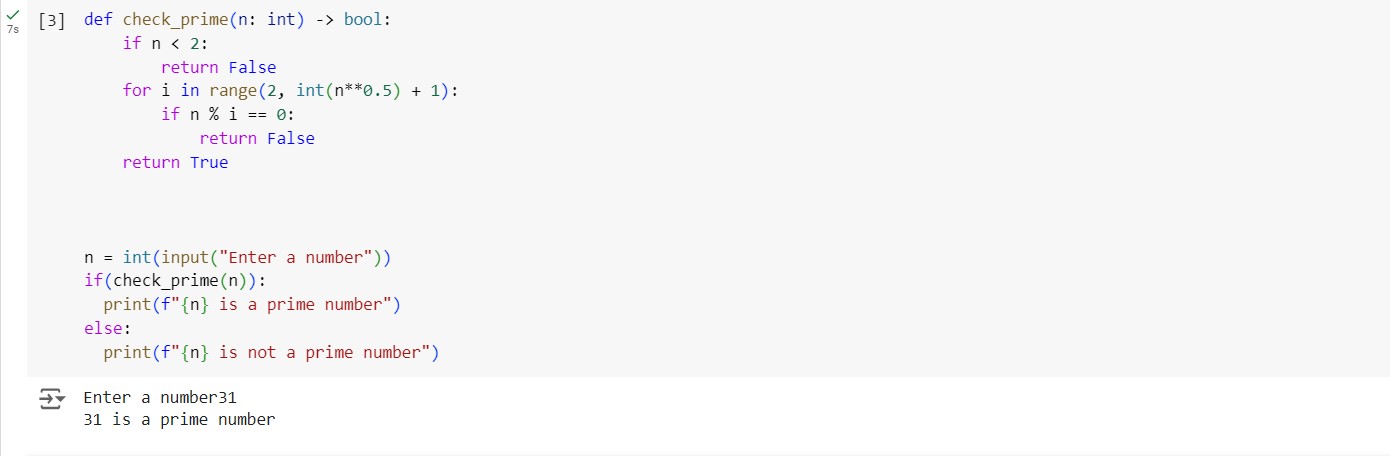
Practical – 1

**Problem Statement:** Write a program to determine whether the given number is Prime or not.

## Code:



### Algorithm Explanation:

#### Base Case Check:

* + If n is less than 2, it immediately returns False, because numbers less than 2 are not prime.

#### Prime Checking Loop:

* + The function iterates through numbers from 2 up to the square root of n

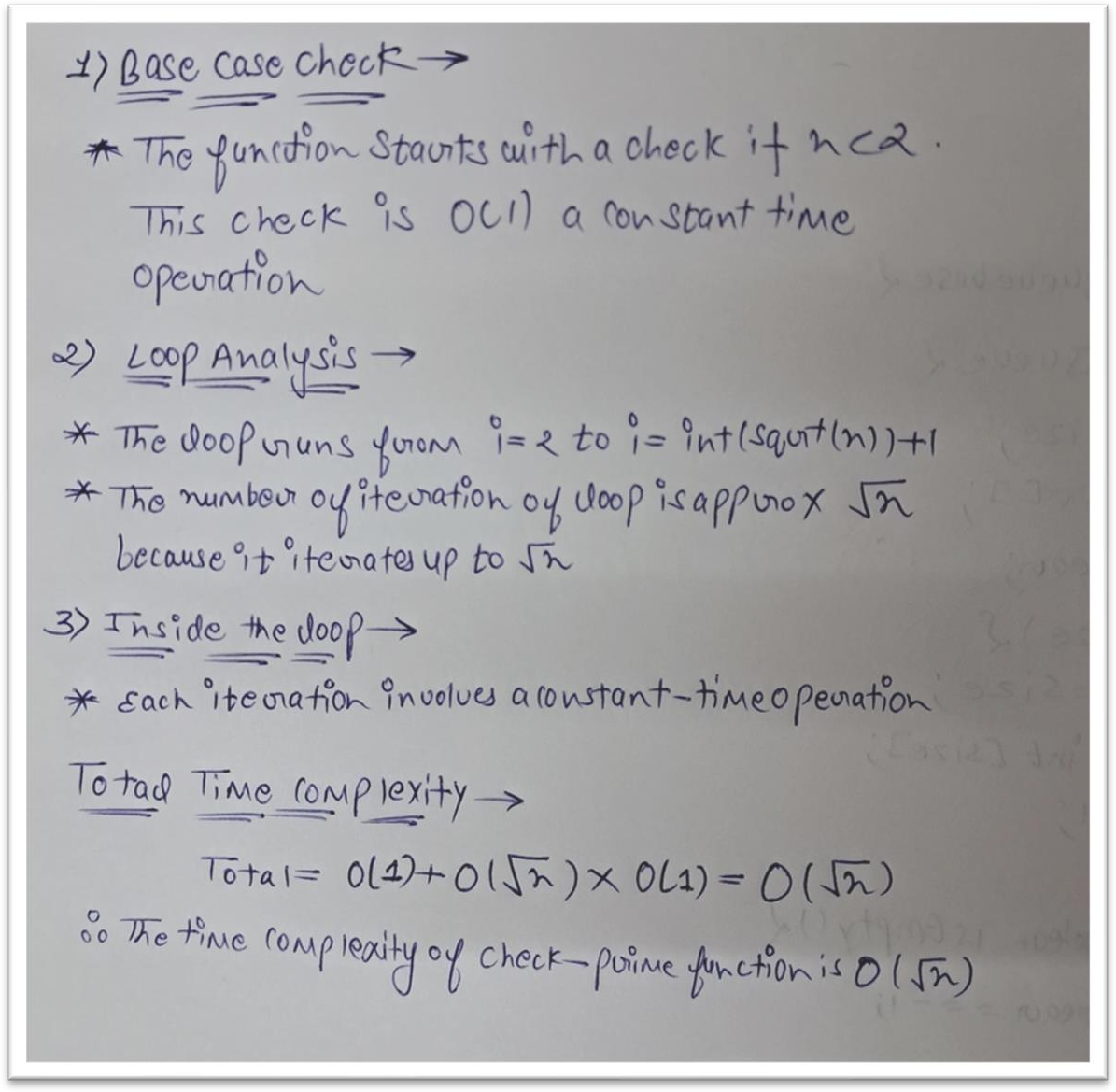
(int(n\*\*0.5) + 1).

* + For each integer i in this range, it checks if n modulo i equals zero (n % i == 0):
    - If true, n is divisible by i and hence not prime, so it returns False.
    - If no divisors are found in this range, it concludes that n is prime and returns True.

#### Efficiency:

* + The loop runs approximately up to sqrt(n) times, which significantly reduces the number of checks compared to iterating up to n.
  + This makes the function efficient for large values of n.

### Time Complexity:



**Space Complexity:**

o **O(1)**: The function uses a constant amount of space.

# Practical –2

**Problem Statement:** Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

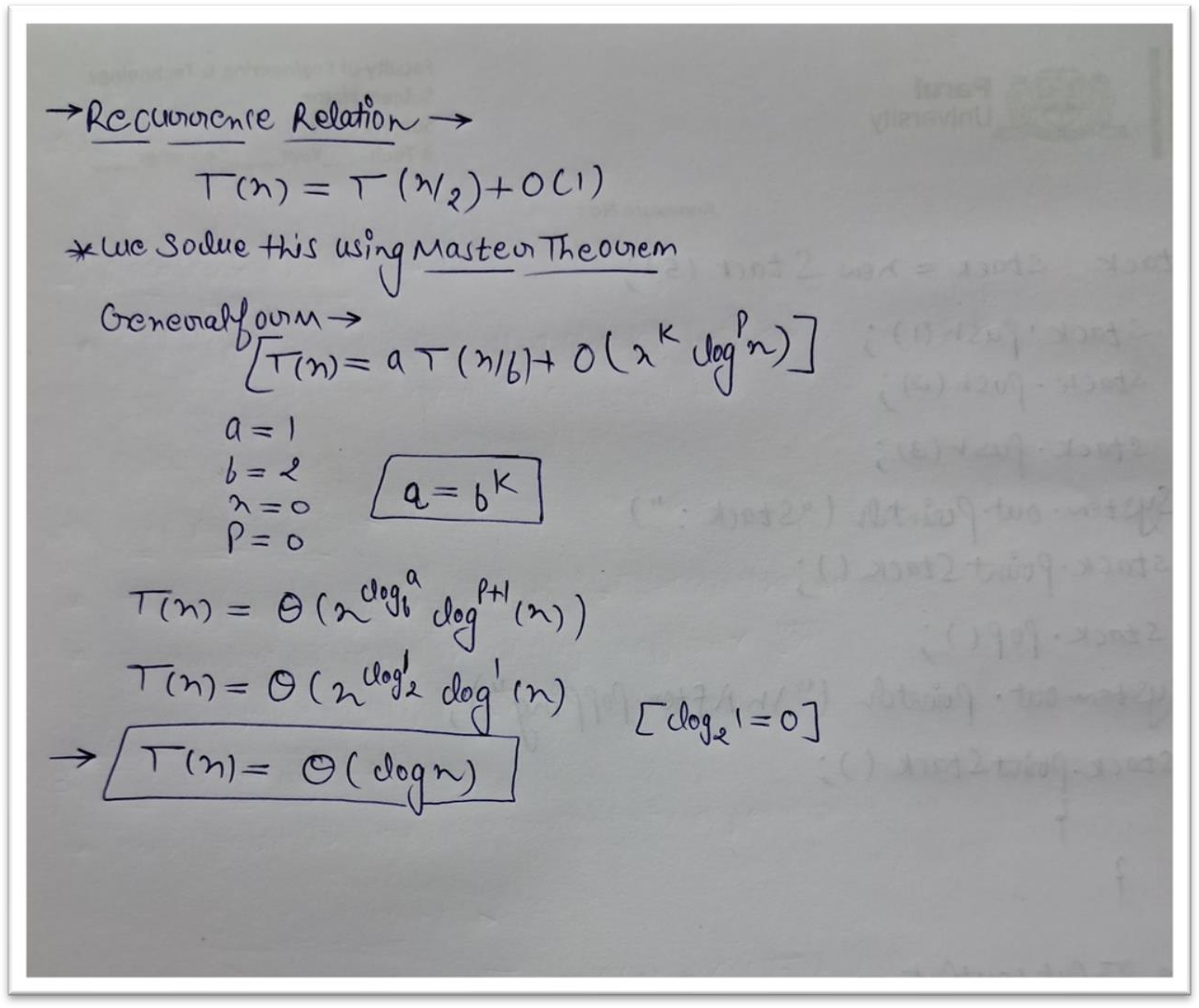
# Code:



### Algorithm Explanation:

* Initialize two pointers, **left** and **right**, to the start and end of the array respectively (left = 0, right = len(nums) - 1).
* While left is less than or equal to right:
  + Calculate the middle index mid as mid = (left + right) // 2.
  + Compare the middle element nums[mid] with target:
    - If nums[mid] == target, return mid (target found).
    - If nums[mid] < target, move the left pointer to mid + 1 (search in the right half).
    - If nums[mid] > target, move the right pointer to mid - 1 (search in the left half).
* If the loop exits without finding the target (left > right), return the left pointer which indicates the insertion point.

**Time Complexity**:



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

**Space Complexity**: O(1) since we use only a constant amount of extra space

# Practical – 3

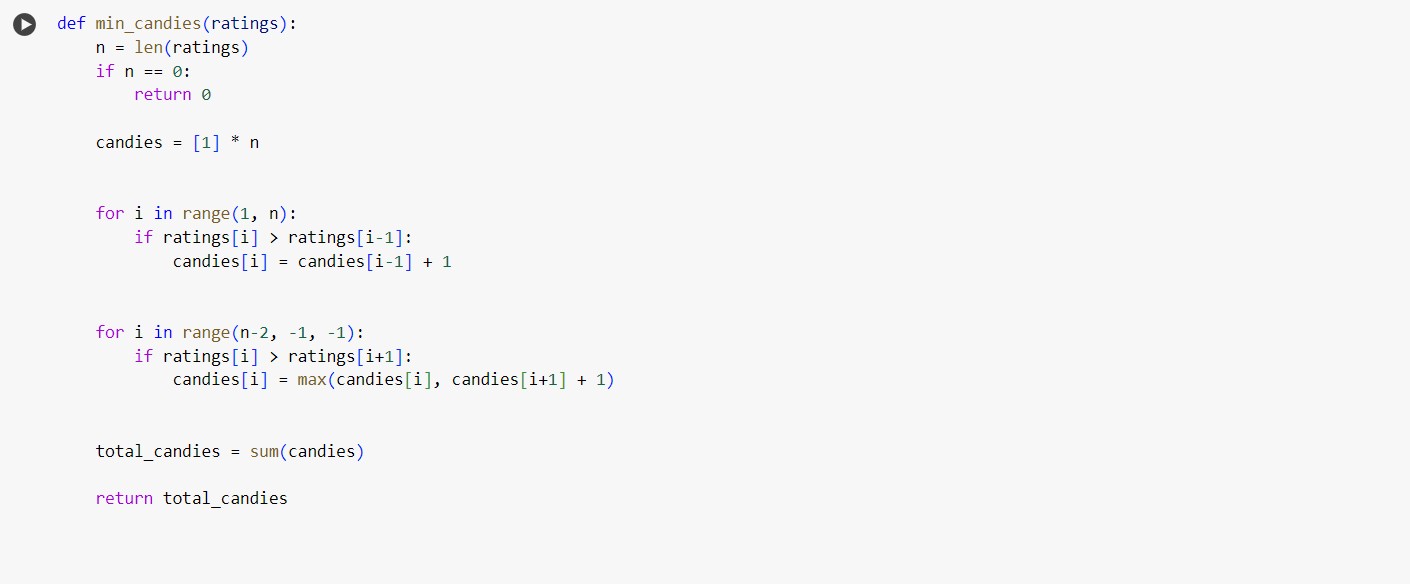
**Problem Statement:** There are n children standing in a line. Each child is assigne rating value given in the integer array ratings.

You are giving candies to these children subjected to the

* Each child must have at least one candy
* Children with a higher rati

Return the minim the

# Code:



### Algorithm Explanation:

#### Initialization:

* + Create an array candies initialized to all ones, as each child must receive at least one candy initially.

#### First Pass (Left to Right):

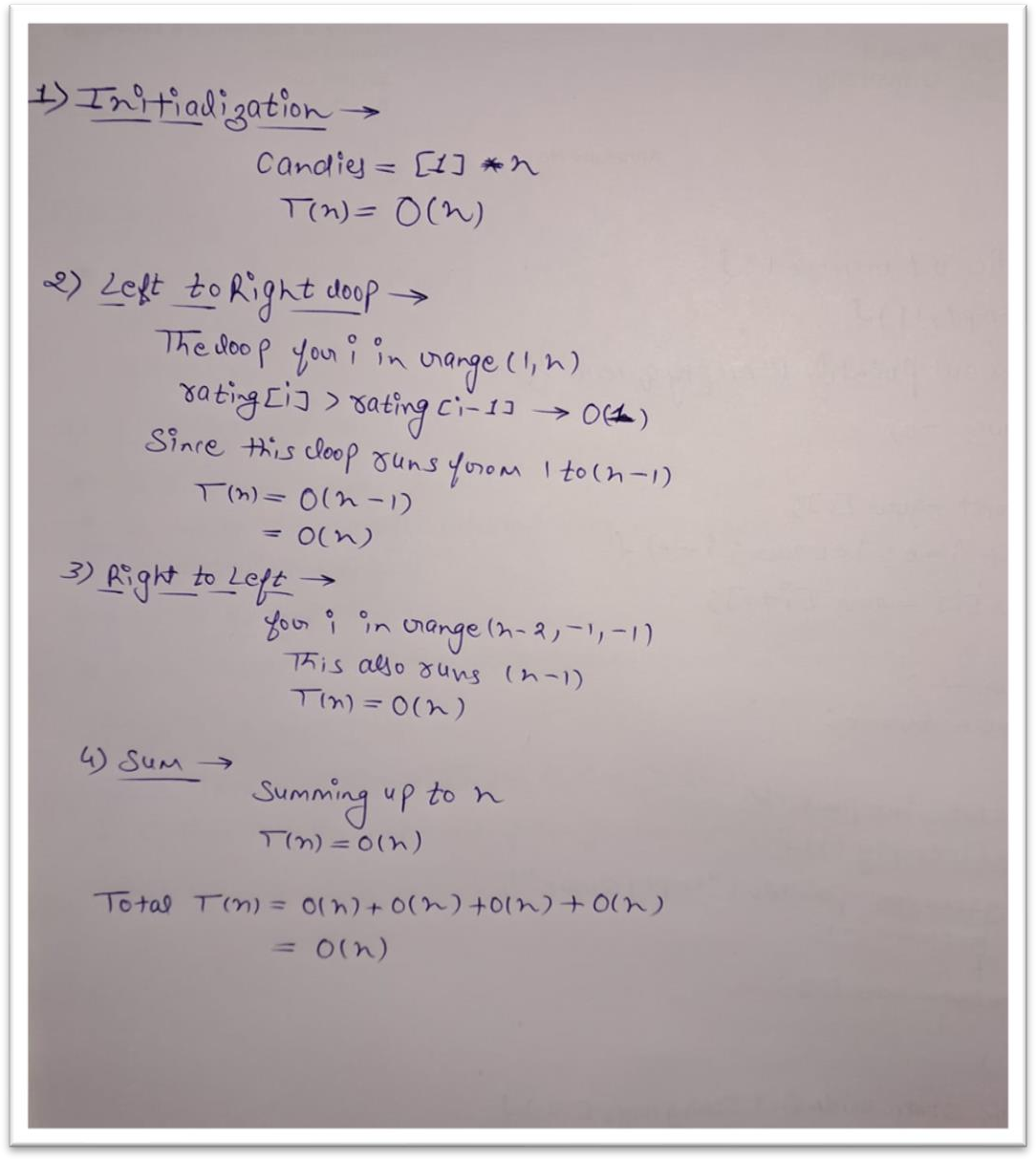
* + Traverse the ratings array from left to right.
  + If a child's rating is greater than the previous child's rating (ratings[i] > ratings[i- 1]), assign candies[i] = candies[i-1] + 1. This ensures that a child with a higher rating gets more candies than the previous child.

#### Second Pass (Right to Left):

* + Traverse the ratings array from right to left.
  + Adjust the candies count for each child again:
  + If a child's rating is greater than the next child's rating (ratings[i] > ratings[i+1]) and the current candies count (candies[i] <= candies[i+1]), update candies[i] to ensure that the child with higher rating gets more candies than the next child.

#### Compute Total Candies:

* + Sum up all the values in the candies array to get the total minimum number of candies needed.

**Time Complexity:**

# Space Complexity:

* + **Input Size**: The ratings array of size n requires O(n) space.
  + **Candies Array**: An additional array candies of size n is used to track candies given to each child, also requiring O(n) space.
  + **Additional Variables**: Variables like n, loop counters (i), and

temporary variables (ratings[i], ratings[i-1], ratings[i+1]) occupy constant O(1) space.

* + **Total Space Complexity**: Combining the above, the total space complexity is **O(n),** where n is the number of children (length of the ratings array).