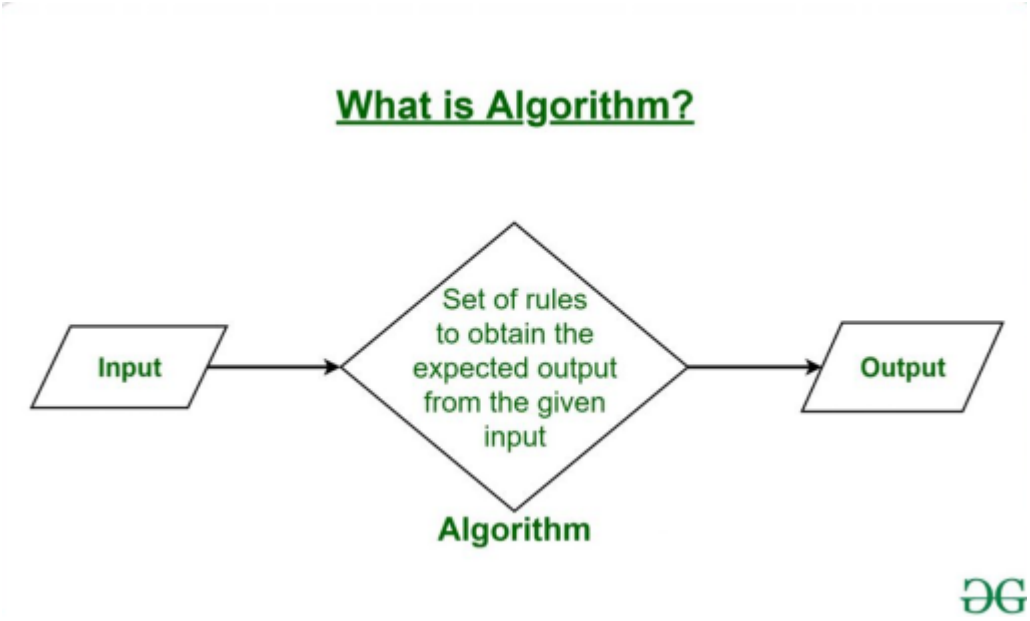
DAA CAE1

1. what is algo and its type

- A finite set of instruction that specifies a sequence of operation is to be carried out in order to solve a specific problem or class of problems is called an Algorithm
- **. An algorithm is a well-defined, step-by-step procedure or set of instructions for solving a particular problem or performing a specific task.**

Algorithms provide a systematic way to solve problems, and they are a fundamental concept in computer science and mathematics. They serve as the foundation for creating efficient and effective software solutions.



Brute Force Algorithms:

Definition: These algorithms try all possible solutions to find the answer. They are simple and straightforward but often inefficient.

Example: Linear search, Bubble sort.

Divide and Conquer Algorithms:

Definition: These algorithms divide the problem into smaller subproblems, solve each subproblem recursively, and then combine the solutions to solve the original problem.
Example: Merge sort, Quick sort, Binary search.

Greedy Algorithms:
Definition: These algorithms make the locally optimal choice at each step with the hope of finding the global optimum.
Example: Prim's algorithm for Minimum Spanning Tree, Kruskal's algorithm, Dijkstra's algorithm for shortest paths.

Advantages of Algorithms

1. **Efficiency**: Algorithms can be optimized for performance, allowing them to solve problems faster and with less resource consumption, leading to more efficient computing.
2. **Automation**: Algorithms enable automation of tasks, reducing the need for human intervention and minimizing errors, which is particularly beneficial in large-scale operations.
3. **Consistency**: Algorithms provide a consistent method of solving problems. Once implemented, they yield the same output for the same input, ensuring reliability and predictability.
4. **Problem Solving**: They can tackle complex problems that might be infeasible to solve manually, breaking down tasks into manageable steps and systematically addressing them.
5. **Reusability**: Well-designed algorithms can be reused across different applications and domains, allowing for quicker development and reduced costs in software engineering.

Disadvantages of Algorithms

1. **Complexity**: Some algorithms can be highly complex, making them difficult to understand, implement, or maintain, which can lead to increased development time and costs.
2. **Resource Intensity**: Certain algorithms may require significant computational resources (CPU time, memory), which can be impractical in resource-constrained environments.
3. **Rigidity**: Algorithms operate based on predefined rules and parameters. They can struggle with unstructured problems or situations that require human intuition and judgment.
4. **Limited Flexibility**: Once an algorithm is implemented, making changes or optimizations can be challenging, especially if the initial design does not accommodate future modifications.

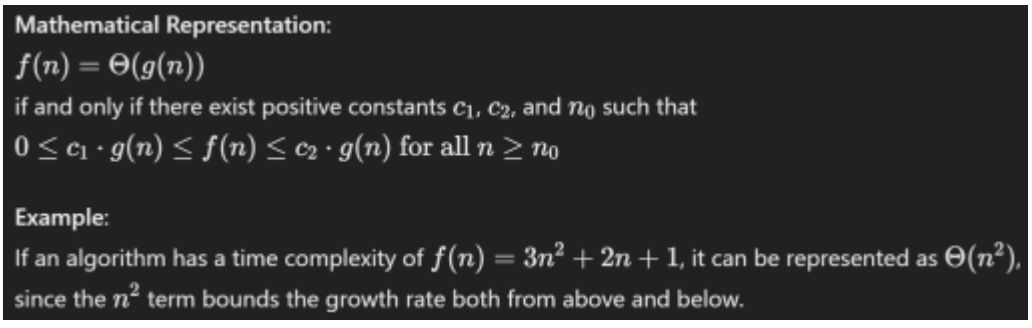
5. **Dependency on Data Quality**: The effectiveness of an algorithm is heavily dependent on the quality and accuracy of the data it processes. Poor data can lead to misleading results and ineffective solutions.

2. Asymptotic notation

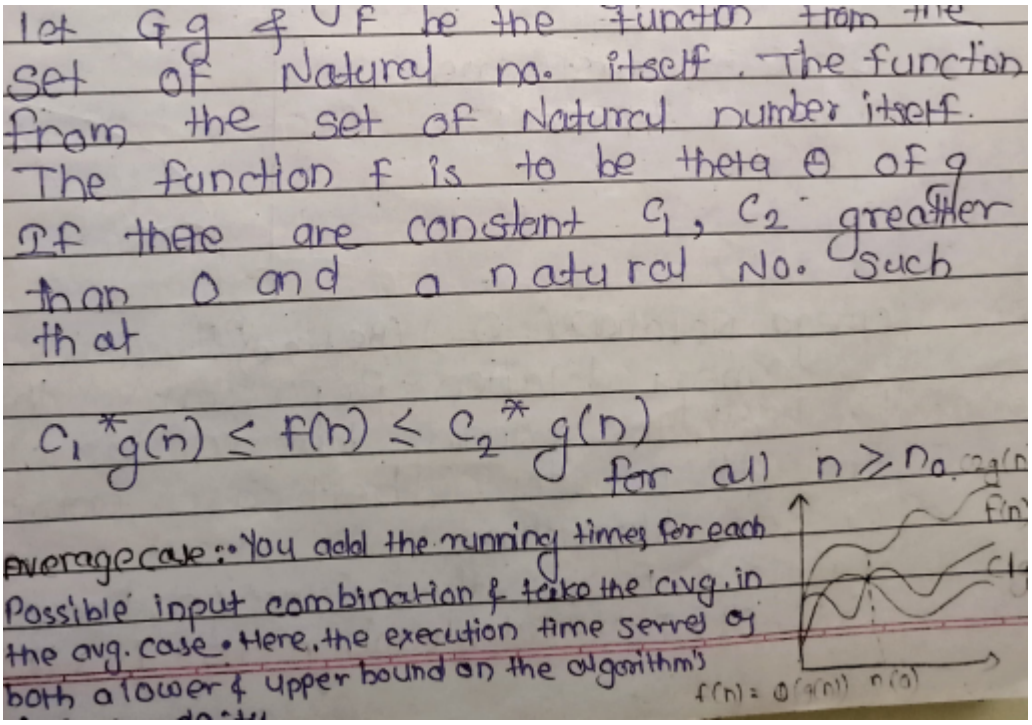
- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value
- efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations
- ASYMPTOTIC NOTATION provide a standardized way to analyze and compare algorithms in terms of their worst-case, best-case, and average-case behavior, and they form the foundation of algorithmic complexity analysis.
- There are mainly three asymptotic notations:
- □ Big-O notation -- It measures the worst case time complexity, Big O notation is used to describe the upper bound of an algorithm in terms of the input size
- □ Omega notation -- It measures the best case time complexity , Omega notation represents the lower bound of an algorithm in terms of the input size
- □ Theta notation -- describe the average case, encompassing both the upper and lower bounds. It describes the exact growth rate of the algorithm's running time as the input size increases.

3. Explain theta notation with example

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.
- describe the average case, encompassing both the upper and lower bounds
- Theta notation provides a tight bound on the time complexity of an algorithm
- It describes the exact growth rate of the algorithm's running time as the input size increases.
- it is reflexive , symmetric and transitive



-



4. Omega notation

- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time.
- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.
- It gives an asymptotic lower limit on how fast the algorithm can possibly perform with respect to the input size.
- For instance, if an algorithm's time complexity is Ω(n), it implies that the algorithm's running time is at least linear with respect to the input size.
- it is reflexive and transitive but not symmetric

Mathematical Representation:

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

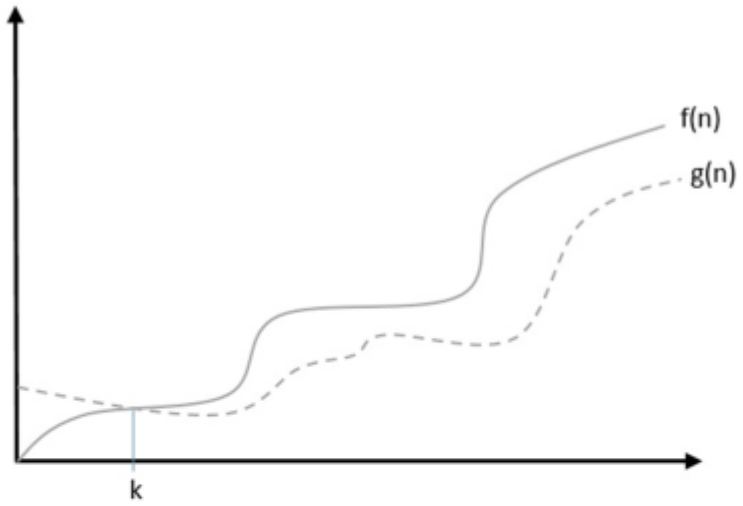
if and only if there exist positive constants c and n_0 such that

$0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$

Example:

If an algorithm has a time complexity of $f(n) = 3n^2 + 2n + 1$, it can be represented as $\Omega(n^2)$, since the n^2 term provides the minimum growth rate.

-



For example, for a function ***f(n)***

Ω(f(n)) ≥ { g(n) : there exists c > 0 and n0 such that g(n) ≤ c.f(n) for all n > n0. }

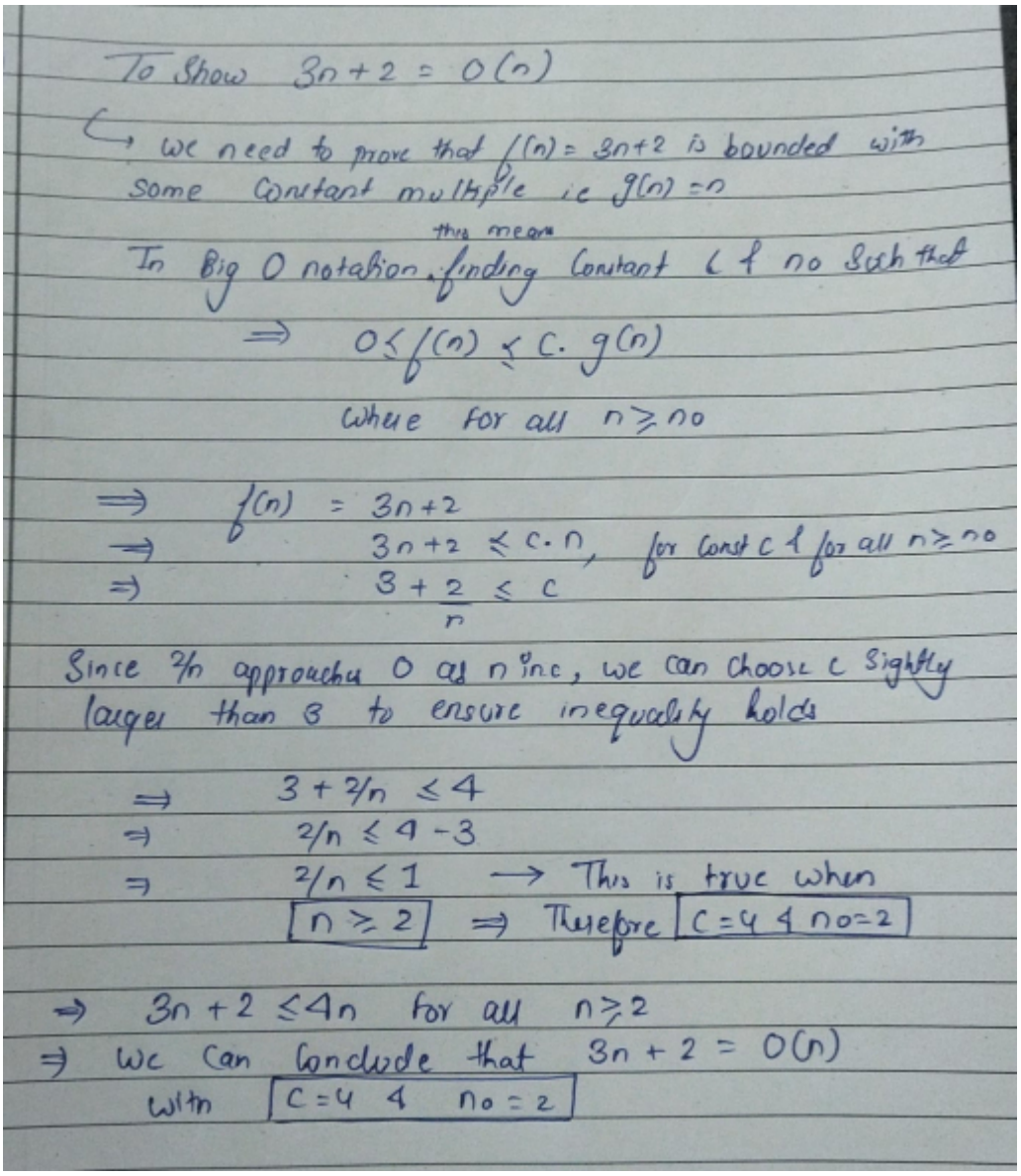
Example

Let us consider a given function, $f(n) = 4.n^3+10.n^2+5.n+1$

Considering $g(n) = n^3$, $f(n) \geq 4.g(n)$ for all the values of $n > 0$.

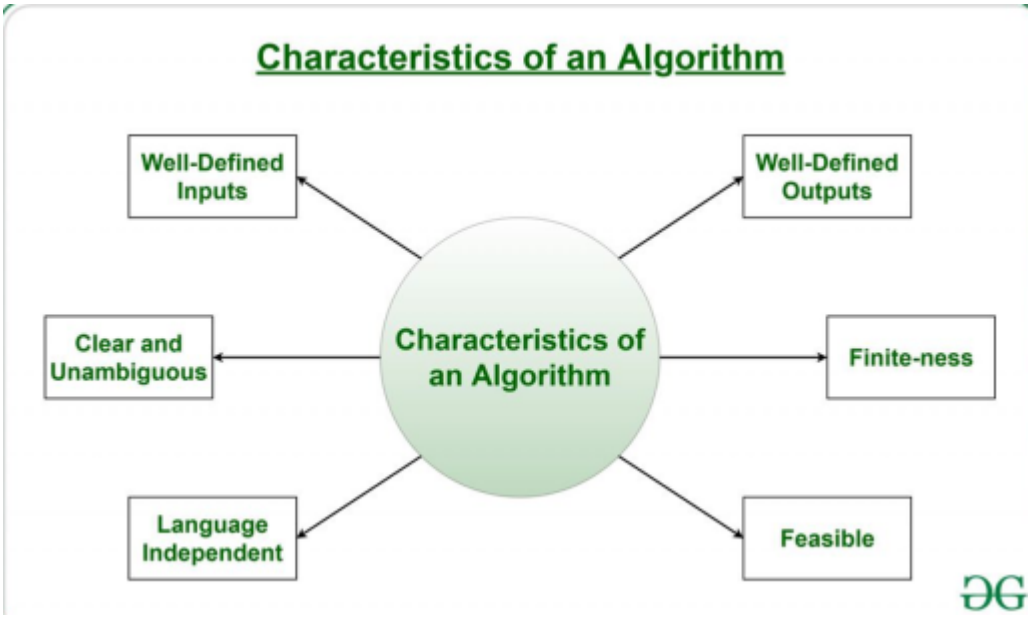
Hence, the complexity of $f(n)$ can be represented as $\Omega(g(n))$,i.e. $\Omega(n^3)$.

5.Show that 3n+2 = O(n) where f(n) = 3n+2 and g(n) = n.



-

6. characteristics and specification of algo



- **Clear and Unambiguous**: The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- **Well-Defined Outputs**: The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.
- **Finite-ness**: The algorithm must be finite, i.e. it should terminate after a finite time
- **Feasible**: The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- **Language Independent**: The Algorithm designed must be languageindependent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.
- **Effectiveness**. All operations to be accomplished must be sufficiently basic that they can be done exactly and in finite length

7. **Explain best, worst, average case analysis**

Best Case Analysis:

- Describes the scenario where the algorithm performs the minimum number of steps. It is the most optimistic case.
- Omega measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

- Example: In a linear search, the best case occurs when the target element is the first element in the list.
- equation and graph

Worst Case Analysis:

- Describes the scenario where the algorithm performs the maximum number of steps. It is the most pessimistic case.
- This represents the maximum amount of time an algorithm will take, considering the input that leads to the longest execution time
- Big O notation is used to describe the upper bound or worst-case time complexity of an algorithm in terms of the input size.
- Example: In a linear search, the worst case occurs when the target element is the last element in the list or not present at all.

Average Case Analysis:

- Describes the expected number of steps an algorithm takes, averaged over all possible inputs. It is usually calculated using probability and statistics.
- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.
- describe the average case, encompassing both the upper and lower bounds
- Example: In a linear search, if the target element is equally likely to be at any position, the average case occurs when the target is in the middle of the list.

8. Different type of control struture analysis

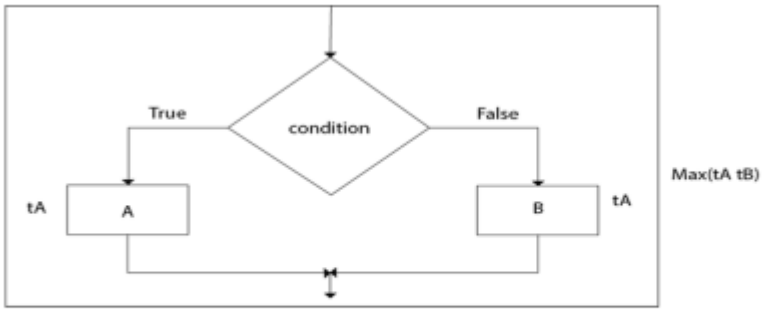
-- Control structures determine the flow of execution of the program

-- control structre involves examining the flow of control within an algorithm to understand its behavior, efficiency, and correctness.

-- This analysis focuses on various control structures such as sequencing,if then else, for loop, while loop

- sequencing -- Suppose our algorithm consists of two parts A and B. A takes time tA and B takes time tB for computation. The total computation "tA + tB " is according to the sequence rule. According to maximum rule, this computation time is (max (tA ,tB))
- if then else -- Execute different blocks of code based on conditions.

2. If-then-else:



The total time computation is according to the condition rule-"if-then-else." According to the maximum rule, this computation time is max (t_A,t_B).

for & while loops --: Repeatedly execute a block of code. focuses on Number of iterations, impact on time complexity

9. time and space complexity in detail

TIME COMPLEXITY :

- Time complexity is the computational complexity describing the amount of time required for the execution of an algorithm.
- Time complexity measures the time taken by every statement of the algorithm
- it highly depends on the size of processed data. Additionally, it helps to define the effectiveness of an algorithm and to evaluate its performance
- Time complexity is usually expressed using Big O notation, which provides an upper bound on the growth rate of the running time
- There are different time complexity such as best , average, worst case every case ahs its own time complexity
- constant -- O(1)
- log - O(log n)
- linear - O(n)
- quad - O(n^2)
- cubic

- expo - 2^O(n)

SPACE COMPLEXITY :

- space complexity refers to the amount of memory space an algorithm uses to complete its execution as a function of the input size.
- It includes both the memory required for storing the input data and any additional memory used during the algorithm's execution, such as variables, data structures, and function call stacks.
- Space complexity is important because excessive memory usage can lead to performance issues, especially in resource-constrained environments.
- It's also a critical consideration when dealing with large datasets.
- Space complexity is often denoted using Big O notation in terms of the input size `n`.
- general equation for space complexity in terms of Big O notation: Space Complexity = f(n), where f(n) represents the upper bound of the memory used by the algorithm as a function of the input size `n`.
- Constant Space Complexity (O (1))
- Linear Space Complexity (O(n))
- Quadratic Space Complexity (O(n^2)):
- Logarithmic Space Complexity (O(log n)):
- Exponential Space Complexity (O(2^n)):

diff:

Aspect	Space Complexity	Time Complexity
Definition	Measures the amount of memory space required by an algorithm as a function of the input size.	Measures the amount of time an algorithm takes to complete as a function of the input size.
Focus	Concerned with memory usage, including both fixed and dynamic space allocations.	Concerned with the execution time and the number of operations performed.
Components	Considers both the auxiliary space (extra space) and the space required for input.	Considers the number of basic operations or steps taken during execution.
Measurement Units	Typically expressed in bytes, kilobytes, or in terms of Big O notation (e.g., O(n)).	Expressed in terms of time units (seconds, milliseconds) or in Big O notation (e.g., O(n)).
Importance	Important for applications with limited memory resources or when optimizing memory usage.	Crucial for performance-critical applications where speed is a priority.
Impacts	High space complexity can lead to inefficient memory use, potentially causing out-of-memory errors.	High time complexity can lead to longer execution times, affecting overall performance and responsiveness.
Trade-offs	Reducing space complexity may increase time complexity, as more computations might be required.	Reducing time complexity can lead to increased space complexity due to additional data structures or caches.
Example	Recursive algorithms can have high space complexity due to call stack usage.	Sorting algorithms can have varying time complexities based on the method used (e.g., O(n log n) for quicksort).
Optimization	Space-efficient algorithms are important in environments with limited memory (e.g., embedded systems).	Time-efficient algorithms are prioritized in real-time systems where quick responses are needed.

11. binary search

- Binary Search algorithm is an interval searching method that performs the searching in intervals only.
- The input taken by the binary search algorithm must always be in a sorted array
- Binary search is a fast search algorithm with run-time complexity of O(log n).
- This search algorithm works on the principle of divide and conquer
- Binary search looks for a particular key value by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. But if the middle item has a value greater than the key value, the right sub-array of the middle item is searched. Otherwise, the left sub-array is searched. This process continues recursively until the size of a subarray reduces to zero
- Step 1 − Select the middle item in the array and compare it with the key value to be searched. If it is matched, return the position of the median.

- Step 2 − If it does not match the key value, check if the key value is either greater than or less than the median value.
- Step 3 − If the key is greater, perform the search in the right subarray; but if the key is lower than the median value, perform the search in the left sub-array.
- Step 4 − Repeat Steps 1, 2 and 3 iteratively, until the size of subarray becomes 1.
- Step 5 − If the key value does not exist in the array, then the algorithm returns an unsuccessful search.

14. Describe maximum and minimum finding algorithm with example.

Algorithm to Find Maximum and Minimum:

1. Initialize two variables to hold the maximum and minimum values.
2. Traverse through the array and compare each element with the current maximum and minimum.
3. Update the maximum and minimum values accordingly.

Steps:

- Initialize: max = arr[0], min = arr[0]
- For each element in the array:
    - If the element > max, update max
    - If the element < min, update min

Example:Array: [3, 5, 7, 2, 8, -1, 4]

- Initial: max = 3, min = 3
- Traverse:
    - Compare 5: max = 5
    - Compare 7: max = 7
    - Compare 2: min = 2
    - Compare 8: max = 8
    - Compare -1: min = -1
    - Compare 4: no change

Result: max = 8, min = -1

17. Elaborate the concept of master method in recursion.

- The master theorem is a mathematical tool used to analyze the time complexity of certain types of recursive algorithms.
- It provides a framework for determining the time complexity of divide-andconquer algorithms that have a specific structure.
- The master theorem is particularly useful for solving recurrence relations that arise in algorithms like binary search, merge sort, and some divide-andconquer algorithms.
- The master theorem provides a formulaic approach to solving recurrence relations of the form: T(n) = a * T(n/b) + f(n),

  -- where: - T(n) represents the time complexity of the algorithm for an input of size n. -
  -- a - is the number of subproblems the algorithm is divided into. -
  -- n/b is the size of each subproblem. -
  -- f(n) is the time complexity of any extra work done outside of the recursive calls.

- The master theorem has three cases, each with conditions and outcomes that help determine the time complexity of the algorithm:
- 1. Case 1: If f(n) is O(n^c) where c < log_b(a):
  - The work done outside of the recursive calls dominates the time complexity.
  - Time complexity: T(n) = Θ(n^c).
- 2. Case 2: If f(n) is Θ(n^c * log^k n) where c = log_b(a):
  - The work done outside of the recursive calls is equally significant as the recursive calls. -
  Time complexity: T(n) = Θ(n^c * log^(k+1) n).
- 3. Case 3: If f(n) is Ω(n^c) where c > log_b(a), and a * f(n/b) <= k * f(n) for some constant k < 1 and sufficiently large n:
  - The recursive calls dominate the time complexity.
  - Time complexity: T(n) = Θ(f(n)).
- master theorem is applicable only to specific forms of recurrence relations. If the recurrence relation doesn't match any of the master theorem cases, other techniques or methods might be required for analysis.

18. Divide and conqure strategy

- Divide and conquer is a powerful algorithmic paradigm used to solve complex problems by breaking them down into smaller, more manageable subproblems.
- The key idea is to divide the problem into smaller instances, solve these instances recursively, and then combine their solutions to obtain the solution to the original problem.
- This approach often leads to elegant and efficient algorithms.
- Divide and Conquer algorithm consists of a dispute using the following three steps.
  1. Divide the original problem into a set of subproblems.
  2. Conquer: Solve every subproblem individually, recursively.
  3. Combine: Put together the solutions of the subproblems to get the solution to the whole problem.
- The specific computer algorithms are based on the Divide & Conquer approach:
  1. Maximum and Minimum Problem
  2. Binary Search
  3. Sorting (merge sort, quick sort)
  4. Tower of Hanoi

20. itertion and recursion

Aspect	Iteration	Recurrence
Definition	Repeating a set of instructions using loops until a condition is met.	A function calling itself with smaller instances of the problem until a base case is met.
Control Structure	Uses loops (`for`, `while`).	Uses function calls.
Memory Usage	Generally more memory efficient as it uses a fixed amount of memory space.	Can be less memory efficient due to additional stack space used for function calls.
Ease of Understanding	Can be more intuitive for problems that involve straightforward repetition.	Can be easier to understand for problems with a natural recursive structure.
Performance	Typically faster due to lower overhead.	Can be slower due to overhead of multiple function calls.
Examples	Summing first n natural numbers using a loop.	Summing first n natural numbers using a recursive function.

Aspect	Iteration Example	Recurrence Example
Code	```python	```python
	def iterative_sum(n):	def recursive_sum(n):
	total = 0	if n == 1:
	for i in range(1, n + 1):	return 1
	total += i	else:
	return total	return n + recursive_sum(n - 1)
	``	```
Steps	1. Initialize `total` to 0.	1. Base case: if `n` is 1, return 1.
	2. Loop from 1 to `n`, adding each number to `total`.	2. Recursive case: return `n` + `recursive_sum(n - 1)`.
	3. Return `total`.	3. Each recursive call reduces the problem size by 1 until the base case is reached.

Summary:

Aspect	Iteration	Recurrence
Sum of first n natural numbers	`iterative_sum(n)`	`recursive_sum(n)`
Steps	Loop through 1 to `n`	Function calls itself with $n - 1$
Memory	Fixed amount (O(1))	Depends on the depth of recursion (O(n))

iteration :

```
int iterative_sum(int n) {
    int total = 0;
    for (int i = 1; i <= n; ++i) {
       total += i;
     }
     return total;
    }
```

recursive :

```
int recursive_sum(int n) {
if (n == 1) {
return 1;
}
else {
return n + recursive_sum(n - 1);
}
```

}