DAA-HOLIDAY ASSIGNMENT

Name: M.Sreeram

Rollno: 2311CS020407(omega)

1. Check whether given number is Palindrome or not.

Ans. bool isPalindrome(int x) {

if (x < 0 || (x % 10 == 0 && x != 0)) return false;

int rev = 0, original = x; while (x > rev) {

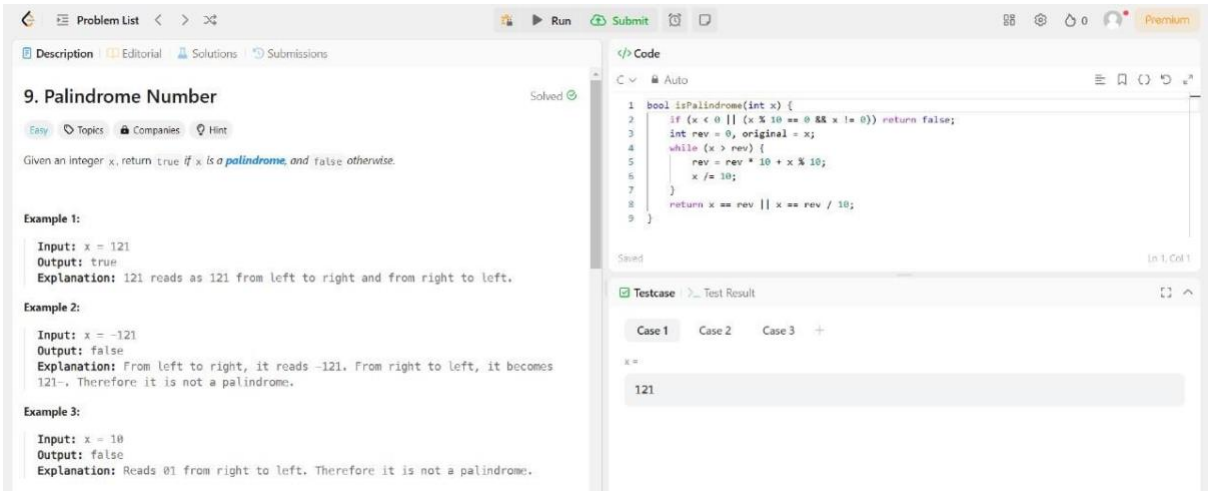
rev = rev * 10 + x % 10; x /= 10;

}

return x == rev || x == rev / 10;

}

Output:



2. Convert the Roman to integer. Ans. int

romanToInt(char* s) { int res = 0, prev = 0, curr = 0;

while (*s) { switch (*s++) { case 'I': curr = 1;

break; case 'V': curr = 5; break; case 'X':

curr = 10; break; case 'L': curr = 50; break;

case 'C': curr = 100; break; case 'D': curr = 500;

break; case 'M': curr = 1000; break;

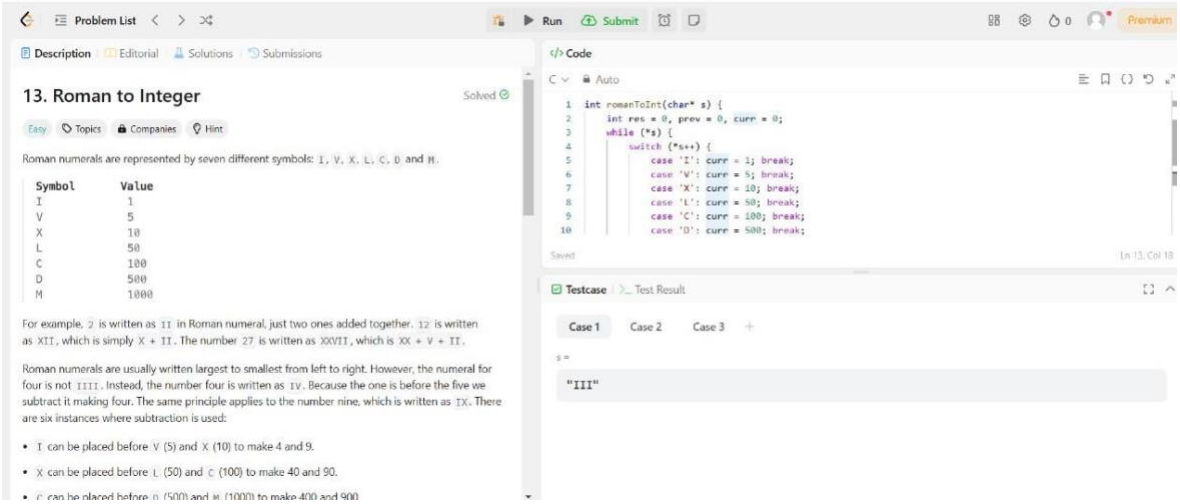
}

res += (curr > prev) ? curr - 2 * prev : curr;

prev = curr;

```
    }     return
res;
}
```

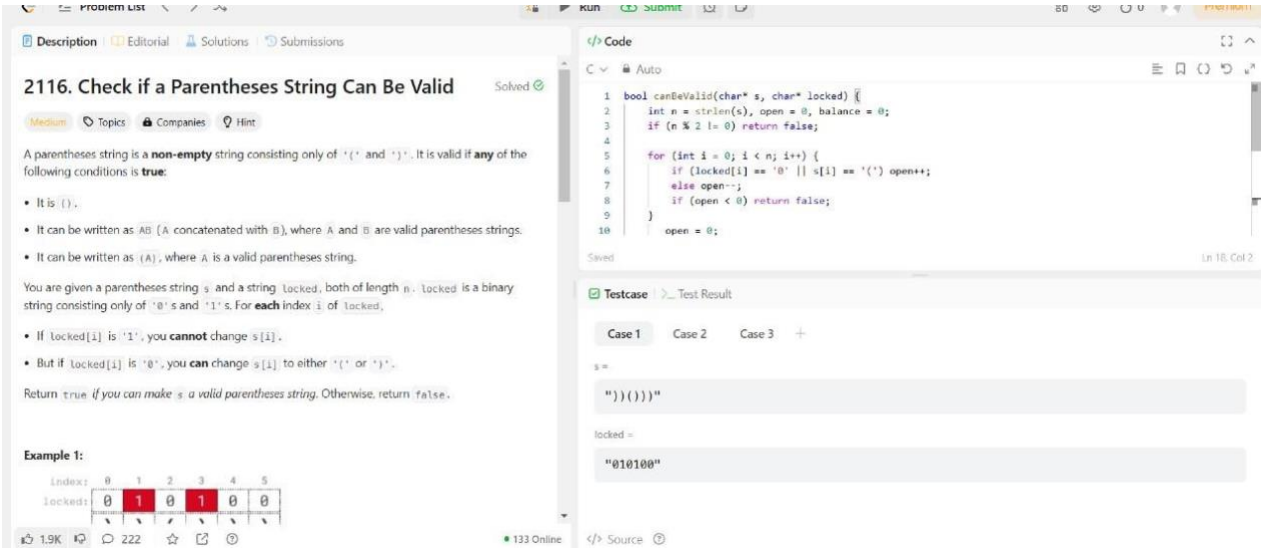
Output:



3. Validating opening and closing parenthesis in a String

```
Ans. ool canBeValid(char* s, char* locked) {     int n =
strlen(s), open = 0, balance = 0;     if (n % 2 != 0) return
false;
   for (int i = 0; i < n; i++) {         if
(locked[i] == '0' || s[i] == '(') open++;
else open--;
       if (open < 0) return false;
   }        open
= 0;
  for (int i = n - 1; i >= 0; i--) {
       if (locked[i] == '0' || s[i] == ')') open++;
      else open--;
       if (open < 0) return false;
   }     return
true;
}
```

Output:

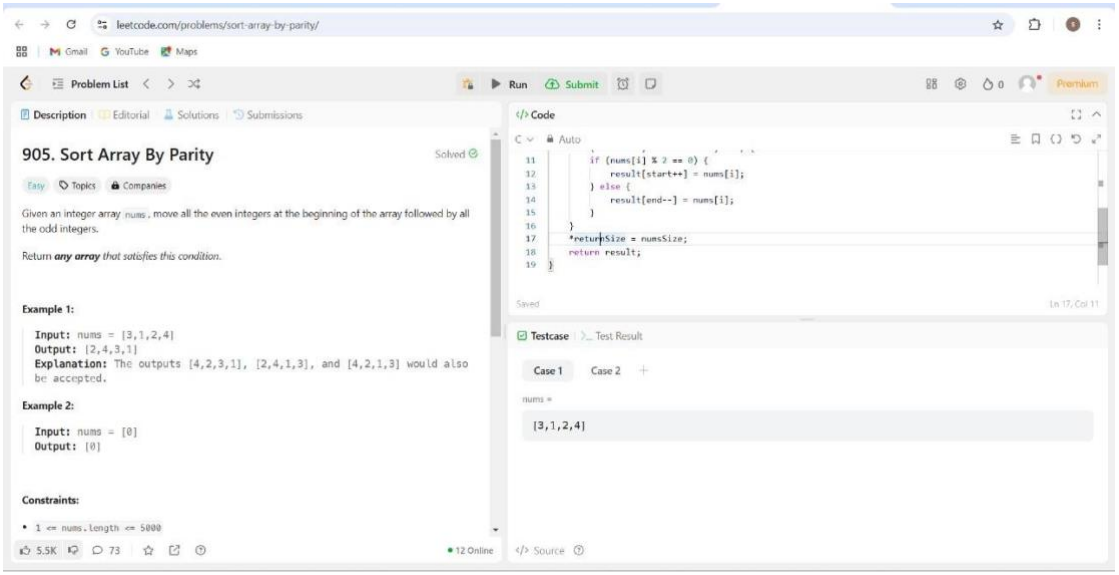


4. Finding Odd and even numbers in an Array

Ans. #include <stdlib.h>

```
int* sortArrayByParity(int* nums, int numsSize, int* returnSize) {
int* result = (int*)malloc(numsSize * sizeof(int));     int start = 0,
end = numsSize - 1;

    for (int i = 0; i < numsSize; i++) {
if (nums[i] % 2 == 0) {
result[start++] = nums[i];
        } else {
            result[end--] = nums[i];
        }
    }
    *returnSize = numsSize;
    return result;
}
```

Output:



5. Find all symmetric pairs in array of pairs. Given an array of pairs of integers, find all symmetric pairs, i.e., pairs that mirror each other. For instance, pairs (x, y) and (y, x) are mirrors of each other.

Ans. #include <stdlib.h>

int cmp(const void* a, const void* b) {

return (*(int*)a - *(int*)b);

}

int findPairs(int* nums, int numsSize, int k) {

qsort(nums, numsSize, sizeof(int), cmp);

int count = 0, left = 0, right = 1; while

(right < numsSize) {

    if (left == right || nums[right] - nums[left] < k) {

right++;

    } else if (nums[right] - nums[left] > k) {

left++;

    } else {

count++;

        left++;

right++;

        while (right < numsSize && nums[right] == nums[right - 1]) right++;

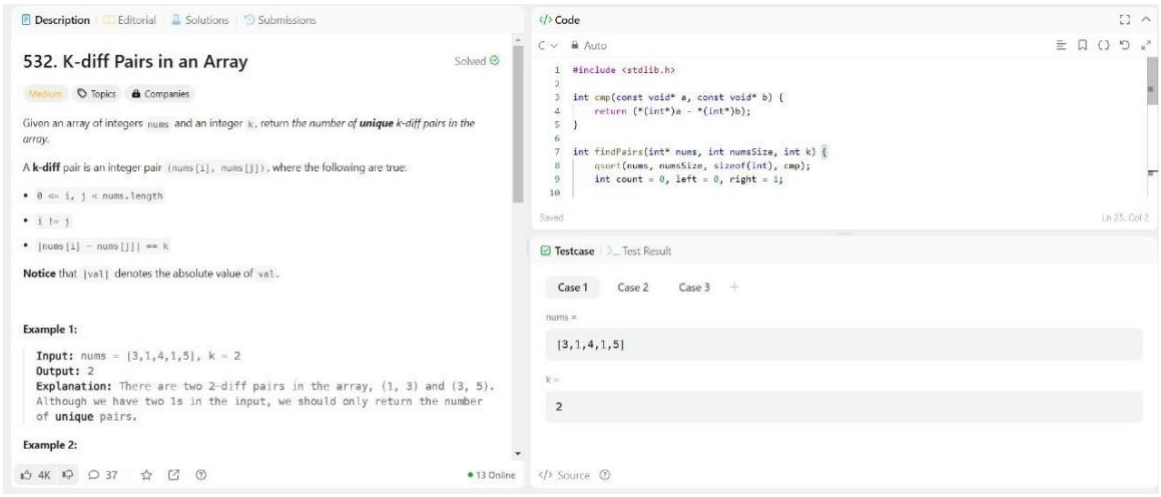
    }

  }

 return count;

}

Output:



6. Find the Kth smallest element in an Array using function.

Ans. #include <stdio.h>

#include <stdlib.h> struct

MinHeap {

int* arr;

int size; int

capacity;

};

void swap(int* a, int* b) {

int temp = *a; *a = *b;

*b = temp;

}

void heapify(struct MinHeap* heap, int idx) {

int smallest = idx; int left = 2 * idx + 1;

int right = 2 * idx + 2;

if (left < heap->size && heap->arr[left] < heap->arr[smallest]) {

smallest = left;

}

if (right < heap->size && heap->arr[right] < heap->arr[smallest]) {

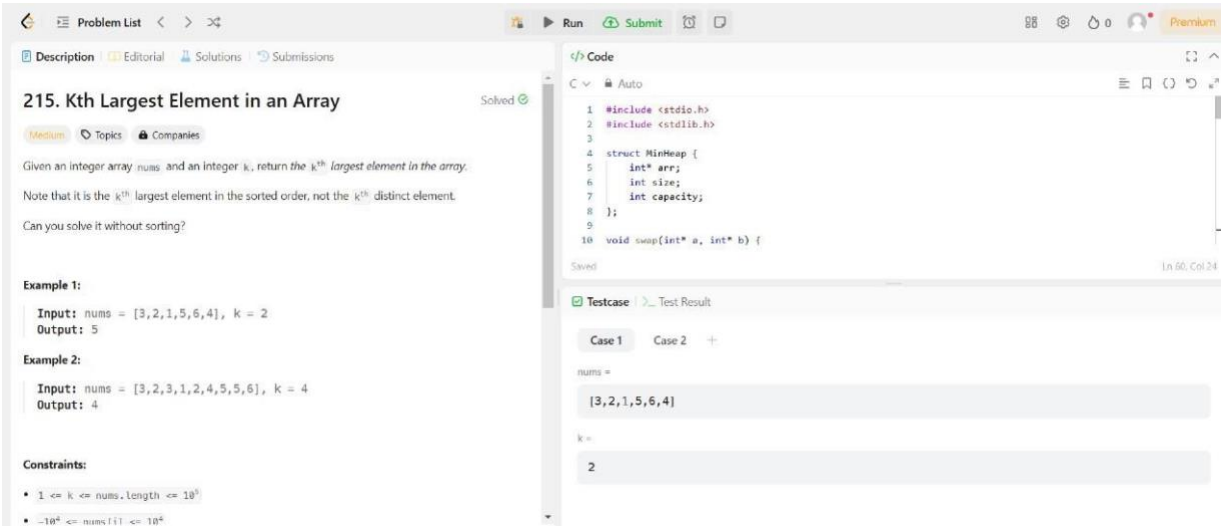
smallest = right;

} if (smallest !=

idx) {

```
        swap(&heap->arr[smallest], &heap->arr[idx]);
heapify(heap, smallest);
    }
}
void insertMinHeap(struct MinHeap* heap, int val) {
if (heap->size < heap->capacity) {         heap-
>arr[heap->size] = val;         heap->size++;         int i
= heap->size - 1;
        while (i > 0 && heap->arr[(i - 1) / 2] > heap->arr[i]) {
swap(&heap->arr[(i - 1) / 2], &heap->arr[i]);
            i = (i - 1) / 2;
        }
    } else if (val > heap->arr[0]) {
heap->arr[0] = val;         heapify(heap,
0);
    }
}
int findKthLargest(int* nums, int numsSize, int k) {
struct MinHeap heap;     heap.arr = (int*)malloc(k *
sizeof(int));     heap.size = 0;     heap.capacity = k;
for (int i = 0; i < numsSize; i++) {
insertMinHeap(&heap, nums[i]);
    }     return
heap.arr[0];
}
```

Output:



7. Create a structure named Complex to represent a complex number with real and imaginary parts. Write a C program to add and multiply two complex numbers.

Ans. #include <stdio.h> #include

<stdlib.h>

int* plusOne(int* digits, int digitsSize, int* returnSize) {

int carry = 1;

for (int i = digitsSize - 1; i >= 0; i--) {

digits[i] += carry; if (digits[i] ==

10) { digits[i] = 0; carry

= 1; } else { carry = 0;

break;

}

}if (carry) {

*returnSize = digitsSize + 1;

int* result = (int*)malloc(sizeof(int) * (*returnSize));

result[0] = 1;

for (int i = 1; i < *returnSize; i++) {

result[i] = digits[i - 1];

}

return result;

} else {

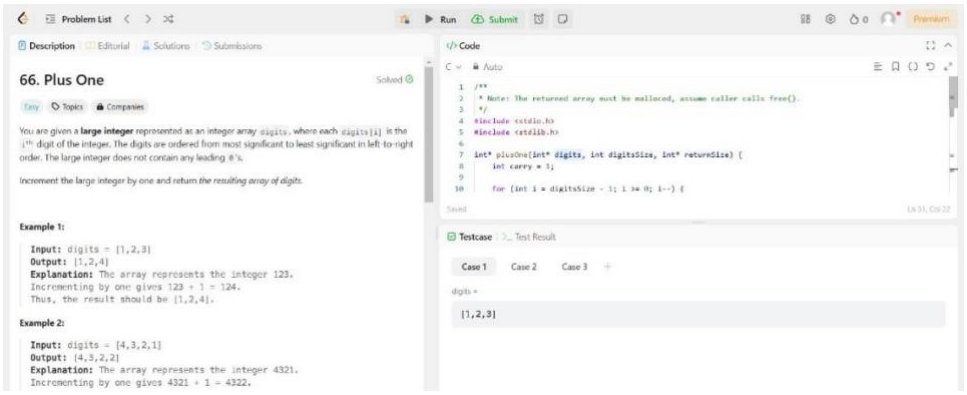
*returnSize = digitsSize;

return digits;

}

}

Output:



8. Find the missing and duplicate number in an Array Ans. int findDuplicate(int* nums, int numsSize) { int slow = nums[0], fast = nums[0]; do {

slow = nums[slow];

fast = nums[nums[fast]]; }

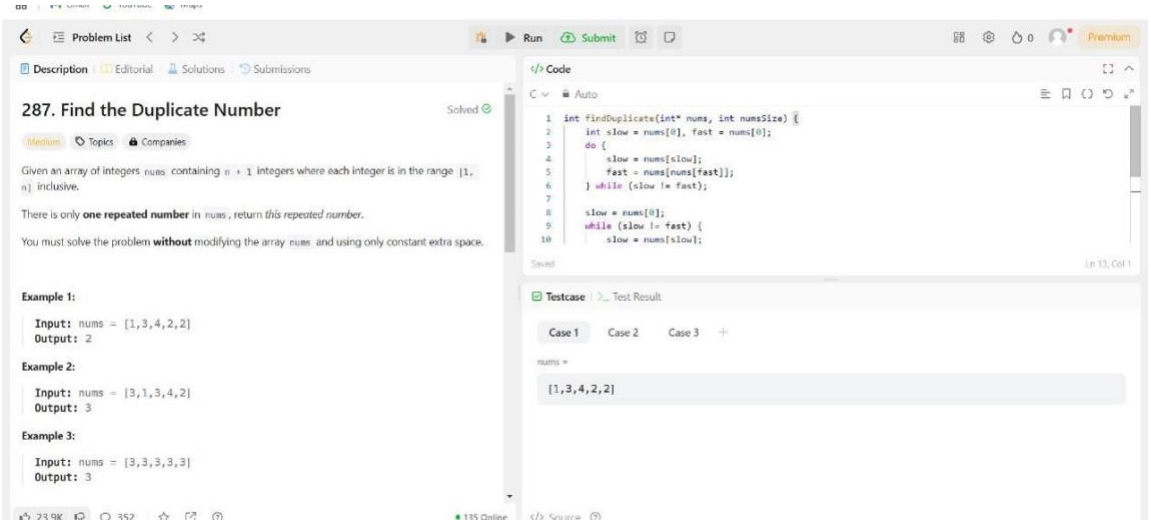
while (slow != fast); slow

= nums[0];

while (slow != fast) { slow = nums[slow];

fast = nums[fast];

}return slow;}



9. Write C program to determine if a number n is happy. A happy number is a number defined by the following process: **Input**: n = 19

Output: true

Explanation:

$1^2 + 9^2 = 82$

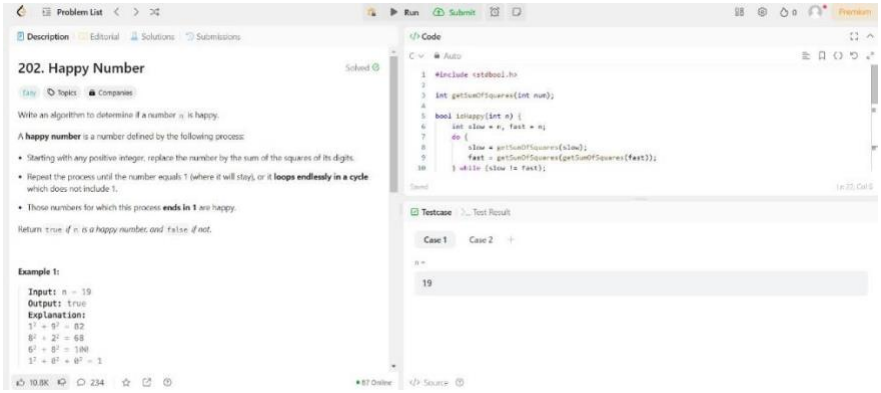
$8^2 + 2^2 = 68$

$6^2 + 8^2 = 100$

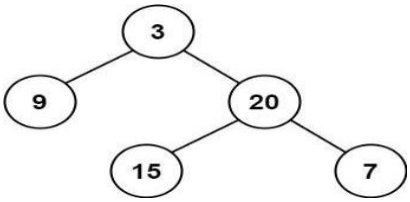
$1^2 + 0^2 + 0^2 = 1$ Ans.

```
#include <stdbool.h> int
getSumOfSquares(int num);
bool isHappy(int n) {     int
slow = n, fast = n;     do {
        slow = getSumOfSquares(slow);
        fast = getSumOfSquares(getSumOfSquares(fast));
    } while (slow != fast);
return slow == 1;
}
int getSumOfSquares(int num) {
int sum = 0;     while (num > 0)
{         int digit = num % 10;
sum += digit * digit;         num
/= 10;
    }
   return sum;}
```

Output:



10. Given a binary tree, determine if it is height-balanced:

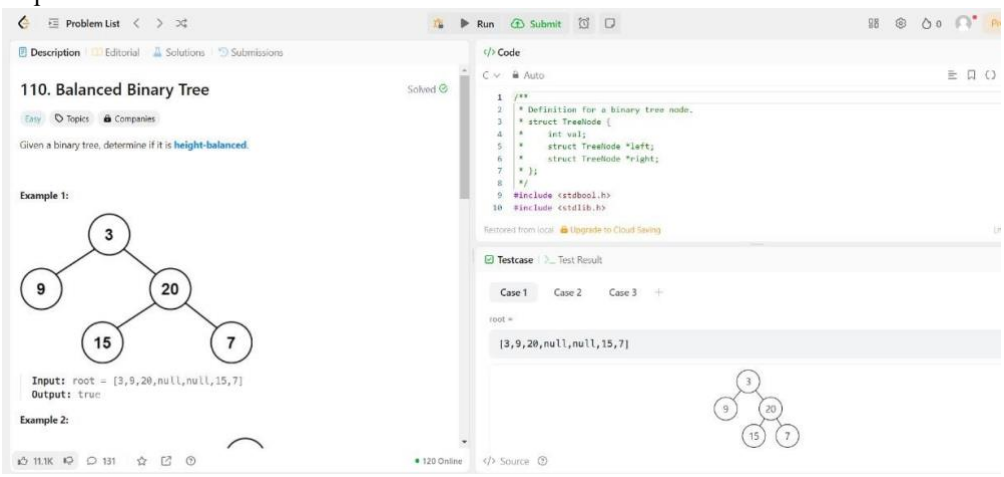


Input: root = [3, 9, 20, null, null, 15, 7]
Output: true

Ans. #include <stdbool.h>

```
#include <stdlib.h>

int height(struct TreeNode* root) {
    if (root == NULL) return 0;

    int leftHeight = height(root->left);
    if (leftHeight == -1) return -1;

    int rightHeight = height(root->right);
if (rightHeight == -1) return -1;
    if (abs(leftHeight - rightHeight) > 1) return -1;

    return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
}

bool isBalanced(struct TreeNode* root) {
    return height(root) != -1; }
```

Output:



Case Study:

Perform Quick sort for the following Array.

10, 16, 8, 12, 15, 6, 3, 9, 5.

Analyze Best case, Worst case and Average case Time complexities.

Give an example for array of elements which takes Maximum Time and explain.

Ans. Time Complexities of Quick Sort: Detailed Mathematical Explanation

The time complexity of Quick Sort depends on the number of comparisons made during partitioning and the number of recursive calls. Let us analyze the best case, worst case, and average case step by step.

1. Best Case: O(nlogn)

When does it happen? The best case occurs when the pivot divides the array into two equal (or nearly equal) halves at each step. For example, choosing the median as the pivot ensures a balanced partition.

Mathematical Analysis:

Number of Levels in Recursion Tree: At each level of recursion, the array is divided into two halves. Starting with an array of size n, the number of levels required to reduce each subarray to size 1 is approximately logn, since halving an array repeatedly results in n divisions.

Number of Comparisons per Level: At each level of the recursion tree, all n

elements of the array are compared during the partitioning step.

Total Number of Comparisons: Since the partitioning happens for n levels and each level processes all n elements, the total number of comparisons is:

Total Comparisons=n+n+n+…(log n times)=n·log n Hence,

the time complexity in the best case is:

$O(n\log n)$

2. Worst Case:

$O(n\text{^}2)$

When does it happen? The worst case occurs when the pivot chosen is the smallest or largest element in the array at each step, resulting in highly unbalanced partitions. For example, sorting an already sorted array or reverse-sorted array with the first or last element as the pivot will lead to this situation.

Mathematical Analysis:

Number of Levels in Recursion Tree: At each level, only one element (the pivot) is placed in its correct position, leaving the rest of the array (size n−1) to be sorted. This means there are n levels in the recursion tree.

Number of Comparisons per Level: At the first level, all n elements are compared during partitioning. At the second level, n−1 elements are compared. At the third level, n−2 elements are compared, and so on.

Total Number of Comparisons: The total number of comparisons is the sum of the first n natural numbers:

Total Comparisons=n+(n−1)+(n−2)+⋯+1

Using the formula for the sum of the first n natural numbers:

Total Comparisons= 2n(n+1)

This simplifies to O(n^2).

3. Average Case:

O(nlogn)

When does it happen? In the average case, the pivot divides the array into two partitions that are roughly proportional in size, but not necessarily equal. On average, each pivot divides the array into partitions of sizes approximately 4n and 3n Mathematical Analysis:

Number of Levels in Recursion Tree: Similar to the best case, the number of levels in the recursion tree is approximately logn, because the array is divided into smaller and smaller partitions.

Number of Comparisons per Level: At each level, the partitioning step processes all n elements of the array

Expected Total Number of Comparisons: To calculate the expected number of comparisons, we use a recurrence relation. Let T(n) represent the total time taken to sort an array of size n. Partitioning takes O(n) time, and the array is divided into two subarrays of sizes i and n−i−1 (where i is the position of the pivot). Thus, we write:

T(n)=T(i)+T(n−i−1)+O(n)

Averaging over all possible pivots, we assume i is equally likely to be any position 0,1,2,…,n−1. Taking the average, we sum over all values of i:

$T(n) = \sum_{i=0}^{n-1}[T(i) + T(n-i-1)] + O(n)$

Simplifying the recurrence relation and solving using advanced techniques (such as integration or the Master Theorem), the solution converges to:

$T(n)=O(n\log n)$