EXPERIMENT 10-20

-Prisha D(192311018)

1. Write a program to find the reverse of a given number using recursive.

```
def reverse_number(n, reversed_num=0):
    if n == 0:
        return reversed_num
    else:
        remainder = n % 10
        reversed_num = (reversed_num * 10) + remainder
        return reverse_number(n // 10, reversed_num)

# Example usage:
number = int(input("Enter a number: "))
reversed_number = reverse_number(number)
print(f"The reverse of {number} is {reversed_number}")
```

2. Write a program to find the perfect number.

```
def is_perfect_number(n):
    if n < 1:
        return False
    divisors_sum = sum([i for i in range(1, n) if n % i == 0])
    return divisors_sum == n

# Example usage:
number = int(input("Enter a number: "))
if is_perfect_number(number):
    print(f"{number} is a perfect number")
else:
    print(f"{number} is not a perfect number")
```

```
# Find perfect numbers in a range
def find_perfect_numbers_in_range(start, end):
    perfect_numbers = []
    for num in range(start, end + 1):
        if is_perfect_number(num):
            perfect_numbers.append(num)
    return perfect_numbers

# Example usage for finding perfect numbers in a range:
start = int(input("Enter the start of the range: "))
end = int(input("Enter the end of the range: "))
perfect_numbers = find_perfect_numbers_in_range(start, end)
print(f"Perfect numbers between {start} and {end}: {perfect_numbers}")
```

3. Write C program that demonstrates the usage of these notations by analyzing the time complexity of some example algorithms.

```
import time
import matplotlib.pyplot as plt

# Example algorithms
def constant_time_example(n):
    return 42  # O(1)

def linear_time_example(n):
    total = 0
    for i in range(n):
        total += i  # O(n)
    return total

def quadratic_time_example(n):
    total = 0
    for i in range(n):
```

```
        for j in range(n):
            total += i * j  # O(n^2)
    return total

# Time complexity analysis
def time_algorithm(algorithm, n_values):
    times = []
    for n in n_values:
        start_time = time.time()
        algorithm(n)
        end_time = time.time()
        times.append(end_time - start_time)
    return times

# Example input sizes
n_values = [10, 100, 1000, 10000]

# Measure time for each algorithm
times_constant = time_algorithm(constant_time_example, n_values)
times_linear = time_algorithm(linear_time_example, n_values)
times_quadratic = time_algorithm(quadratic_time_example, n_values)

# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(n_values, times_constant, label="O(1) - Constant Time", marker='o')
plt.plot(n_values, times_linear, label="O(n) - Linear Time", marker='o')
plt.plot(n_values, times_quadratic, label="O(n^2) - Quadratic Time", marker='o')

plt.xlabel("Input Size (n)")
plt.ylabel("Time (seconds)")
```

```
plt.title("Time Complexity Analysis")
plt.legend()
plt.yscale('log')
plt.grid(True)
plt.show()
```

4. Write C programs that demonstrate the mathematical analysis of non-recursive and recursive algorithms.

import time

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

def analyze_non_recursive():
    input_sizes = [10, 100, 1000, 10000, 100000]
    times = []

    for size in input_sizes:
        arr = list(range(size))
        target = size - 1  # Ensure worst-case scenario

        start_time = time.time()
        linear_search(arr, target)
        end_time = time.time()

        times.append(end_time - start_time)

    print("Non-Recursive Algorithm (Linear Search)")
    for size, t in zip(input_sizes, times):
```

```
        print(f"Input size: {size}, Time taken: {t:.5f} seconds")

analyze_non_recursive()



import time

def recursive_factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * recursive_factorial(n - 1)

def analyze_recursive():
    input_sizes = [5, 10, 15, 20, 25]
    times = []

    for size in input_sizes:
        start_time = time.time()
        recursive_factorial(size)
        end_time = time.time()

        times.append(end_time - start_time)

    print("Recursive Algorithm (Factorial)")
    for size, t in zip(input_sizes, times):
        print(f"Input size: {size}, Time taken: {t:.5f} seconds")

analyze_recursive()
```

5. Write C programs for solving recurrence relations using the Master Theorem, Substitution Method, and Iteration Method will demonstrate how to calculate the time complexity of an example recurrence relation using the specified technique.

```
import math

def master_theorem(a, b, f_n, n):
    log_b_a = math.log(a, b)
    if f_n == n:
        if f_n == n**log_b_a:
            return f"Theta(n^{log_b_a} * log(n))"
        elif f_n < n**log_b_a:
            return f"Theta(n^{log_b_a})"
        else:
            return f"Theta(f(n))"
    return "Master Theorem conditions not satisfied"

# Example usage
a = 2
b = 2
f_n = lambda n: n
n = 100
print(master_theorem(a, b, f_n(n), n))


def substitution_method(T, n):
    if n <= 1:
        return T
    else:
        return 2 * substitution_method(T, n//2) + n

# Example usage
n = 16
print(f"T({n}) =", substitution_method(1, n))
```

```
def iteration_method(T, n):
    iteration = 0
    while n > 1:
        T += n
        n = n // 2
        iteration += 1
    return T

# Example usage
n = 16
T = 1
print(f"T({n}) =", iteration_method(T, n))
```

6. Given two integer arrays nums1 and nums2, return an array of their Intersection. Each element in the result must be unique and you may return the result in any order.

```
def intersection(nums1, nums2):
    # Convert both lists to sets to remove duplicates and for fast intersection
    set1 = set(nums1)
    set2 = set(nums2)

    # Find the intersection of the two sets
    intersection_set = set1 & set2

    # Convert the set back to a list
    return list(intersection_set)

# Example usage
nums1 = [1, 2, 2, 1]
nums2 = [2, 2]
```

```
print(intersection(nums1, nums2))  # Output: [2]


nums1 = [4, 9, 5]
nums2 = [9, 4, 9, 8, 4]
print(intersection(nums1, nums2))  # Output: [9, 4]
```

7. Given two integer arrays nums1 and nums2, return an array of their intersection. Each element in the result must appear as many times as it shows in both arrays and you may return the result in any order.

```
from collections import Counter

def intersect(nums1, nums2):
    # Count the occurrences of each element in both arrays
    count1 = Counter(nums1)
    count2 = Counter(nums2)

    # Find the intersection
    intersection = []
    for num in count1:
        if num in count2:
            # Append the element min(count1[num], count2[num]) times
            intersection.extend([num] * min(count1[num], count2[num]))

    return intersection

# Example usage
nums1 = [1, 2, 2, 1]
nums2 = [2, 2]
print(intersect(nums1, nums2))  # Output: [2, 2]

nums1 = [4, 9, 5]
nums2 = [9, 4, 9, 8, 4]
```

```
print(intersect(nums1, nums2))  # Output: [4, 9] or [9, 4] (order doesn't matter)
```

8. Given an array of integers nums, sort the array in ascending order and return it.You must solve the problem without using any built-in functions in O(nlog(n)) time complexity and with the smallest space complexity possible.

```
def merge_sort(nums):
    if len(nums) <= 1:
        return nums

    mid = len(nums) // 2
    left, right = merge_sort(nums[:mid]), merge_sort(nums[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    while left and right:
        result.append(left.pop(0) if left[0] < right[0] else right.pop(0))
    return result + left + right

# Example usage:
nums = [5, 2, 9, 1, 6, 4]
sorted_nums = merge_sort(nums)
print(sorted_nums)  # Output: [1, 2, 4, 5, 6, 9]
```

9. Given an array of integers nums, half of the integers in nums are odd, and the other half are even.

10. Sort the array so that whenever nums[i] is odd, i is odd, and whenever nums[i] is even, i is even. Return any answer array that satisfies this condition.

```
def sort_array_by_parity(nums):
    even_nums = [x for x in nums if x % 2 == 0]  # Get even numbers
    odd_nums = [x for x in nums if x % 2 != 0]   # Get odd numbers
```

```
    result = []
    for i in range(len(nums)):
        if i % 2 == 0:
            result.append(even_nums.pop(0))  # Add even number at even index
        else:
            result.append(odd_nums.pop(0))   # Add odd number at odd index

    return result

# Example usage:
nums = [4, 2, 5, 7]
sorted_nums = sort_array_by_parity(nums)
print(sorted_nums)  # Output: [4, 5, 2, 7] or [2, 5, 4, 7] (any valid answer is acceptable)
```