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

Aim: A program to find the reverse of a given number using recursive

Algorithm:

define a recursive function reverse_number that takes two Arguments: num (the number to reverse) and rev (the reversed number so far)

if the num is zero, return rev

other wise, update rev to rev * 10 + num % 10 and update num to num//10.

Recursively call reverse_number with the updated num and rev.

Program:

```
def reverse_number(num, rev=0):
    if num==0:
        return rev
    else:
        return
reverse = number(num//10, rev* 10+ num%10)
num = 12345
print (reverse_number (num))
```

input: 28

output: True

12

Write a program to find the perfect number

Aim: A program to find the perfect number

Algorithm: Define a function is_perfect that takes an integer n

- initialize a variable sum_divisors to 0.
- loop from 1 to n//2 and if divides n evenly, add to sum_divisors
- if sum_divisors equation, return true, otherwise return false.

program:

```
def is_perfect(n):
    sum_divisors=0
    for i in range(1, n//2+1):
        if i in range(1, n//2+1):
            if n%i==0:
                sum_divisors+=i
                return sum_divisors==n

n=28
print(is_perfect(n))
```

13. Write a program that demonstrates the usage of three notation by analysing the time complexity of some example algorithms.

Aim: A program to find demonstrate using of Big-O Notation

Algorithm: Define a constant time function (constant_time) perform a single operation and return the result (a(1))

Define a linear time function (linear_time):

- initialise a sum variable to 0.
- iterate through the Array.string. adding each element to the sum (o(n)).
- Return the sum

Define a quandratic time function (quadratic, time)

- use nested loops to print each pairs of indices (o(n^2))

Program:

```
def constant_time(n):
    return n+1
def linear_time(arr):
    total=0
    for num in arr:
        total += num #O(n)
    return total
def quadratic_time(arr):
    for i in range(len(arr)):
        for i in range(len(arr)):
            for j in range(len(arr1)):
                print(i,j)          # O(n)
print(constant_time(5))             # O(1)
print(linear_time([1,2,3,4,5]))     # O(n)
quadratic_time([1,2,3])             # O(n^2)
```

14. Write a program that defmanstrate the mathematical Analysis of non-recursive and recursive algorithm

Aim: A program to find mathematical Analysis of non-recursive and recursive algorithms

Algorithm: A program to find mathematical Analysis of non-recursive and recursive algorithms

- define a non-recursive linear search function (linear search):
- loop through the Array
- if the largest target element element is found, return index (o(n)).
- if the loops ends without finding the target. return -1
- A recursive factorial function (factorial):

if his 0 or 1, returns (base case).

return n factorial (n-1) (recursive case) (o(n)).

program:

```
non-recursive algorithm: (linear search)
def linear-search (arr, target):
    for i in range (in(arr)):
        if arr[i] == target:
            return: # o(n)
    return -1
Recursive algorithm: factorial
def factorial (n):
    if n==0 or n==1:
        return 1 # o(1)
    else: return n factorial (n-1) # o(n)
print (linear-search [1,2,3,4,5],4)] # o(n)
print (factorial (1)) # o(n).
```

15. write a program that demonstrate 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 specification the chrique.

Aim: A program to solving recurrence relations

Algorithm:

Master theorem:

Define a function master_theorm for $T(n) = aT(n/2) + n$

Base case: if n is; return 1

Recursive case: return 2* master_theorm $(n/2) + n$ $(O(n \log n))$.

Program:

```
def master_theorm(n):
    if n==1:
        return 1
        return 2*
    master_theorm (n/2)th  # O(nlogn)
    print(master_theorm()) # O(nlogn)
```

substitution method

Define a function substitution_method for $T(1) = T(n-1) + 1$

Base case: if n(s), return 1.

Recursive case: Return substitution method $(n-1) + 1$ $(O(n))$.

```
def substitution_method(n):
    if n==1:
        return 1
    return substitutionmethod(n-1) + 1    # O(n)
print (substitution_method(5)) # O(n)
```

iteration method

- Define a function iteration method for $T(n) = 2T(n/2) + n$,

  Base case: if n=1; return 1.

  Recursive case: return 2^n iteration method $(n//2) + n$ $(2T(n/2) + n)$

```
def iteration_method(n):
    if n==1:
        return 1
    return 2^n iteration_method(n//2)^th   # O(n log n)
print(iteration_method(9))   # O(n log n)
```

Input:

nums1 = [1,2,2,1]

nums2 = [2,2]

output: [2]

16

Aim: A program to find intersection of two arrays (unique elements)

Algorithm: Convert both arrays into sets to remove duplicates. Find the intersection of the two sets using the & operator. Convert the resulting set back to a list and return it.

Program:

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

nums1 = [1,2,2,1]
nums2 = [2,2]
print(intersection_unique(nums1, nums2))
```

Input:

$n_1 = [1, 2, 2, 1]$

$n_2 = [2, 2]$

output:

$[2, 2]$

17. Given 2 integer arrays nums 1 and nums 2, return an array of their intersection. each element in the result must be unique and you may return the result in any other.

Aim: A program to find intersection of two Arrays (elements Appearing multiple Times).

Algorithm:

use counter then collections to count the elements in num1 and num2.

find the intersection of the counts by iterating over counts 1.

for each common element. extend the result list by the minimum count from both Arrays.

program:

```
def intersection_multiset(nums1, nums2):
    count1 = counter(nums 1)
    counts 2 = counter(nums 2)
    intersection = []

    for num in counts 1:
        if num in counts 2:
intersection, extend [num] min [counts 1 [num], counts 2
    return intersection.                          [num]))

num1 = [1,2,2,1]

nums 2 = [2,2]
print (intersection.multiset (num1, num 2)) # output: [2,2]
```

18. Given an array of integers nums, sort the array in ascending order and return it. you must solve the problem without using any build-in functions in O(nlogn) time complexity and with the smallest space complexity possible.

Aim: A program to find sorted an Array in, Assembly order (o(n(logn)).

Algorithm:

Define a merge sort function (merge_sort)

- if the array has more has than one element:
- find the middle index and divide the array into two halves recursively apply merge, sort to each half
- merge the sorted halves

merge functions:

initialize three pointers for the left half, right half, and the main array

compare elements of left and right halt

Program:

```
def merge_sort(arr):
    if len(arr)>1:
        mid = len(arr)//2
        L = arr[mid]
        R = arr[mid:]
merge_sort[L)
    merge_sort(R]
i = j = k = 0
while i< len(L) and j < len(R):
if L[i] < R[j]:
        arr[k] = L[i]
```

```
            i += 1
        else:
            arr[k] = R[j]
            j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]  i += 1  k += 1
    while j < len(R):
        arr[k] = R[j]  j += 1  k += 1
    return arr
print(merge_sort(nums))
```

Input:

nums = [4, 1, 2, 3, 6, 7, 8, 5]

output:

[1, 4, 3, 2, 7, 6, 5, 8]

can valid arrangment

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

Aim: To find sort an Array of integer such that half are odd and half are even

Algorithm:

initialize two pointers: odd index starting at the begining of the array (index 0) and even index starting at the begining array (index 0).

loop through the array:

if the current element is odd and even index is at an increment by 2.

After swap, increment both odd_index and even_index by 2 continue this process until all elements are correctly placed.

program:

```
def sort_half_odd_half_even(nums):
    odd_index = 0
    even_index = 1
    n = len(nums)
while odd_index < n and even_index < n:
    if nums[odd_index] % 2 == 0
while even_index < n and nums[even_index] % 2 == 0:
        even_index += 2
    if even_index < n:
        nums[odd_index], nums[even_index] =
num[even_index], nums[odd_index]
        odd_index += 2
    return nums
nums = [4, 1, 2, 3, 6, 7, 8, 5]
print(sort_half_odd_half_even(nums))
```

Input :

Nums = [4, 1, 2, 3, 6, 7, 8, 5]

output :

[4, 3, 2, 1, 6, 7, 8, 5]

(any valid Arrangement)

80. sort the Array so that whenever nums[i] is odd, i is odd and when even nums[i] is even, i is even. return any answer array that satisfies this condition

Aim: A program to find sort an Array such that whenever nums[i] is odd, i is odd and whenever nums[i] is even, i is even

Algorithm:

- seperate the numbers into odd and even
- sort them
- merge them back to ensure the condition

If the element at even index is odd, odd index is even swap them

- continue until all elements are in the correct positions.

program:

```
def sort_array_by_parity(nums):
    odd_index = 1
    even_index = 0
    n = len(nums)
    while odd_index < n and even_index < n:
        if nums[even_index] % 2 == 0
            even_index += 2
        dif nums[odd_index] % 2 == 1:
            odd_index += 2
        else:
            nums[even_index], nums[odd_index] =
            nums[odd_index], nums[even_index]
            even_index += 2
            odd_index += 2
    return nums

nums = [4,1,2,3,6,7,8,5]
print(sort_array_by_parity(nums))
```

7/6/24