

DAA-HOLIDAY ASSIGNMENT

Name: L.Yaswanth

Rollno: 2311CS020372(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:

The screenshot shows a code editor interface for a problem titled "9. Palindrome Number". The problem description states: "Given an integer x, return true if x is a palindrome, and false otherwise." It includes three examples: Example 1 (Input: 121, Output: true), Example 2 (Input: -121, Output: false), and Example 3 (Input: 10, Output: false). The code editor shows the following C++ code:

```
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;
}
```

The test case section shows "Case 1" with input "121" and output "true".

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:

13. Roman to Integer

Easy Topics Companies Hint

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

- I can be placed before V (5) and X (10) to make 4 and 9.
- X can be placed before L (50) and C (100) to make 40 and 90.
- C can be placed before D (500) and M (1000) to make 400 and 900.

```

1 int romanToInt(char* s) {
2     int res = 0, prev = 0, curr = 0;
3     while (*s) {
4         switch (*s++) {
5             case 'I': curr = 1; break;
6             case 'V': curr = 5; break;
7             case 'X': curr = 10; break;
8             case 'L': curr = 50; break;
9             case 'C': curr = 100; break;
10            case 'D': curr = 500; break;

```

Testcase 1: s = "III"

3. Validating opening and closing parenthesis in a String

Ans. bool 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:

2116. Check if a Parentheses String Can Be Valid Solved

Medium Topics Companies Hint

A parentheses string is a **non-empty** string consisting only of '(' and ')'. It is valid if **any** of the following conditions is **true**:

- It is ().
- It can be written as AB (A concatenated with B), where A and B are valid parentheses strings.
- It can be written as (A), where A is a valid parentheses string.

You are given a parentheses string s and a string locked, both of length n. locked is a binary string consisting only of '0's and '1's. For **each** index i of locked,

- If locked[i] is '1', you **cannot** change s[i].
- But if locked[i] is '0', you **can** change s[i] to either '(' or ')'.
Return true if you can make s a valid parentheses string. Otherwise, return false.

Example 1:

Index:	0	1	2	3	4	5
locked:	0	1	0	1	0	0

1.9K 222 133 Online

```

1 bool canBeValid(char* s, char* locked) {
2     int n = strlen(s), open = 0, balance = 0;
3     if (n % 2 != 0) return false;
4
5     for (int i = 0; i < n; i++) {
6         if (locked[i] == '0' || s[i] == '(') open++;
7         else open--;
8         if (open < 0) return false;
9     }
10    open = 0;

```

Testcase Test Result

Case 1 Case 2 Case 3 +

s =
"))()())"

locked =
"010100"

4. Finding Odd and even numbers in an Array

Ans. #include <stdlib.h>

```

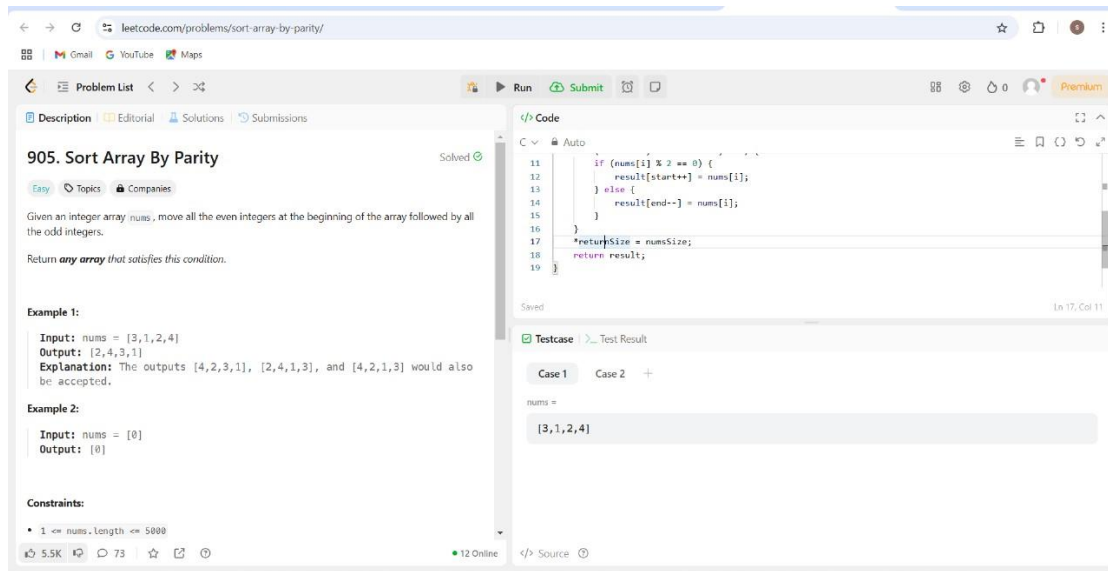
int* sortArrayByParity(int* nums, int numsSize, int* returnSize) {
    int* result = (int*)malloc(numsSize * sizeof(int));
    int start = 0, end = numsSize - 1;

    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:

The screenshot shows the LeetCode interface for problem 532, "K-diff Pairs in an Array". The problem description states: "Given an array of integers `nums` and an integer `k`, return the number of **unique** `k`-diff pairs in the array." It defines a `k`-diff pair as an integer pair `(nums[i], nums[j])` where `0 <= i, j < nums.length`, `i != j`, and `|nums[i] - nums[j]| == k`. A notice mentions that `|val|` denotes the absolute value of `val`. Example 1 shows input `nums = [3,1,4,1,5]` and `k = 2`, with output `2`. The explanation states there are two 2-diff pairs: (1, 3) and (3, 5), and that duplicate 1s should not be counted. Example 2 is also listed. The code editor on the right shows a C++ solution using a map and a two-pointer technique. The test case section shows the input `nums = [3,1,4,1,5]` and `k = 2` resulting in an output of `2`.

6. Find the K^{th} 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;

```

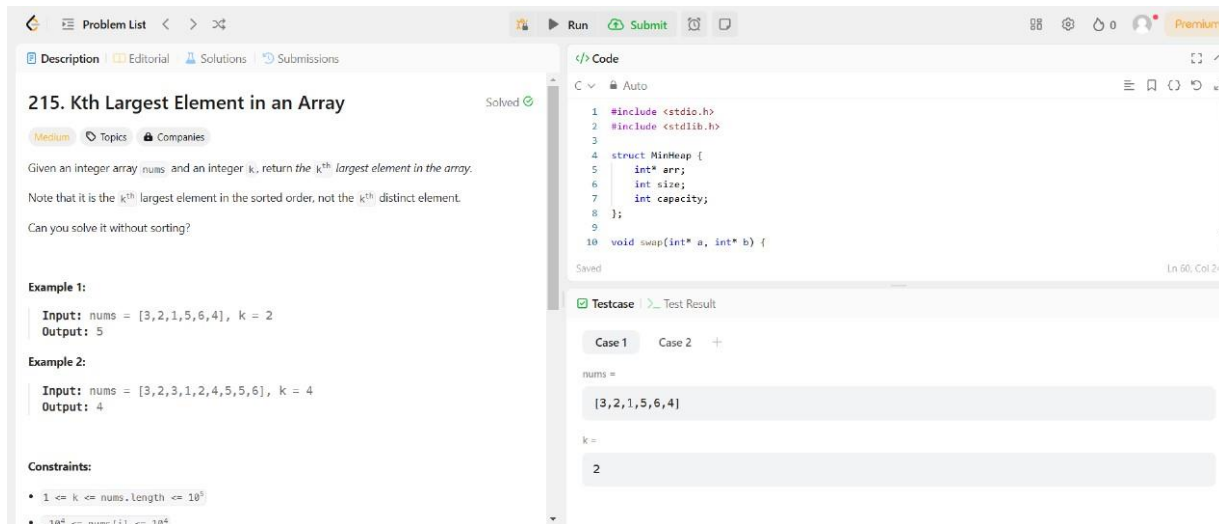
    }
    if (smallest != idx) {
        swap(&heap->arr[smallest], &heap->arr[idx]);
        heapify(heap, smallest);
    }
}

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:

The screenshot shows a C++ IDE with the following content:

202. Happy Number Solved

Write an algorithm to determine if a number n is happy.

A **happy number** is a number defined by the following process:

- Starting with any positive integer, replace the number by the sum of the squares of its digits.
- Repeat the process until the number equals 1 (where it will stay), or it **loops endlessly in a cycle** which does not include 1.
- Those numbers for which this process **ends in 1** are happy.

Return true if n is a happy number, and false if not.

Example 1:

Input: n = 19
Output: true
Explanation:
 $1^2 + 9^2 = 82$
 $8^2 + 2^2 = 68$
 $6^2 + 8^2 = 100$
 $1^2 + 0^2 + 0^2 = 1$

Code:

```
1 #include <stdbool.h>
2
3 int getSumOfSquares(int num);
4
5 bool isHappy(int n) {
6     int slow = n, fast = n;
7     do {
8         slow = getSumOfSquares(slow);
9         fast = getSumOfSquares(getSumOfSquares(fast));
10    } while (slow != fast);
11}
```

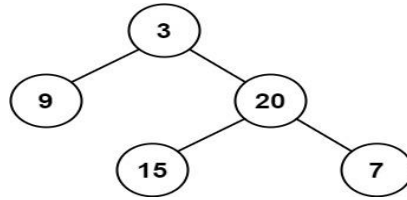
Testcase: Test Result

Case 1 Case 2

n =

19

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:

The screenshot shows the LeetCode interface for problem 110. On the left, the problem description states: "Given a binary tree, determine if it is height-balanced." Below this is "Example 1:" with a diagram of the tree (root 3, left child 9, right child 20, 20's left child 15, 20's right child 7). The input is given as "root = [3,9,20,null,null,15,7]" and the output is "true". On the right, the "Code" editor shows a C++ solution. The code defines a TreeNode struct and a height function that recursively checks the balance of the tree. The height function returns the height of the tree if it is balanced, or -1 if it is not. The isBalanced function then checks if the height is not -1. The code is as follows:

```
1 /**
2  * Definition for a binary tree node.
3  * struct TreeNode {
4  *     int val;
5  *     struct TreeNode *left;
6  *     struct TreeNode *right;
7  * };
8  */
9 #include <stdbool.h>
10 #include <stdlib.h>
```

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:

$O(n \log n)$

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 $\log n$, 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 = $n + n + n + \dots (\log n \text{ times}) = n \cdot \log n$

Hence, the time complexity in the best case is:

$O(n \log n)$

2. Worst Case:

$O(n^2)$

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 = $n + (n-1) + (n-2) + \dots + 1$

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:

$O(n \log n)$

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 $\frac{1}{2}n$ and $\frac{1}{2}n$

Mathematical Analysis:

Number of Levels in Recursion Tree: Similar to the best case, the number of levels in the recursion tree is approximately $\log n$, 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 $0, 1, 2, \dots, n-1$. 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:

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