solved independently. This step usually involves applying the same algorithm or technique recursively to solve the subproblems. The solutions to the subproblems may be obtained by further dividing them or using a different approach specific to the subproblem type.

3. **Combine**: The solutions obtained from the subproblems are combined to obtain the solution to the original problem. This step involves merging or aggregating the subproblem solutions in a way that builds up to the final solution. The combination step should ensure that the final solution is correct and complete.

The Divide and Conquer approach is particularly useful for solving problems that can be broken down into smaller, similar subproblems. It enables efficient problem solving by reducing the complexity of the original problem into smaller, more manageable parts.

The main advantage of the Divide and Conquer approach is that it often leads to efficient algorithms with better time and space complexity compared to other methods. By solving smaller subproblems independently and combining their solutions, the approach can provide solutions to larger and more complex problems.

However, it is important to note that the Divide and Conquer approach is not suitable for all types of problems. Some problems may not be easily divisible into sub problems, or the overhead of combining the solutions may negate the benefits gained from the division. Therefore, careful analysis and understanding of the problem's characteristics are necessary to determine if the Divide and Conquer approach is appropriate.

The divide-and-conquer approach is a problem-solving strategy commonly used in algorithms. It involves breaking down a complex problem into smaller, more manageable sub problems, solving them independently, and then combining the solutions to obtain the solution for the original problem.

The process consists of three steps at each level of recursion:

1. **Divide:** The problem is divided into multiple sub problems that are similar to the original problem but smaller in size. This step involves breaking down the problem into smaller, more easily solvable parts.

2. **Conquer:** The sub problems are solved recursively. If the sub problems are small

enough, they can be solved directly without further division. In this step, the algorithm applies the same divide-and-conquer approach to solve each subproblem independently.

3. **Combine:** The solutions to the sub problems are combined to obtain the solution for the original problem. This step involves merging or integrating the individual solutions obtained from solving the sub problems.

By repeatedly applying these steps, the algorithm divides the problem into smaller and more manageable sub problems, solves them independently, and then combines the solutions until the original problem is solved.

The divide-and-conquer approach is beneficial because it simplifies complex problems by breaking them into smaller, more understandable parts. It often leads to efficient algorithms and can significantly improve the overall performance of a solution.

ANALYZING THE DIVIDE AND CONQUER ALGORITHM

When analyzing divide-and-conquer algorithms, we often use recurrence equations to describe their running time. A recurrence equation expresses the overall running time of an algorithm on a problem of size n in terms of the running time on smaller inputs.

To understand this concept in simpler terms:

Recursive Calls:- When an algorithm calls itself recursively, we can describe its running time using a recurrence equation. This equation helps us understand how the algorithm's performance relates to the size of the input.

Steps of the Paradigm: The recurrence equation is based on the three steps of the divide-and-conquer paradigm: divide, conquer, and combine.

Divide and Conquer: The algorithm divides the problem into smaller subproblems, typically of size n/b, where b is a constant. It solves each subproblem recursively.

Base Case: When the problem size becomes small enough (e.g., n ≤ c for some constant c), the algorithm can solve it directly in constant time (Θ(1)).

When we divide a problem as part of the divide-and-conquer approach, we end up with smaller subproblems. The size of each subproblem is a fraction of the original problem's size.

In this context, the fraction is denoted as 1/b, where "b" is a constant. It represents how much smaller each subproblem is compared to the original problem.

Division and Combination Time: The time taken to divide the problem into sub problems is denoted as D(n), and the time taken to combine the subproblem solutions into the original problem's solution is denoted as C(n).

Recurrence Equation: Putting everything together, we can write a recurrence equation to describe the running time T(n) of the algorithm on an input of size n. The equation incorporates the time spent on dividing the problem (D(n)), the time spent on combining the solutions (C(n)), and the recursive calls on sub problems of size n/b. This equation helps us understand and analyze the performance of the algorithm.

By solving the recurrence equation using mathematical tools and techniques, we can derive bounds on the algorithm's performance and understand how the running time grows as the input size increases.

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

Here are some well-known algorithms that utilize the Divide and Conquer approach:

- ➔ Merge sort
- ➔ Binary Search
- ➔ Quick Sort

MERGE SORT

Merge sort is a sorting algorithm that arranges a list of elements in ascending (or descending) order. It follows a divide-and-conquer approach to break down the sorting process into smaller, more manageable parts.

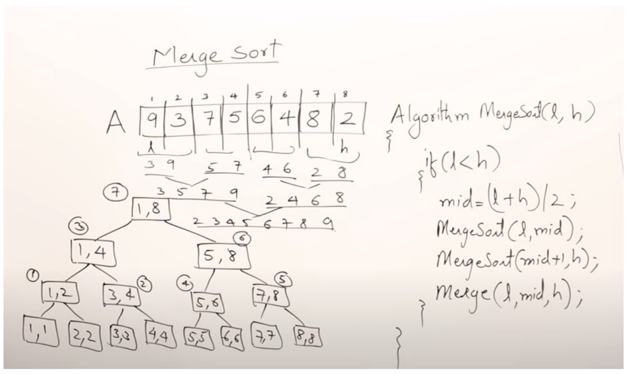
Here's a step-by-step explanation of merge sort in simpler terms:

1. **Divide**: The algorithm starts by dividing the unsorted list into smaller sublists. It repeatedly divides the list in half until each sublist contains only one element or is empty.
2. **Conquer**: After dividing the list, the algorithm recursively sorts each sublist. This is done by applying the same merge sort algorithm to each sublist. The base case is reached when a sublist contains only one element, as a single element is always considered sorted.
3. **Combine**: Once all the sublists are sorted, the algorithm merges them back together to create a sorted list. It compares the elements from the sublists pairwise and places them in the correct order. This merging process continues until all the elements are merged into a single sorted list.
4. **Repeat:** The divide, conquer, and combine steps are repeated recursively until the original unsorted list is completely sorted.

The key idea behind merge sort is that merging two sorted lists is relatively easy. By recursively sorting smaller sublists and then merging them back together, the algorithm can gradually build a sorted list from the bottom up.

Merge sort has a time complexity of O(n log n), where "n" represents the number of elements in the list. It is known for its efficiency and stable sorting, meaning that the order of equal elements is preserved.

In summary, merge sort is a sorting algorithm that divides the list, sorts each divided part recursively, and then merges the sorted parts to produce a final sorted list. This process continues until the entire list is sorted.



TIME COMPLEXITY OF MERGE SORT

The time complexity of merge sort is a measure of how the running time of the algorithm increases as the size of the input list grows.

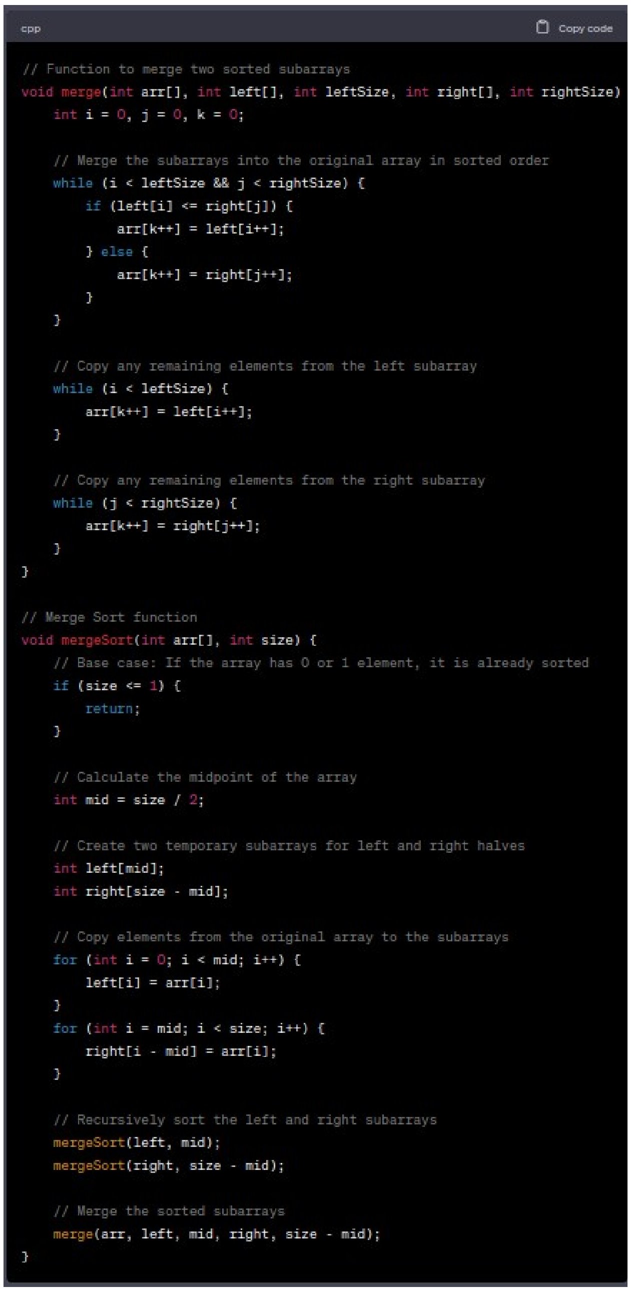
In simpler terms, the time complexity of merge sort tells us how many comparisons and operations the algorithm needs to perform to sort a list of elements.

Merge sort has a time complexity of O(n log n), where "n" represents the number of elements in the list. The "log n" term indicates that the algorithm's running time increases in proportion to the logarithm of the input size, while the "n" term represents the linear relationship with the input size.

The "n log n" time complexity of merge sort makes it an efficient sorting algorithm for large lists. It ensures that the algorithm's running time grows at a manageable rate, even as the input size increases.

To summarize, the time complexity of merge sort, expressed as O(n log n), tells us that the number of operations performed by the algorithm grows logarithmically with the input size. This makes merge sort an efficient sorting algorithm for large lists.

SIMPLE IMPLEMENTATION OF MERGE SORT



```cpp
// Function to merge two sorted subarrays
void merge(int arr[], int left[], int leftSize, int right[], int rightSize)
    int i = 0, j = 0, k = 0;

    // Merge the subarrays into the original array in sorted order
    while (i < leftSize && j < rightSize) {
        if (left[i] <= right[j]) {
            arr[k++] = left[i++];
        } else {
            arr[k++] = right[j++];
        }
    }

    // Copy any remaining elements from the left subarray
    while (i < leftSize) {
        arr[k++] = left[i++];
    }

    // Copy any remaining elements from the right subarray
    while (j < rightSize) {
        arr[k++] = right[j++];
    }
}

// Merge Sort function
void mergeSort(int arr[], int size) {
    // Base case: If the array has 0 or 1 element, it is already sorted
    if (size <= 1) {
        return;
    }

    // Calculate the midpoint of the array
    int mid = size / 2;

    // Create two temporary subarrays for left and right halves
    int left[mid];
    int right[size - mid];

    // Copy elements from the original array to the subarrays
    for (int i = 0; i < mid; i++) {
        left[i] = arr[i];
    }
    for (int i = mid; i < size; i++) {
        right[i - mid] = arr[i];
    }

    // Recursively sort the left and right subarrays
    mergeSort(left, mid);
    mergeSort(right, size - mid);

    // Merge the sorted subarrays
    merge(arr, left, mid, right, size - mid);
}
```

QUICK SORT

Quick sort is a sorting algorithm that arranges a list of elements in ascending (or descending) order. It follows a divide-and-conquer approach and works by selecting a pivot element from the list and partitioning the other elements into two sublists: one with elements smaller than the pivot and another with elements greater than the pivot.

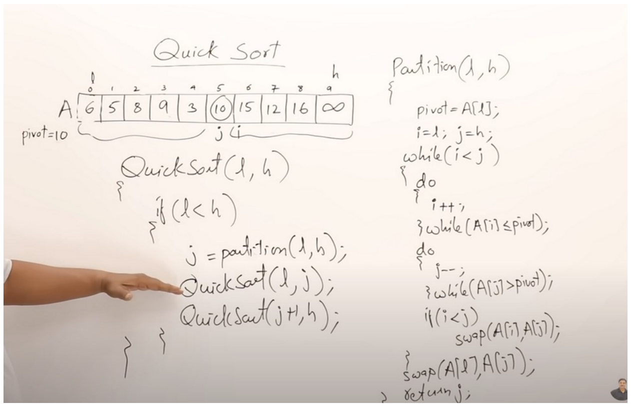
Here's a step-by-step breakdown of quick sort in simpler terms:

1. **Choose a Pivot**: The algorithm selects a pivot element from the list. The pivot can be chosen in different ways, such as selecting the first, middle, or random element.
2. **Partition the List:** The algorithm rearranges the other elements in the list such that all elements smaller than the pivot come before the pivot, and all elements greater than the pivot come after it. This step effectively partitions the list into two sublists.
3. **Recursively Sort Sublists**: The algorithm then recursively applies the same process to the two sublists created in the previous step. This involves selecting a pivot for each sublist and partitioning the elements around that pivot.
4. **Combine the Sorted Sublists:** Finally, the algorithm combines the sorted sublists by placing the elements in the correct order. The result is a fully sorted list.

The key idea behind quick sort is that it repeatedly partitions the list into smaller sublists and sorts them independently. By dividing the problem into smaller parts and solving them recursively, quick sort efficiently sorts the entire list.

Quick sort has an average time complexity of O(n log n), where "n" represents the number of elements in the list. It is known for its efficiency and is widely used in practice for sorting large datasets.

To summarize, quick sort is a sorting algorithm that selects a pivot, partitions the list based on the pivot, recursively sorts the sublists, and combines them to obtain a sorted list. By dividing and conquering the sorting process, quick sort efficiently handles large lists.



```
QuickSort(l, h)
{
    if(l<h)
    {
        j = partition(l,h);
        QuickSort(l,j);
        QuickSort(j+1,h);
    }
}

Partition(l, h)
{
    pivot = A[l];
    i=l; j=h;
    while(i<j)
    {
        do
        {
            i++;
        }while(A[i]≤pivot);
        do
        {
            j--;
        }while(A[j]>pivot);
        if(i<j)
            swap(A[i],A[j]);
    }
    swap(A[l],A[j]);
    return j;
}
```

TIME COMPLEXITY OF A QUICK SORT

The time complexity of quick sort is a measure of how the running time of the algorithm increases as the size of the input list grows.

In simpler terms, the time complexity of quick sort tells us how many comparisons and operations the algorithm needs to perform to sort a list of elements.

The average time complexity of quick sort is O(n log n), where "n" represents the number of elements in the list. This means that as the input size doubles, the number of operations required by quick sort increases roughly in proportion to the logarithm of the input size multiplied by the input size itself.

To understand this, imagine you have 10 elements in the list. Quick sort may require around 100 operations. If you double the size to 20 elements, it may require around 200 operations. The number of operations grows at a moderate rate compared to the input size.

However, in the worst-case scenario, when the chosen pivot is consistently the smallest or largest element, quick sort can have a time complexity of O(n ∧ 2), which means the number of operations can grow rapidly as the input size increases. But this worst-case scenario is rare in practice.

In summary, the time complexity of quick sort, on average, is O(n log n), indicating that the number of operations grows in a manageable way as the input size increases. Quick sort is known for its efficiency, especially for large lists, making it a popular sorting algorithm in practice.

HOW QUICK SORT WORKS

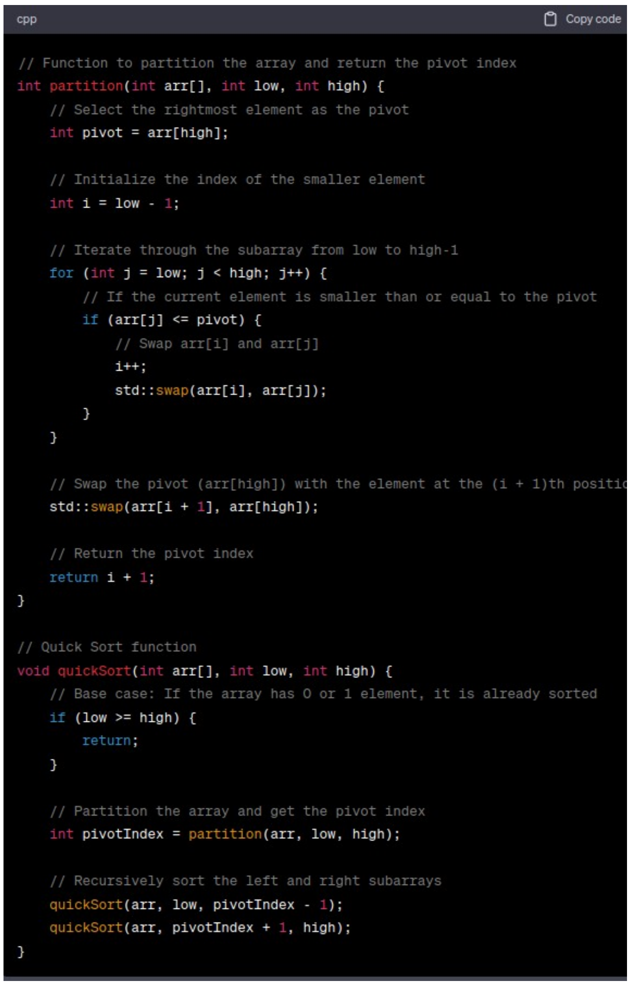
In quick sort, the increments of the "i" and "j" indices are based on the pivot element and the partitioning process. The goal is to efficiently partition the elements in the list around the pivot.

Here's how the increments of "i" and "j" are typically determined in the partitioning step of quick sort :-

1. Start with "i" and "j" indices pointing to the left and right ends of the sub list respectively.
2. Increment "i" until an element greater than or equal to the pivot is found. This means moving "i" to the right until an element that should be on the right side of the pivot is encountered.
3. Decrement "j" until an element smaller than or equal to the pivot is found. This means moving "j" to the left until an element that should be on the left side of the pivot is encountered.
4. If "i" is still less than or equal to "j", swap the elements at "i" and "j" to ensure that the smaller elements are on the left side and the larger elements are on the right side of the pivot.
5. Repeat steps 2 to 4 until "i" becomes greater than "j". At this point, all the elements have been partitioned correctly around the pivot.

Here's a code snippet illustrating the increments of "i" and "j" in the partitioning step of quick sort:

QUICK SORT IMPLEMENTATION



```cpp
// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    // Select the rightmost element as the pivot
    int pivot = arr[high];

    // Initialize the index of the smaller element
    int i = low - 1;

    // Iterate through the subarray from low to high-1
    for (int j = low; j < high; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            // Swap arr[i] and arr[j]
            i++;
            std::swap(arr[i], arr[j]);
        }
    }

    // Swap the pivot (arr[high]) with the element at the (i + 1)th positi
    std::swap(arr[i + 1], arr[high]);

    // Return the pivot index
    return i + 1;
}

// Quick Sort function
void quickSort(int arr[], int low, int high) {
    // Base case: If the array has 0 or 1 element, it is already sorted
    if (low >= high) {
        return;
    }

    // Partition the array and get the pivot index
    int pivotIndex = partition(arr, low, high);

    // Recursively sort the left and right subarrays
    quickSort(arr, low, pivotIndex - 1);
    quickSort(arr, pivotIndex + 1, high);
}
```

CHAPTER THREE

GREEDY MODEL ALGORITHM

Introduction: The Greedy algorithmic paradigm is an approach to problem-solving that follows the principle of making locally optimal choices at each step. In this paradigm, the algorithm makes the best decision at the current moment without considering the overall global optimal solution. The Greedy approach is often used when a problem can be solved by making a series of choices, and each choice can be optimized locally.

Detailed Explanation: The Greedy algorithmic paradigm follows a set of steps to solve a problem :-

1. **Define the problem**: Clearly understand the problem and identify the goal or objective that needs to be achieved. Determine the constraints and the criteria for making choices.

2. **Identify the sub problems:** Break down the problem into smaller sub problems or steps. Determine how the optimal solution to the overall problem can be obtained by solving these sub problems individually.

3. **Make a greedy choice:** At each step or sub problem, make a locally optimal choice that appears to be the best at that moment. This choice is based on some heuristics or criteria defined by the problem.

4. **Update the solution:** Update the current solution by incorporating the locally optimal choice made in the previous step. Keep track of the progress and ensure that the solution remains feasible and improves towards the global optimal solution.

5. **Repeat steps 3 and 4:** Continue making greedy choices and updating the solution until the problem is solved or a satisfactory solution is achieved. At each step, consider the current state and make choices that maximize the immediate benefit or utility.

6. **Verify the solution:** Finally, evaluate and verify the solution obtained using the Greedy approach. Ensure that it satisfies the desired criteria and objectives of the problem. Note that Greedy algorithms do not guarantee an optimal solution in every case, so it is important to validate the solution's correctness and optimality.

The Greedy algorithmic paradigm is often used in optimization problems, scheduling problems, and various other scenarios where making locally optimal choices can lead to a reasonably good solution. However, it is important to note that Greedy algorithms may not always provide the globally optimal solution, and careful consideration of problem-specific characteristics and constraints is required.

Overall, the Greedy algorithmic paradigm offers an intuitive and efficient approach to problem-solving, allowing us to make decisions based on immediate gains or benefits, even if they might not lead to the best overall solution.

FEASIBLE SOLUTION AND OPTIMAL SOLUTION

Feasible Solution: A feasible solution, in the context of the Greedy algorithm, refers to a solution that satisfies all the constraints and requirements of the problem. It is a solution that is valid and permissible within the given problem domain. Feasible solutions are typically determined by checking if they meet the specific conditions or constraints defined by the problem.

For example, if you have a scheduling problem where you need to assign tasks to workers, a feasible solution would be one that assigns each task to a worker without violating any constraints, such as worker availability or task dependencies. Feasible solutions do not necessarily need to be optimal; they only need to adhere to the problem's constraints.

Optimal Solution: An optimal solution, in the context of the Greedy algorithm, refers to the best possible solution among all feasible solutions. It is the solution that maximizes or minimizes a certain objective or criterion, depending on the problem's goal. The objective could be to maximize profit, minimize cost, minimize time, or achieve any other optimization goal specific to the problem.

Finding an optimal solution is the primary objective of the Greedy algorithm, although it does not guarantee that the solution will be globally optimal for every problem. Greedy algorithms make locally optimal choices at each step, aiming to maximize immediate gain or benefit. While the Greedy approach may provide an optimal solution for certain problems, it can fall short in others, and additional techniques may be required.

It is essential to evaluate the solution obtained using the Greedy algorithm to determine if it meets the problem's optimality criteria. This evaluation involves comparing the solution to known optimal solutions or using mathematical analysis to ensure that the solution meets the desired optimization goals.

ADVANTAGE AND DISADVANTAGE OF GREEDY ALGORITHM

Advantages of the Greedy algorithm:

1. **Simplicity**: Greedy algorithms are often straightforward to implement and understand. They follow a simple logic of making locally optimal choices at each step, which can make them easier to design and analyze compared to more complex algorithms.

2. **Efficiency**: Greedy algorithms often have efficient runtimes, making them suitable for solving large-scale problems. They typically have a linear or near-linear time complexity, which can be advantageous when dealing with datasets of significant sizes.

3. **Intuitive approach**: The Greedy paradigm aligns with human intuition. It mimics the way people often make decisions by considering immediate gains or benefits, which can lead to reasonably good solutions in many scenarios.

4. **Approximation solutions**: Although not always providing the globally optimal solution, Greedy algorithms can often provide reasonably good solutions that are close to the optimal solution. In some cases, the difference between the Greedy solution and the optimal solution may be negligible.

Disadvantages of the Greedy algorithm:

1. **Lack of global optimization**: Greedy algorithms make locally optimal choices at each step without considering the overall global optimization. This can lead to

situations where the locally optimal choice does not result in the best overall solution. Therefore, Greedy algorithms may not always guarantee the globally optimal solution.

2. **Limited applicability:** Greedy algorithms are suitable for specific types of problems where making locally optimal choices leads to an acceptable solution. However, they may not be applicable or effective for all problem types. Problems with complex dependencies or constraints may require more sophisticated algorithms.
3. **Lack of backtracking**: Greedy algorithms do not typically revisit or undo choices made at earlier steps. Once a choice is made, it is assumed to be the best local choice. This lack of backtracking can restrict the algorithm's ability to correct or revise decisions, potentially leading to suboptimal solutions.
4. **Sensitivity to input**: Greedy algorithms can be sensitive to the order or structure of the input data. Small changes in the input or the order of processing can sometimes result in significantly different outcomes. This sensitivity can be a disadvantage when dealing with unpredictable or varying inputs.

It's important to note that the advantages and disadvantages of the Greedy algorithm depend on the specific problem at hand. While the Greedy approach has its limitations, it remains a valuable algorithmic paradigm in various domains where its strengths align with the problem characteristics.

Here are some well-known algorithms that utilize the Greedy approach:

- **Dijkstra's Algorithm**
- **Kruskal's Algorithm**
- **Prim's Algorithm**
- **Job Sequencing Algorithm**

JOB SEQUENCING ALGORITHM

The job sequencing algorithm is a problem-solving approach that aims to maximize the profit or efficiency of completing a set of jobs within a given deadline. The algorithm determines the order in which the jobs should be executed to achieve the best outcome.

In simpler terms, the job sequencing algorithm helps us decide which jobs to prioritize and in what order to complete them to achieve the highest possible profit or optimal utilization of resources. It considers both the deadlines associated with the jobs and their respective profits or benefits.

Here's a step-by-step breakdown of the job sequencing algorithm:

1. **Gather Job Details**: The algorithm collects information about each job, including its deadline and associated profit or benefit.
2. **Sort Jobs**: The jobs are sorted based on their profit in descending order. This step ensures that the jobs with higher profits are considered first during the sequencing process.
3. **Determine Feasible Schedule**: Starting from the highest-profit job, the algorithm attempts to schedule each job within its deadline while avoiding any clashes with previously scheduled jobs. It checks if the time slot corresponding to the job's deadline is available or if there is a vacant slot earlier than the deadline.
4. **Update Schedule and Profit**: If a job can be scheduled, it is assigned to the corresponding time slot. The profit is also updated by adding the profit of the scheduled job. If a job cannot be scheduled due to clash or lack of available slots, it is skipped.
5. **Repeat Steps 3-4:** The algorithm continues with the next job in the sorted order, repeating the process until all jobs are considered or scheduled.

6. **Final Output**: The algorithm provides the optimal schedule of jobs along with the total profit achieved by executing those jobs within their respective deadlines.

The goal of the job sequencing algorithm is to maximize the overall profit or achieve the desired objective while adhering to the given deadlines. By prioritizing higher-profit jobs and intelligently scheduling them within the available time slots, the algorithm helps to make the best use of resources and maximize the outcome.

It's worth noting that different variants of the job sequencing algorithm exist, such as considering penalties for missing deadlines or considering time-dependent profits. However, the basic concept remains the same: sequencing jobs to optimize the desired objective within given constraints.

Job sequencing with deadlines is a problem where we have a set of jobs, each with a deadline and a profit associated with it. The goal is to maximize the total profit by completing the jobs within their respective deadlines.

Here's a breakdown of the simplified concepts:

1. Job Details: Each job has a deadline (a specific time by which it should be completed) and a profit associated with it. The profit is earned only if the job is completed on time.

2. Single Processor: There is only one machine available to process the jobs, and it takes one unit of time to complete each job.

3. Problem Objective: The objective is to find the best way to schedule the jobs to maximize the total profit. We want to complete the jobs within their deadlines to earn the associated profits.

4. Constraints: The constraints include having only one processor and the requirement to complete all jobs within their respective deadlines.

5. Feasible Solution: A feasible solution is a subset of jobs that can be completed within their deadlines. The profit of a feasible solution is the sum of the profits of the jobs included in that subset.

6. Optimal Solution: An optimal solution is a feasible solution that gives the maximum total profit. It is the best possible solution for maximizing the profits within the given constraints

Here's a simpler explanation of the greedy algorithm for job sequencing with deadlines:

1. **Sort Jobs:** First, we sort all the given jobs in decreasing order of their profits. This means we prioritize the jobs with higher profits.

2. **Create Gantt Chart**: Next, we create a Gantt chart with a time axis representing the maximum deadline among all the jobs. The Gantt chart helps visualize the scheduling of jobs within their deadlines.

3. **Schedule Jobs**: Starting with the job that has the highest profit, we try to place each job on the Gantt chart as far to the right as possible while ensuring it gets completed before its deadline. This means we schedule jobs one by one in the order of their sorted profits, making sure they are placed in the Gantt chart without overlapping with other jobs or exceeding their respective deadlines.

The key idea behind this greedy algorithm is to prioritize jobs with higher profits and schedule them as late as possible within their deadlines. By doing this, we aim to maximize the total profit by completing the most profitable jobs while considering their time constraints.

In simpler terms, the algorithm sorts jobs by their profits and schedules them on a Gantt chart, taking care to place each job as late as possible without missing its deadline. By following this approach, we can achieve an optimal solution that maximizes the total profit.

Example :- Given the following jobs, their deadlines and associated profits as shown-

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

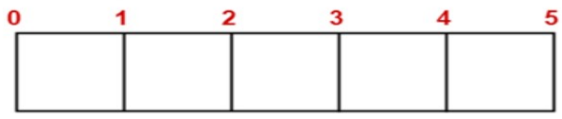
Answer the following questions :-

- Write the optimal schedule that gives maximum profit.
- Are all the jobs completed in the optimal schedule?
- What is the maximum earned profit?

Step-1: Sort all the given jobs in the decreasing order of their profits-

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

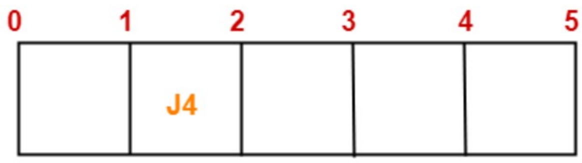
Step-2 :- Value of maximum deadline = 5, so draw a Gantt chart with maximum time on Gantt chart = 5 units



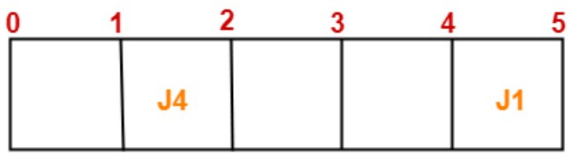
Step-3: Now, we will take each job one by one in the order they appear in step-1 and place them on Gantt

- chart as far as possible from 0 (starting point)

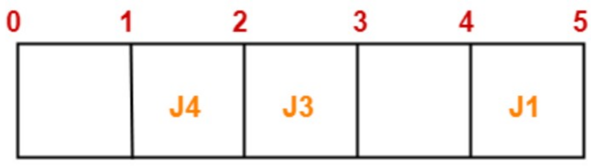
First, we take job J4. Since, its deadline is 2, so we place it in the first empty cell before deadline 2 as



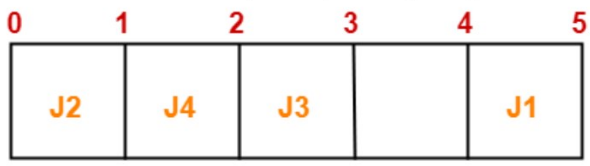
Now, we take job J1. Since, its deadline is 5, so we place it in the first empty cell before deadline 5



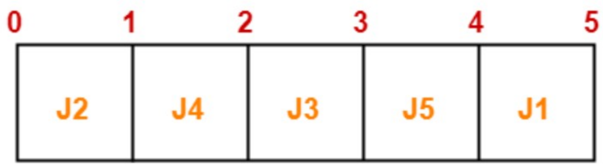
Now, we take job J3. Since, its deadline is 3, so we place it in the first empty cell before deadline 3



Now, we take job J2. Since, its deadline is 3, so we place it in the first empty cell before deadline 3. Since, the second and third cells are already filled, so we place job J2 in the first cell as-



Now, we take job J5. Since, its deadline is 4, so we place it in the first empty cell before deadline 4



Now, the only job left is job J6 whose deadline is 2. All the slots before deadline 2 are already occupied. Thus, job J6 cannot be completed.

Now, the given questions may be answered as-

Part-01: The optimal schedule is- J2, J4, J3, J5, J1. This is the required order in which the jobs must be completed in order to obtain the maximum profit.

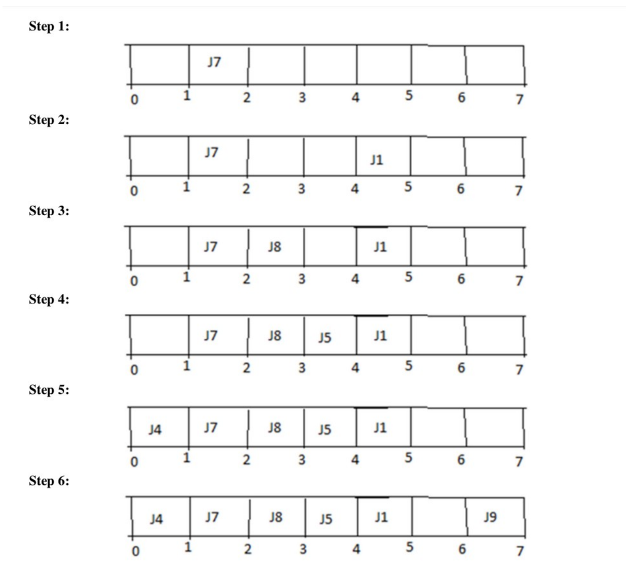
Part-02: All the jobs are not completed in the optimal schedule. This is because job J6 could not be completed within its deadline.

Part-03: Maximum earned profit
= Sum of profits of all jobs in optimal schedule

= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1
= 180 + 300 + 190 + 120 + 200
= 990 units //

Example 2 :- Given a set of 9 jobs where each job has a deadline and profit associated to it .Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit if and only if the job is completed by its deadline. The task is to find the maximum profit and the number of jobs done.

Jobs	Profit	Deadline
J1	85	5
J2	25	4
J3	16	3
J4	40	3
J5	55	4
J6	19	5
J7	92	2
J8	80	3
J9	15	7



So, the maximum profit = 40 + 92 + 80 + 55 + 85 + 15 = 367 //

OPTIMAL MERGE PATTERN

The optimal merge pattern is an approach used to merge multiple sorted lists into a single sorted list in the most efficient way. It aims to minimize the number of comparisons and operations needed to perform the merging.

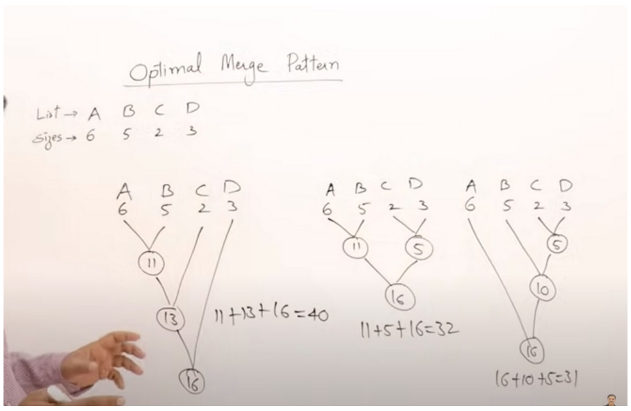
Here's a breakdown of the simplified details :-

1. **Sorted Lists**: We start with multiple lists that are already sorted in ascending order. Each list may have a different number of elements.
2. **Merge Process**: The goal is to merge all the sorted lists into a single sorted list. This involves comparing the elements from the different lists and arranging them in the correct order.
3. **Optimal Merge**: The optimal merge pattern determines the most efficient way to merge the lists. It minimizes the number of comparisons needed to merge the elements.
4. **Pairwise Merging**: The optimal merge pattern works by merging pairs of lists at a time. It compares the elements from each pair of lists and merges them into a new sorted list. This process continues until all the lists are merged into a single sorted list.
5. **Efficient Comparisons**: The optimal merge pattern ensures that each comparison is done between elements that have the closest values. By doing so, it minimizes the number of comparisons required to merge the lists.

The key idea behind the optimal merge pattern is to merge pairs of lists efficiently, prioritizing comparisons between elements with similar values. This approach reduces the overall number of comparisons needed to merge the lists and improves the efficiency of the merging process.

In summary, the optimal merge pattern is a technique used to merge multiple sorted lists into a single sorted list with minimum comparisons. It achieves this by merging pairs of lists efficiently, considering elements with similar values first. By following this approach, the merging process can be performed in an optimized and time-efficient manner.

Example :-



The optimal merge pattern is a greedy method used to merge multiple sorted lists into a single sorted list. The goal is to minimize the number of comparisons needed for the merging process.

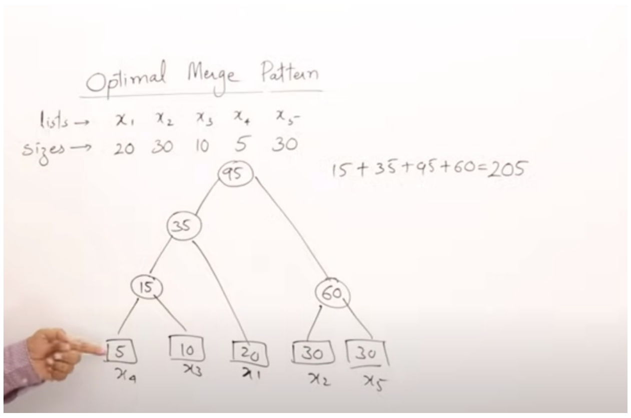
In simpler terms:

1. **Greedy Approach**: The optimal merge pattern follows a greedy approach, which means it makes locally optimal choices at each step to achieve the best overall result.
2. **Merging Small Size Lists**: To get the best result, the optimal merge pattern suggests merging pairs of lists that have the smallest sizes. This means prioritizing merging smaller lists together.

The reasoning behind this is that merging smaller lists requires fewer comparisons compared to merging larger lists. By merging smaller lists first, the overall number of comparisons needed for the entire merging process can be minimized.

So, in summary, the optimal merge pattern is a greedy method that recommends merging pairs of lists with smaller sizes. This approach aims to reduce the total number of comparisons and optimize the merging process to obtain the best overall result.

Example :-



MINIMUM SPANNING TREE

What is spanning tree ?

A spanning tree is a concept in graph theory that refers to a subset of a graph that includes all the vertices (points or nodes) of the original graph while forming a tree-like structure without any cycles.

Here's a simpler explanation:

1. **Graph** :- A graph is a collection of vertices (points or nodes) connected by edges (lines). It represents relationships or connections between different elements.
2. **Subset of the Graph :-** A spanning tree is a subset of the original graph. It contains some vertices and edges from the original graph but not necessarily all of them.
3. **Tree-like Structure** :- In a spanning tree, the selected vertices and edges form a connected structure without any cycles. This means there are no closed loops or repeated edges.
4. **Including All Vertices** :- The spanning tree must include all the vertices of the original graph. This means every vertex in the graph is part of the spanning tree.

The concept of a spanning tree is useful for understanding the structure and connectivity of a graph. It helps identify the minimal subset of vertices and edges needed to connect all the vertices in the graph without any cycles.

In simpler terms, a spanning tree is a subset of a graph that includes all the vertices and forms a connected structure without any cycles. It provides a way to connect all the vertices while maintaining a tree-like structure.

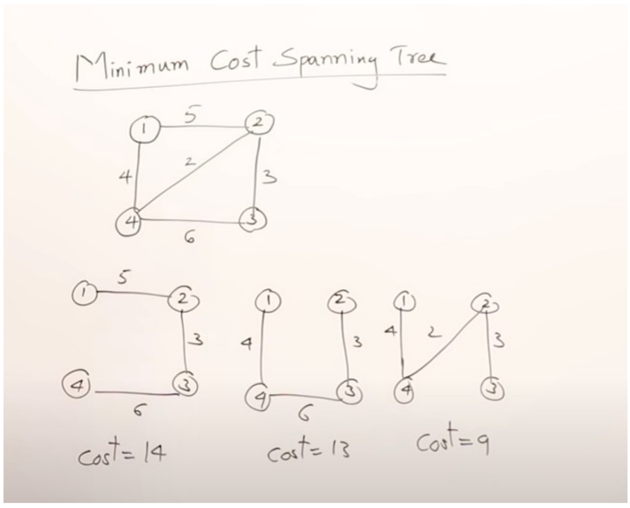
A minimum spanning tree is a concept in graph theory that involves finding a connected subgraph of a weighted graph that connects all the vertices while minimizing the total sum of the edge weights.

Let's break it down further:

1. **Weighted Graph:** A weighted graph consists of vertices (points or nodes) connected by edges (lines) that have associated weights or costs. These weights can represent distances, costs, or any other measure.
2. **Connected Subgraph**: A subgraph is a subset of the original graph that includes some vertices and edges from the original graph. A minimum spanning tree is a connected subgraph that contains all the vertices of the original graph.
3. **Connecting all Vertices**: In a minimum spanning tree, every vertex of the original graph must be connected to form a connected subgraph. This means there should be a path between any two vertices in the minimum spanning tree.
4. **Minimizing Total Edge Weights:** The goal of a minimum spanning tree is to minimize the sum of the weights of the edges in the tree. The tree should be constructed in such a way that the total weight is as small as possible compared to other possible spanning trees.

The concept of a minimum spanning tree is often used in optimization problems, network design, and efficient communication networks. It helps find the most efficient way to connect all the vertices in a weighted graph while minimizing the overall cost or distance.

In simpler terms, a minimum spanning tree is a subset of a weighted graph that connects all the vertices with the least possible total weight. It aims to find the most efficient way to connect the vertices while minimizing the overall cost or distance.



To find the best minimum spanning tree in a weighted graph, you can use various algorithms. The two most commonly used algorithms are Prim's algorithm and Kruskal's algorithm.

Finding the minimum spanning tree (MST) of a weighted graph without using an algorithm can be an arduous task. The number of possibilities to consider increases exponentially with the size of the graph, making it impractical for large graphs.

However, if the graph is small or you want to explore different possibilities, you can manually try different combinations of edges to create multiple spanning trees and calculate their costs. The general approach would be:

1. Start with an empty set of edges, which represents an empty tree.
2. Choose an edge from the graph that connects two vertices. Add this edge to the set of edges.
3. Repeat step 2, selecting edges that are not already in the set, until you have a connected tree that spans all the vertices.
4. Calculate the cost of the tree by summing the weights of the edges in the set.
5. Repeat steps 2-4, considering different combinations of edges, until you have considered all possibilities.

6. Compare the costs of the different spanning trees you obtained and choose the one with the minimum cost.

It's important to note that this manual approach becomes unfeasible for larger graphs due to the exponential growth in the number of possibilities. In such cases, Prim's algorithm or Kruskal's algorithm, as mentioned earlier, provide efficient and reliable methods to find the minimum spanning tree.

Using an algorithm ensures that you find the minimum spanning tree in an efficient manner, avoiding the need to manually consider all possibilities. These algorithms are designed to optimize the search for the minimum spanning tree by considering the weights of the edges and the connectivity of the vertices.

In summary, while manually considering different possibilities and calculating costs can be done for small graphs, it becomes impractical for larger graphs. Algorithms like Prim's algorithm and Kruskal's algorithm offer efficient approaches to finding the minimum spanning tree, taking into account the edge weights and connectivity to provide the best solution.

Prim's Algorithm

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a weighted graph. It starts with an arbitrary vertex and gradually grows the MST by adding the edge with the smallest weight that connects a vertex in the MST to a vertex outside the MST.

Here's a step-by-step breakdown of Prim's algorithm:

1. **Start with an arbitrary vertex**: Begin by selecting any vertex from the graph as the starting point of the MST.
2. **Grow the MST**: At each step, choose the edge with the smallest weight that connects a vertex in the current MST to a vertex outside the MST.
3. **Add the selected edge and vertex:** Add the selected edge to the MST and include the connected vertex in the MST.
4. **Continue the process**: Repeat steps 2 and 3 until all vertices are included in the MST. This ensures that the MST remains connected and spans all the vertices.
5. **Termination:** Once all vertices are included, the algorithm terminates, and you have obtained the minimum spanning tree.

By iteratively selecting the smallest weight edges, Prim's algorithm guarantees that the resulting tree is a minimum spanning tree with the minimum possible total weight.

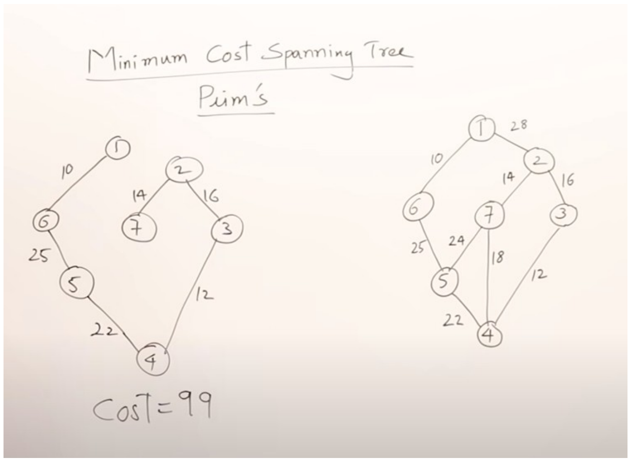
Here's a simplified example to illustrate Prim's algorithm:

Consider a weighted graph with five vertices: A, B, C, D, and E.

1. Start with vertex A as the initial vertex.
2. Find the smallest weight edge connected to vertex A. Suppose it is the edge from A to B with weight 3.
3. Add the edge from A to B and include vertex B in the MST.
4. Look for the smallest weight edge that connects the MST (vertices A and B) to a vertex outside the MST. Suppose it is the edge from B to D with weight 2.
5. Add the edge from B to D and include vertex D in the MST.
6. Repeat this process, selecting edges with the smallest weight, until all vertices (A, B, C, D, and E) are included in the MST.
7. The resulting MST will have the minimum total weight among all possible spanning trees of the graph.

In summary, Prim's algorithm is a greedy algorithm that starts with an arbitrary vertex and grows the minimum spanning tree by iteratively selecting the edge with the smallest weight. It ensures that the resulting tree is a minimum spanning tree by considering the connectivity and weights of the edges.

Example



Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a weighted graph. It starts with an empty set of edges and iteratively adds edges with the smallest weights while avoiding cycles. The final set of edges forms the minimum spanning tree.

Here's a step-by-step breakdown of Kruskal's algorithm:

1. Sort the edges: Begin by sorting all the edges of the graph in ascending order based on their weights.
2. Initialize an empty set of edges: Start with an empty set that will eventually contain the edges of the minimum spanning tree.
3. Iterate through the sorted edges: Begin iterating through the sorted edges in ascending order of weights.
4. Add edges without creating cycles: For each edge, if adding it to the current set of edges does not create a cycle (i.e., it does not connect two vertices already connected by a path), add the edge to the set of edges.
5. Track connected components: To check for cycles efficiently, use a disjoint set data structure (such as a union-find data structure) to keep track of connected components. It helps determine whether adding an edge creates a cycle or not.
6. Repeat until all vertices are connected: Continue the process until all vertices are connected, i.e., until the number of edges in the set of edges reaches (number of vertices - 1).
7. Termination: Once all vertices are connected, the algorithm terminates, and you have obtained the minimum spanning tree.

By selecting edges with the smallest weights and ensuring that no cycles are formed, Kruskal's algorithm guarantees that the resulting tree is a minimum spanning tree with the minimum possible total weight.

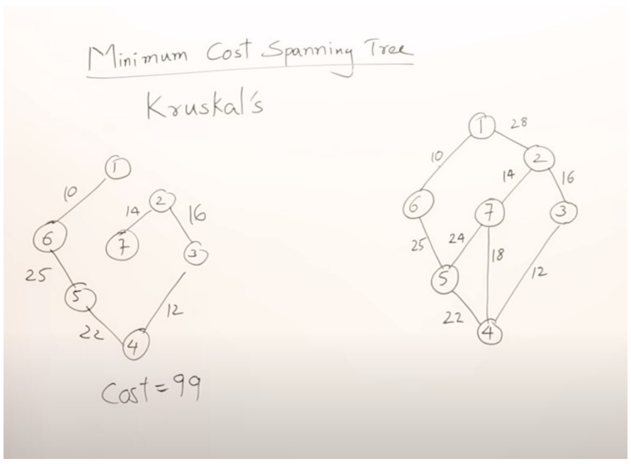
Here's a simplified example to illustrate Kruskal's algorithm:

Consider a weighted graph with five vertices: A, B, C, D, and E.

1. Sort the edges in ascending order of weights.
2. Begin with an empty set of edges.
3. Select the edge with the smallest weight, let's say it is the edge from A to B with weight 3.
4. Add the edge from A to B to the set of edges.
5. Select the next smallest weight edge that does not create a cycle, such as the edge from B to D with weight 2.
6. Add the edge from B to D to the set of edges.
7. Repeat this process, selecting edges with the smallest weight and avoiding cycles, until all vertices (A, B, C, D, and E) are connected.

8. The resulting set of edges forms the minimum spanning tree with the minimum total weight among all possible spanning trees of the graph.

In summary, Kruskal's algorithm is a greedy algorithm that sorts the edges and iteratively adds edges with the smallest weights while avoiding cycles. It guarantees that the resulting tree is a minimum spanning tree by considering the weights of the edges and ensuring that no cycles are formed.



DIFFERENCE BETWEEN PRIM'S AND Kruskal's algorithm

The main differences between Prim's algorithm and Kruskal's algorithm lie in their approaches to constructing the minimum spanning tree (MST) and the order in which they consider edges.

Here are the key differences between Prim's algorithm and Kruskal's algorithm:

Approach :-

- Prim's algorithm follows a grow-the-tree approach. It starts with an arbitrary vertex and gradually adds edges to connect the tree until all vertices are included. It grows the MST from a single vertex.
- Kruskal's algorithm follows a merge-the-trees approach. It starts with an empty set of edges and iteratively adds the smallest weighted edges that do not create cycles. It merges the MST by considering all edges independently.

1. **Edge Selection**:

- Prim's algorithm selects edges based on their weights that connect the current MST to a vertex outside the MST. It prioritizes the smallest weighted edges that extend the existing tree.
- Kruskal's algorithm selects edges based on their weights regardless of their connectivity to the current MST. It considers edges in ascending order of weights, ensuring that they do not create cycles when added.

2. **Graph Traversal**:

- Prim's algorithm explores the graph starting from a single vertex and extends the MST by connecting the current tree to a vertex outside the MST. It relies on the connectivity of the graph to grow the MST.
- Kruskal's algorithm explores the entire graph by considering all edges independently. It keeps track of disjoint sets to determine whether adding an edge creates a cycle or not.

3. **Efficiency:**

- Prim's algorithm is typically more efficient for dense graphs, where the number of edges is close to the maximum possible number of edges. It has a time complexity of O(V ∧ 2), where V is the number of vertices.
- Kruskal's algorithm is typically more efficient for sparse graphs, where the number of edges is much smaller than the maximum possible number of edges. It has a time complexity of O(E log E), where E is the number of edges.

In summary, Prim's algorithm grows the MST from a single vertex, selects edges based on their weights connected to the current MST, and explores the graph by extending the tree. Kruskal's algorithm merges the MST by selecting edges based solely on their weights, considers all edges independently, and explores the entire graph. The choice between the two algorithms depends on the characteristics of the graph and the efficiency requirements of the problem at hand.

SINGLE SOURCE SHORTEST PATH

A single-source shortest path algorithm is a type of algorithm that calculates the shortest path from a single source vertex to all other vertices in a weighted graph. It aims to find the most efficient path in terms of the sum of edge weights between the source vertex and all other vertices.

The problem of finding the shortest path arises in various real-world scenarios, such as navigation systems, network routing, and transportation planning. Single-source shortest path algorithms provide a solution by determining the shortest path for each vertex in the graph from a given source vertex.

One of the most well-known algorithms for solving the single-source shortest path problem is Dijkstra's algorithm. Dijkstra's algorithm starts from the source vertex and iteratively explores neighboring vertices, updating the shortest path distance to each vertex as it progresses. It uses a priority queue or a min-heap data structure to efficiently select the vertex with the shortest distance for exploration at each step.

What is the difference between single source shortest path and minimum spanning tree

The minimum spanning tree (MST) and single-source shortest path (SSSP) are two different concepts related to graph theory and optimization problems. Here are the key differences between them:

Minimum Spanning Tree (MST):

- The MST is a tree that spans all the vertices of a connected, weighted graph.
- It aims to find the subset of edges that form a tree with the minimum total weight.
- The MST does not have any cycles and includes all the vertices of the original graph.
- The MST is useful for applications such as network design, where the goal is to connect all vertices with the least possible total edge weight.
- Examples of MST algorithms include Prim's algorithm and Kruskal's algorithm.

Single-Source Shortest Path (SSSP):

- SSSP focuses on finding the shortest path from a single source vertex to all other vertices in a graph.
- It calculates the minimum total weight or cost of the path from the source vertex to each destination vertex.
- The SSSP problem can be solved for both directed and undirected graphs, with or without negative edge weights.
- SSSP algorithms determine the shortest path distances and may also track the actual paths themselves.
- Well-known SSSP algorithms include Dijkstra's algorithm for non-negative edge weights and the Bellman-Ford algorithm for graphs with negative edge weights.

In summary, the main difference lies in the objective of these algorithms. Minimum spanning tree algorithms seek to find a connected tree that spans all vertices with the minimum total weight, while single-source shortest path algorithms focus on finding the shortest paths from a single source vertex to all other vertices in terms of total weight or cost.

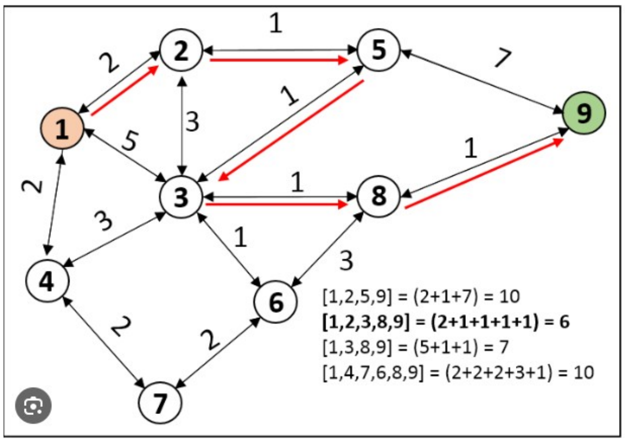
Dijkstra's Algorithm

Dijkstra's Algorithm is a popular algorithm used to find the shortest path between two nodes in a graph. It is named after its inventor, Edsger Dijkstra, a Dutch computer scientist.

Why do we need Dijkstra's Algorithm? Imagine you have a network of connected nodes, like cities connected by roads, and you want to find the shortest path from one node to another. Dijkstra's Algorithm comes in handy for solving such problems. It helps us find the most efficient route by considering the distances or costs associated with each connection between nodes.

Dijkstra's algorithm is a popular algorithm for finding the shortest path from a single source vertex to all other vertices in a weighted graph. It operates on graphs with non-negative edge weights and guarantees correct results when executed properly. Here's a step-by-step explanation of how Dijkstra's algorithm works:

- **Initialization:**
  - Set the distance of the source vertex to 0 and all other vertices to infinity.
  - Create an empty set of visited vertices.
- **Select the vertex with the minimum distance:**
  - Choose the vertex with the smallest distance among the vertices not yet visited.
  - Initially, the source vertex will be selected first.
- **Explore neighbors and update distances :-**
  - For the selected vertex, consider all its neighboring vertices.
  - Calculate the tentative distance from the source vertex to each neighboring vertex.
  - If the tentative distance is smaller than the current distance, update the distance.
- **Mark the current vertex as visited:**
  - Once all the neighbors of the current vertex have been processed, mark it as visited.
  - Adding a vertex to the visited set ensures that its shortest path has been determined.
- **Repeat steps 2-4:**
  - Continue the process by selecting the vertex with the smallest distance among the unvisited vertices.
  - Repeat steps 3 and 4 for the newly selected vertex.
  - Continue until all vertices have been visited or the destination vertex has been reached.
- **Termination:**
  - The algorithm terminates when all vertices have been visited, or when the destination vertex has been visited.
  - At this point, the algorithm has computed the shortest path distances from the source vertex to all other vertices.
- **Trace back the shortest path:**
  - Once the algorithm terminates, the shortest path from the source vertex to any other vertex can be determined.
  - Starting from the destination vertex, follow the path of minimum distances back to the source vertex.
  - This will give the shortest path from the source vertex to the destination vertex.



Dijkstra's algorithm employs a greedy strategy by continuously selecting the vertex with the minimum distance. It gradually builds the shortest paths from the source vertex to other vertices by iteratively exploring and updating distances. The algorithm guarantees optimality because it always chooses the vertex with the smallest distance, ensuring that the shortest path to each vertex is determined before considering longer paths.

It's worth noting that Dijkstra's algorithm may not work correctly if the graph contains negative edge weights. In such cases, alternative algorithms like the Bellman-Ford algorithm should be used.

CHAPTER FOUR

DYNAMIC PROGRAMMING

Dynamic programming is a problem-solving technique that involves breaking down complex problems into overlapping sub problems, solving each subproblem only once, and storing the solutions for efficient retrieval and reuse. It aims to optimize the time complexity of solving a problem by eliminating redundant computations.

In dynamic programming, the emphasis is on finding optimal solutions to problems by making use of previously computed subproblem solutions. It is particularly useful when a problem can be divided into overlapping sub problems and the optimal solution to the problem can be expressed in terms of the optimal solutions to its sub problems.

The dynamic programming approach typically follows these steps:

1. **Identify the problem and define the objective:** Clearly understand the problem and determine the goal or objective that needs to be achieved. Identify the specific constraints or criteria that need to be considered.
2. **Formulate the recursive relationship:** Break down the problem into smaller subproblems and define the relationship between the original problem and its subproblems. Express the optimal solution to the problem in terms of the optimal solutions to its subproblems.
3. **Define the base case(s):** Identify the simplest subproblems that can be solved directly without further recursion. These base cases serve as the starting point for building the solution.
4. **Design the dynamic programming algorithm:** Use memoization or tabulation to store the solutions to subproblems as they are computed. This enables efficient retrieval and reuse of the solutions instead of recomputing them multiple times.
5. **Compute the solution:** Apply the dynamic programming algorithm to solve the problem. Start with the base cases and iteratively compute the solutions to larger subproblems until the optimal solution to the original problem is obtained.

Dynamic programming is often used to solve optimization problems, such as finding the shortest path, maximizing or minimizing a value, or determining the optimal arrangement of objects. It can also be applied to problems involving sequencing, scheduling, resource allocation, and many other domains.

The key idea behind dynamic programming is to break down a problem into smaller overlapping subproblems, solve them once, and store the solutions for efficient retrieval. By avoiding redundant computations, dynamic programming can significantly improve the efficiency of solving complex problems and provide optimal solutions.

Here are some well-known algorithms that follow the dynamic programming approach:

- ➔ Fibonacci Sequence:
- ➔ Knapsack Problem:

Dynamic programming is a problem-solving technique that is used to solve optimization problems by breaking them down into smaller subproblems and combining their solutions. It is called "dynamic programming" because it involves breaking down a problem into a series of simpler and overlapping subproblems, and then solving each subproblem just once, saving its solution in a table.

The development of a dynamic programming algorithm typically involves the following four steps:

1. **Characterize the structure of an optimal solution:**

- Identify the key elements or choices that make up an optimal solution.
- Determine how these choices relate to subproblems and the overall problem.

2. **Recursively define the value of an optimal solution:**

- Define the value of the optimal solution recursively in terms of the values of smaller subproblems.
- Express the value of the optimal solution in a way that allows it to be computed efficiently.

3. **Compute the value of an optimal solution in a bottom-up fashion:**

- Start with the smallest subproblems and build up to the solution of the overall problem.
- Solve each subproblem only once and store its solution in a table for future reference.

4. Construct an optimal solution from computed information:

- If needed, maintain additional information during the computation to facilitate the construction of the optimal solution.
- Use the stored solutions of subproblems to construct the optimal solution for the overall problem.

Dynamic programming relies on two key elements for its application:

1. **Optimal substructure:**

- An optimization problem exhibits optimal substructure if an optimal solution to the problem can be constructed from optimal solutions of its subproblems.
- This property allows us to build up the solution to the original problem using the solutions of smaller subproblems.

2. **Overlapping subproblems:**

- Overlapping subproblems refer to the property that a recursive algorithm for a problem repeatedly solves the same subproblems instead of generating new ones.
- Dynamic programming takes advantage of this property by solving each subproblem only once and storing its solution for future reference.

By using optimal substructure and overlapping subproblems, dynamic programming algorithms can efficiently solve optimization problems. The time complexity of a dynamic programming algorithm depends on the number of subproblems and the number of choices examined for each subproblem.

It is important to note that dynamic programming and greedy algorithms are different approaches. While both exploit optimal substructure, they differ in terms of when choices are made. In dynamic programming, choices are made after solving subproblems, whereas in greedy algorithms, choices are made before solving subproblems.

In summary, dynamic programming is a technique used to solve optimization problems by breaking them down into smaller subproblems and solving them efficiently by exploiting optimal substructure and overlapping subproblems. It involves characterizing the structure of an optimal solution, defining the value of an optimal solution recursively, computing the value of the optimal solution, and constructing the optimal solution from computed information.

ALL PAIRS SHORTEST PATH (APSP)

The All Pairs Shortest Path (APSP) algorithm is a fundamental graph algorithm used to find the shortest paths between all pairs of vertices in a weighted directed graph. It provides a matrix representation of the shortest path distances between every pair of vertices.

The APSP algorithm solves the problem of finding the shortest path between every pair of vertices in a graph by building on the concept of the Single-Source Shortest Path (SSSP) algorithm, such as Dijkstra's algorithm or the Bellman-Ford algorithm.

Here is a step-by-step explanation of the APSP algorithm :-

1. **Input:** The algorithm takes as input a weighted directed graph, where each edge has an associated weight or cost. The graph can be represented as an adjacency matrix or an adjacency list.

2. **Initialization**: Create a matrix of size n x n, where n is the number of vertices in the graph. Initialize the matrix with the direct edge weights between vertices, i.e., if there is an edge between vertex i and vertex j, then the corresponding entry in the matrix will be the weight of that edge. If there is no edge between vertices i and j, then the corresponding entry in the matrix will be infinity.

3. **Dynamic Programming :-** The APSP algorithm utilizes the concept of dynamic programming to solve the problem efficiently. It iteratively updates the matrix by considering intermediate vertices in the path.

a. For each vertex k from 1 to n :-

- Consider each pair of vertices (i, j) and check if the shortest path from vertex i to vertex j can be improved by including vertex k as an intermediate vertex.
- Update the entry matrix[i][j] with the minimum of the current value matrix[i][j] and the sum of matrix[i][k] and matrix[k][j]. This step checks if there is a shorter path by going through vertex k.

4. Repeat Step 3 for all values of k from 1 to n. This ensures that the algorithm considers all possible intermediate vertices and updates the matrix accordingly.

5. Output: After completing the dynamic programming step, the final matrix represents the shortest path distances between all pairs of vertices in the graph.