

ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

MR23-1CS0223 - Design and Analysis of Algorithms

Holiday Assignment

by 2311cs020458

Nagaram Tarun

Aiml -Omega

1. Check whether given number is Palindrome or not.

Ans. #include <stdio.h>

```
int main() {  
  
    int num, reversedNum = 0, remainder, originalNum;  
  
    // Input from the user  
  
    printf("Enter an integer: ");  
  
    scanf("%d", &num);  
  
  
    originalNum = num; // Store the original number  
  
  
    // Reverse the number  
  
    while (num != 0) {  
  
        remainder = num % 10;  
  
        reversedNum = reversedNum * 10 + remainder;  
  
        num /= 10;  
    }  
}
```

```

}

// Check if the original number and reversed number are the same

if (originalNum == reversedNum) {

    printf("%d is a palindrome number.\n", originalNum);

} else {

    printf("%d is not a palindrome number.\n", originalNum);

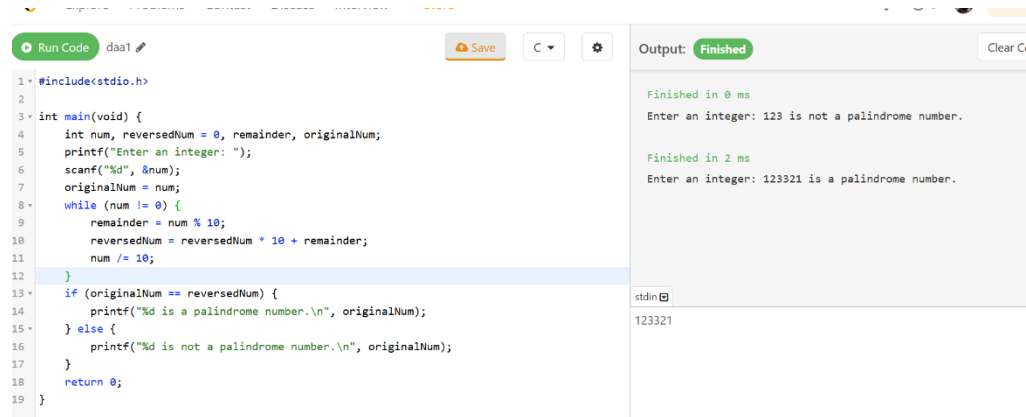
}

return 0;

}

```

Output:



The screenshot shows a C++ IDE with a code editor on the left and an output window on the right. The code in the editor is a C++ program that checks if a number is a palindrome. It uses a while loop to reverse the number and then compares the original number with the reversed number. The output window shows the results of two test cases: 123 is not a palindrome, and 123321 is a palindrome.

```

1 #include<stdio.h>
2
3 int main(void) {
4     int num, reversedNum = 0, remainder, originalNum;
5     printf("Enter an integer: ");
6     scanf("%d", &num);
7     originalNum = num;
8     while (num != 0) {
9         remainder = num % 10;
10        reversedNum = reversedNum * 10 + remainder;
11        num /= 10;
12    }
13    if (originalNum == reversedNum) {
14        printf("%d is a palindrome number.\n", originalNum);
15    } else {
16        printf("%d is not a palindrome number.\n", originalNum);
17    }
18    return 0;
19 }

```

Output: Finished

```

Finished in 0 ms
Enter an integer: 123 is not a palindrome number.

Finished in 2 ms
Enter an integer: 123321 is a palindrome number.

```

2. Convert the Roman to integer.

Ans. #include <stdio.h>

#include <string.h>

```
// Function to return the integer value of a Roman numeral character
```

```
int romanToInt(char c) {  
  
    switch (c) {  
  
        case 'I': return 1;  
  
        case 'V': return 5;  
  
        case 'X': return 10;  
  
        case 'L': return 50;  
  
        case 'C': return 100;  
  
        case 'D': return 500;  
  
        case 'M': return 1000;  
  
        default: return 0; // Invalid character  
  
    }  
  
}
```

```
// Function to convert a Roman numeral string to an integer
```

```
int convertRomanToInt(char *roman) {  
  
    int i, total = 0, current, next;  
  
    int length = strlen(roman);  
  
    for (i = 0; i < length; i++) {  
  
        current = romanToInt(roman[i]);  
  
        if (i + 1 < length) {
```

```

    next = romanToInt(roman[i + 1]);

    // If the current value is less than the next value, subtract it
    if (current < next) {

        total -= current;

    } else {

        total += current;

    }

} else {

    total += current; // Add the last value

}

}

return total;

}

int main() {

    char roman[20];

    // Input Roman numeral from the user

    printf("Enter a Roman numeral: ");

    scanf("%s", roman);

```

```
// Convert Roman numeral to integer and print the result
```

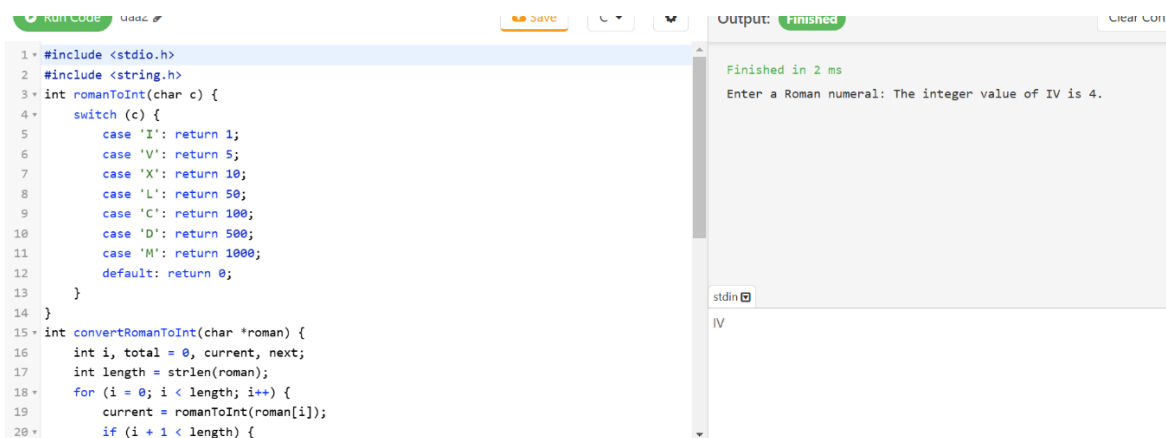
```
int result = convertRomanToInt(roman);
```

```
printf("The integer value of %s is %d.\n", roman, result);
```

```
return 0;
```

```
}
```

Output:



The screenshot shows a C++ IDE with a code editor on the left and an output window on the right. The code in the editor defines a function `romanToInt` that maps Roman numerals to their integer values using a `switch` statement, and a function `convertRomanToInt` that iterates through a string of Roman numerals, applying the `romanToInt` function to each character and summing the results. The output window shows the message "Finished in 2 ms" and "Enter a Roman numeral: The integer value of IV is 4.", indicating that the program was executed with the input "IV".

```
1 #include <stdio.h>
2 #include <string.h>
3 int romanToInt(char c) {
4     switch (c) {
5         case 'I': return 1;
6         case 'V': return 5;
7         case 'X': return 10;
8         case 'L': return 50;
9         case 'C': return 100;
10        case 'D': return 500;
11        case 'M': return 1000;
12        default: return 0;
13    }
14 }
15 int convertRomanToInt(char *roman) {
16     int i, total = 0, current, next;
17     int length = strlen(roman);
18     for (i = 0; i < length; i++) {
19         current = romanToInt(roman[i]);
20         if (i + 1 < length) {
```

Output: Finished in 2 ms
Enter a Roman numeral: The integer value of IV is 4.

stdin
IV

3. Validating opening and closing parenthesis in a String

Ans. #include <stdio.h>

#include <stdbool.h>

#include <string.h>

```
// Function to validate if the parentheses are balanced
```

```
bool isValidParentheses(char *str) {
```

```
    int stack[100]; // Stack to keep track of open parentheses
```

```
    int top = -1; // Top of the stack
```

```

for (int i = 0; str[i] != '\0'; i++) {

    char ch = str[i];

    // Push opening parentheses onto the stack

    if (ch == '(' || ch == '{' || ch == '[') {

        stack[++top] = ch;

    }

    // Check closing parentheses

    else if (ch == ')' || ch == '}' || ch == ']') {

        if (top == -1) {

            // Stack is empty, so no matching opening parenthesis

            return false;

        }

        // Check if the current closing parenthesis matches the top of the stack

        char topChar = stack[top--];

        if ((ch == ')' && topChar != '(') ||

            (ch == '}' && topChar != '{') ||

            (ch == ']' && topChar != '[')) {

            return false;

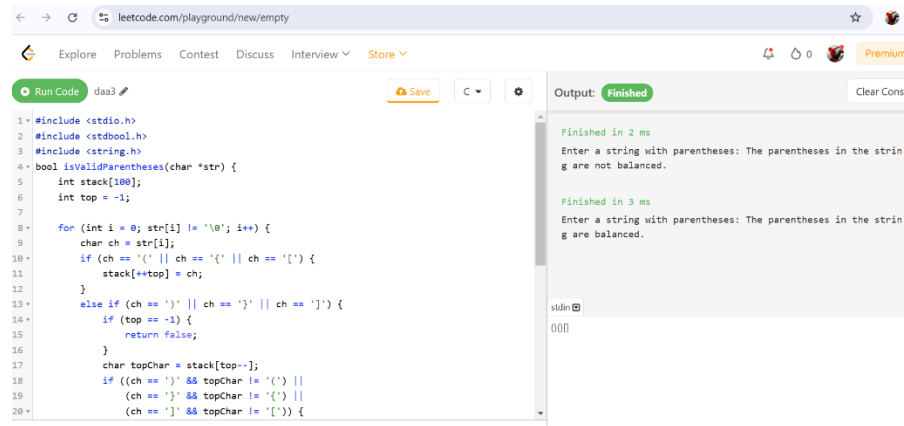
        }

```

```
    }  
}  
  
    // If stack is empty, all parentheses are matched  
    return top == -1;  
}  
  
int main() {  
    char str[100];  
  
    // Input string from the user  
    printf("Enter a string with parentheses: ");  
    scanf("%s", str);  
  
    // Validate the parentheses  
    if (isValidParentheses(str)) {  
        printf("The parentheses in the string are balanced.\n");  
    } else {  
        printf("The parentheses in the string are not balanced.\n");  
    }  
  
    return 0;  
}
```

```
}
```

Output:



```
1 * #include <stdio.h>
2 * #include <stdbool.h>
3 * #include <string.h>
4 * bool isValidParentheses(char *str) {
5 *     int stack[100];
6 *     int top = -1;
7 *
8 *     for (int i = 0; str[i] != '\0'; i++) {
9 *         char ch = str[i];
10 *         if (ch == '(' || ch == '{' || ch == '[') {
11 *             stack[++top] = ch;
12 *         }
13 *         else if (ch == ')' || ch == '}' || ch == ']') {
14 *             if (top == -1) {
15 *                 return false;
16 *             }
17 *             char topChar = stack[top--];
18 *             if ((ch == ')' && topChar != '(') ||
19 *                 (ch == '}' && topChar != '{') ||
20 *                 (ch == ']' && topChar != '[')) {
```

Output: Finished

```
Finished in 2 ms
Enter a string with parentheses: The parentheses in the string are not balanced.

Finished in 3 ms
Enter a string with parentheses: The parentheses in the string are balanced.

stdin
000
```

4. Finding Odd and even numbers in an Array

Ans. #include <stdio.h>

```
int main() {
```

```
    int n, i;
```

```
    // Input size of the array
```

```
    printf("Enter the size of the array: ");
```

```
    scanf("%d", &n);
```

```
    int arr[n], odd[n], even[n];
```

```
    int oddCount = 0, evenCount = 0;
```

```
    // Input array elements
```



```
printf("Enter %d elements of the array:\n", n);

for (i = 0; i < n; i++) {

    scanf("%d", &arr[i]);

}


// Separate odd and even numbers

for (i = 0; i < n; i++) {

    if (arr[i] % 2 == 0) {

        even[evenCount++] = arr[i]; // Add to even array

    } else {

        odd[oddCount++] = arr[i]; // Add to odd array

    }

}


// Print even numbers

printf("Even numbers in the array: ");

for (i = 0; i < evenCount; i++) {

    printf("%d ", even[i]);

}

printf("\n");


// Print odd numbers
```

```

printf("Odd numbers in the array: ");

for (i = 0; i < oddCount; i++) {

    printf("%d ", odd[i]);

}

printf("\n");

return 0;

}

```

Output:

```

1 #include <stdio.h>
2
3 int main() {
4     int n, i;
5     printf("Enter the size of the array: ");
6     scanf("%d", &n);
7
8     int arr[n], odd[n], even[n];
9     int oddCount = 0, evenCount = 0;
10    printf("Enter %d elements of the array:\n", n);
11    for (i = 0; i < n; i++) {
12        scanf("%d", &arr[i]);
13    }
14    for (i = 0; i < n; i++) {
15        if (arr[i] % 2 == 0) {
16            even[evenCount++] = arr[i]; // Add to even array
17        } else {
18            odd[oddCount++] = arr[i]; // Add to odd array
19        }
20    }

```

Output: Finished

```

Finished in 3 ms
Enter the size of the array: Enter 5 elements of the array:
Even numbers in the array: 6 8
Odd numbers in the array: 5 7 9

```

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 <stdio.h>

```

void findSymmetricPairs(int pairs[][2], int n) {

    printf("Symmetric pairs are:\n");

    for (int i = 0; i < n; i++) {

```

```
    for (int j = i + 1; j < n; j++) {  
        if (pairs[i][0] == pairs[j][1] && pairs[i][1] == pairs[j][0]) {  
            printf("(%d, %d) and (%d, %d)\n", pairs[i][0], pairs[i][1], pairs[j][0], pairs[j][1]);  
        }  
    }  
}  
}
```

```
int main() {  
    int n;  
  
    printf("Enter the number of pairs: ");  
  
    scanf("%d", &n);  
  
    int pairs[n][2];  
  
    printf("Enter the pairs:\n");  
  
    for (int i = 0; i < n; i++) {  
        scanf("%d %d", &pairs[i][0], &pairs[i][1]);  
    }  
  
    findSymmetricPairs(pairs, n);  
  
    return 0;
```

```
}
```

Output:



The screenshot shows a LeetCode playground interface. On the left, there is a C++ code editor with the following code:

```
1 #include <stdio.h>
2
3 void findSymmetricPairs(int pairs[][2], int n) {
4     printf("Symmetric pairs are:\n");
5     for (int i = 0; i < n; i++) {
6         for (int j = i + 1; j < n; j++) {
7             if (pairs[i][0] == pairs[j][1] && pairs[i][1] == pairs[j][0]) {
8                 printf("(%d, %d) and (%d, %d)\n", pairs[i][0], pairs[i][1], pairs[j][0],
9                     pairs[j][1]);
10            }
11        }
12    }
13 }
14
15 int main() {
16     int n;
17     printf("Enter the number of pairs: ");
18     scanf("%d", &n);
19     int pairs[n][2];
```

On the right, the 'Output' panel shows the result of running the code:

```
Finished in 2 ms
Enter the number of pairs: Enter the pairs:
Symmetric pairs are:
(1, 2) and (2, 1)
(3, 5) and (5, 3)
```

Below the output, the 'stdin' input is shown:

```
5
12
35
21
67
53
```

6. Find the K^{th} smallest element in an Array using function.

Ans. #include <stdio.h>

```
int partition(int arr[], int low, int high) {
```

```
    int pivot = arr[high];
```

```
    int i = low - 1;
```

```
    for (int j = low; j < high; j++) {
```

```
        if (arr[j] < pivot) {
```

```
            i++;
```

```
            int temp = arr[i];
```

```
            arr[i] = arr[j];
```

```
            arr[j] = temp;
```

```
        }
```

```
    }
```

```
int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1;

}
```

```
int kthSmallest(int arr[], int low, int high, int k) {

    if (low <= high) {

        int pi = partition(arr, low, high);

        if (pi == k - 1) {

            return arr[pi];

        } else if (pi > k - 1) {

            return kthSmallest(arr, low, pi - 1, k);

        } else {

            return kthSmallest(arr, pi + 1, high, k);

        }

    }

    return -1;

}
```

```
int main() {

    int n, k;
```

```
printf("Enter the size of the array: ");

scanf("%d", &n);

int arr[n];

printf("Enter %d elements of the array:\n", n);

for (int i = 0; i < n; i++) {

    scanf("%d", &arr[i]);

}

printf("Enter the value of k: ");

scanf("%d", &k);

int result = kthSmallest(arr, 0, n - 1, k);

if (result != -1) {

    printf("The %dth smallest element is %d.\n", k, result);

} else {

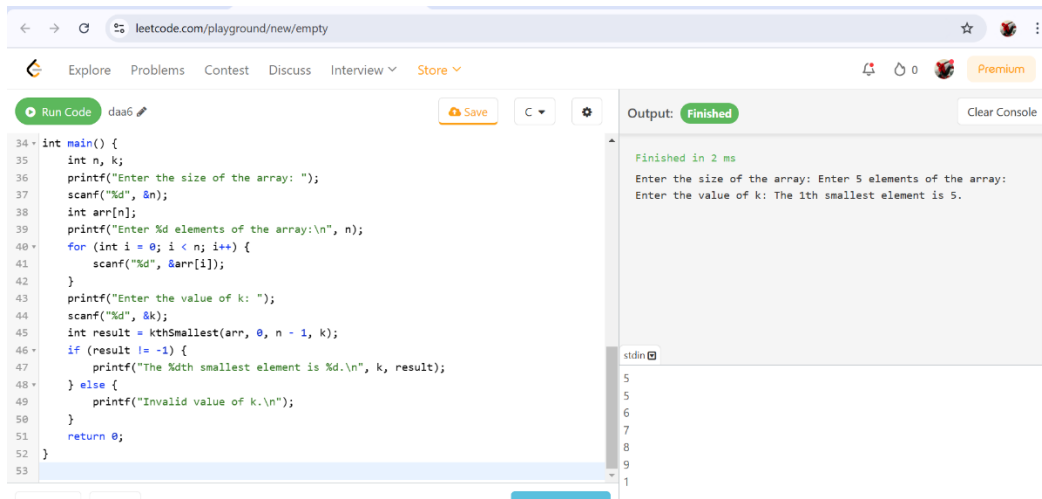
    printf("Invalid value of k.\n");

}

return 0;

}
```

Output:



```
34 int main() {
35     int n, k;
36     printf("Enter the size of the array: ");
37     scanf("%d", &n);
38     int arr[n];
39     printf("Enter %d elements of the array:\n", n);
40     for (int i = 0; i < n; i++) {
41         scanf("%d", &arr[i]);
42     }
43     printf("Enter the value of k: ");
44     scanf("%d", &k);
45     int result = kthSmallest(arr, 0, n - 1, k);
46     if (result != -1) {
47         printf("The %dth smallest element is %d.\n", k, result);
48     } else {
49         printf("Invalid value of k.\n");
50     }
51     return 0;
52 }
53
```

Output: Finished

Finished in 2 ms

Enter the size of the array: Enter 5 elements of the array:

Enter the value of k: The 1th smallest element is 5.

stdin

5

5

6

7

8

9

1

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>

typedef struct {

float real;

float imag;

} Complex;

Complex addComplex(Complex c1, Complex c2) {

Complex result;

result.real = c1.real + c2.real;

result.imag = c1.imag + c2.imag;

return result;

}

```
Complex multiplyComplex(Complex c1, Complex c2) {
```

```
    Complex result;
```

```
    result.real = c1.real * c2.real - c1.imag * c2.imag;
```

```
    result.imag = c1.real * c2.imag + c1.imag * c2.real;
```

```
    return result;
```

```
}
```

```
int main() {
```

```
    Complex c1, c2, sum, product;
```

```
    printf("Enter the real and imaginary parts of the first complex number: ");
```

```
    scanf("%f %f", &c1.real, &c1.imag);
```

```
    printf("Enter the real and imaginary parts of the second complex number: ");
```

```
    scanf("%f %f", &c2.real, &c2.imag);
```

```
    sum = addComplex(c1, c2);
```

```
    product = multiplyComplex(c1, c2);
```

```
    printf("Sum of the complex numbers: %.2f + %.2fi\n", sum.real, sum.imag);
```

```
    printf("Product of the complex numbers: %.2f + %.2fi\n", product.real, product.imag);
```



```

return 0;

}

```

Output:

The screenshot shows a LeetCode playground interface. On the left, there is a code editor with C code for adding and multiplying complex numbers. The code defines a 'Complex' struct with 'real' and 'imag' float members. It includes functions 'addComplex' and 'multiplyComplex'. The 'addComplex' function returns the sum of two complex numbers, and 'multiplyComplex' returns their product. The code is saved and run. On the right, the 'Output' panel shows the execution results: 'Finished in 3 ms', followed by prompts for real and imaginary parts of two complex numbers, and the calculated sum and product.

```

1 #include <stdio.h>
2
3 typedef struct {
4     float real;
5     float imag;
6 } Complex;
7
8 Complex addComplex(Complex c1, Complex c2) {
9     Complex result;
10    result.real = c1.real + c2.real;
11    result.imag = c1.imag + c2.imag;
12    return result;
13 }
14
15 Complex multiplyComplex(Complex c1, Complex c2) {
16     Complex result;
17    result.real = c1.real * c2.real - c1.imag * c2.imag;
18    result.imag = c1.real * c2.imag + c1.imag * c2.real;
19    return result;
20 }

```

Output: Finished

```

Finished in 3 ms
Enter the real and imaginary parts of the first complex number:
r: 4.000000 7.000000
Enter the real and imaginary parts of the second complex number:
er: 5.000000 8.000000
Sum of the complex numbers: 9.00 + 15.00i
Product of the complex numbers: -36.00 + 67.00i

```

8. Find the missing and duplicate number in an Array

Ans. #include <stdio.h>

```
void findMissingAndDuplicate(int arr[], int n) {
```

```
    int i, missing = -1, duplicate = -1;
```

```
    int count[n + 1];
```

```
    for (i = 0; i <= n; i++) {
```

```
        count[i] = 0;
```

```
    }
```

```
    for (i = 0; i < n; i++) {
```

```
        count[arr[i]]++;
```

```
}
```

```
for (i = 1; i <= n; i++) {
```

```
    if (count[i] == 0) {
```

```
        missing = i;
```

```
    }
```

```
    if (count[i] > 1) {
```

```
        duplicate = i;
```

```
    }
```

```
}
```

```
printf("Duplicate number: %d\n", duplicate);
```

```
printf("Missing number: %d\n", missing);
```

```
}
```

```
int main() {
```

```
    int n, i;
```

```
    printf("Enter the size of the array (n): ");
```

```
    scanf("%d", &n);
```

```
    int arr[n];
```

```
    printf("Enter %d elements (1 to n):\n", n);
```

```

for (i = 0; i < n; i++) {

    scanf("%d", &arr[i]);

}

findMissingAndDuplicate(arr, n);

return 0;

}

```

Output:

The screenshot shows a C programming IDE with the following code in the editor:

```

24     printf("Duplicate number: %d\n", duplicate);
25     printf("Missing number: %d\n", missing);
26 }
27
28 int main() {
29     int n, i;
30     printf("Enter the size of the array (n): ");
31     scanf("%d", &n);
32
33     int arr[n];
34     printf("Enter %d elements (1 to n):\n", n);
35     for (i = 0; i < n; i++) {
36         scanf("%d", &arr[i]);
37     }
38
39     findMissingAndDuplicate(arr, n);
40
41     return 0;
42 }
43

```

The output window shows the following text:

```

Finished in 3 ms
Enter the size of the array (n): Enter 4 elements (1 to n):
Duplicate number: 4
Missing number: 3

```

The input window shows the following text:

```

4
1
2
4
4

```

9. Write C program 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.

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 <stdio.h>

#include <stdbool.h>

```
int sumOfSquares(int n) {
```

```
    int sum = 0;
```

```
    while (n > 0) {
```

```
        int digit = n % 10;
```

```
        sum += digit * digit;
```

```
        n /= 10;
```

```
    }
```

```
    return sum;
```

```
}
```

```
bool isHappy(int n) {
```

```
    int slow = n, fast = n;
```

```
    do {
```

```
        slow = sumOfSquares(slow);
```

```

        fast = sumOfSquares(sumOfSquares(fast));

    } while (slow != fast);

    return slow == 1;

}

int main() {

    int n;

    printf("Enter a number: ");

    scanf("%d", &n);

    if (isHappy(n)) {

        printf("%d is a happy number.\n", n);

    } else {

        printf("%d is not a happy number.\n", n);

    }

    return 0;

}

```

Output:

The screenshot shows a web browser window with the URL `leetcode.com/playground/new/empty`. The page has a navigation bar with links for Explore, Problems, Contest, Discuss, Interview, and Store. Below the navigation bar, there's a section for running code. On the left, there's a text area with C code for a function `sumOfSquares` and a function `isHappy`. The `isHappy` function uses a while loop to check if a number is happy by comparing the original number with the sum of its squares. On the right, there's an 'Output' section with a 'Finished' status. The output text shows the program's execution for the input 19: 'Finished in 2 ms', 'Enter a number: 19', and '19 is a happy number.'.

```

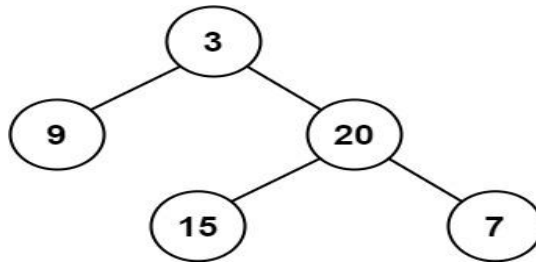
1- #include <stdio.h>
2- #include <stdbool.h>
3-
4- int sumOfSquares(int n) {
5-     int sum = 0;
6-     while (n > 0) {
7-         int digit = n % 10;
8-         sum += digit * digit;
9-         n /= 10;
10-    }
11-    return sum;
12-}
13-
14- bool isHappy(int n) {
15-     int slow = n, fast = n;
16-     do {
17-         slow = sumOfSquares(slow);
18-         fast = sumOfSquares(sumOfSquares(fast));
19-     } while (slow != fast);
20-     return slow == 1;

```

Output: Finished

Finished in 2 ms
Enter a number: 19
19 is a happy number.

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



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

Output: true

Ans. #include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

```
typedef struct BinaryTreeNode {  
    int val;  
    struct BinaryTreeNode* left;  
    struct BinaryTreeNode* right;  
} BinaryTreeNode;
```

```
BinaryTreeNode* createNode(int val) {  
    BinaryTreeNode* node = (BinaryTreeNode*)malloc(sizeof(BinaryTreeNode));  
    node->val = val;  
    node->left = NULL;  
    node->right = NULL;  
    return node;  
}
```

```
BinaryTreeNode* insertNode() {  
    int val;  
    printf("Enter node value (-1 for NULL): ");  
    scanf("%d", &val);  
    if (val == -1) {  
        return NULL;  
    }  
    BinaryTreeNode* node = createNode(val);  
    printf("Enter left child of %d:\n", val);  
    node->left = insertNode();
```

```

    printf("Enter right child of %d:\n", val);
    node->right = insertNode();
    return node;
}

int height(BinaryTreeNode* root) {
    if (root == NULL) {
        return 0;
    }
    int leftHeight = height(root->left);
    int rightHeight = height(root->right);
    if (leftHeight == -1 || rightHeight == -1 || abs(leftHeight - rightHeight) > 1) {
        return -1;
    }
    return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
}

bool isBalanced(BinaryTreeNode* root) {
    return height(root) != -1;
}

int main() {
    printf("Create the binary tree:\n");
    BinaryTreeNode* root = insertNode();

    if (isBalanced(root)) {
        printf("The tree is height-balanced.\n");
    } else {
        printf("The tree is not height-balanced.\n");
    }

    return 0;
}

```

Output:

```
13     return 1 + (leftHeight > rightHeight ? leftHeight : rightHeight);
14 }
15
16 bool isBalanced(BinaryTreeNode* root) {
17     return height(root) != -1;
18 }
19
20 int main() {
21     printf("Create the binary tree:\n");
22     BinaryTreeNode* root = insertNode();
23
24     if (isBalanced(root)) {
25         printf("The tree is height-balanced.\n");
26     } else {
27         printf("The tree is not height-balanced.\n");
28     }
29
30     return 0;
31 }
```

Output: Finished

Create the binary tree:
Enter node value (-1 for NULL): Enter left child of 3:
Enter node value (-1 for NULL): Enter left child of 9:
Enter node value (-1 for NULL): Enter right child of 9:
Enter node value (-1 for NULL): Enter right child of 3:
Enter node value (-1 for NULL): Enter left child of 20:
Enter node value (-1 for NULL): Enter left child of 15:
Enter node value (-1 for NULL): Enter right child of 15:
Enter node value (-1 for NULL): Enter right child of 20:
Enter node value (-1 for NULL): Enter left child of 7:
Enter node value (-1 for NULL): Enter right child of 7:
Enter node value (-1 for NULL): The tree is height-balanced.

stdin

-1
-1
20
15
-1
-1
7
-1
-1

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:

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

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:

$$\text{Total Comparisons} = n + (n-1) + (n-2) + \dots + 1$$

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

$$\text{Total Comparisons} = 2n(n+1)$$

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 $\frac{n}{4}$ and $\frac{3n}{4}$

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:

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