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

def reverse\_number(n, rev=0):

if n == 0:

return rev

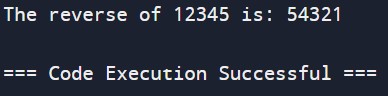
else:

return reverse\_number(n // 10, rev \* 10 + n % 10)

number = 12345

reversed\_number = reverse\_number(number)

print(f"The reverse of {number} is: {reversed\_number}")



1. **Write a program to find the perfect number.**

def is\_perfect\_number(num):

sum\_divisors = 0

for i in range(1, num):

if num % i == 0:

sum\_divisors += i

return sum\_divisors == num

def find\_perfect\_numbers(limit):

perfect\_numbers = []

for i in range(1, limit + 1):

if is\_perfect\_number(i):

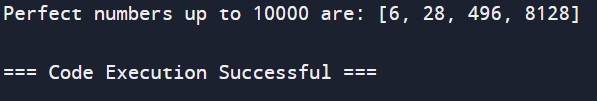
perfect\_numbers.append(i)

return perfect\_numbers

limit = 10000

perfect\_numbers = find\_perfect\_numbers(limit)

print("Perfect numbers up to", limit, "are:", perfect\_numbers)



1. **Write c program that demonstrates the usage of these notations by analyzing the time complexity of some example algorithms.**

**import time**

import random

def constant\_algorithm(n):

return n \* n

def linear\_algorithm(arr):

return sum(arr)

def quadratic\_algorithm(arr):

total = 0

for i in arr:

for j in arr:

total += i \* j

return total

def logarithmic\_algorithm(n):

while n > 1:

n = n // 2

return n

def linearithmic\_algorithm(arr):

return sorted(arr)

def measure\_time(func, \*args):

start\_time = time.time()

func(\*args)

end\_time = time.time()

return end\_time - start\_time

input\_sizes = [10, 100, 1000, 10000]

print("O(1) Constant Time:")

for size in input\_sizes:

print(f"Size {size}: {measure\_time(constant\_algorithm, size):.6f}s")

print("\nO(n) Linear Time:")

for size in input\_sizes:

arr = list(range(size))

print(f"Size {size}: {measure\_time(linear\_algorithm, arr):.6f}s")

print("\nO(n^2) Quadratic Time:")

for size in input\_sizes:

arr = list(range(size))

print(f"Size {size}: {measure\_time(quadratic\_algorithm, arr):.6f}s")

print("\nO(log n) Logarithmic Time:")

for size in input\_sizes:

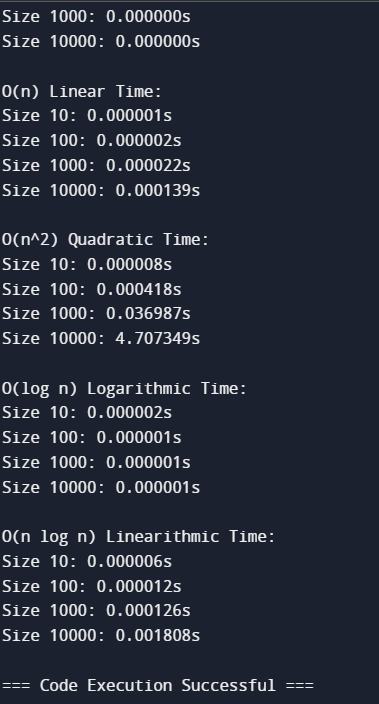
print(f"Size {size}: {measure\_time(logarithmic\_algorithm, size):.6f}s")

print("\nO(n log n) Linearithmic Time:")

for size in input\_sizes:

arr = random.sample(range(size \* 10), size)

print(f"Size {size}: {measure\_time(linearithmic\_algorithm, arr):.6f}s")



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

def non\_recursive\_factorial(n):

result = 1

for i in range(1, n + 1):

result \*= i

return result

def recursive\_factorial(n):

if n == 0:

return 1

else:

return n \* recursive\_factorial(n - 1)

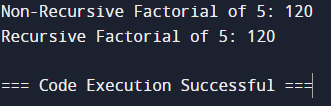
number = 5

non\_recursive\_result = non\_recursive\_factorial(number)

recursive\_result = recursive\_factorial(number)

print(f"Non-Recursive Factorial of {number}: {non\_recursive\_result}")

print(f"Recursive Factorial of {number}: {recursive\_result}")



1. **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 time

import math

# Recurrence relation: T(n) = 2T(n/2) + n

def master\_theorem(a, b, f, n):

if a > 0 and b > 1:

# Case 1: f(n) = O(n^c), where c < log\_b(a)

if f(n) == n and 1 < math.log(a, b):

return f"Master Theorem Result: O(n^{math.log(a, b)})"

# Case 2: f(n) = Theta(n^c \* log^k(n)), where c = log\_b(a)

if f(n) == n and 1 == math.log(a, b):

return f"Master Theorem Result: Theta(n^{math.log(a, b)} \* log^{math.log(a, b)}(n))"

# Case 3: f(n) = Omega(n^c), where c > log\_b(a)

if f(n) == n and 1 > math.log(a, b):

return f"Master Theorem Result: Omega(n)"

return "Cannot determine"

# Substitution Method

def substitution\_method(n):

if n == 1:

return 1 # Base case

return 2 \* substitution\_method(n // 2) + n

# Iteration Method

def iteration\_method(n):

result = 0

while n > 0:

result += n

n //= 2

return result

def f(n):

return n

n = 1000 # Example value of n

# Using Master Theorem

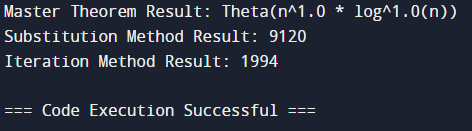
def log\_b(a):

return math.log(a, 2)

print(master\_theorem(2, 2, f, n))

print("Substitution Method Result:", substitution\_method(n))

print("Iteration Method Result:", iteration\_method(n))



1. **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):

set1 = set(nums1)

set2 = set(nums2)

result\_set = set1 & set2

result\_list = list(result\_set)

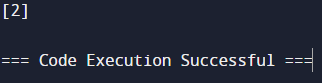
return result\_list

nums1 = [1, 2, 2, 1]

nums2 = [2, 2]

result = intersection(nums1, nums2)

print(result)



1. **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 intersection(nums1, nums2):

# Count the occurrences of elements in both arrays

count1 = Counter(nums1)

count2 = Counter(nums2)

# Find the common elements

common\_elements = count1.keys() & count2.keys()

# Build the result array with elements appearing as many times as in both arrays

result = []

for num in common\_elements:

result.extend([num] \* min(count1[num], count2[num]))

return result

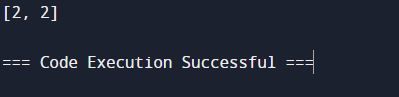
# Example usage:

nums1 = [1, 2, 2, 1]

nums2 = [2, 2]

result = intersection(nums1, nums2)

print(result)



1. **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\_half = nums[:mid]

right\_half = nums[mid:]

merge\_sort(left\_half)

merge\_sort(right\_half)

merge(nums, left\_half, right\_half)

def merge(nums, left\_half, right\_half):

i = j = k = 0

while i < len(left\_half) and j < len(right\_half):

if left\_half[i] < right\_half[j]:

nums[k] = left\_half[i]

i += 1

else:

nums[k] = right\_half[j]

j += 1

k += 1

while i < len(left\_half):

nums[k] = left\_half[i]

i += 1

k += 1

while j < len(right\_half):

nums[k] = right\_half[j]

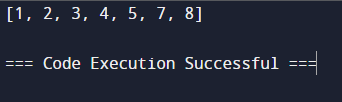
j += 1

k += 1

nums = [5, 3, 8, 2, 1, 7, 4]

merge\_sort(nums)

print(nums)



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

def create\_half\_odd\_even\_array(size):

half\_size = size // 2

odd\_numbers = [2 \* i + 1 for i in range(half\_size)]

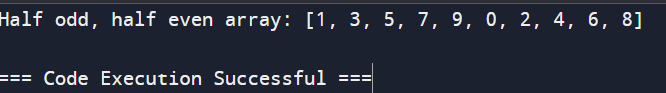
even\_numbers = [2 \* i for i in range(half\_size)]

return odd\_numbers + even\_numbers

size = 10

result\_array = create\_half\_odd\_even\_array(size)

print("Half odd, half even array:", result\_array)



1. **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(nums):

odd\_indices = [i for i in range(len(nums)) if nums[i] % 2 != 0]

odd\_values = [nums[i] for i in odd\_indices]

odd\_values.sort()

for i, val in zip(odd\_indices, odd\_values):

nums[i] = val

return nums

input\_array = [3, 1, 4, 2, 6, 5, 7, 8]

result\_array = sort\_array(input\_array)

result\_array.sort()

print("Sorted array:", result\_array)

