

# DAA HOLIDAY ASSIGNMENT

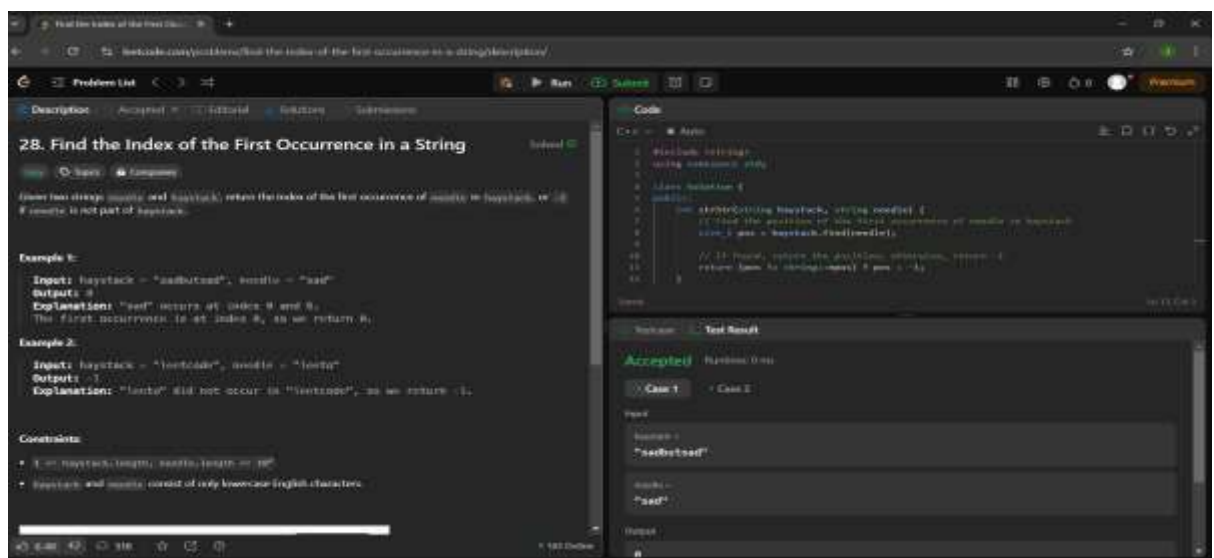
P.JANAKI

2211CS020388

AIML-THETA

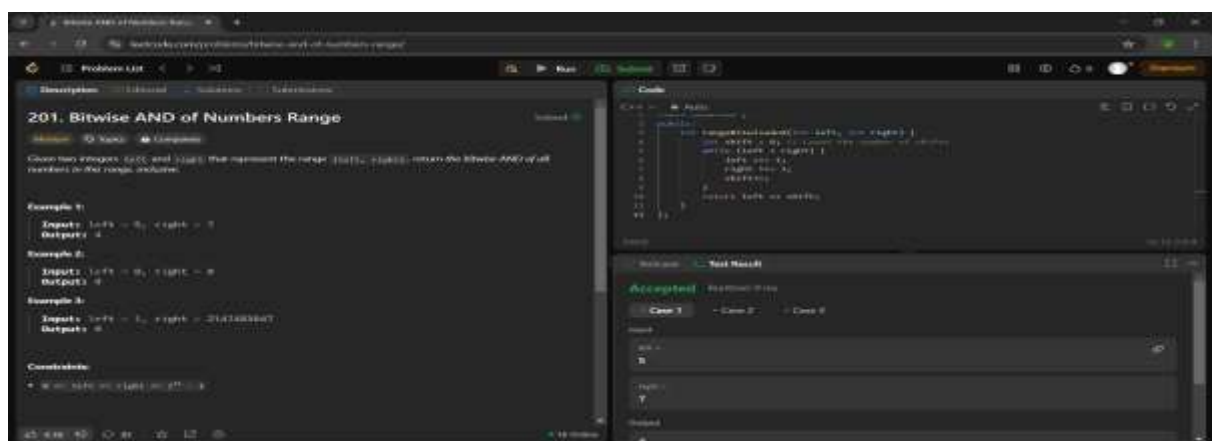
1) find-the-index-of-the-first-occurrence-in-a-string

<https://leetcode.com/problems/find-the-index-of-the-first-occurrence-in-a-string/description/>



2) Bitwise AND of numbers range

<https://leetcode.com/problems/bitwise-and-of-numbers-range/description/>



### 3)SQRT(X)

<https://leetcode.com/problems/sqrtx>

The screenshot shows the LeetCode interface for problem 69, 'Sqrt(x)'. The left panel contains the problem description, which asks for the integer square root of a non-negative integer x. It includes two examples: Example 1 with input 4 and output 2, and Example 2 with input 8 and output 2. The right panel shows a C++ solution using binary search. The code initializes left and right pointers and iteratively narrows down the search range until the square of the midpoint is less than or equal to x. The 'Test Result' section at the bottom shows the solution is 'Accepted' with a runtime of 0 ms.

**69. Sqrt(x)**

Given a non-negative integer  $x$ , return the square root of  $x$  rounded down to the nearest integer. The returned integer should be non-negative as well.

You must not use any built-in exponent function or operator.

- For example, do not use `pow(x, 0.5)` in C++ or `x ** 0.5` in Python.

**Example 1:**

Input:  $x = 4$   
Output: 2  
Explanation: The square root of 4 is 2, so we return 2.

**Example 2:**

Input:  $x = 8$   
Output: 2  
Explanation: The square root of 8 is 2.82842..., and since we round it down to the nearest integer, 2 is returned.

**Constraints:**

- $0 \leq x \leq 2^{31} - 1$

**Code:**

```
C++
class Solution {
public:
    int mySqrt(int x) {
        if (x == 0) return 0;
        long left = 1, right = x;
        while (left <= right) {
            long mid = left + (right - left) / 2;
            long result = mid * mid;
            if (result == x) return mid;
            else if (result < x) left = mid + 1;
            else right = mid - 1;
        }
        return right;
    }
};
```

**Test Result:** Accepted Runtime: 0 ms

**Case 1:**

Input: 4  
Output: 2  
Expected: 2

### 4)Largest Number

<https://leetcode.com/problems/largest-number/description/>

The screenshot shows the LeetCode interface for problem 179, 'Largest Number'. The left panel contains the problem description, which asks to arrange a list of non-negative integers to form the largest possible number. It includes two examples: Example 1 with input [10, 2] and output '210', and Example 2 with input [3, 30, 34, 5, 4] and output '534330'. The right panel shows a C++ solution that sorts the numbers as strings in descending order. The 'Test Result' section at the bottom shows the solution is 'Accepted' with a runtime of 0 ms.

**179. Largest Number**

Given a list of non-negative integers  $nums$ , arrange them such that they form the largest number and return it.

Since the result may be very large, so you need to return a string instead of an integer.

**Example 1:**

Input:  $nums = [10, 2]$   
Output: "210"

**Example 2:**

Input:  $nums = [3, 30, 34, 5, 4]$   
Output: "534330"

**Constraints:**

- $1 \leq nums.length \leq 100$
- $0 \leq nums[i] \leq 10^6$

See this question in a real interview before? 1/5

**Code:**

```
C++
class Solution {
public:
    string largestNumber(vector<int> &nums) {
        string result;
        for (int i = 0; i < nums.size(); i++) {
            result += to_string(nums[i]);
        }
        if (result[0] == '0') return "0";
        return result;
    }
};
```

**Test Result:** Accepted Runtime: 0 ms

**Case 1:**

Input: [10, 2]  
Output: "210"  
Expected: "210"

## 5) Valid Parenthesis

<https://leetcode.com/problems/valid-parentheses/>

**20. Valid Parentheses**

Given a string `s` containing just the characters `'('`, `)'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

**Example 1:**

Input: `s = "()"`

Output: `true`

**Example 2:**

Input: `s = "({[]})"`

Output: `true`

**Example 3:**

Input: `s = "()"`

Output: `true`

**Code:**

```
C++
class Solution {
public:
    bool isValid(string s) {
        stack<char> stack;
        if (s.empty()) return true;
        if (s[0] == '(' || s[0] == '{' || s[0] == '[') stack.push(s[0]);
        else return false;
        for (int i = 1; i < s.length(); i++) {
            if (s[i] == '(' || s[i] == '{' || s[i] == '[') stack.push(s[i]);
            else {
                if (stack.empty()) return false;
                char top = stack.top();
                if ((s[i] == ')' && top == '(') || (s[i] == '}' && top == '{') || (s[i] == ']' && top == '[')) {
                    stack.pop();
                } else {
                    return false;
                }
            }
        }
        return stack.empty();
    }
};
```

**Test Result:** Accepted

**Case 1:**

Input: `s = "()"`

Output: `true`

Expected: `true`

## 6) Merge Two Sorted Lists

<https://leetcode.com/problems/merge-two-sorted-lists/>

**21. Merge Two Sorted Lists**

You are given the heads of two sorted linked lists, `list1` and `list2`.

Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

**Example 1:**

Input: `list1 = [1, 2, 4]`, `list2 = [1, 3, 4]`

Output: `[1, 1, 2, 3, 4, 4]`

**Code:**

```
C++
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        if (!list1) return list2;
        if (!list2) return list1;
        if (list1->val < list2->val) {
            list1->next = mergeTwoLists(list1->next, list2);
            return list1;
        } else {
            list2->next = mergeTwoLists(list1, list2->next);
            return list2;
        }
    }
};
```

**Test Result:** Accepted

**Case 1:**

list1: `[1, 2, 4]`

list2: `[1, 3, 4]`

Output: `[1, 1, 2, 3, 4, 4]`

## 7) Remove Duplicates from sorted list

<https://leetcode.com/problems/remove-duplicates-from-sorted-list/>

**83. Remove Duplicates from Sorted List**

Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

**Example 1:**

Input: head = [1,1,2]  
Output: [1,2]

**Example 2:**

Input: head = [1,1,2,3,3]  
Output: [1,2,3]

```
C++
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if (!head) return head;
        ListNode* curr = head;
        while (curr && curr->next) {
            if (curr->val == curr->next->val) {
                curr->next = curr->next->next;
            } else {
                curr = curr->next;
            }
        }
        return head;
    }
};
```

Accepted

Test Result

Case 1

Input: [1,1,2]  
Output: [1,2]  
Expected: [1,2]

## 8) Find peak Element

<https://leetcode.com/problems/find-peak-element/>

**162. Find Peak Element**

A peak element is an element that is strictly greater than its neighbors.

Given a 0-indexed integer array `nums`, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

You may imagine that `nums[-1] = nums[n] = -∞`. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in  $O(\log n)$  time.

**Example 1:**

Input: `nums = [1,2,3,1]`  
Output: `2`  
Explanation: 3 is a peak element and your function should return the index number 2.

**Example 2:**

Input: `nums = [1,2,1,3,5,6,4]`  
Output: `5`  
Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

```
C++
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int left = 0, right = nums.size() - 1;
        while (left < right) {
            int mid = (left + right) / 2;
            if (nums[mid] > nums[mid + 1]) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }
};
```

Accepted

Test Result

Case 1

Input: [1,2,3,1]  
Output: 2  
Expected: 2

## 9) Binary Tree Inorder Traversal

<https://leetcode.com/problems/binary-tree-inorder-traversal/>

**94. Binary Tree Inorder Traversal**

Given the `root` of a binary tree, return the inorder traversal of its nodes' values.

**Example 1:**

Input: `root = [1,null,2,3]`

Output: `[1,3,2]`

**Explanation:**

```
graph TD
    1((1)) --> 3((3))
    1 --> 2((2))
```

**Example 2:**

Input: `root = [1,2,3,null,4]`

Output: `[1,3,2,4]`

**Explanation:**

```
graph TD
    1((1)) --> 3((3))
    1 --> 2((2))
    2 --> 4((4))
```

**Code:**

```
C++
1 // Definition for a binary tree node.
2 struct TreeNode {
3     int val;
4     TreeNode *left;
5     TreeNode *right;
6     TreeNode() : val(0), left(nullptr), right(nullptr) {}
7     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
8     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
9 };
10
11 class Solution {
12 public:
13     vector<int> inorderTraversal(TreeNode* root) {
14         vector<int> result;
15         if (root == nullptr) return result;
16         inorderTraversal(root->left);
17         result.push_back(root->val);
18         inorderTraversal(root->right);
19         return result;
20     }
21 };
22
23 // Test Result
24 Accepted Runtime: 0 ms
```

## 10) N-Queens

<https://leetcode.com/problems/n-queens/>

**51. N-Queens**

The *n*-queens puzzle is the problem of placing *n* queens on an *n* × *n* chessboard such that no two queens attack each other.

Given an integer *n*, return all distinct solutions to the *n*-queens puzzle. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the *n*-queens' placement, where `"Q"` and `"."` both indicate a queen and an empty space, respectively.

**Example 1:**

**Inputs:** `n = 4`

**Output:**

```
["..Q.", "Q...", "....", "...Q."], ["Q...", "...Q.", "....", "Q..."]
```

**Code:**

```
C++
1 // Definition of a chessboard
2 class Solution {
3 public:
4     vector<vector<string>> solveNQueens(int n) {
5         vector<string> board(n, string(n, '.'));
6         vector<int> queens(n, -1);
7         solveNQueensHelper(board, queens, 0);
8         return board;
9     }
10
11     void solveNQueensHelper(vector<string> &board, vector<int> &queens, int row) {
12         if (row == board.size()) {
13             vector<string> solution(board);
14             solutions.push_back(solution);
15             return;
16         }
17         for (int col = 0; col < board[0].size(); col++) {
18             if (!isSafe(board, queens, row, col)) continue;
19             queens[row] = col;
20             board[row][col] = 'Q';
21             solveNQueensHelper(board, queens, row + 1);
22             board[row][col] = '.';
23             queens[row] = -1;
24         }
25     }
26
27     bool isSafe(vector<string> &board, vector<int> &queens, int row, int col) {
28         for (int r = 0; r < row; r++) {
29             if (queens[r] == col || abs(queens[r] - col) == row - r) return false;
30         }
31         return true;
32     }
33 };
34
35 // Test Result
36 Accepted Runtime: 0 ms
```

DAA Case-StudyScenario:-

An e-Commerce platform is implementing a feature where products need to be sorted by various attributes (e.g. price, rating and name). The product list contains millions of items, and the sorting operation needs to be efficient and scalable.

1. What are the time and space complexities of the commonly used sorting algorithms (QuickSort, MergeSort)?
  2. How do the characteristics of the data (e.g. range of prices, product name length) impact the choice of sorting algorithm?
1. Time and Space Complexities of Common Sorting Algorithm

Quick Sort:-

Time Complexity (Best): -  $O(\log n)$

Time Complexity (Average): -  $O(n \log n)$

Time Complexity (Worst): -  $O(n^2)$  (unbalanced pivot)

Space Complexity: -  $O(\log n)$  auxiliary.

Merge Sort:-

Time Complexity (Best): -  $O(n \log n)$

Time Complexity (Average): -  $O(n \log n)$

Time Complexity (Worst): -  $O(n \log n)$

Space complexity:  $O(n)$  auxiliary.



## 2. Impact of Data Characteristics on Sorting Algorithm choice.

### 1. Range of Prices or Numerical Data: -

- For a Small range of values (e.g., product prices), Counting Sort or radix Sort may be more efficient than Comparison-based algorithms, achieving linear time complexity  $O(n+k)$ .

- For a large or arbitrary range, Comparison-based algorithms like Quick Sort or Merge Sort are preferred.

### 2. Product Name Lengths (String Sorting): -

- Lexicographic Sorting involves comparing strings character by character. Algorithms like QuickSort and MergeSort still perform well but may be slower due to increased comparison times.

- Radix Sort can be used for fixed-length strings or cases where the length is bounded and known, providing near-linear time complexity.

### 3. Distribution and Size of Data: -

- Uniformly Distributed Data: QuickSort performs well with randomized pivot selection strategies.

- Nearly Sorted Data: InsertionSort (or) Trim Sort may outperform others due to better handling of such input.

### 4. Stability Requirements: -

- MergeSort is stable, meaning equal elements retain their relative order. QuickSort is generally not stable unless modified.

### 5. Memory Constraints: -

- If memory usage is a concern, QuickSort (with-in place sorting) is often more suitable than MergeSort.