Assignment 2: Concept of Programming

Introduction

This assignment is designed to test your understanding of conditional and looping statements, recursion, and functions in JavaScript. You will have to write code snippets to solve various problems using these concepts. You will also have to explain your logic and output for each code snippet.

Instructions

- Write your code in a text editor like Notepad or Visual Studio Code.
- Save your file with a .js extension (for example, assignment1.js).
- You can use a web browser such as Chrome or Firefox to run your code. Open the browser's developer tools (Ctrl+Shift+I) and go to the console tab. Then drag and drop your file into the browser window or use the File > Open File option to select your file. You should see the output of your code in the console.
- Alternatively, you can use an online tool such as https://replit.com/languages/javascript to write and run your code.
- You have to complete this assignment individually.
- You have to submit your assignment as a single PDF file that contains your code snippets, explanations, and outputs.
- You have to use proper indentation, comments, and naming conventions for your code snippets.
- You have to cite any sources that you use for reference or inspiration.
- You have to follow the marking scheme given below for each question.

Questions

Question 1: Display prime numbers between 1 and 100 or 1 and n (10 marks)

Write a JavaScript function named `displayPrimes` that takes a positive integer `n` as a parameter and displays all the prime numbers between 1 and `n`. If `n` is not given, display all the prime numbers between 1 and 100 by default. A prime number is a natural number that has exactly two positive divisors: 1 and itself.

For example:

```
displayPrimes(); // displays 2, 3, 5, ..., 97
displayPrimes(50); // displays 2, 3, 5, ..., 47
```

Explain how your function works and what output it produces for different values of `n`.

Question 2: Find the factorial of a number (10 marks)

Write a JavaScript function named `factorial` that takes a non-negative integer `n` as a parameter and returns its factorial. The factorial of `n`, denoted by `n!`, is the product of all positive integers less than or equal to `n`. For example:

```
factorial(0); // returns 1
factorial(5); // returns 120
```

Explain how your function works and what output it produces for different values of `n`.

Question 3: Check if a number is palindrome or not (10 marks)

Write a JavaScript function named `isPalindrome` that takes an integer `n` as a parameter and returns true if it is palindrome or false otherwise. A palindrome is a number that remains the same when its digits are reversed. For example:

```
isPalindrome(121); // returns true
isPalindrome(123); // returns false
```

Explain how your function works and what output it produces for different values of `n`.

Question 4: Add two integer variables in five different ways using functions and control statements (10 marks)

Write five different JavaScript functions that take two integer variables `a` and `b` as parameters and return their sum. Each function should use a different control statement such as if-else, switch-case, for loop, while loop, or do-while loop. For example:

```
function addWithIf(a,b) {
  if (a > b) {
    return b + a;
  } else {
    return a + b;
  }
}
```

Explain how each function works and what output it produces for different values of `a` and `b`.

Question 5: Find square root of a number without sqrt method (10 marks)

Write a JavaScript function named `sqrt` that takes an non-negative number `x` as a parameter and returns its square root without using the built-in Math.sqrt method. You can use any algorithm or technique you want such as binary search or Newton's method. For example:

```
sqrt(25); // returns 5
sqrt(2); // returns approximately 1.4142135623730951
```

Explain how your function works and what output it produces for different values of `x`.

Question 6: Check Armstrong number (10 marks)

Write a JavaScript function named `isArmstrong` that takes an integer n as parameter with three digits only (between [100 -999])and checks whether it is an Armstrong number or not . An Armstrong number is an n-digit number that is equal to the sum of its digits raised to the power n . For example:

```
isArmstrong(153); //returns true because
//153 = (1^3)+(5^3)+(3^3)

isArmstrong(370);//returns true because
//370 = (3^3)+(7^3)+(0^3)
```

Question 7: Calculate grades of students using their marks (10 marks)

Write a JavaScript function named calculateGrade that takes an integer representing a student's marks (between 0 and 100) as a parameter and returns the corresponding letter grade based on the following grading scale:

- 90-100: A
- 80-89: B
- 70-79: C
- 60-69: D
- 0-59: F

For example:

```
calculateGrade(85); // returns 'B'
 calculateGrade(95); // returns 'A'
```

Explain how your function works and what output it produces for different values of marks.

Question 8: Print patterns using loops (10 marks)

Write a JavaScript function named printPattern that takes a positive integer n as a parameter and prints the following pattern using loops:

```
1
12
123
...
123...n
```

For example:

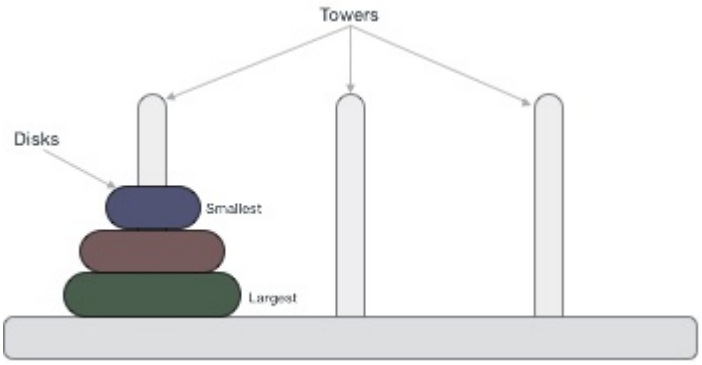
```
```

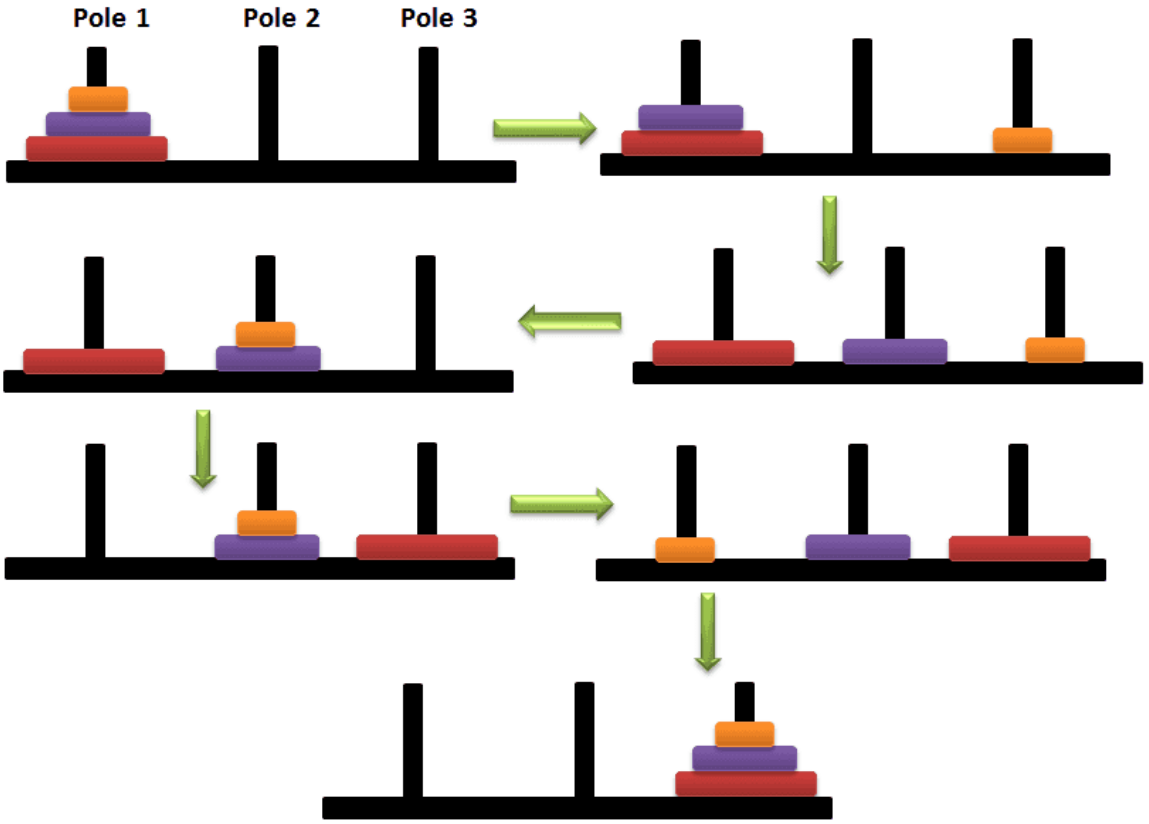
```
printPattern(3);
Output:
        1
        12
        123
```

Explain how your function works and what output it produces for different values of n.

Question 9: Solve the Tower of Hanoi problem using recursion (20 marks)

Write a JavaScript function named towerOfHanoi that takes a positive integer n as a parameter, representing the number of disks in the Tower of Hanoi problem, and prints the steps to solve the problem using recursion. The function should also take three string parameters representing the names of the three pegs (A, B, and C by default).






```
For example:
towerOfHanoi(2, "A", "B", "C");
Output:
        Move disk 1 from A to B
        Move disk 2 from A to C
        Move disk 1 from B to C
```

First try on your own, for hints refer at the end of this document.

Appendix

Recursion

What is recursion?

- A function is recursive if the body of that function calls itself, either directly or indirectly[1].
- Recursive functions often operate on increasingly smaller instances of a problem.
- For example, consider the problem of finding the sum of digits of a number:

```
def sum_digits(n):
    if n < 10: # base case
        return n
    else: # recursive case
        return n % 10 + sum_digits(n // 10) # add last digit and recurse on remaining digits
```

Why use recursion?

- Recursion can make some problems easier to express and solve.
- Recursion can avoid explicit loops and mutable variables.
- Recursion can mimic the structure of data or problem domains that are inherently recursive, such as trees, fractals, grammars, etc.

How to write recursive functions?

- Identify the base case(s) where the problem can be solved directly without recursion.
- Identify the recursive case(s) where the problem can be reduced to a smaller or simpler subproblem of the same kind.
- Write a function that checks for the base case(s) and returns the solution, or makes one or more recursive calls for the recursive case(s) and combines their results.

Types of recursion

- Single recursion: when a function makes only one recursive call in its body. For example:

```
def factorial(n):
    if n == 0: # base case
        return 1
    else: # recursive case
```

```
    return n * factorial(n - 1) # multiply by factorial of previous number
```

- Multiple recursion: when a function makes more than one recursive call in its body. For example:

```
def fibonacci(n):
  if n == 0 or n == 1: # base cases
    return n
  else: # recursive case
    return fibonacci(n - 1) + fibonacci(n - 2) # add previous two fibonacci numbers
```

Challenges of recursion

- Recursion can be hard to understand and debug at first.
- Recursion can cause stack overflow errors if there are too many nested function calls.
- Recursion can be inefficient if it repeats unnecessary work or creates too many intermediate values.

Hints:

Q5:

To find the square root of a non-negative number x without using the built-in Math.sqrt method, you can use Newton's method or the binary search algorithm. Here are hints for both techniques:

Newton's method:

a. Start with an initial guess for the square root, such as x/2.
b. Iterate the following formula to refine the guess:
    next_guess = (current_guess + x / current_guess) / 2
c. Continue iterating until the difference between the current guess and the next guess is smaller than a chosen tolerance level (e.g., 1e-7).
d. Return the final guess as the square root of x.

Binary search algorithm:

a. Initialize two variables, 'low' and 'high', representing the search interval. Set 'low' to 0 and 'high' to x.
b. While the difference between 'high' and 'low' is greater than a chosen tolerance level (e.g., 1e-7):
    i. Calculate the midpoint of the interval: mid = (low + high) / 2.
    ii. If mid * mid is equal to x (within the tolerance level), return 'mid' as the square root of x.
    iii. If mid * mid is less than x, set 'low' to 'mid'.
    iv. If mid * mid is greater than x, set 'high' to '

Q10:

The Tower of Hanoi problem is a classic puzzle that involves moving a set of disks from one peg to another, following certain rules. To solve the Tower of Hanoi problem, you can use the following recursive algorithm:

Consider three pegs: source (A), auxiliary (B), and target (C).

The goal is to move n disks from the source peg (A) to the target peg (C) using the auxiliary peg (B) as an intermediate storage.

Follow these steps:

a. Move the top n-1 disks from the source peg (A) to the auxiliary peg (B) using the target peg (C) as the intermediate storage. This is done by recursively applying the Tower of Hanoi algorithm.

b. Move the largest disk, disk n, from the source peg (A) to the target peg (C). This step doesn't require any recursion since it's a single move.

c. Move the n-1 disks from the auxiliary peg (B) to the target peg (C) using the source peg (A) as the intermediate storage. This is done by recursively applying the Tower of Hanoi algorithm again.