MODEL QUESTION PAPER

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

FIRST SEMESTER B. TECH DEGREE EXAMINATION, DECEMBER 2024

Course Code: UCEST105

Course Name: Algorithmic Thinking with Python

Max. Marks: 60 Duration: 2 hours 30 minutes

PART A

Answer all questions. Each question carries 3 marks

1. How do you use a decomposition strategy to design a menu-driven calculator that supports four basic arithmetic operators - addition, subtraction, multiplication, and division?(3marks) [CO 1]

To design a menu-driven calculator using a **decomposition strategy**, you break the program into smaller, manageable functions or modules, each handling a specific aspect of the calculator's functionality. Here's how you can approach it:

1. **Main Menu Function:**
   - Design a function to display the menu options (e.g., addition, subtraction, multiplication, division, and exit).
   - Accept the user's choice and route them to the appropriate function for the selected operation.
2. **Arithmetic Operation Functions:**
   - Create separate functions for each arithmetic operation:
     - **Addition Function:** Accept two numbers and return their sum.
     - **Subtraction Function:** Accept two numbers and return their difference.
     - **Multiplication Function:** Accept two numbers and return their product.
     - **Division Function:** Accept two numbers, ensure the denominator is not zero, and return the quotient.
3. **Input and Output Management:**
   - Implement a function to handle input (e.g., read two numbers from the user).
   - Include validation for division to prevent division by zero.
   - Use a function to display results or error messages.
4. **Loop for Continuous Operation:**

- Use a loop in the main function to allow the user to perform multiple operations until they choose to exit.

By decomposing the program into these functional units, you achieve modularity, reusability, and simplicity in design and debugging.

2. A mad scientist wishes to make a chain out of plutonium and lead pieces. There is a problem, however. If the scientist places two pieces of plutonium next to each other, BOOM! The question is, in how many ways can the scientist safely construct a chain of length ***n*** ? ?(3marks) [CO 4]

This problem is a classic combinatorial problem where we want to count the number of ways to construct a chain of length nnn using two types of pieces (Plutonium PPP and Lead LLL), such that no two PPP's are adjacent.

Approach

To solve this problem, we can use **dynamic programming**. Let:

- f(n): The number of safe ways to construct a chain of length n.

Base Cases:

- f(1)=2: There are two ways to construct a chain of length 1: P or L.
- f(2)=3 There are three safe configurations for a chain of length 2: PL,LP,LL.

Recurrence Relation:

for n≥3:

- If the n th piece is L, the first n−1 pieces can be in any valid configuration (there are f(n−1) ways for this).
- If the n-th piece is P, the (n−1)-th piece must be L, and the first n−2 pieces can be in any valid configuration (there are f(n−2) ways for this).

Thus, the recurrence relation is:

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

This is the Fibonacci sequence shifted by one position.

General Formula:

The number of ways to safely construct the chain of length n is the (n+1)-th Fibonacci number.

Example Calculations:

- f(1)=2
- f(2)=3
- f(3)=f(2)+f(1)=3+2=5
- f(4)=f(3)+f(2)=5+3=8

So, for n=4, there are 8 ways to safely construct the chain.

This approach gives us an efficient solution using the Fibonacci sequence!

3.

Write a **case** statement that will examine the value of ***flag*** and print one of the following messages, based on its value. | 3 | (3)

Flag value	Message
1	Hot
2	Luke warm
3	Cold
Any other value	Out of range

```
BEGIN
  READ flag  // Input the value of flag

  SWITCH flag
    CASE 1:
      PRINT "Hot"
      BREAK
    CASE 2:
      PRINT "Luke warm"
      BREAK
    CASE 3:
      PRINT "Cold"
      BREAK
    DEFAULT:
      PRINT "Out of range"
  END SWITCH
END
```

Explanation:

1. **Input the value**: Read the value of flag from the user or another source.
2. **Switch case**:

    - If flag is 1, print "Hot".
    - If flag is 2, print "Luke warm".
    - If flag is 3, print "Cold".
    - For any other value, the DEFAULT case handles it by printing "Out of range".
3. **End the program**: Exit after the appropriate message is printed.

This pseudocode ensures a clear and structured approach to evaluating the flag value.

4. Draw a flowchart to print the numbers that are divisible by 4 but not by 3 in a list of ***n*** positive numbers. [CO-3] [3-marks]

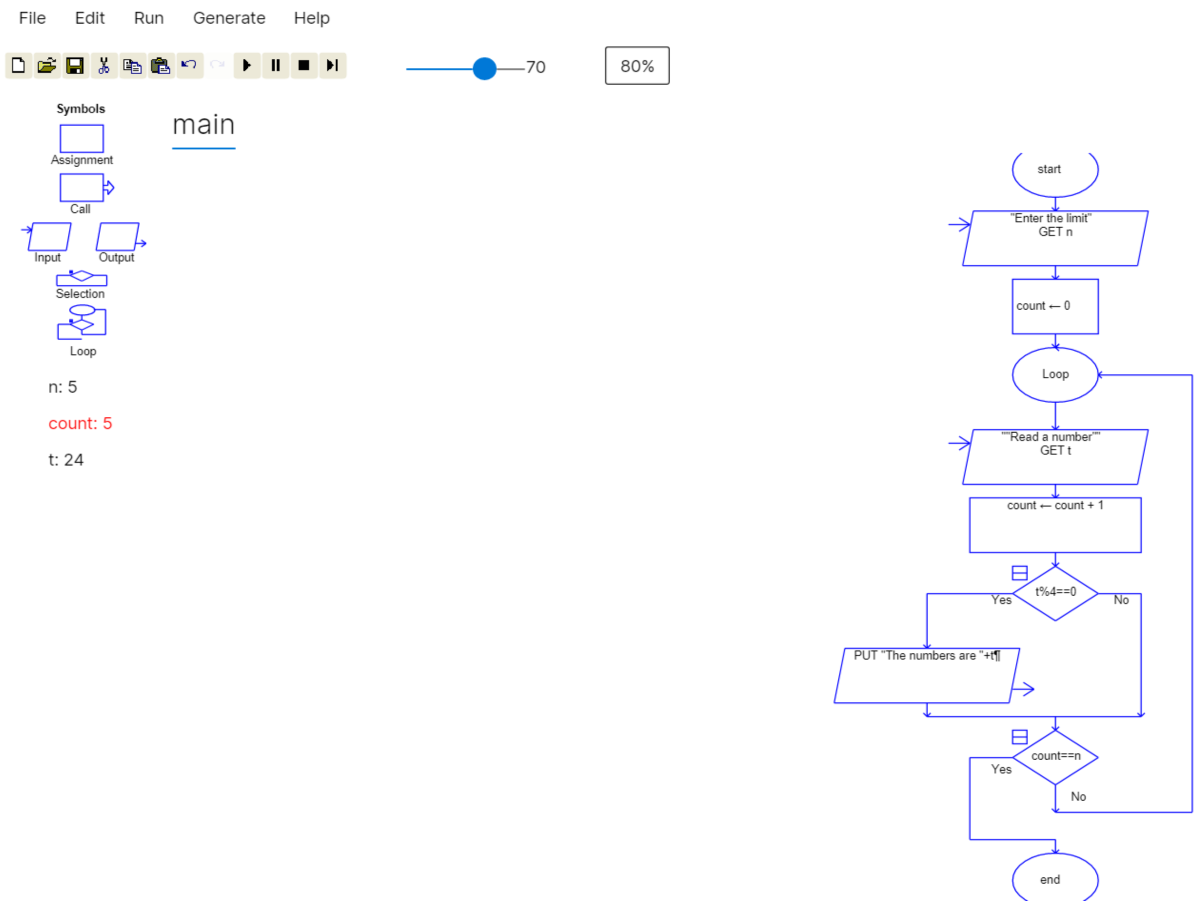
Start

Input the list of n positive numbers (Parallelogram shape)

For each number in the list (Rectangle shape, representing a loop)

- **Check if the number is divisible by 4 but not by 3** (Diamond shape for decision)
    - If **Yes**, **Print the number** (Parallelogram shape)
    - If **No**, skip to the next number

End



n: 5

count: 5

t: 24

5. Identify and rectify the problem with the following recursive definition to find the greatest common divisor of two positive integers. ABC (n , m) if n == 2 return m else return ABC(m , n mod m) (3 marks) [CO 4]

The given recursive definition for finding the **Greatest Common Divisor (GCD)** of two positive integers n and m contains some issues:

Problem in the given definition:

The base case in the recursive definition is flawed:

ABC(n, m) if n == 2 return m

- This is incorrect because the base case for finding the GCD should be when n or m is zero (i.e., n==0 or m==0). The GCD of any number and zero is the number itself. The condition n == 2 is not a correct or general base case for GCD computation.

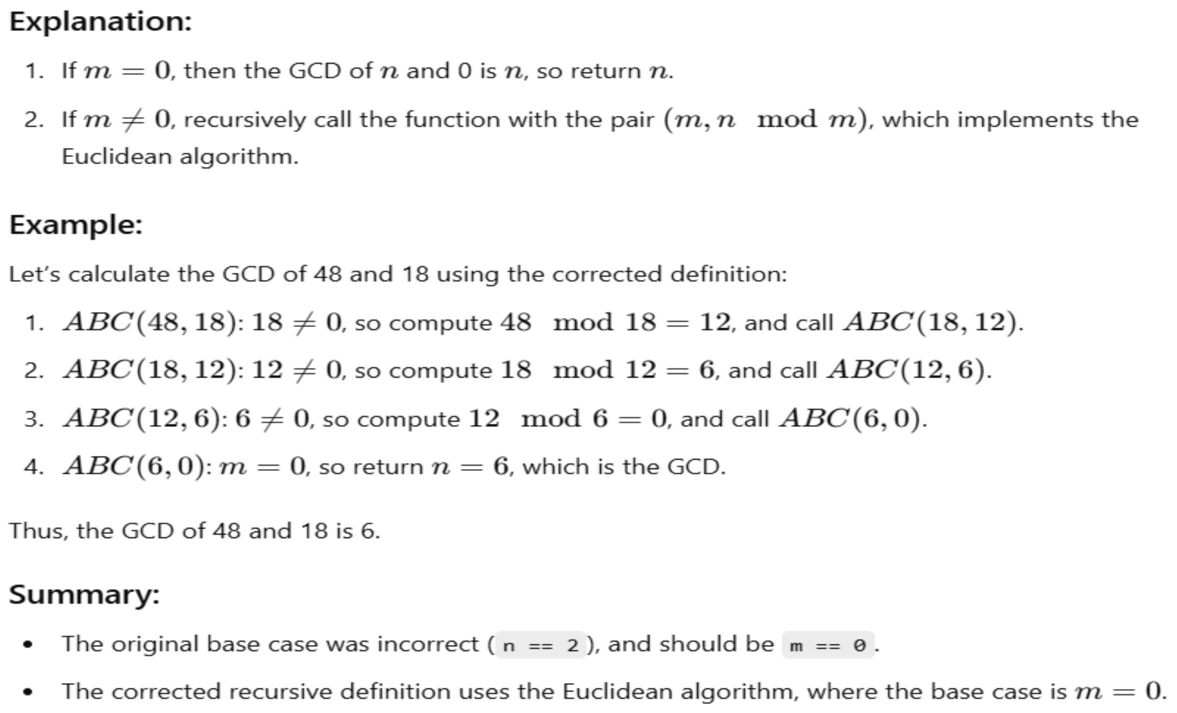
Correct Recursive Definition (Euclidean Algorithm):

The correct recursive definition for the **Euclidean Algorithm** to find the GCD is:

1. **Base Case**: If m=0, then return n, because the GCD of n and 0 is n.
2. **Recursive Case**: Otherwise, recursively call the function with ABC(m,n mod m)

   So, the correct recursive definition would look like:

```
ABC(n, m)
 if m == 0 then return n
 else return ABC(m, n mod m)
```



Explanation:

1. If $m = 0$, then the GCD of n and 0 is n, so return n.
2. If $m \neq 0$, recursively call the function with the pair $(m, n \mod m)$, which implements the Euclidean algorithm.

Example:

Let's calculate the GCD of 48 and 18 using the corrected definition:

1. $ABC(48, 18)$: $18 \neq 0$, so compute $48 \mod 18 = 12$, and call $ABC(18, 12)$.
2. $ABC(18, 12)$: $12 \neq 0$, so compute $18 \mod 12 = 6$, and call $ABC(12, 6)$.
3. $ABC(12, 6)$: $6 \neq 0$, so compute $12 \mod 6 = 0$, and call $ABC(6, 0)$.
4. $ABC(6, 0)$: $m = 0$, so return $n = 6$, which is the GCD.

Thus, the GCD of 48 and 18 is 6.

Summary:

- The original base case was incorrect (`n == 2`), and should be `m == 0`.
- The corrected recursive definition uses the Euclidean algorithm, where the base case is $m = 0$.

6. Write a recursive procedure to search for a ***key*** in a list of ***n*** integers.

To write a **recursive procedure** to search for a key in a list of n integers, we can follow these steps:

Recursive Approach:

- **Base Case**: If the list is empty (or the index exceeds the list length), return **False** because the key is not found.
- **Recursive Case**: If the current element matches the key, return **True**. Otherwise, recursively search the rest of the list by moving to the next index.

Pseudocode for the Recursive Search:

```
Procedure SearchKey(List, key, index)
  if index >= length(List) then
    return False  // Base case: if the index exceeds the list length, key is not found
  else if List[index] == key then
    return True  // Key found, return True
  else
    return SearchKey(List, key, index + 1)  // Recursively search the next element


// Example usage: Search for key in the list starting at index 0
result = SearchKey(List, key, 0)
```

Explanation:

1. **Base Case**:
   - If the index is greater than or equal to the length of the list (index >= length(List)), it means we have searched through the entire list and the key is not present, so we return False.
2. **Recursive Case**:
   - If the element at List[index] equals the key, return True (indicating the key is found).
   - Otherwise, call the function recursively with the next index (index + 1) to continue searching through the rest of the list.

Example:

Let's say the list is: [3, 5, 7, 9, 11] and the key to search is 7.

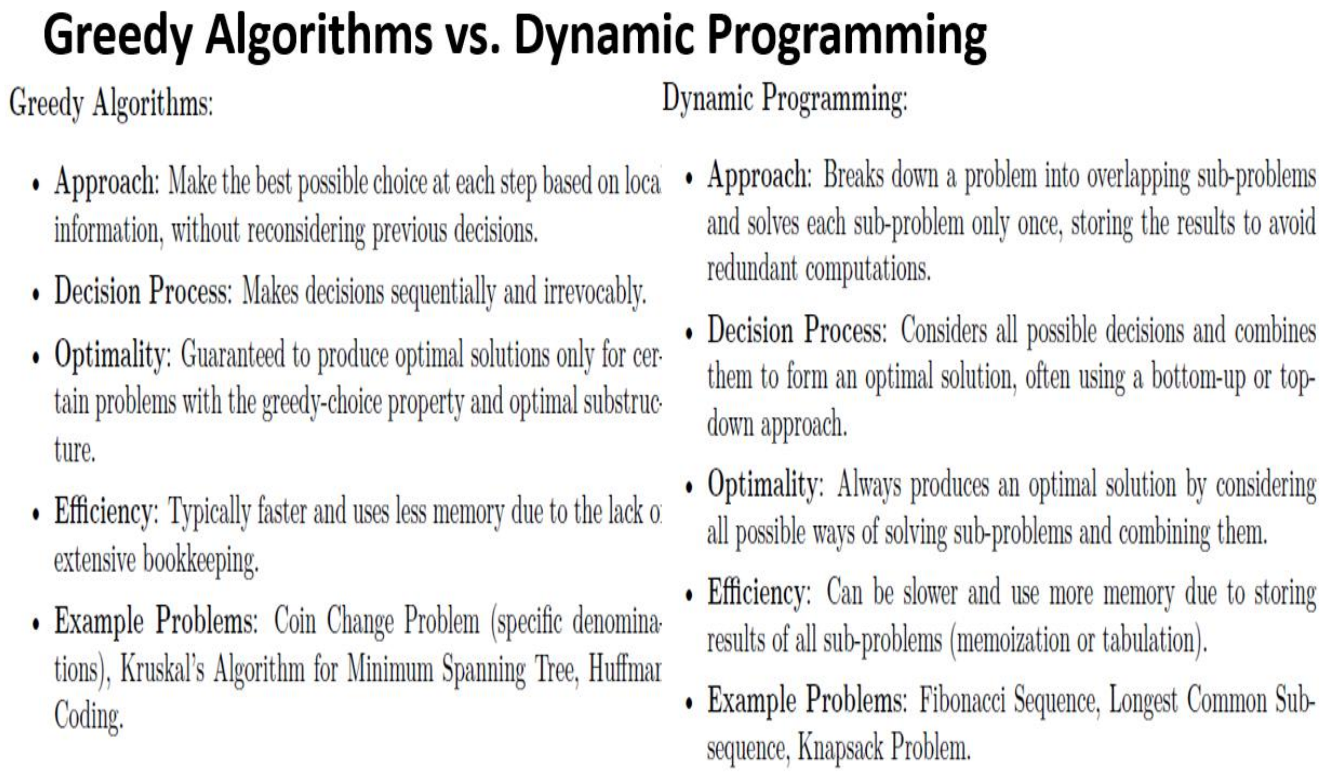
1. SearchKey([3, 5, 7, 9, 11], 7, 0) will check if List[0] == 7, which is false, then recursively call SearchKey([3, 5, 7, 9, 11], 7, 1).
2. SearchKey([3, 5, 7, 9, 11], 7, 1) checks List[1] == 7, which is false, then calls SearchKey([3, 5, 7, 9, 11], 7, 2).
3. SearchKey([3, 5, 7, 9, 11], 7, 2) checks List[2] == 7, which is true, so it returns True.

Thus, the function returns True, indicating that the key 7 was found in the list.

Time Complexity:

- **Best case**: O(1) (if the key is found at the first position).
- **Worst case**: O(n) (if the key is at the end of the list or not in the list).

7. Compare and contrast greedy and dynamic programming strategies. (3 marks) [CO 3]



Greedy Algorithms vs. Dynamic Programming

Greedy Algorithms:

- **Approach:** Make the best possible choice at each step based on loca information, without reconsidering previous decisions.
- **Decision Process:** Makes decisions sequentially and irrevocably.
- **Optimality:** Guaranteed to produce optimal solutions only for certain problems with the greedy-choice property and optimal substructure.
- **Efficiency:** Typically faster and uses less memory due to the lack o extensive bookkeeping.
- **Example Problems:** Coin Change Problem (specific denominations), Kruskal's Algorithm for Minimum Spanning Tree, Huffmar Coding.

Dynamic Programming:

- **Approach:** Breaks down a problem into overlapping sub-problems and solves each sub-problem only once, storing the results to avoid redundant computations.
- **Decision Process:** Considers all possible decisions and combines them to form an optimal solution, often using a bottom-up or top-down approach.
- **Optimality:** Always produces an optimal solution by considering all possible ways of solving sub-problems and combining them.
- **Efficiency:** Can be slower and use more memory due to storing results of all sub-problems (memoization or tabulation).
- **Example Problems:** Fibonacci Sequence, Longest Common Subsequence, Knapsack Problem.

8. Give the pseudocode for brute force technique to find the mode of elements in an array. Mode is the value that appears most frequently in the array. (3 marks) [CO 3]

The **brute force technique** to find the mode of an array involves iterating over the array to count the frequency of each element, then identifying the element with the highest frequency.

Pseudocode for Brute Force to Find the Mode:

```
Procedure FindMode(Array)
   maxFrequency = 0    // Initialize the maximum frequency
   mode = None         // Initialize the mode as None

   // Loop through each element in the array
   for i from 0 to length(Array) - 1 do
      currentElement = Array[i]
      frequency = 0

      // Count the frequency of currentElement
      for j from 0 to length(Array) - 1 do
         if Array[j] == currentElement then
            frequency = frequency + 1
      // Check if currentElement has higher frequency than previous mode
      if frequency > maxFrequency then
         maxFrequency = frequency
         mode = currentElement

   return mode
```

Explanation:

1. **Initialization**:
   - maxFrequency: This keeps track of the highest frequency found so far (initialized to 0).
   - mode: This stores the element that has the highest frequency (initialized to None).
2. **Outer Loop**:
   - We iterate over each element of the array using index i. For each element currentElement, we count its frequency by comparing it to every other element in the array.
3. **Inner Loop**:
   - For each element currentElement, we compare it to every other element in the array (using index j) and increment the frequency count if they match.
4. **Update the Mode**:
   - After counting the frequency of currentElement, we check if its frequency is greater than maxFrequency. If so, we update maxFrequency and set mode to the current element.
5. **Return the Mode**:
   - After all elements have been processed, mode will hold the value that appears most frequently in the array.

Example:

Consider the array [1, 2, 2, 3, 3, 3, 4].

1. Start with maxFrequency = 0 and mode = None.
2. Check element 1, its frequency is 1. Update maxFrequency = 1, mode = 1.
3. Check element 2, its frequency is 2. Update maxFrequency = 2, mode = 2.
4. Check element 3, its frequency is 3. Update maxFrequency = 3, mode = 3.
5. Check element 4, its frequency is 1, no update to maxFrequency or mode.
6. The final mode is 3, which appears most frequently.

Time Complexity:

- **Time Complexity**: O(n^2), because we have two nested loops: the outer loop runs n times, and for each element, the inner loop runs n times to count its frequency.
- **Space Complexity**: O(1), as we're only using a few variables for tracking frequencies and the mode.

		PART B		
		Answer any one full question from each module. Each question carries 9 marks		
		Module 1		
9		Walk through the six problem-solving steps to find the largest number out of three numbers.	1	(9)
10	a)	Your professor has given you an assignment on "Algorithmic thinking" to be submitted by this Wednesday. How do you employ means-end analysis to devise a strategy for completing your assignment before the deadline?	2	(5)
	b)	Name two current problems in your life that might be solved through a heuristic approach. Explain why each of these problems can be solved using heuristics.	1	(4)

9. The six problem-solving steps are a structured approach to solving problems methodically. Let's walk through these steps to find the **largest number out of three numbers**.

Problem: Find the largest number out of three numbers a, b, and c

1. Understand the Problem:

- We are given three numbers: aaa, bbb, and ccc.
- The goal is to identify the largest number among the three.
- The largest number is the one that is greater than or equal to the other two numbers.

2. Devise a Plan:

To solve this problem, we can use a series of comparisons:

- Compare aaa and bbb.
- The larger of these two can then be compared to ccc.
- This will give us the largest of the three numbers.

We can break it down into the following steps:

1. Compare aaa and bbb to find the larger of the two.
2. Compare the result from step 1 with ccc to determine the largest of all three numbers.

3. **Carry Out the Plan**:

We'll implement the plan using conditional comparisons:

- First, compare a and b:
  - If $a \geq b$, then the larger of a and b is a.
  - If $b > a$, then the larger of a and b is b.
- Next, compare the result from the first comparison to c:
  - If the larger of a and b (let's say max_ab) is greater than or equal to c, then max_ab is the largest number.
  - Otherwise, c is the largest.

4. **Look Back and Check**:

After performing the comparison, ensure that each step follows logically and correctly identifies the largest number.

Example:

- For $a = 5$, $b = 8$, and $c = 3$:
  - Step 1: Compare $a = 5$ and $b = 8$, the larger is 8.
  - Step 2: Compare 8 with $c = 3$, the larger is 8, so the result is correct.

Thus, the logic is sound, and we've correctly identified the largest number.

---

5. **Implement the Solution**:

We can write the solution in pseudocode or as a simple function:

```
def find_largest(a, b, c):
  if a >= b and a >= c:
    return a
  elif b >= a and b >= c:
    return b
  else:
```

```
return c
```

6. Review and Reflect:

Reflect on the process to make sure the solution is correct and efficient.

- The approach uses simple comparisons and runs in constant time, so it is efficient.
- The solution works for all valid inputs, including negative numbers and cases where two or more numbers are equal.

Final Example Walkthrough:

For $a = 7$, $b = 4$, and $c = 9$:

1. Compare $a = 7$ and $b = 4$, the larger is 7.
2. Compare 7 with $c = 9$, the larger is 9.
3. The largest number is 9.

Conclusion:

By following the six problem-solving steps, we systematically find the largest number out of three numbers using logical comparisons. The approach is straightforward and effective.

10. (a) Your professor has given you an assignment on “Algorithmic thinking” to be submitted by this Wednesday. How do you employ means-end analysis to devise a strategy for completing your assignment before the deadline? (5 marks)

Means-End Analysis is a problem-solving technique that involves breaking down a task into smaller, manageable steps and identifying the resources (means) needed to achieve the final goal (end).
-Here's how you can employ **Means-End Analysis** to devise a strategy for completing your assignment on **"Algorithmic Thinking"** before the deadline:

1. Identify the Goal (End):

- The **end goal** is to complete and submit the assignment on "Algorithmic Thinking" by **Wednesday**.
- Ensure that the assignment is fully done, well-organized, and submitted on time.

2. Analyze the Current Situation (Means):

- Break down the assignment to understand what needs to be done:
  Research and Understanding:
  - Do you need to review any material on algorithmic thinking?
  - Are there specific topics you need to study or explore?

  Writing:
  How long is the assignment?
  Do you need to write explanations, pseudocode, or code?
  - **Formatting and Finalization**: Do you need to format the assignment and check for any errors?

- **Submission**: Are there specific submission guidelines (e.g., online portal, file format)?

3. Create Sub-goals and Break Down Tasks (Means):

- Break the overall task into smaller sub-tasks and prioritize them:
  1. **Day 1 (Today - Monday)**:
     - **Review the assignment prompt**: Understand exactly what is required. Read the instructions carefully.
     - **Research algorithmic thinking**: Study relevant materials, textbooks, or online resources to refresh your knowledge or fill in any gaps.
  2. **Day 2 (Tuesday)**:
     - **Write the introduction** and **outline** the major sections (e.g., introduction, methodology, examples, conclusion).
     - **Start solving problems** or creating algorithms (pseudocode, diagrams, or actual code). Focus on the most important parts of the assignment.
  3. **Day 3 (Wednesday)**:
     - **Finish writing** any remaining sections.
     - **Review and revise** the assignment for clarity, correctness, and completeness.
     - **Format and finalize** the document according to submission guidelines.
     - **Submit the assignment**.

4. Evaluate Possible Means to Achieve Each Sub-goal:

- For each sub-goal, consider how you will accomplish it:
  - **Research**: Gather your materials (textbooks, online articles, etc.). Spend a set amount of time researching key concepts, algorithms, and their applications.
  - **Writing**: Use any notes or drafts you've already made. If necessary, seek assistance from online resources or classmates to clarify any difficult parts.
  - **Reviewing**: Allocate specific time for proofreading, fixing formatting issues, and ensuring the content flows logically.

5. Monitor Progress and Adjust as Needed:

- As you progress through each sub-goal, evaluate whether you're on track.
- **Adjust your strategy** if you find you need more time for research, writing, or revision.
- If you finish early, use the extra time to review and polish your work. If you're behind, adjust your remaining tasks to ensure you complete the assignment by the deadline.

Example Timeline Using Means-End Analysis:

- **Monday**: Research and gather materials on algorithmic thinking. Plan your approach.
  - **End of Monday**: Clear understanding of assignment requirements.
- **Tuesday**: Write the assignment's core sections (examples, pseudocode, explanations). Ensure clarity.
  - **End of Tuesday**: A draft that needs revision and formatting.
- **Wednesday**: Finalize the draft, check for grammar, formatting, and clarity, then submit the assignment.
  - **End of Wednesday**: Assignment submitted on time.

Conclusion:

By employing **Means-End Analysis**, you break down the task into manageable sub-goals, clearly identify the resources and methods needed for each step, and monitor your progress towards completing the assignment before the deadline.
This strategy ensures a structured, focused approach to meet the deadline efficiently.

10.
b) Name two current problems in your life that might be solved through a heuristic approach. Explain why each of these problems can be solved using heuristics. (4 marks)

Heuristic approaches are useful for solving problems where there is no guaranteed, optimal solution, or when a perfect solution is computationally infeasible.
Instead, heuristics provide practical and efficient ways to find a good enough solution.

Below are two examples of problems that might be solved using heuristics:
1. Prioritizing Daily Tasks

- **Problem**: Managing a long list of tasks, where some tasks are urgent and others are important but not immediately pressing, can be overwhelming. There is often too little time to finish everything, and it's hard to determine the best order to tackle them.
- **Heuristic Solution**: A common heuristic to address this problem is the **Eisenhower Matrix**. This matrix categorizes tasks based on urgency and importance into four quadrants:
    1. Urgent and Important (Do immediately)
    2. Not Urgent but Important (Schedule for later)
    3. Urgent but Not Important (Delegate if possible)
    4. Not Urgent and Not Important (Eliminate or postpone)
- **Why Heuristics Work**: This heuristic approach simplifies decision-making by prioritizing tasks based on a quick evaluation of urgency and importance, rather than analyzing every task exhaustively. It doesn't guarantee the absolute best order, but it provides a reasonable solution that is practical and time-efficient.

2. Navigating Traffic to Find the Fastest Route

- **Problem**: When driving to a destination in an unfamiliar area, choosing the fastest route is difficult due to constantly changing traffic conditions, construction, or accidents. Navigating with maps and real-time traffic data can still involve some uncertainty.
- **Heuristic Solution**: A heuristic commonly used in GPS navigation systems is the **Shortest Path Algorithm** (e.g., Dijkstra's algorithm or A*). These algorithms approximate the fastest route by considering current traffic conditions, the number of available routes, and possible delays. They may also factor in the likelihood of congestion and use real-time data to adjust suggestions.
- **Why Heuristics Work**: Heuristics like these provide a **good enough solution** in real-time without needing exhaustive searches. They simplify the complex decision-making involved in navigating through a constantly changing environment, and while they may not always guarantee the optimal route, they are typically effective in reaching the destination more quickly than random trial and error.

Conclusion:
Both problems can be solved through heuristics because:

1. **Task prioritization** requires simplification to make decisions quickly in an environment with multiple, sometimes conflicting, priorities.
2. **Navigation** benefits from a heuristic solution by using efficient algorithms that provide practical, real-time choices without needing to compute every possible outcome.

In both cases, heuristics allow for effective problem-solving in environments where time or resources are limited.

Module 2				
11	a)	Mr. Shyam, a history professor, would like to know the percentage increase in the population of our country per decade given the first decade and the last decade. Other given data include the population at the beginning of each decade. Draw a flowchart for determining the percentage increase in the population.	3	(6)
	b)	Draw a flowchart to find the average mileage of a car in kilometers per litre after six fill-ups at petrol pumps. Input data include the number of litres of diesel, the starting odometer reading, and the odometer reading at each fillup.	3	(3)
12	a)	A standard science experiment is to drop a ball and see how high it bounces. Once the "bounciness" of the ball has been determined, the ratio gives a bounciness index. For example, if a ball dropped from a height of 10 feet bounces 6 feet high, the index is 0.6, and the total distance traveled by the ball is 16 feet after one bounce. If the ball were to continue bouncing, the distance after two bounces would be 10 ft + 6 ft + 6 ft + 3.6 ft = 25.6 ft. Note that the distance traveled for each successive bounce is the distance to the floor plus 0.6 of that distance as the ball comes back up. Write an algorithm that lets the user enter the initial height of the ball, bounciness index and the number of times the ball is allowed to continue bouncing. Output should be the total distance traveled by the ball.	3	(6)
	b)	Light travels at 3×10^8 meters per second. A light-year is the distance a light beam travels in one year. Write an algorithm that inputs a large distance value (in meters) and displays it in light-years.	3	(3)

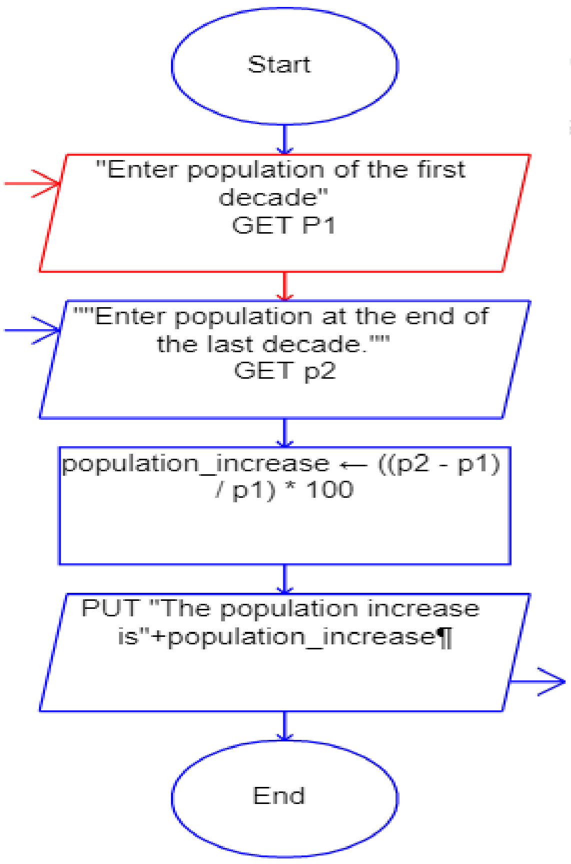
11 a) [6 marks]

To determine the **percentage increase in the population** from the first decade to the last decade, we can follow these steps in the form of a flowchart. The formula for calculating the percentage increase is:

$$\text{Percentage Increase} = \left(\frac{\text{Population at the end of the last decade} - \text{Population at the beginning of the first decade}}{\text{Population at the beginning of the first decade}}\right) \times 100$$

Flowchart Explanation:

1. **Start**
   - The process begins with the user initiating the calculation.
2. **Input Population Data**
   - Prompt the user to input the **population at the beginning of the first decade**.
   - Prompt the user to input the **population at the end of the last decade**.
3. **Calculate Percentage Increase**
   - Apply the formula for the percentage increase in population:
     $$\text{Percentage Increase} = \left(\frac{\text{Population at the end of the last decade} - \text{Population at the beginning of the first decade}}{\text{Population at the beginning of the first decade}}\right) \times 100$$
4. **Display the Result**
   - Output the **percentage increase** to the user.
5. **End**
   - The process ends after displaying the result.


Start
"Enter population of the first decade"
GET P1
""Enter population at the end of the last decade.""
GET p2
population_increase ← ((p2 - p1) / p1) * 100
PUT "The population increase is"+population_increase¶
End

Example Walkthrough:

Suppose the population at the beginning of the first decade is **50 million** and the population at the end of the last decade is **75 million**:

- P1 = 50 million
- P2 = 75 million

The percentage increase is:

$$\text{Percentage Increase} = \left(\frac{75 - 50}{50}\right) \times 100 = 50\%$$

The flowchart would help Mr. Shyam calculate the population growth efficiently based on the given inputs.

11. B) [3 marks]

Algorithm to Calculate Average Mileage:

1. **Start**: Initialize necessary variables:
   - TotalDistance = 0
   - TotalLitres = 0
2. **Input Data**:
   - Record the **starting odometer reading** (StartOdometer).
   - For each of the six fill-ups:
     - Record the **odometer reading** after the fill-up.
     - Record the **litres of diesel** added.
3. **Calculate Total Distance**:
   - For each fill-up, calculate the distance traveled since the last reading: Distance=Current Odometer Reading−Previous Odometer ReadingDistance = \text{Current Odometer Reading} - \text{Previous Odometer Reading}Distance=Current Odometer Reading−Previous Odometer Reading
   - Accumulate the total distance: TotalDistance+=DistanceTotalDistance += DistanceTotalDistance+=Distance
4. **Calculate Total Fuel Consumed**:
   - Accumulate the total fuel consumed from all six fill-ups: TotalLitres+=Litres of DieselTotalLitres += \text{Litres of Diesel}TotalLitres+=Litres of Diesel
5. **Calculate Average Mileage**:
   - Compute the average mileage using the formula: Average Mileage=TotalDistanceTotalLitres\text{Average Mileage} = \frac{\text{TotalDistance}}{\text{TotalLitres}}Average Mileage=TotalLitresTotalDistance
6. **Output Result**:
   - Display the average mileage in kilometers per litre.
7. **End**: The process ends.

Flowchart Explanation:

Input Data:

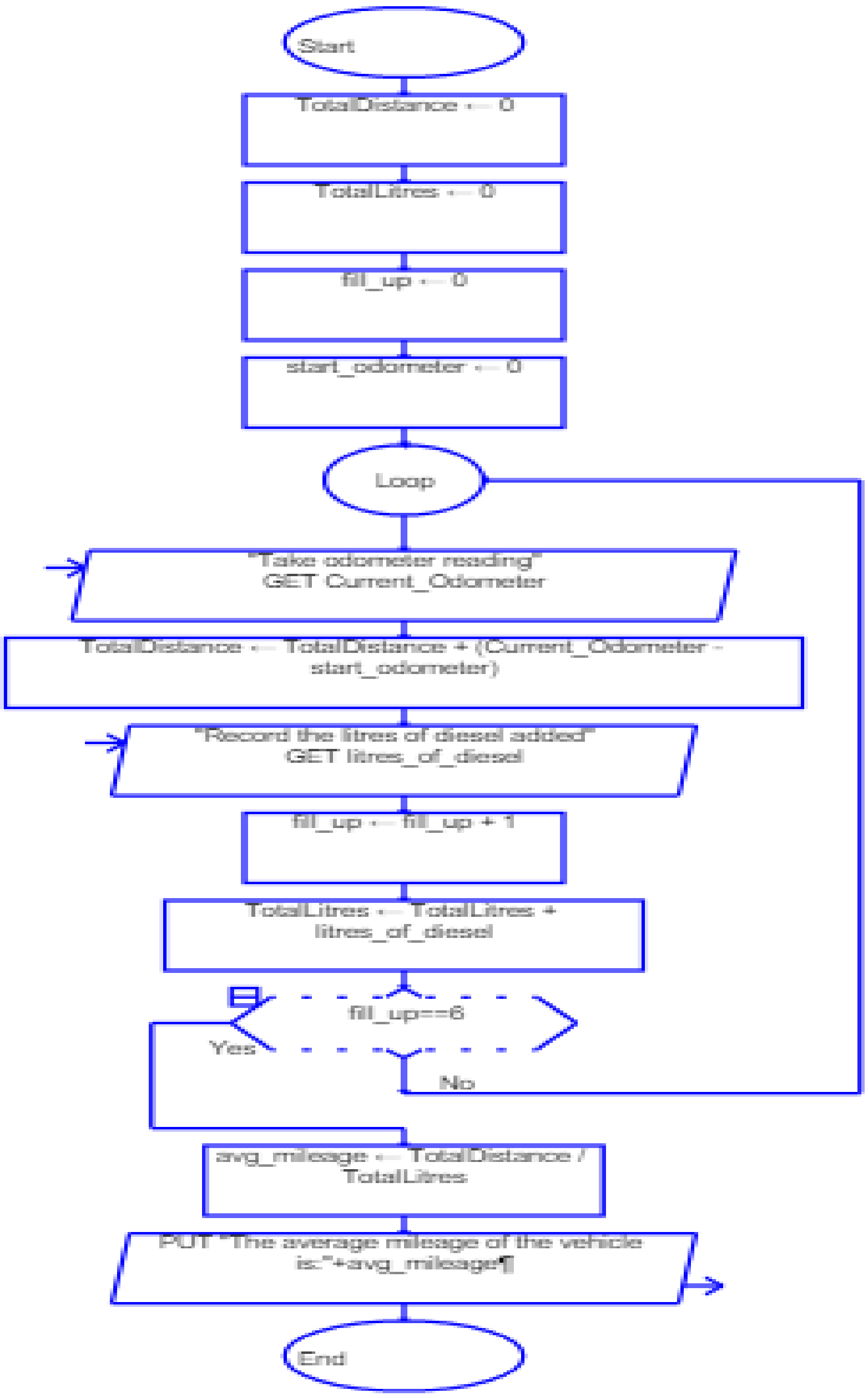
- The flowchart prompts the user for the starting odometer reading, the odometer reading after each fill-up, and the litres of diesel added.

Processing:

- Iteratively calculate the distance traveled between fill-ups and accumulate both the total distance and total litres of diesel.

Output:

- Calculate the average mileage and display the result.



Start
TotalDistance ← 0
TotalLitres ← 0
fill_up ← 0
start_odometer ← 0
Loop
"Take odometer reading"
GET Current_Odometer
TotalDistance ← TotalDistance + (Current_Odometer - start_odometer)
"Record the litres of diesel added"
GET litres_of_diesel
fill_up ← fill_up + 1
TotalLitres ← TotalLitres + litres_of_diesel
fill_up==6
Yes
No
avg_mileage ← TotalDistance / TotalLitres
PUT "The average mileage of the vehicle is:"+avg_mileage¶
End

12 a) [6 -marks]

Algorithm:

1. **Input**:
    - initial_height: The height from which the ball is dropped.
    - bounciness_index: The fraction of height the ball bounces back.
    - num_bounces: The number of bounces to calculate.
2. **Initialize Variables**:
    - Set total_distance to 0. This will store the cumulative distance traveled by the ball.
    - Set current_height to initial_height. This tracks the height of each bounce.
3. **Process**:
    - Add the initial_height to total_distance (for the initial drop).
    - For each bounce (up to num_bounces):
        - Calculate the height of the bounce using current_height * bounciness_index.
        - Add the height of the downward travel (current_height) and the upward travel (current_height * bounciness_index) to total_distance.
        - Update current_height to current_height * bounciness_index.
4. **Output**:
    - Display the total_distance traveled by the ball.
5. **End**.

Pseudocode:

```
Start
   Input initial_height
   Input bounciness_index
   Input num_bounces

   total_distance = 0
   current_height = initial_height

   total_distance = total_distance + initial_height  # Initial drop

   For i = 1 to num_bounces
      bounce_height = current_height * bounciness_index
      total_distance = total_distance + current_height + bounce_height
      current_height = bounce_height
   End For

   Output total_distance
End
```

Example Walkthrough:
Input:

- initial_height = 10 ft
- bounciness_index = 0.6
- num_bounces = 2

Step-by-Step Calculation:

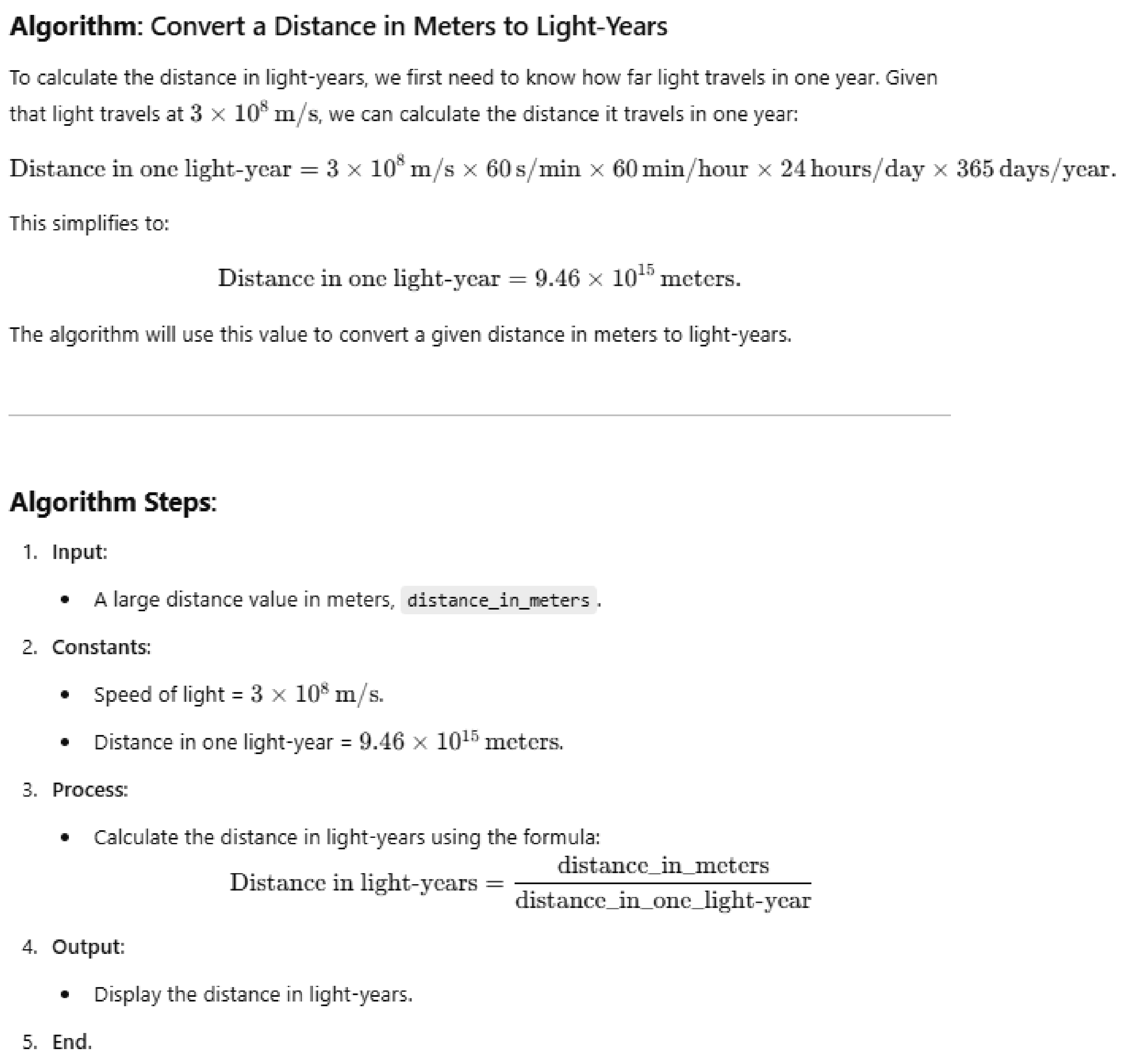
1. **Initial Drop**:
    - total_distance = 10

2. **First Bounce**:
   - Bounce height = 10 * 0.6 = 6
   - Add downward and upward travel: total_distance = 10 + 6 + 6 = 22
3. **Second Bounce**:
   - Bounce height = 6 * 0.6 = 3.6
   - Add downward and upward travel: total_distance = 22 + 6 + 3.6 = 31.6

Output:

- Total Distance = 31.6 ft

12. B) [3 marks]



Algorithm: Convert a Distance in Meters to Light-Years

To calculate the distance in light-years, we first need to know how far light travels in one year. Given that light travels at 3×10^8 m/s, we can calculate the distance it travels in one year:

$$\text{Distance in one light-year} = 3 \times 10^8\,\text{m/s} \times 60\,\text{s/min} \times 60\,\text{min/hour} \times 24\,\text{hours/day} \times 365\,\text{days/year}.$$

This simplifies to:

$$\text{Distance in one light-year} = 9.46 \times 10^{15}\ \text{meters}.$$

The algorithm will use this value to convert a given distance in meters to light-years.

Algorithm Steps:

1. **Input:**
   - A large distance value in meters, `distance_in_meters`.
2. **Constants:**
   - Speed of light = 3×10^8 m/s.
   - Distance in one light-year = 9.46×10^{15} meters.
3. **Process:**
   - Calculate the distance in light-years using the formula:

$$\text{Distance in light-years} = \frac{\text{distance\_in\_meters}}{\text{distance\_in\_one\_light-year}}$$

4. **Output:**
   - Display the distance in light-years.
5. **End.**

Pseudocode:

plaintext

Copy code

```
Start
    Input distance_in_meters
    Set speed_of_light = 3 × 10^8  # meters per second
    Set seconds_per_year = 60 × 60 × 24 × 365  # total seconds in a year
    Set distance_per_light_year = speed_of_light × seconds_per_year  # meters in one light

    Calculate distance_in_light_years = distance_in_meters / distance_per_light_year

    Output distance_in_light_years
End
```

Example Walkthrough:

Input:

- `distance_in_meters = 1 × 10^{17}` meters.

Step-by-Step Calculation:

1. **Distance in one light-year:**

$$3 \times 10^8 \, \text{m/s} \times 31,536,000 \, \text{s/year} = 9.46 \times 10^{15} \, \text{meters}.$$

2. **Convert Distance:**

$$\text{Distance in light-years} = \frac{1 \times 10^{17}}{9.46 \times 10^{15}} \approx 10.57 \, \text{light-years}.$$

Output:
The program calculates and displays the equivalent distance in light-years for any given large distance in meters.

Module 3				
13	a)	Write a recursive function to find an array's minimum and maximum elements. Your method should return a tuple ***(a, b)***, where ***a*** is the minimum element and ***b*** is the maximum.	4	(5)
	b)	Write a program to input a matrix and determine its type: lower triangular, upper triangular, or diagonal.	4	(4)
14	a)	Write a program to read *N* words and display them in the increasing order of their lengths. The length of each word is also to be displayed.	4	(6)
	b)	There are 500 light bulbs (numbered 1 to 500) arranged in a row. Initially, they are all OFF. Starting with bulb 2, all even numbered bulbs are turned ON. Next, starting with bulb 3, and visiting every third bulb, it is turned ON if it is OFF, and it is turned OFF if it is ON. This procedure is repeated for every fourth bulb, then every fifth bulb, and so on up to the 500th bulb. Devise an algorithm to determine which bulbs glow at the end of the above exercise.	4	(3)

13 (a) [5 marks]

```
def find_min_max(arr, low, high):
    # Base case: If the array has only one element
    if low == high:
        return (arr[low], arr[low])

    # Base case: If the array has two elements
    if high == low + 1:
        if arr[low] < arr[high]:
            return (arr[low], arr[high])
        else:
            return (arr[high], arr[low])

    # Recursive case: Divide the array into two halves
    mid = (low + high) // 2
    min1, max1 = find_min_max(arr, low, mid)   # Left half
    min2, max2 = find_min_max(arr, mid + 1, high)  # Right half

    # Combine results
    return (min(min1, min2), max(max1, max2))
```

How It Works:

1. **Base Case**:
    - If the array has only one element, both the minimum and maximum are that element.
    - If the array has two elements, compare them to find the minimum and maximum.
2. **Recursive Case**:
    - Divide the array into two halves.
    - Recursively find the minimum and maximum of each half.
    - Combine the results by taking the smaller minimum and the larger maximum.

Example Usage:

```
# Array to find min and max
array = [3, 1, 4, 1, 5, 9, 2, 6, 5]

# Call the function
result = find_min_max(array, 0, len(array) - 1)

# Output the result
print("Minimum:", result[0])
print("Maximum:", result[1])
```

Step-by-Step Example:

For the array [3, 1, 4, 1, 5, 9, 2, 6, 5]:

1. The array is divided recursively into halves until subarrays of size 1 or 2 are reached.
2. Min and max values are computed for these subarrays:
    - [3, 1] → (1, 3)
    - [4, 1] → (1, 4)
    - [5, 9] → (5, 9)
    - [2, 6, 5] → (2, 6) via further division.
3. Results from each half are combined to compute the overall minimum and maximum.

Final output:

```
Minimum: 1
Maximum: 9
```

b) Write a program to input a matrix and determine its type: lower triangular, upper triangular, or diagonal. [4 marks]

Here's a Python program using the numpy module to input a matrix and determine its type (lower triangular, upper triangular, or diagonal):

```
import numpy as np

# Function to check if a matrix is lower triangular
def is_lower_triangular(matrix):
    return np.all(np.triu(matrix, k=1) == 0)

# Function to check if a matrix is upper triangular
def is_upper_triangular(matrix):
    return np.all(np.tril(matrix, k=-1) == 0)

# Function to check if a matrix is diagonal
def is_diagonal(matrix):
    return np.all(np.tril(matrix, k=-1) == 0) and np.all(np.triu(matrix, k=1) == 0)

# Input matrix size
n = int(input("Enter the number of rows/columns for the square matrix: "))
matrix = []

print("Enter the elements row by row:")
for i in range(n):
    row = list(map(int, input().split()))
    matrix.append(row)

matrix = np.array(matrix)

# Determine the type of the matrix
if is_diagonal(matrix):
    print("The matrix is Diagonal.")
elif is_lower_triangular(matrix):
    print("The matrix is Lower Triangular.")
elif is_upper_triangular(matrix):
    print("The matrix is Upper Triangular.")
else:
    print("The matrix is neither Lower Triangular, Upper Triangular, nor Diagonal.")
```

Explanation:

1. **is_lower_triangular(matrix)**: This function uses np.triu(matrix, k=1) to extract the upper triangular part of the matrix starting from the first diagonal above the main diagonal. It then checks if this part contains only zeros, indicating the matrix is lower triangular.
2. **is_upper_triangular(matrix)**: This function uses np.tril(matrix, k=-1) to extract the lower triangular part of the matrix starting from the first diagonal below the main diagonal. It checks if this part contains only zeros, indicating the matrix is upper triangular.
3. **is_diagonal(matrix)**: This function uses both np.tril(matrix, k=-1) and np.triu(matrix, k=1) to extract the parts of the matrix that are below and above the diagonal. It checks if both parts contain only zeros, indicating the matrix is diagonal.

Steps:

1. The user is asked to input the size of the square matrix (n).
2. The user enters the elements of the matrix row by row.
3. The program checks if the matrix is diagonal, lower triangular, or upper triangular and prints the result accordingly.

Example:

Input:

```
Enter the number of rows/columns for the square matrix: 3
Enter the elements row by row:
1 0 0
0 2 0
0 0 3
```

Output:

```
The matrix is Diagonal.
```

For a **lower triangular matrix**:

```
Enter the number of rows/columns for the square matrix: 3
Enter the elements row by row:
1 0 0
2 3 0
4 5 6
```

Output:

```
The matrix is Lower Triangular.
```

For an **upper triangular matrix**:

```
Enter the number of rows/columns for the square matrix: 3
Enter the elements row by row:
1 2 3
0 4 5
0 0 6
```

Output:

```
The matrix is Upper Triangular.
```

14

a) [6 marks]

Here's a Python program that reads N words, displays them in increasing order of their lengths, and also displays the length of each word:

```
# Read the number of words
N = int(input("Enter the number of words: "))

# Read N words into a list
words = []
for _ in range(N):
    word = input("Enter a word: ")
    words.append(word)

# Sort the words by their length
words.sort(key=len)

# Display the words and their lengths in increasing order of length
print("\nWords in increasing order of their lengths:")
for word in words:
    print(f"{word} - Length: {len(word)}")
```

Explanation:

1. **Input**: The program first reads an integer N (number of words). It then collects N words from the user and stores them in a list called words.
2. **Sorting**: The program uses the sort() method with the key=len to sort the words by their length.
3. **Output**: After sorting, the program displays each word along with its length.

Example:

Input:

```
Enter the number of words: 5
Enter a word: elephant
Enter a word: dog
Enter a word: cat
Enter a word: giraffe
Enter a word: apple
```

Output:

```
Words in increasing order of their lengths:
cat - Length: 3
dog - Length: 3
apple - Length: 5
giraffe - Length: 7
elephant - Length: 8
```

This will display the words sorted in increasing order of their lengths, along with their respective lengths.

14.
b)
The problem involves a sequence of operations on 500 light bulbs, where each bulb is toggled (ON/OFF) based on a systematic pattern. We need to determine which bulbs remain ON at the end of this sequence of operations.

Problem Breakdown:

1. **Initialization**:
   - We start with 500 light bulbs, all initially OFF.
2. **Toggling Process**:
   - The bulbs are toggled in a pattern:
     - Starting with the 2nd bulb, every 2nd bulb is toggled (i.e., bulbs 2, 4, 6, 8, ..., 500).
     - Then, starting with the 3rd bulb, every 3rd bulb is toggled (i.e., bulbs 3, 6, 9, 12, ..., 498).
     - This process continues up to every 500th bulb.
3. **Key Insight**:
   - A bulb's state (ON or OFF) is toggled once for each divisor of its position number.
   - For example, bulb 12 will be toggled for each of its divisors: 1, 2, 3, 4, 6, and 12.
   - A bulb ends up being ON if it is toggled an odd number of times and OFF if toggled an even number of times.
4. **Observation**:
   - A number (bulb number) has an odd number of divisors only if it is a perfect square.
   - This is because divisors usually come in pairs (for example, for 12: divisors are (1, 12), (2, 6), and (3, 4)), except when the number is a perfect square (e.g., 9 has divisors (1, 9) and (3, 3), where 3 is repeated).
   - Therefore, the bulbs that remain ON at the end are those numbered with perfect squares.

Algorithm:

1. **Identify perfect squares**:
   - The bulbs that will remain ON are those whose numbers are perfect squares (1, 4, 9, 16, 25, ..., up to 500).
2. **Output the result**:
   - List all perfect squares from 1 to 500.

Python Code:

```
import math

# Number of bulbs
n = 500

# List to store bulbs that remain ON (perfect squares)
on_bulbs = []

# Check for perfect squares
for i in range(1, int(math.sqrt(n)) + 1):
   perfect_square = i * i
   on_bulbs.append(perfect_square)

# Output the bulbs that are ON
print("Bulbs that remain ON (perfect squares):")
print(on_bulbs)
```

Explanation of Code:

- **math.sqrt(n)**: This calculates the square root of n (500). We only need to consider numbers up to the square root of 500 because the perfect squares beyond this range will exceed 500.
- **i * i**: For each number i from 1 to the integer part of the square root of 500, i * i gives the perfect square, which corresponds to a bulb that remains ON.
- **Output**: The bulbs that remain ON are those numbered with perfect squares. These are collected in the on_bulbs list and printed.

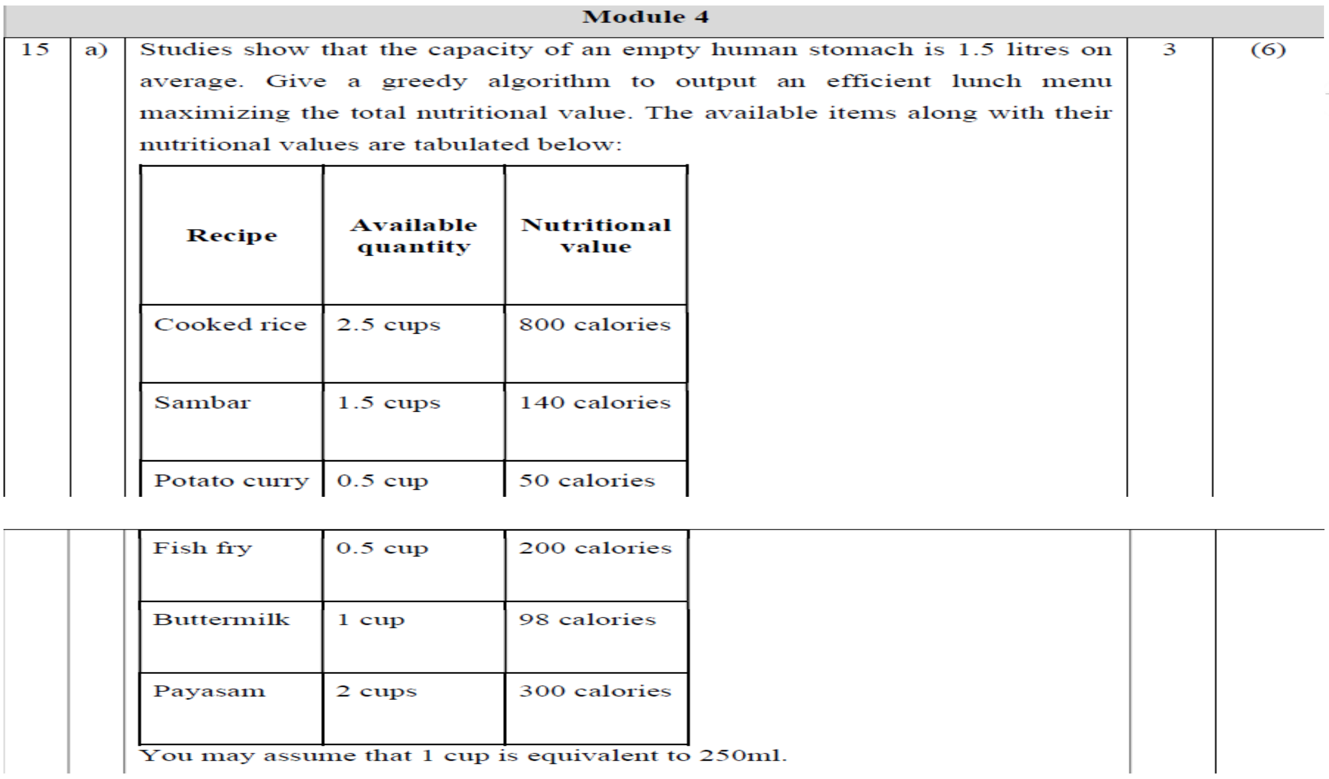
  Output:

  graphql

  Copy code

  ```
  Bulbs that remain ON (perfect squares):
  [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484]
  ```

  Conclusion:
- The bulbs that remain ON are the ones numbered with perfect squares: 1, 4, 9, 16, 25, 36, 49, ..., 484.
- These bulbs are toggled an odd number of times and remain ON at the end of the procedure.



Module 4

15 a) Studies show that the capacity of an empty human stomach is 1.5 litres on average. Give a greedy algorithm to output an efficient lunch menu maximizing the total nutritional value. The available items along with their nutritional values are tabulated below: 3 (6)

Recipe	Available quantity	Nutritional value
Cooked rice	2.5 cups	800 calories
Sambar	1.5 cups	140 calories
Potato curry	0.5 cup	50 calories
Fish fry	0.5 cup	200 calories
Buttermilk	1 cup	98 calories
Payasam	2 cups	300 calories

You may assume that 1 cup is equivalent to 250ml.

To solve this problem using a greedy algorithm, we need to maximize the total nutritional value (calories) of the lunch while staying within the capacity of 1.5 liters (or 1500 ml) of the stomach. Here's a step-by-step breakdown of how the problem can be approached:

Step 1: Convert all the quantities to the same unit (milliliters)
Since the capacity of the stomach is given in milliliters (1500 ml), we need to convert the available quantities of food items into milliliters as well.

1 cup = 250 ml

Cooked rice: 2.5 cups = 2.5 * 250 = 625 ml
Sambar: 1.5 cups = 1.5 * 250 = 375 ml
Potato curry: 0.5 cup = 0.5 * 250 = 125 ml
Fish fry: 0.5 cup = 0.5 * 250 = 125 ml
Buttermilk: 1 cup = 1 * 250 = 250 ml
Payasam: 2 cups = 2 * 250 = 500 ml

Step 2: Calculate the nutritional value per milliliter for each item

-We want to maximize the nutritional value per unit of volume, so we calculate the nutritional value per milliliter for each food item.

Cooked rice: 800 calories / 625 ml = 1.28 calories per ml
Sambar: 140 calories / 375 ml = 0.37 calories per ml
Potato curry: 50 calories / 125 ml = 0.40 calories per ml
Fish fry: 200 calories / 125 ml = 1.60 calories per ml
Buttermilk: 98 calories / 250 ml = 0.392 calories per ml
Payasam: 300 calories / 500 ml = 0.60 calories per ml

Step 3: Sort the items by their nutritional value per milliliter (in descending order)

-To maximize the nutritional value, we should consider the food items with the highest nutritional value per unit of volume first.

Here is the sorted list:

Fish fry: 1.60 calories per ml
Cooked rice: 1.28 calories per ml
Payasam: 0.60 calories per ml
Potato curry: 0.40 calories per ml
Buttermilk: 0.392 calories per ml
Sambar: 0.37 calories per ml

Step 4: Greedily select the food items
-Now, we will greedily pick the food items, starting from the one with the highest calories per ml, until we reach the stomach's capacity limit of 1500 ml.

Step 5: Algorithm to maximize the nutritional value

```
# Define the items along with their quantities (in ml) and nutritional values (in calories)
items = [
    ("Fish fry", 125, 200),        # 125 ml, 200 calories
    ("Cooked rice", 625, 800),     # 625 ml, 800 calories
    ("Payasam", 500, 300),         # 500 ml, 300 calories
    ("Potato curry", 125, 50),     # 125 ml, 50 calories
    ("Buttermilk", 250, 98),       # 250 ml, 98 calories
    ("Sambar", 375, 140)           # 375 ml, 140 calories
]

# Sort the items by their calories per ml in descending order
items.sort(key=lambda x: x[2] / x[1], reverse=True)

# Capacity of the stomach (in ml)
capacity = 1500

# Greedy selection of items
total_calories = 0
selected_items = []
remaining_capacity = capacity

for item in items:
    name, quantity, calories = item
    if quantity <= remaining_capacity:
        # Take the full item
        selected_items.append((name, quantity, calories))
        total_calories += calories
        remaining_capacity -= quantity
    else:
        # Take only the remaining part if the item doesn't fit completely
        if remaining_capacity > 0:
            selected_items.append((name, remaining_capacity, calories * remaining_capacity /
quantity))
            total_calories += calories * remaining_capacity / quantity
            remaining_capacity = 0
        break

# Output the result
print("Selected items and their nutritional values:")
for item in selected_items:
    name, quantity, calories = item
    print(f"{name}: {quantity} ml, {calories} calories")

print(f"\nTotal calories: {total_calories} calories")
```

Explanation:
Data Representation: Each item is represented as a tuple with its name, quantity (in milliliters), and nutritional value (in calories).
Sorting: The items are sorted based on their calories per milliliter in descending order.

Greedy Selection: The algorithm selects items one by one, starting from the one with the highest calories per milliliter, and fills the stomach until its capacity is reached.
Handling Partial Items: If an item doesn't fit completely within the remaining capacity, only a part of it is taken, and its nutritional value is proportionally scaled.

Output: The selected items and their total nutritional value are printed.

15.
b) How are recursion and dynamic programming (DP) related? Is it possible to construct a DP version for all recursive solutions? [3 marks]

Recursion and Dynamic Programming (DP) Relationship:
Recursion and Dynamic Programming (DP) are closely related techniques used to solve problems, especially optimization problems and problems that can be broken down into overlapping subproblems.

1. **Recursion**:
    - A recursive solution involves solving a problem by breaking it down into smaller subproblems, solving each recursively, and combining their solutions.
    - The function calls itself to solve smaller instances of the problem.
    - Recursion is often used when a problem exhibits the **divide and conquer** property and the solution to the problem can be built upon the solutions to smaller subproblems.
2. **Dynamic Programming (DP)**:
    - Dynamic Programming is a technique used to optimize recursive algorithms by **storing the results of subproblems** that are solved multiple times.
    - DP solves the problem by solving each subproblem only once and storing its solution (usually in a table or array) to avoid redundant work.
    - It is typically used when the problem exhibits **overlapping subproblems** and **optimal substructure** (where the optimal solution to the problem can be constructed from optimal solutions to its subproblems).

Relationship Between Recursion and DP:

- **Overlapping Subproblems**: Both recursion and DP are effective for problems that involve repeated solving of the same subproblems. In the case of recursion, the same subproblems may be recomputed multiple times, leading to inefficiency. DP eliminates this inefficiency by storing and reusing the results of subproblems.
- **Optimal Substructure**: In both recursive and DP approaches, the optimal solution to the problem is built by combining the optimal solutions to its subproblems.

Is It Possible to Construct a DP Version for All Recursive Solutions?

- **Not always**: While many recursive problems can be transformed into dynamic programming problems, there are cases where a direct DP formulation is not possible or does not provide a clear benefit.
- **When DP is applicable**:
    - DP is particularly useful for **optimization problems** where solutions to subproblems are reused and the problem has an **optimal substructure**.
    - For example, problems like Fibonacci sequence, shortest path problems (like in Dijkstra's algorithm), knapsack problem, and others fit naturally into DP.

- **When DP may not be possible or effective**:
    - Some problems may not exhibit overlapping subproblems or optimal substructure in a way that makes dynamic programming applicable.
    - Problems that don't have a **clear recursive structure** or require complex state transitions might not be easily solvable using DP.

Summary:

- **Recursion** is a technique to break down a problem into smaller subproblems, and **dynamic programming** optimizes recursive solutions by storing and reusing the results of subproblems.
- **Not all recursive problems** can be efficiently solved with DP, but many optimization problems can be improved with dynamic programming techniques.

16	a)	Write a Python program for a random walk simulation in a 2D grid starting from the origin **(0, 0)**. At each step, randomly move up, down, left, or right. Print the final position after 10 steps.	3	(4)
	b)	Use divide and conquer to find the majority element in an array, where the majority element appears more than n/2 times. Divide the array into two halves, find the majority element in each half, and combine the results to identify if there is a majority element in the entire array.	2	(5)

*****

16.
a)
Here's a simple Python program to simulate a random walk in a 2D grid, starting from the origin (0, 0). At each step, the program randomly moves up, down, left, or right. The program prints the final position after 10 steps.

```
import random
# Function to simulate a random walk
def random_walk(steps):
    # Starting position at the origin (0, 0)
    x, y = 0, 0
        # List of possible moves: up, down, left, right
    moves = [("up", (0, 1)), ("down", (0, -1)), ("left", (-1, 0)), ("right", (1, 0))]

    # Simulate the walk for the given number of steps
    for _ in range(steps):
        # Randomly choose a move
        move = random.choice(moves)
        # Update the position based on the move
        x += move[1][0]
        y += move[1][1]

    return x, y
# Simulate a random walk with 10 steps
final_position = random_walk(10)
# Print the final position after 10 steps
print(f"Final position after 10 steps: {final_position}")
```

Explanation:

1. **Starting Point**: The walk starts at the origin (0, 0).
2. **Move Choices**: The possible moves are:
   - **Up**: Increase the y coordinate.
   - **Down**: Decrease the y coordinate.
   - **Left**: Decrease the x coordinate.
   - **Right**: Increase the x coordinate.
3. **Random Movement**: In each step, a random direction is chosen using random.choice(moves).
4. **Final Position**: After 10 steps, the final position is printed.

Example Output:

```
Final position after 10 steps: (1, -3)
```

Note that the output will vary each time you run the program because the walk is random

16.

b)

To solve the problem of finding the majority element in an array using the **divide and conquer** approach, we can break the problem into smaller subproblems by dividing the array into two halves, recursively finding the majority element in each half, and then combining the results to identify the majority element in the entire array.

Approach:

1. **Base Case**: If the array has only one element, that element is the majority by default.
2. **Divide**: Recursively divide the array into two halves.
3. **Conquer**: For each half, find the majority element.
4. **Combine**: After finding the majority element in each half, check if they match. If they match, that element is the majority element for the entire array. If they don't match, check the frequency of each majority element in the entire array to determine which one is the majority.

Algorithm Steps:

1. **Divide** the array into two halves.
2. **Recurse** to find the majority element in each half.
3. **Combine**:
   - If the majority element of the left half and the majority element of the right half are the same, then that element is the majority element of the whole array.
   - If they are different, count the occurrences of both elements in the entire array. The element with more than n/2 occurrences is the majority element.

Code Implementation:

```
def majority_element(arr, left, right):
  # Base case: if the array contains only one element, that element is the majority element
  if left == right:
    return arr[left]

  # Divide the array into two halves
  mid = (left + right) // 2

  # Recursively find the majority element in the left and right halves
  left_majority = majority_element(arr, left, mid)
  right_majority = majority_element(arr, mid + 1, right)
```

```
    # If both halves have the same majority element, return that element
    if left_majority == right_majority:
        return left_majority

    # If the majority elements are different, count their occurrences in the entire array
    left_count = arr[left:right+1].count(left_majority)
    right_count = arr[left:right+1].count(right_majority)

    # The majority element must appear more than n/2 times in the whole array
    if left_count > (right - left + 1) // 2:
        return left_majority
    elif right_count > (right - left + 1) // 2:
        return right_majority

    # No majority element found
    return None

# Helper function to call the recursive function
def find_majority_element(arr):
    return majority_element(arr, 0, len(arr) - 1)

# Example Usage
arr = [2, 3, 9, 2, 2, 2, 5, 2, 2]
result = find_majority_element(arr)

if result is not None:
    print(f"The majority element is {result}")
else:
    print("No majority element found.")
```

Explanation:

1. **Base Case**: If the array has only one element, that element is trivially the majority element.
2. **Divide**: The array is divided into two halves by calculating the middle index mid.
3. **Recursive Call**: The majority_element function is called recursively on both halves of the array.
4. **Combine**:
   - If the majority element of the left half (left_majority) is equal to the majority element of the right half (right_majority), then this element is the majority of the entire array.
   - Otherwise, count how many times each of the two elements appears in the array and determine which one appears more than n/2 times. This element is the majority element.

Example:

Given the array: [2, 3, 9, 2, 2, 2, 5, 2, 2]

- After applying the divide and conquer approach, the program finds that 2 is the majority element because it appears more than 4 times (which is n/2 for n = 9).

Output:

```
The majority element is 2
```

Time Complexity:

- **Divide Step**: Each recursive call divides the array into two halves, which takes O(log n) levels of recursion.
- **Conquer Step**: At each level, we may need to count the occurrences of the majority elements in the array, which takes O(n) time.
- Therefore, the overall time complexity is **O(n log n)** due to the recursive divide and the counting step at each level.

Space Complexity:

- The space complexity is **O(log n)** due to the recursion stack.