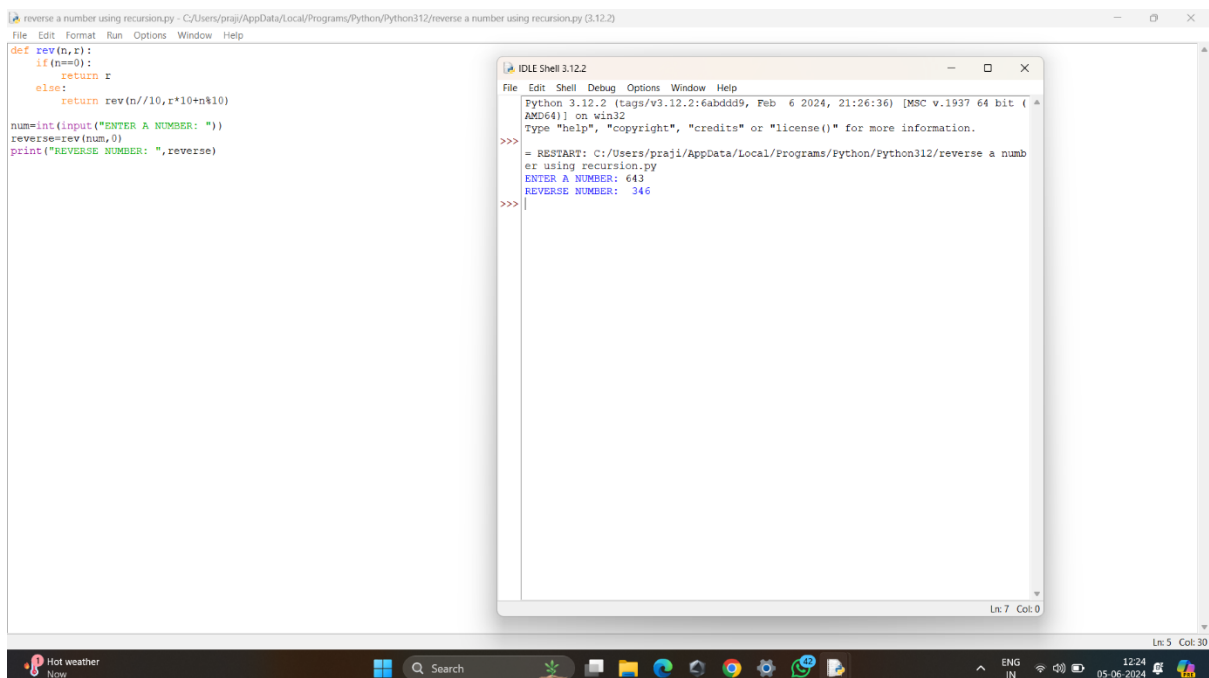


1. Write a program to find the reverse of a given number using recursive.

PROGRAM:

```
def rev(n,r):  
    if(n==0):  
        return r  
    else:  
        return rev(n//10,r*10+n%10)  
  
num=int(input("ENTER A NUMBER: "))  
reverse=rev(num,0)  
print("REVERSE NUMBER: ",reverse)
```

Time:O(n)

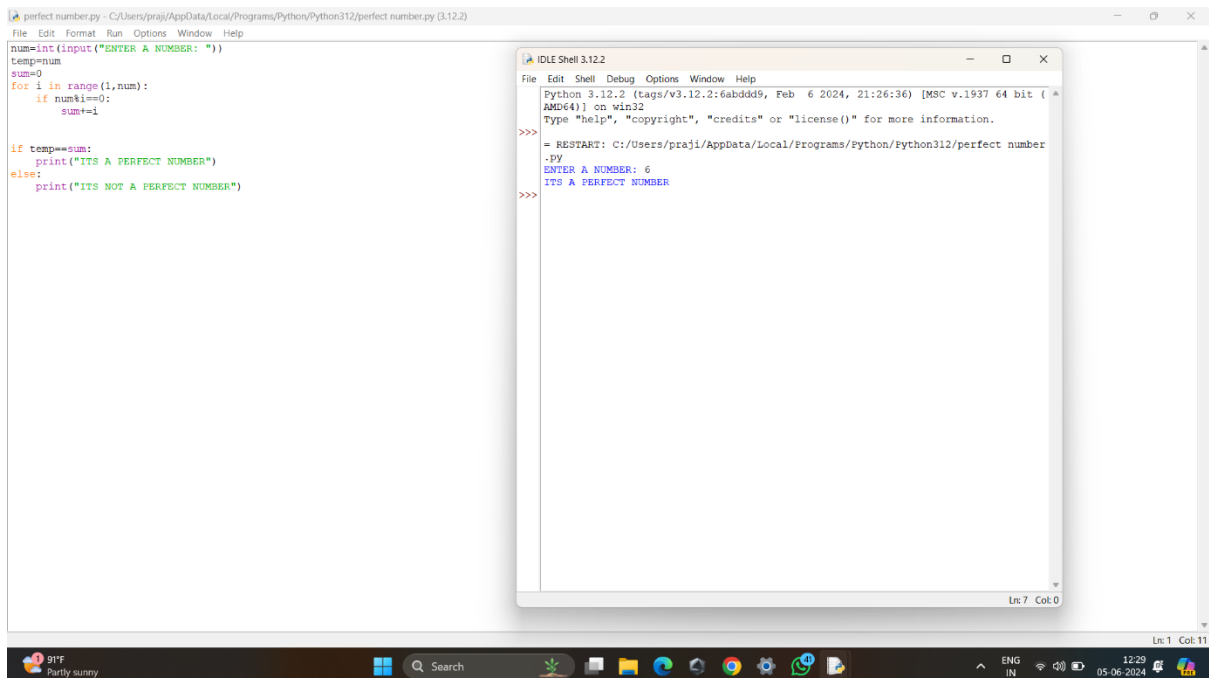


2. Write a program to find the perfect number.

PROGRAM:

```
num=int(input("ENTER A NUMBER: "))
temp=num
sum=0
for i in range(1,num):
    if num%i==0:
        sum+=i
if temp==sum:
    print("ITS A PERFECT NUMBER")
else:
    print("ITS NOT A PERFECT NUMBER")
```

Time:O(n)



The screenshot displays a Python IDE window titled "perfect number.py" with the following code:

```
num=int(input("ENTER A NUMBER: "))
temp=num
sum=0
for i in range(1,num):
    if num%i==0:
        sum+=i
if temp==sum:
    print("ITS A PERFECT NUMBER")
else:
    print("ITS NOT A PERFECT NUMBER")
```

An "IDLE Shell 3.12.2" window is open, showing the execution of the program. It displays the prompt "ENTER A NUMBER: 6" and the output "ITS A PERFECT NUMBER". The Windows taskbar at the bottom shows the date as 05-06-2024 and the time as 12:29.

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

PROGRAM:

$O(n)$

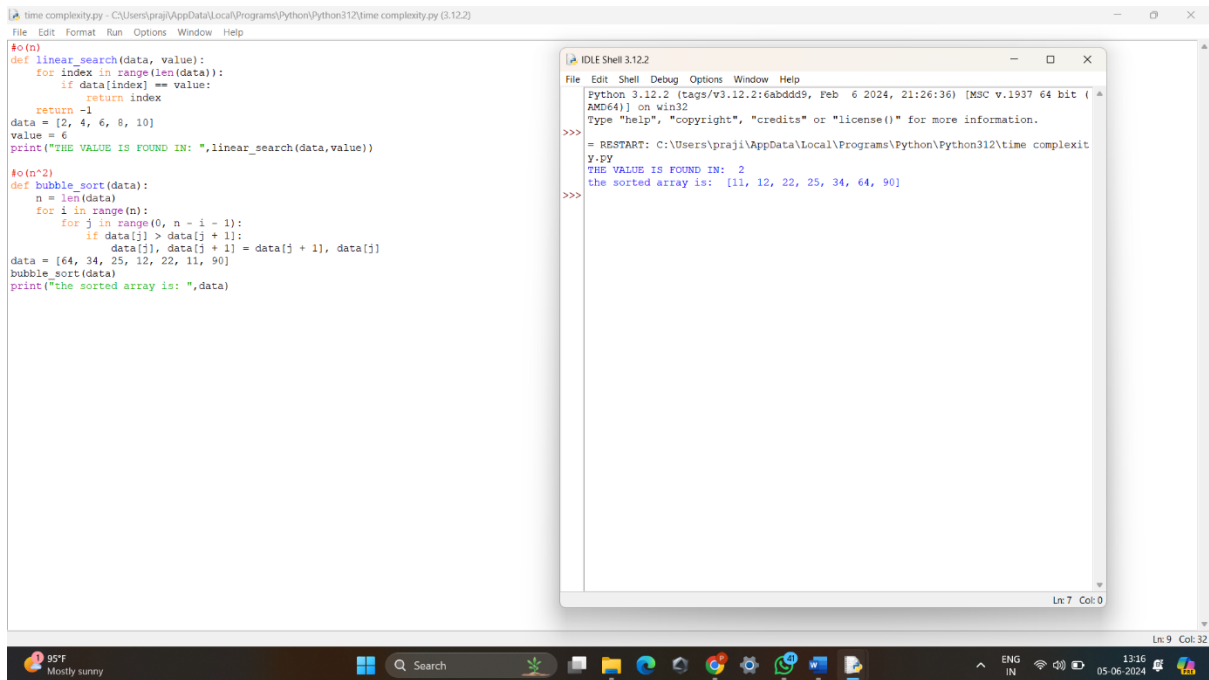
```
def linear_search(data, value):
    for index in range(len(data)):
        if data[index] == value:
            return index
    return -1

data = [2, 4, 6, 8, 10]
value = 6
print("THE VALUE IS FOUND IN: ", linear_search(data, value))
```

$O(n^2)$

```
def bubble_sort(data):
    n = len(data)
    for i in range(n):
        for j in range(0, n - i - 1):
            if data[j] > data[j + 1]:
                data[j], data[j + 1] = data[j + 1], data[j]

data = [64, 34, 25, 12, 22, 11, 90]
bubble_sort(data)
print("the sorted array is: ", data)
```



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

PROGRAM:

#time: $O(n)$

def factorial(n):

 result = 1

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

 result *= i

 return result

num = 5

print("The factorial of the number is ", factorial(num))

#time: $O(2^n)$

def fibonacci(n):

if n <= 1:

return n

else:

return fibonacci(n - 1) + fibonacci(n - 2)

num = 10

print("The fibonacci series is ")

for i in range(0,num):

print(fibonacci(i))

The screenshot shows a Python IDE window titled 'recursive non recursive.py' with the following code:

```
#time: O(2^n)
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
num = 5
print("The factorial of the number is ",factorial(num))

#time: o(2^n)
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
num = 10
print("The fibonacci series is ")
for i in range(0,num):
    print(fibonacci(i))
```

An 'IDLE Shell 3.12.2' window is open, displaying the output of the program:

```
= RESTART: C:/Users/praji/AppData/Local/Programs/Python/Python312/recursive non recursive.py
The factorial of the number is  120
The fibonacci series is
0
1
1
2
3
5
8
13
21
34
```

The Windows taskbar at the bottom shows the date and time as 05-06-2024, 13:19.

5. Write C programs for solving recurrence relations using the Master Theorem, Substitution Method, and Iteration Method will demonstrate how to calculate the time

PROGRAM:

```
def master_theorem(a, b, k):
```

```
    if a < b**k:
```

```
        return "O(log n^b)"
```

```
    elif a == b**k:
```

```
        return "O(n^k)"
```

```
    else:
```

```
        return "O(n^(log a / log b))"
```

```
recurrence = "T(n) = 2T(n/2) + n^2"
```

```
a, b, k = 2, 2, 2
```

```
time_complexity = master_theorem(a, b, k)
```

```
print(f"Time complexity of the recurrence relation: {time_complexity}")
```

```
def iteration(recurrence, n):
```

```
    if recurrence == "T(n) = T(n-1) + n":
```

```
        solution = 0
```

```
        for i in range(n):
```

```
            solution += i
```

```
        return solution
```

```
recurrence = "T(n) = T(n-1) + n"
```

```
n = 3
```

```
solution = iteration(recurrence, n)
```

```
print(f"Solution of the recurrence T(n) at n={n} using iteration: {solution}")
```

```
def substitution(recurrence, n):
```

```
if recurrence == "T(n) = T(n-1) + 1":
```

```
    if n == 0:
```

```
        return 0
```

```
    else:
```

```
        return substitution(recurrence, n-1) + 1
```

```
recurrence = "T(n) = T(n-1) + 1"
```

```
n = 3
```

```
solution = substitution(recurrence, n)
```

```
print(f"Solution of the recurrence T(n) at n={n} using substitution: {solution}")
```

Time: $O(N^2)$

The screenshot shows a Python IDE with two windows. The main window displays a Python script for solving recurrence relations using the Master Theorem, iteration, and substitution methods. The script defines functions for time complexity, iteration, and substitution, and then applies them to the recurrence $T(n) = T(n-1) + 1$ with $n=3$. The output window shows the execution results, including the time complexity $O(\log n^b)$ and the solution for $n=3$ using both iteration and substitution methods.

```
def master_theorem(a, b, k):  
    if a < b**k:  
        return "O(log n^b)"  
    elif a == b**k:  
        return "O(n^k)"  
    else:  
        return "O(n*(log a / log b))"  
  
recurrence = "T(n) = 2T(n/2) + n^2"  
a, b, k = 2, 2, 2  
  
time_complexity = master_theorem(a, b, k)  
print(f"Time complexity of the recurrence relation: {time_complexity}")  
  
def iteration(recurrence, n):  
    if recurrence == "T(n) = T(n-1) + n":  
        solution = 0  
        for i in range(n):  
            solution += i  
        return solution  
  
recurrence = "T(n) = T(n-1) + n"  
n = 3  
  
solution = iteration(recurrence, n)  
print(f"Solution of the recurrence T(n) at n={n} using iteration: {solution}")  
  
def substitution(recurrence, n):  
    if recurrence == "T(n) = T(n-1) + 1":  
        if n == 0:  
            return 0  
        else:  
            return substitution(recurrence, n-1) + 1  
  
recurrence = "T(n) = T(n-1) + 1"  
n = 3  
  
solution = substitution(recurrence, n)  
print(f"Solution of the recurrence T(n) at n={n} using substitution: {solution}")
```

Python 3.12.2 (tags/v3.12.2:6abddd9, Feb 6 2024, 21:26:36) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> = RESTART: C:/Users/prajl/AppData/Local/Programs/Python/Python312/master theorem
Time complexity of the recurrence relation: O(log n^b)
Solution of the recurrence T(n) at n=3 using iteration: 3
Solution of the recurrence T(n) at n=3 using substitution: 3
>>>

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.

PROGRAM:

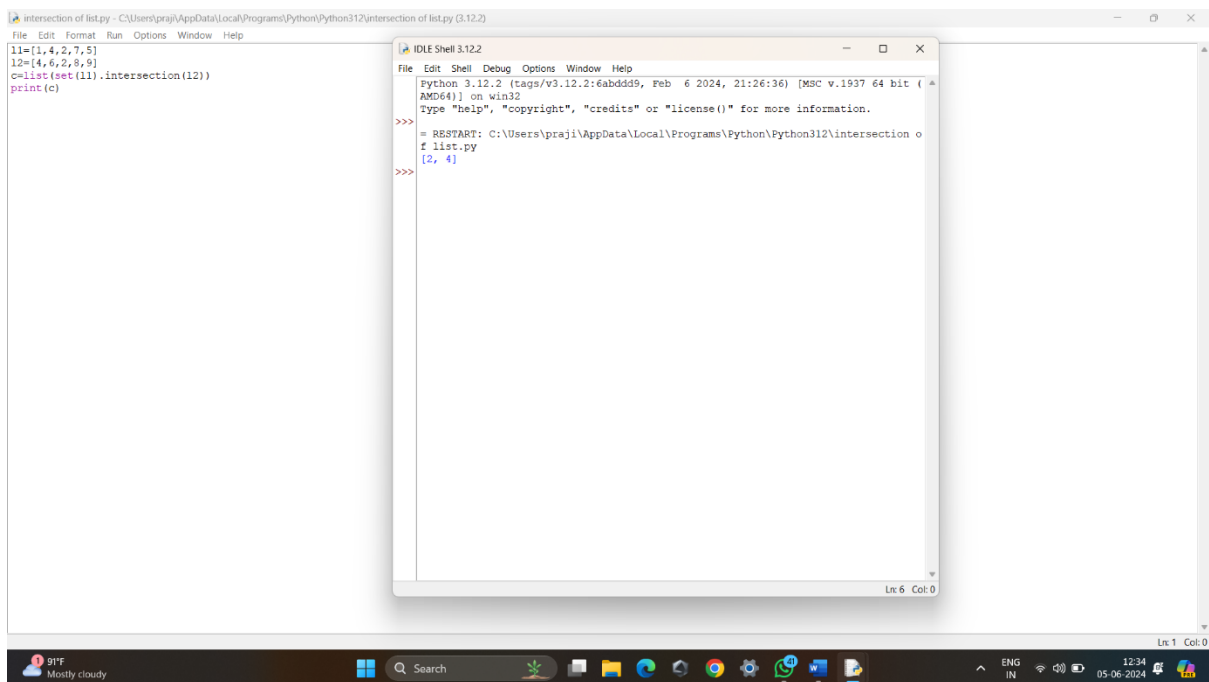
```
l1=[1,4,2,7,5]
```

```
l2=[4,6,2,8,9]
```

```
c=list(set(l1).intersection(l2))
```

```
print(c)
```

Time:O(N)



The screenshot shows a Python IDE with two windows. The main window, titled 'intersection of list.py - C:\Users\praji\AppData\Local\Programs\Python\Python312\intersection of list.py (3.12.2)', contains the following code:

```
l1=[1,4,2,7,5]
l2=[4,6,2,8,9]
c=list(set(l1).intersection(l2))
print(c)
```

The output window, titled 'IDLE Shell 3.12.2', shows the execution results:

```
>>>
= RESTART: C:\Users\praji\AppData\Local\Programs\Python\Python312\intersection o
f list.py
[2, 4]
>>>
```

The taskbar at the bottom shows the system clock as 12:34 on 05-06-2024, with a weather widget indicating 91°F and 'Mostly cloudy'.

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.

PROGRAM:

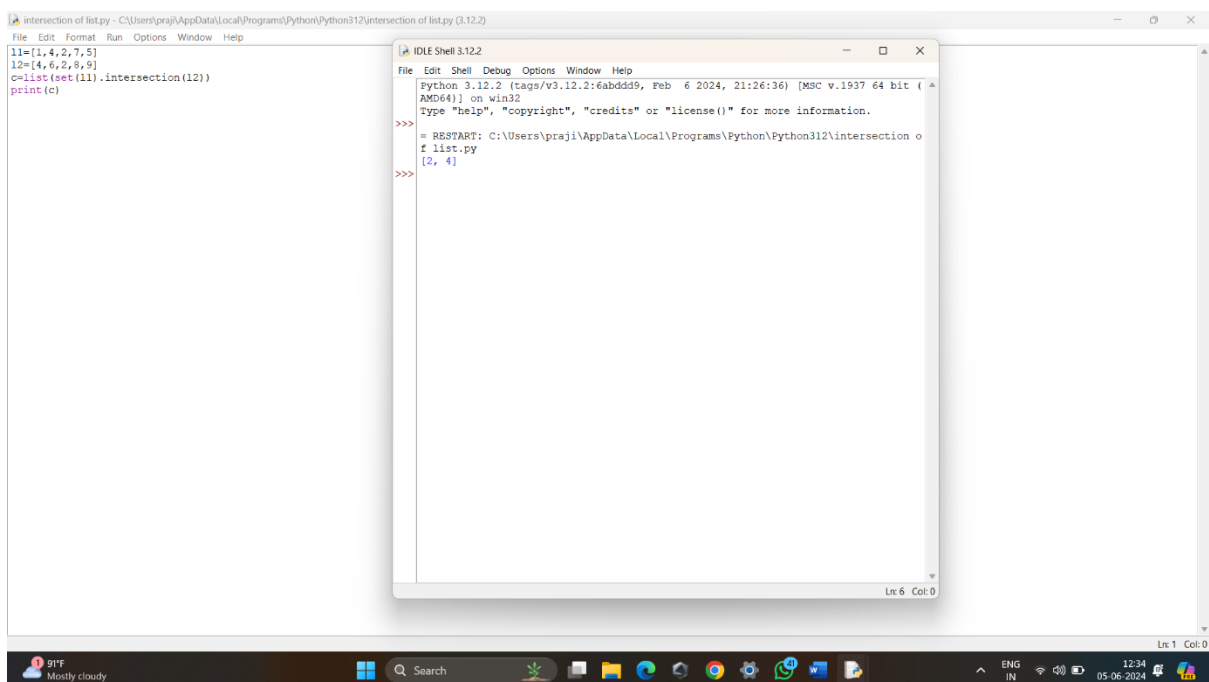
```
l1=[1,4,2,7,5]
```

```
l2=[4,6,2,8,9]
```

```
c=list(set(l1).intersection(l2))
```

```
print(c)
```

Time: $O(n)$



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 $O(n \log(n))$ time complexity and with the smallest space complexity possible.

PROGRAM:

```
def partition(array, low, high):  
    pivot = array[high]  
    i = low - 1  
    for j in range(low, high):  
        if array[j] <= pivot:  
            i = i + 1  
            (array[i], array[j]) = (array[j], array[i])  
    (array[i + 1], array[high]) = (array[high], array[i + 1])  
    return i + 1
```

```
def quickSort(array, low, high):  
    if low < high:  
        pi = partition(array, low, high)  
        quickSort(array, low, pi - 1)  
        quickSort(array, pi + 1, high)
```

```
data = [1, 7, 4, 1, 10, 9, -2]  
print("Unsorted Array")  
print(data)
```

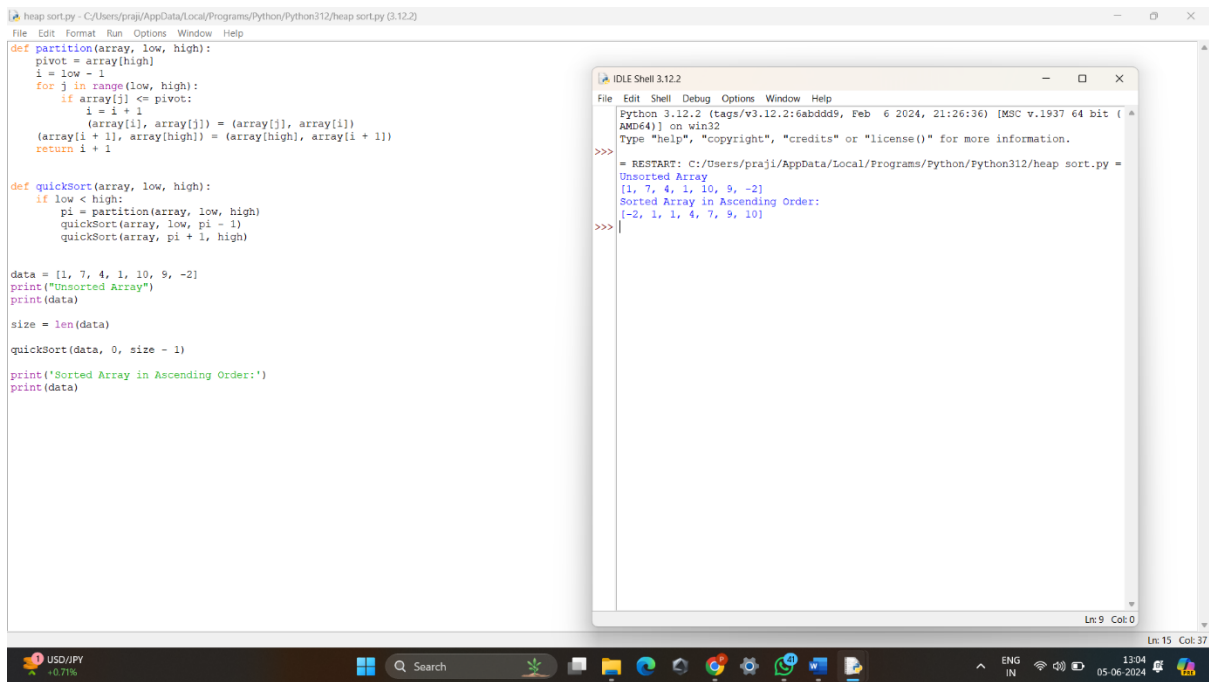
```
size = len(data)
```

```
quickSort(data, 0, size - 1)
```

```
print('Sorted Array in Ascending Order:')
```

```
print(data)
```

Time: $O(n \log n)$



The screenshot shows a Python IDE with a file named 'heap sort.py'. The code implements a quicksort algorithm. The 'partition' function selects the last element as a pivot and rearranges the array. The 'quickSort' function recursively sorts the array. The main code defines a data array [1, 7, 4, 1, 10, 9, -2], prints it as 'Unsorted Array', and then sorts it using quickSort, printing the result as 'Sorted Array in Ascending Order:'. An 'IDLE Shell' window shows the execution output: 'Unsorted Array' followed by '[1, 7, 4, 1, 10, 9, -2]', and 'Sorted Array in Ascending Order:' followed by '[-2, 1, 1, 4, 7, 9, 10]'. The Windows taskbar at the bottom shows the date as 05-06-2024 and time as 13:04.

```
def partition(array, low, high):
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            (array[i], array[j]) = (array[j], array[i])
    (array[i + 1], array[high]) = (array[high], array[i + 1])
    return i + 1

def quickSort(array, low, high):
    if low < high:
        pi = partition(array, low, high)
        quickSort(array, low, pi - 1)
        quickSort(array, pi + 1, high)

data = [1, 7, 4, 1, 10, 9, -2]
print("Unsorted Array")
print(data)

size = len(data)
quickSort(data, 0, size - 1)

print('Sorted Array in Ascending Order:')
print(data)
```

```
>>>
= RESTART: C:/Users/praji/AppData/Local/Programs/Python/Python312/heap sort.py =
Unsorted Array
[1, 7, 4, 1, 10, 9, -2]
Sorted Array in Ascending Order:
[-2, 1, 1, 4, 7, 9, 10]
>>>
```

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

PROGRAM:

```
def sort(nums):
    odd=[]
    even=[]
    res=[]
    for num in nums:
        if num%2==0:
            even.append(num)
        else:
            odd.append(num)
    for i in range(len(nums)):
        if i%2==0:
            res.append(even.pop())
        else:
            res.append(odd.pop())
    return res
nums=[1,2,6,7]
sorted_num= sort(nums)
print(sorted_num)
```

Time:O(N)

