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

```
num= int(input("Enter the integer number: "))
revnum = 0
while (num > 0):
    rem = num%10
    revnum = (revnum*10) + rem
    num = num // 10
print("The reverse number is : {}".format(revnum))
```

output:

```
Enter the integer number: 76
The reverse number is: 67
```

2. Write a program to find the perfect number.

```
n=int(input('enter a num:'))
Sum = 0
for i in range(1, n):
    if(n % i == 0):
        Sum = Sum + i
if (Sum == n):
    print("Number is a Perfect Number.",n)
else:
    print("Number is not a Perfect Number.",n)
```

output:

```
enter a num:6
Number is a Perfect Number. 6
```

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

```
def linear_search(arr, target):
    "linear search "
    for i in range(len(arr)):
        if arr[i] == target:
```

```
            return i  # Found
    return -1  # Not found


def binary_search(arr, target):
    " binary search on the sorted list"
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
def factorial(n):
    "factorial of n"
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    target_linear = 7
    target_binary = 5
    n_factorial = 5

    # Linear search
    print("Linear Search:")
```

```
index = linear_search(arr, target_linear)
if index != -1:
    print(f"Element {target_linear} found at index {index}")
else:
    print(f"Element {target_linear} not found")

# Binary search
print("\nBinary Search:")
index = binary_search(arr, target_binary)
if index != -1:
    print(f"Element {target_binary} found at index {index}")
else:
    print(f"Element {target_binary} not found")

# Factorial
print("\nFactorial Calculation:")
result = factorial(n_factorial)
print(f"Factorial of {n_factorial} is {result}")
```

Output:

Linear Search:

Element 7 found at index 6

Binary Search:

Element 5 found at index 4

Factorial Calculation:

Factorial of 5 is 120

4. Write (python) programs that demonstrate the mathematical analysis of non-recursive and recursive algorithms.

Recursive: factorial

```
def factorial_recursive(n):
```

```
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursive(n - 1)

if __name__ == "__main__":
    n = 5
    result_recursive = factorial_recursive(n)
    print(f"Factorial of {n} (recursive): {result_recursive}")
```

o/p;

Nonrecursive: factorial

```
def factorial_iterative(n):
    if n < 0:
        raise ValueError("Factorial undefined")
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
if __name__ == "__main__":
    n = 5
    print(f"Factorial of {n} (iterative): {factorial_iterative(n)}")
```

output:

Factorial of 5 (iterative): 120

5. Write python 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 merge_sort_master_theorem(n):
    return f"{n}^log(2,2) * log({n}) = {n * math.log2(n)}"
```

```
def merge_sort_substitution_method(n):
    return f"Guess: T(n) = O(n log n)\nVerification: T(n) = 2T(n/2) + Theta(n) = 2O((n/2) log(n/2)) + Theta(n) = O(n log n)"


def merge_sort_iteration_method(n):
    iterations = ["T(n) = 2T(n/2) + Theta(n)"]
    for i in range(1, int(math.log2(n)) + 1):
        iterations.append(f"= 2^({i})T(n/2^{i}) + {2**(i-1)}Theta(n)")
    iterations.append("= n(1 + log(2) + log(3) + ... + log(n))")
    iterations.append(f"= n * log(n!)")
    return "\n".join(iterations)


if __name__ == "__main__":
    n = 8
    print("Using Master Theorem:")
    print(merge_sort_master_theorem(n))
    print()
    print("Using Substitution Method:")
    print(merge_sort_substitution_method(n))
    print()
    print("Using Iteration Method:")
    print(merge_sort_iteration_method(n))
```

Output:

```
Using Master Theorem:
8^log(2,2) * log(8) = 24.0

Using Substitution Method:
Guess: T(n) = O(n log n)
Verification: T(n) = 2T(n/2) + Theta(n) = 2O((n/2) log(n/2)) + Theta(n) = O(n log n)
```

Using Iteration Method:

T(n) = 2T(n/2) + Theta(n)

= 2^(1)T(n/2^1) + 1Theta(n)

= 2^(2)T(n/2^2) + 2Theta(n)

= 2^(3)T(n/2^3) + 4Theta(n)

= n(1 + log(2) + log(3) + ... +

log(n))

= n * log(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):
    set1 = set(nums1)
    set2 = set(nums2)
    return list(set1.intersection(set2))
nums1 = [1, 2, 2, 1]
nums2 = [2, 2, 3]
print(intersection(nums1, nums2))
```

Output: [2]

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.

```
def intersection(nums1, nums2):
    return list(set(nums1) & set(nums2))


nums1 = [1,2,1,4]
nums2 = [4,4,3,1,]
print(intersection(nums1, nums2)
```

output:

[1, 4]

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 = merge_sort(nums[:mid])
    right = merge_sort(nums[mid:])

    return merge(left, right)

def merge(left, right):
    merged = []
    left_index = right_index = 0

    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1

    merged.extend(left[left_index:])
    merged.extend(right[right_index:])

    return merged
nums = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
sorted_nums = merge_sort(nums)
```

```
print(sorted_nums)
```

Output: [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]

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

```
def sort(num):
    odd = sorted([x for x in num if x % 2 != 0])
    even = sorted([x for x in num if x % 2 == 0])
    result = []
    i = j = 0
    for k in range(len(nums)):
        if k % 2 == 0:
            result.append(even[i])
            i += 1
        else:
            result.append(odd[j])
            j += 1
    return result

n= [8,1,7,3,4]
print(sort(n))
```

OUTPUT:

[1,3,4,7,8]

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(nums):
    even= 0
    odd= 1
```

```
    sortednum = [0] * len(nums)

    for num in sorted(nums):
        if num % 2 == 0:
            sortednum[even] = num
            even+= 2
        else:
            sortednum[odd] = num
            odd+= 2

    return sortednum

n= [4, 2, 5, 7]
print(sort(n))
```

OUTPUT:

[2,4,5,7]