**Batch – T7**

**Practical No. 2**

**Title of Assignment : Searching Algorithm**

**Student Name: Om Kulkarni**

**Student PRN: 22510034**

**Problem Statement:**

Q.1You are an IT company's manager. Based on their performance over the last N working days, you must rate your employee. You are given an array of N integers called workload, where workload[i] represents the number of hours an employee worked on an ith day. The employee must be evaluated using the following criteria:

* Rating = the maximum number of consecutive working days when the employee has worked more than 6 hours.

You are given an integer *N*where *N* represents the number of working days. You are given an integer array *workload*where *workload[i]* represents the number of hours an employee worked on an ith day.

**Task**

Determine the employee rating

1. Algorithm/Pseudocode

Initialize a variable max\_consecutive\_days to 0 to keep track of the maximum number of consecutive days where the employee worked more than 6 hours.

Initialize a variable current\_consecutive\_days to 0 to count the current streak of consecutive days where the employee worked more than 6 hours.

Iterate through the workload array:

-If workload[i] is greater than 6, increment current\_consecutive\_days.

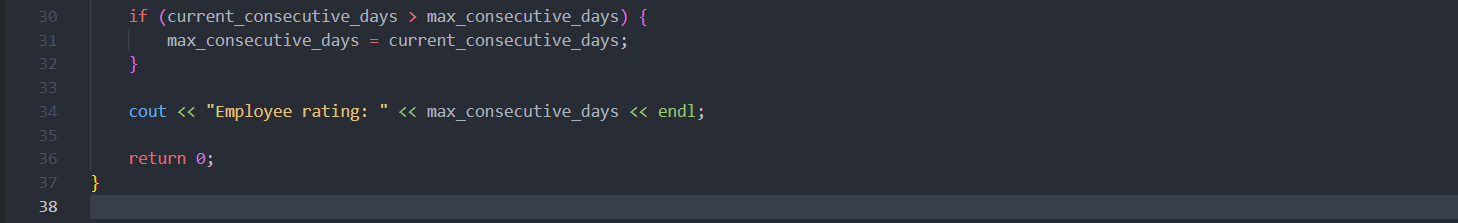
-If workload[i] is less than or equal to 6, compare current\_consecutive\_days with max\_consecutive\_days and update max\_consecutive\_days if current\_consecutive\_days is greater. Then reset current\_consecutive\_days to 0.

After the loop, compare current\_consecutive\_days with max\_consecutive\_days one last time to ensure any streak that might end at the last element is considered.

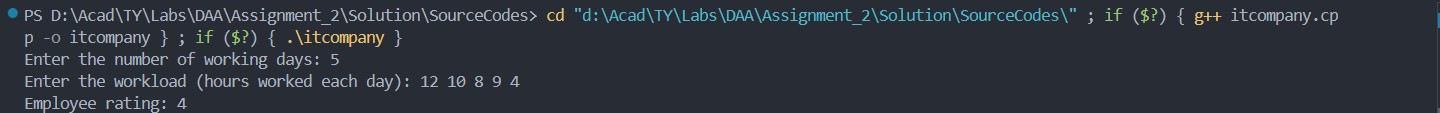
The value of max\_consecutive\_days will be the employee rating.

1. Program Code





1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.
2. Reading Input (Lines 7-11):
   * The loop reads N integers from the user and stores them in the workload array.
   * This loop runs N times.
   * Time Complexity: O(N)
3. Calculating the Maximum Number of Consecutive Days (Lines 14-25):
   * The loop iterates through the workload array to find the maximum number of consecutive days with more than 6 hours of work.
   * This loop also runs N times.
   * Time Complexity: O(N)

Combining both steps, the overall time complexity of the program is: O(N)+O(N)=O(N)

Space Complexity

1. Space for Input Array:
   * A vector<int> of size N is used to store the workload.
   * Space Complexity: O(N)
2. Space for Variables:
   * A few integer variables (N, max\_consecutive\_days, and current\_consecutive\_days) are used.
   * Space Complexity: O(1)

Combining both, the overall space complexity of the program is: O(N)+O(1)=O(N)

**Problem Statement:**

Q.2 You have N boxes numbered 1 through N and K candies numbered 1 through K. You put the candies in the boxes in the following order:

* first candy in the first box,
* second candy in the second box,
* .......
* ........
* so up to N-th candy in the Nth box,
* the next candy in (N - 1)-th box,
* the next candy in (N - 2)-th box
* ........
* .......
* and so on up to the first box,
* then the next candy in the second box
* ......    and so on until there is no candy left.

So you put the candies in the boxes in the following order:

Find the index of the box where you put the K-th candy.

1. Algorithm/Pseudocode

Initialization:

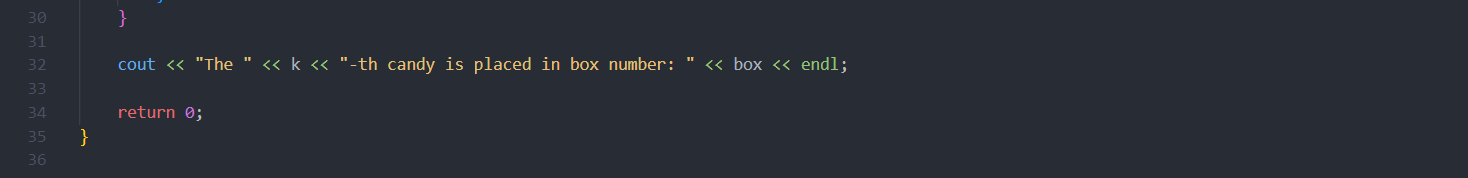
* Initialize a variable current\_box to 1 to start placing candies in the first box.
* Initialize a variable direction to 1, which indicates the current direction (1 means moving forward, -1 means moving backward).
* Initialize a variable steps to 0 to keep track of the number of steps taken.

Candy Distribution:

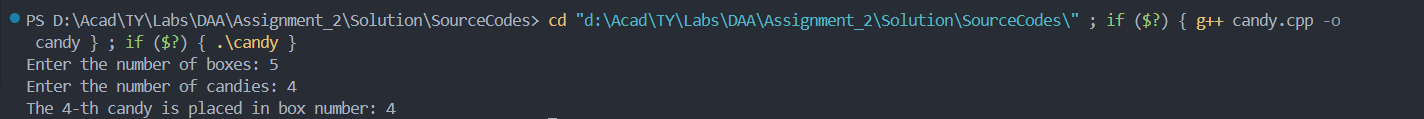
* Iterate from 1 to K (representing the K-th candy):
  + Place the candy in the current box.
  + Increment the steps counter.
  + Check if the current direction is forward (direction == 1):
    - If current\_box is less than N, increment current\_box.
    - If current\_box is equal to N, change direction to -1 and decrement current\_box.
  + Check if the current direction is backward (direction == -1):
    - If current\_box is greater than 1, decrement current\_box.
    - If current\_box is equal to 1, change direction to 1 and increment current\_box.

1. Program Code





1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.

Time Complexity

1. Reading Input (Lines 4-7):
   * Reading the number of boxes n and the number of candies k involves simple input operations.
   * Time Complexity**:** O(1)
2. CandyDistribution (Lines 10-24):
   * The for loop runs from 1 to k−1, iterating a total of k−1 times.
   * Inside the loop, there are constant-time operations (if-else conditions and increments/decrements).
   * Time Complexity: O(k)

Combining these steps, the overall time complexity of the program is: O(1)+O(k)=O(k)

Space Complexity

1. Space for Input Variables:
   * Two integers n and k are used to store the number of boxes and the number of candies.
   * Space Complexity: O(1)
2. Space for Loop Variables:
   * Two integers box and step are used for the candy distribution process.
   * Space Complexity: O(1)

Combining these, the overall space complexity of the program is: O(1)

**Problem Statement:**

Implement and Explain Tower of Hanoi algorithm.

1. Algorithm/Pseudocode

Inputs:

* + n: The number of disks.
  + source: The peg where disks are initially placed.
  + destination: The peg where disks need to be moved.
  + auxiliary: The peg used as a temporary storage.

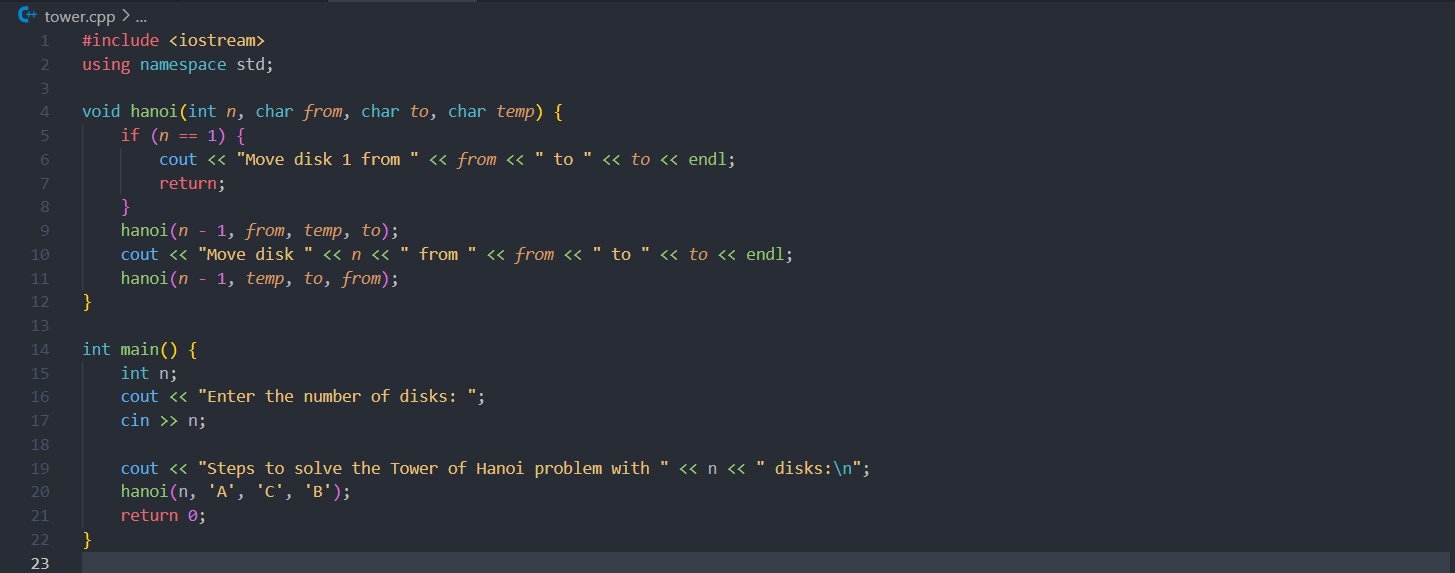
Base Case:

* If there is only one disk (n == 1), move it directly from the source peg to the destination peg.

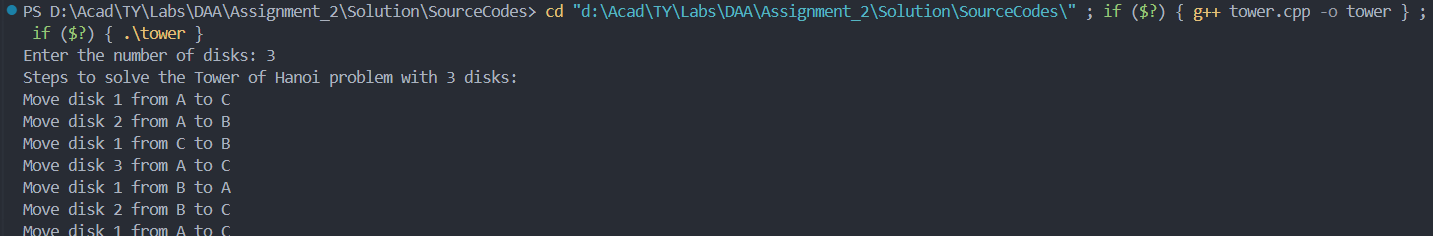
Recursive Case:

* Move the top n-1 disks from the source peg to the auxiliary peg using the destination peg as a temporary peg.
* Move the nth (largest) disk from the source peg to the destination peg.
* Move the n-1 disks from the auxiliary peg to the destination peg using the source peg as a temporary peg.

1. Program Code



1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.

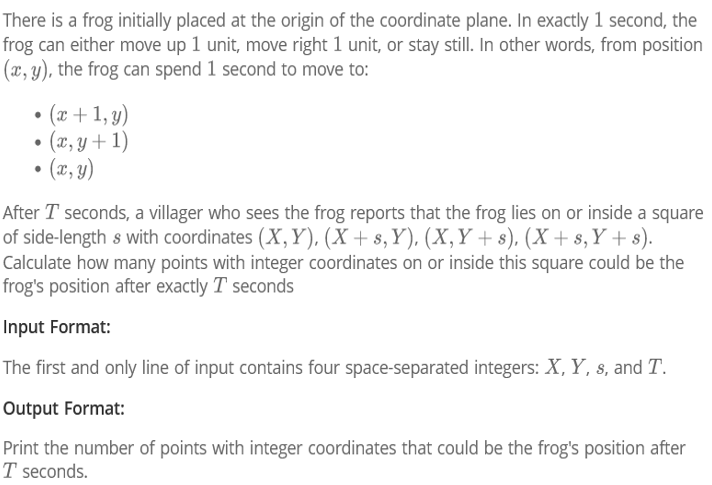
Time Complexity

1. Recursive Function Calls:
   * The Tower of Hanoi algorithm makes recursive calls to solve subproblems. Specifically, to solve the problem for n disks, it involves solving the problem for n-1 disks twice, plus one additional move.
   * If T(n) represents the time complexity for n disks, then the recursive relation is: T(n)=2T(n−1)+O(1)
   * The O(1) term accounts for the time to print the move, which is constant.
2. Solving the Recurrence Relation:
   * By solving the recurrence relation, we find: T(n)=2^n−1
   * This is derived from the fact that each recursive call generates twice the number of operations required for n-1 disks, plus one additional move.
3. Total Moves:
   * The number of moves required to solve the Tower of Hanoi problem is 2^n−1. This directly translates to the time complexity.
4. Time Complexity Summary:
   * Time Complexity: O(2n)

Space Complexity

1. Recursive Call Stack:
   * The depth of the recursion is n because each recursive call reduces the problem size by 1.
   * At each level of recursion, additional space is used for the function call and local variables.
2. Space Complexity Calculation:
   * Since the recursion depth is n, and each recursive call uses a constant amount of space, the space complexity is proportional to the recursion depth.
3. Space Complexity Summary:
   * Space Complexity: O(n)

**Problem Statement:**

****

1. Algorithm/Pseudocode

Determine the Frog's Reachable Positions After T Seconds:

After T seconds, the frog can reach any position (x, y) where x + y <= T and x, y >= 0. This is because the frog can move right x times and up y times with the condition that the total number of moves x + y is exactly T.

Find Valid Points Inside the Square:

The square is defined by the points (X, Y) to (X+s, Y+s).

A valid point (x, y) inside the square must satisfy:

X <= x <= X + s

Y <= y <= Y + s

Count the Number of Valid Points:

Loop over all possible integer values of x and y that satisfy the conditions:

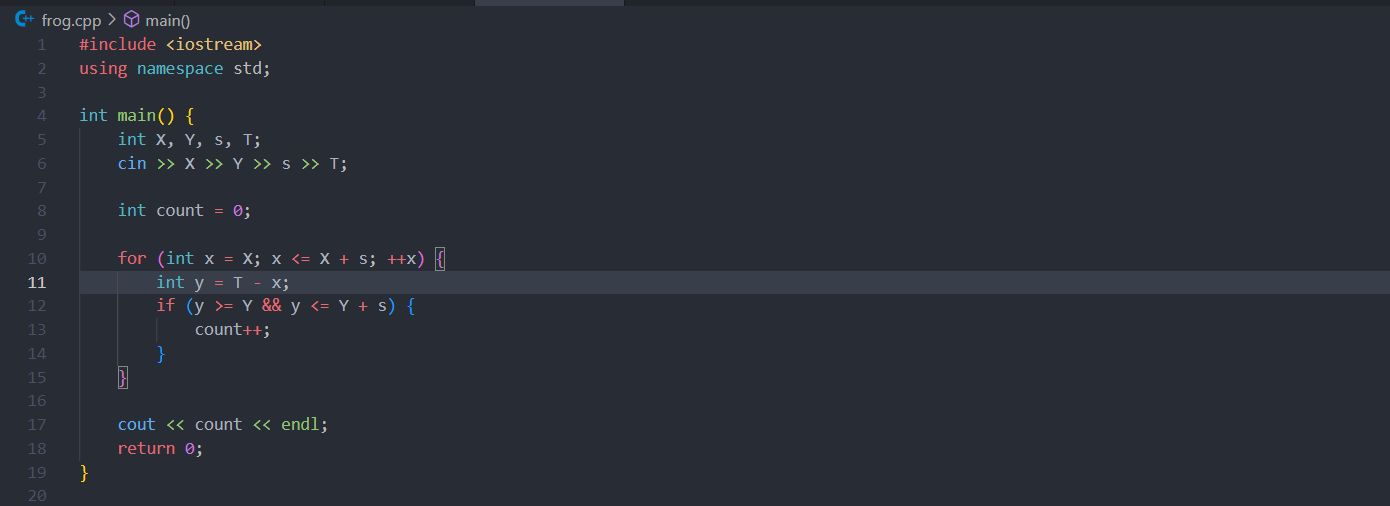
X <= x <= X + s

Y <= y <= Y + s

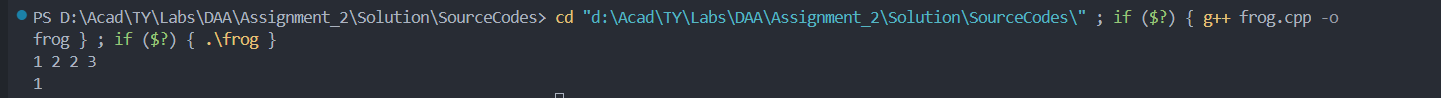
For each (x, y), check if the sum x + y is equal to T. If so, it is a valid point.

Increment a counter for each valid point found.

1. Program Code



1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.

Time Complexity

1. Outer Loop:
   * The loop iterates over all possible x values in the range [X, X+s].
   * This means the loop runs (s + 1) times, as x takes values from X to X + s inclusive.
2. Condition Check:
   * For each value of x, the corresponding y value is calculated as T - x.
   * The check y >= Y && y <= Y + s is a constant time operation, O(1)

Overall Time Complexity: Since the loop runs (s + 1) times and each iteration performs a constant-time check, the overall time complexity is O(s)

Space Complexity

1. Auxiliary Space:
   * The space used is constant, as only a few integer variables (X, Y, s, T, count, x, and y) are stored.
   * No additional data structures (like arrays or vectors) are used.

Overall Space Complexity: The space complexity is O(1) since the space usage does not depend on the input size.

Correctness

1. Looping over x:
   * The loop correctly covers all possible x values within the square from (X, Y) to (X+s, Y+s).
2. Calculating y:
   * The calculation of y as T - x ensures that we consider only those (x, y) pairs that can be reached in exactly T seconds, given the constraint x+y=T
3. Boundary Check:

The condition y >= Y && y <= Y + s ensures that the point (x, y) lies within the specified square.

Conclusion

Time Complexity: O(s)

Space Complexity: O(1)

**Problem Statement:**

**Implement linear search Algorithm.**

1. Algorithm/Pseudocode

**Start from the first element** of the array.

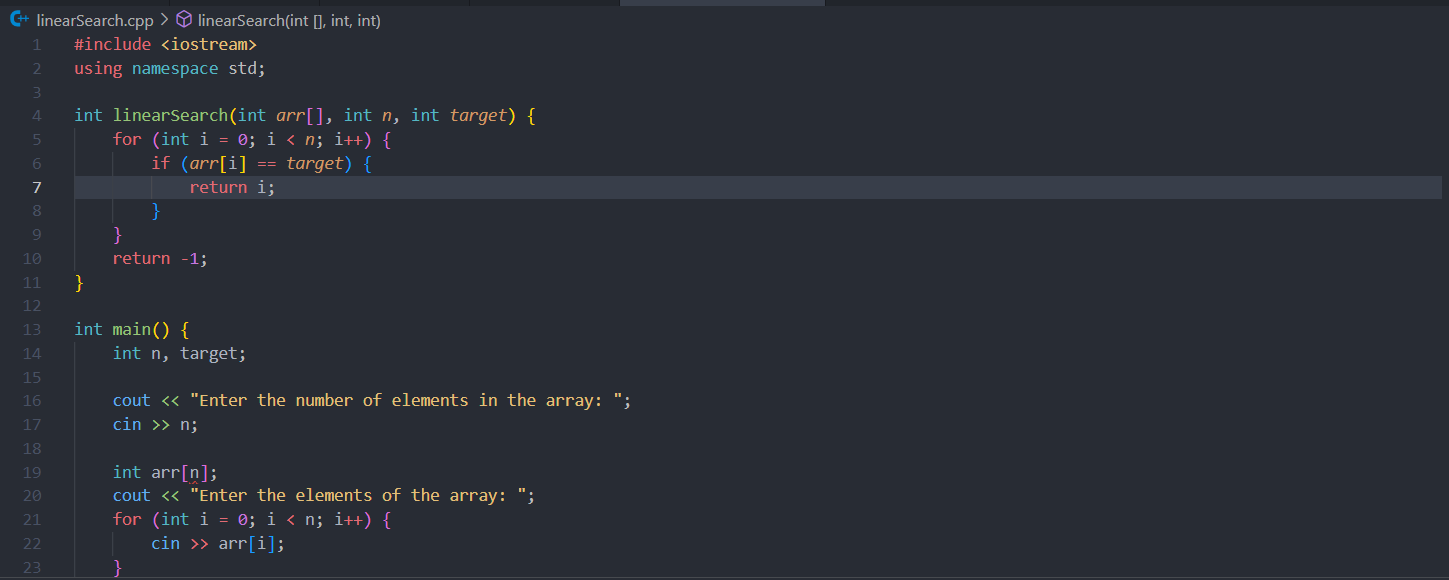
C**ompare** the target element with the current element.

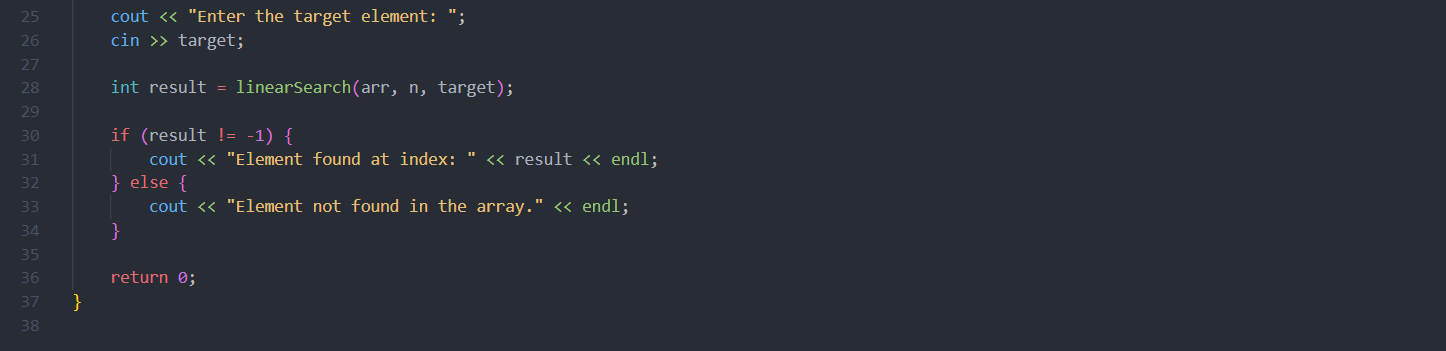
If the element matches the target, **return the index** of the element.

If the element does not match, move to the next element and repeat step 2.

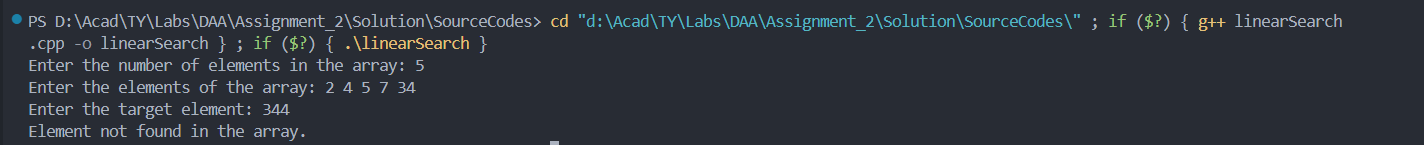
If the end of the array is reached without finding the element, **return -1** to indicate that the element is not present in the array.

1. Program Code





1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.

Time Complexity: O(n)

* The worst-case scenario requires the algorithm to check each element in the array (if the target element is the last element or not present at all).

Space Complexity: O(1)

* The algorithm uses a constant amount of space, independent of the size of the array.

**Problem Statement:**

Implement Binary Search algorithm.

1. Algorithm/Pseudocode

Start with the entire array: Set the low pointer to the first index and the high pointer to the last index.

Calculate the middle index: Find the middle element of the current search interval.

Compare the target element with the middle element:

If the target is equal to the middle element, return the middle index.

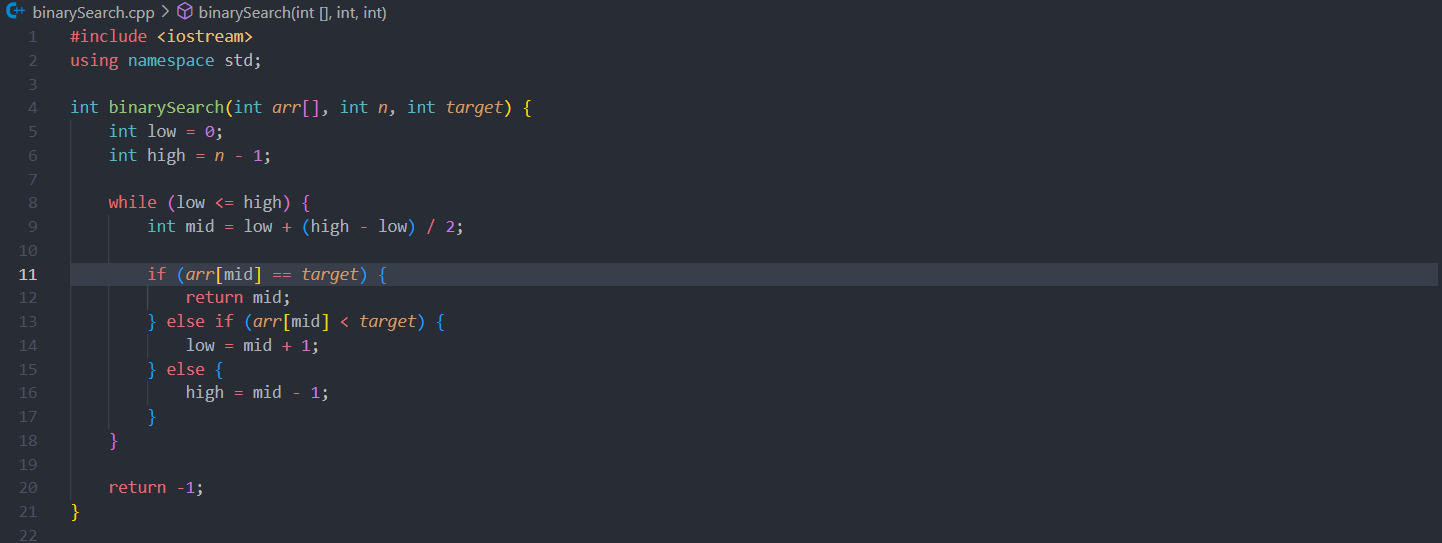
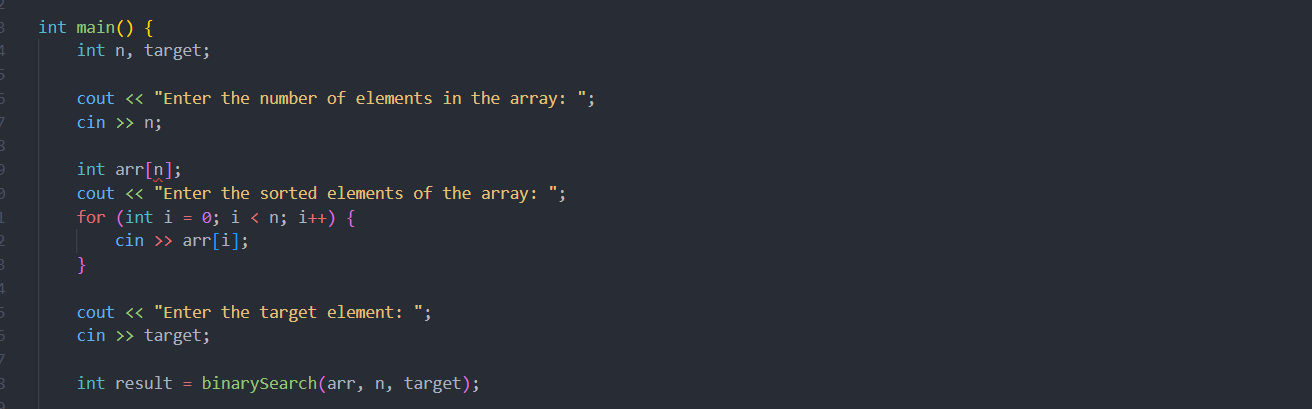
If the target is less than the middle element, narrow the search to the lower half by setting the high pointer to mid - 1.

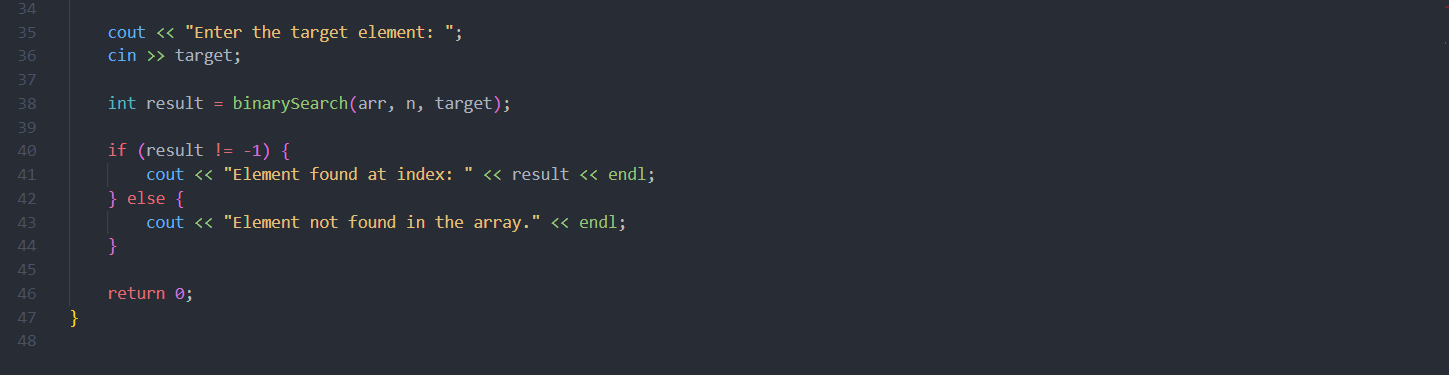
If the target is greater than the middle element, narrow the search to the upper half by setting the low pointer to mid + 1.

Repeat the process until the low pointer is greater than the high pointer.

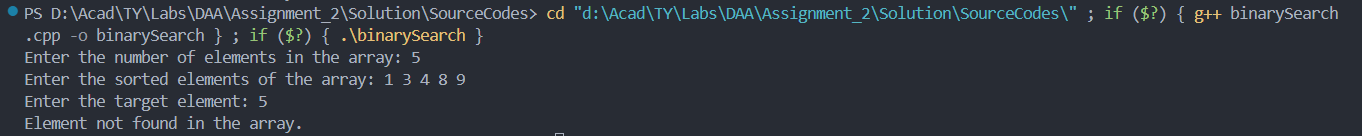
If the element is not found, return -1.

1. Program Code



1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.

Time Complexity**:** O(log n)

In each step, the size of the search interval is halved, which makes the search process logarithmic.

Space Complexity: O(1)

The algorithm uses a constant amount of extra space.