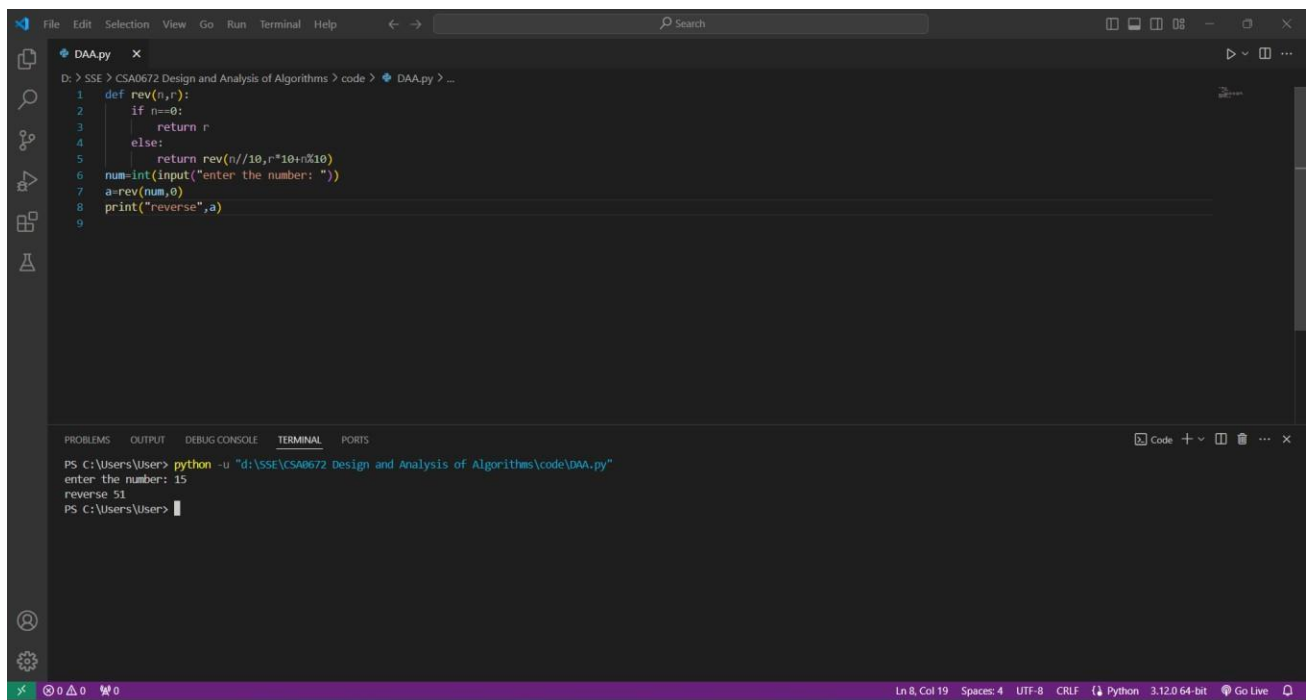


1. Write a program to find the reverse of a given number using recursive.

Code:

```
def rev(n,r):
    if n==0:
        return r
    else:
        return rev(n//10,r*10+n%10)
num=int(input("enter the number: "))
a=rev(num,0)
print("reverse",a)
```

Screenshot for I/O:

A screenshot of a code editor window with a dark theme. The editor shows a Python file named 'DAA.py' containing the recursive function 'rev' and its usage. Below the editor, a terminal window displays the command to run the script and the resulting output. The code in the editor is:

```
1 def rev(n,r):
2     if n==0:
3         return r
4     else:
5         return rev(n//10,r*10+n%10)
6 num=int(input("enter the number: "))
7 a=rev(num,0)
8 print("reverse",a)
9
```

 The terminal output shows:

```
PS C:\Users\User> python -u "d:\SSE\CSA0672 Design and Analysis of Algorithms\code\DAA.py"
enter the number: 15
reverse 51
PS C:\Users\User>
```

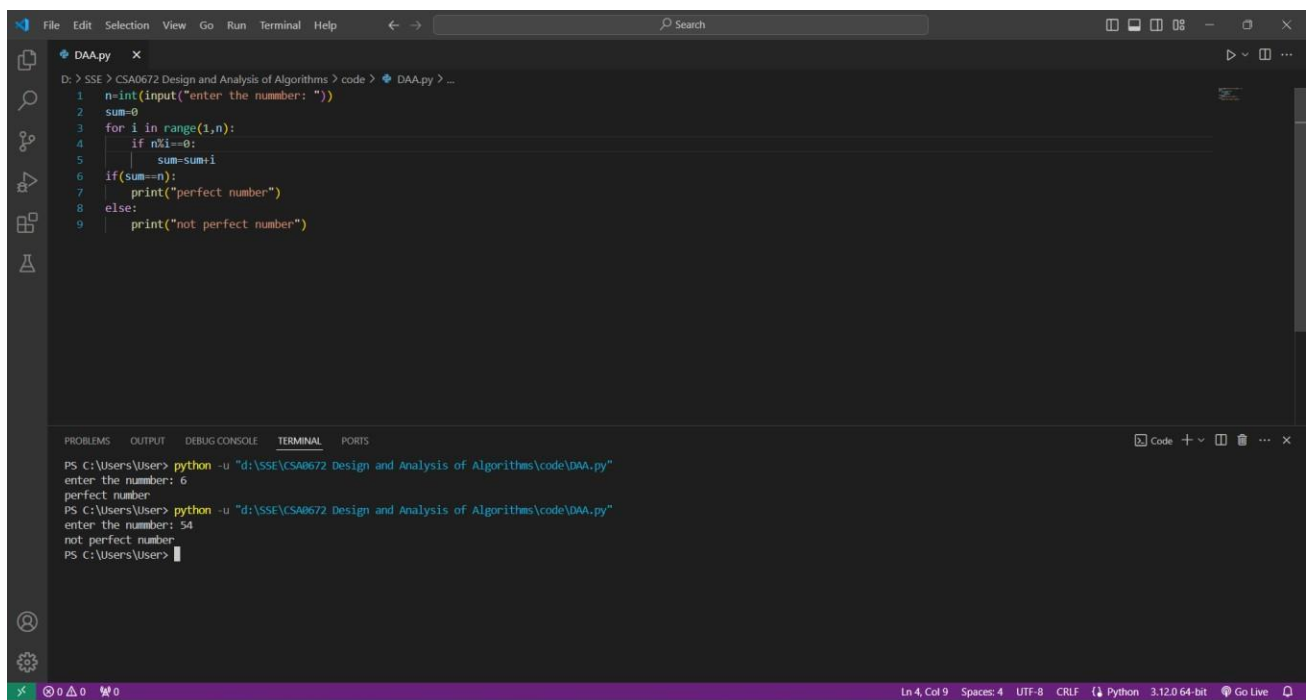
Time Complexity: $O(n)$

2. Write a program to find the perfect number.

Code:

```
n=int(input("enter the nummber: "))
sum=0
for i in range(1,n):
    if n%i==0:
        sum=sum+i
if(sum==n):
    print("perfect number")
else:
    print("not perfect number")
```

Screenshot for I/O:



The screenshot displays a code editor with a Python script named 'DAA.py'. The script implements a function to check if a number is perfect by summing its divisors. Below the code, the terminal window shows the execution of the program twice: first with input '6' resulting in 'perfect number', and then with input '54' resulting in 'not perfect number'.

```
D:\SSE\CSA0672 Design and Analysis of Algorithms\code> python DAA.py
1 n=int(input("enter the number: "))
2 sum=0
3 for i in range(1,n):
4     if n%i==0:
5         sum=sum+i
6 if(sum==n):
7     print("perfect number")
8 else:
9     print("not perfect number")

PS C:\Users\User> python -u "d:\SSE\CSA0672 Design and Analysis of Algorithms\code\DAA.py"
enter the number: 6
perfect number
PS C:\Users\User> python -u "d:\SSE\CSA0672 Design and Analysis of Algorithms\code\DAA.py"
enter the number: 54
not perfect number
PS C:\Users\User>
```

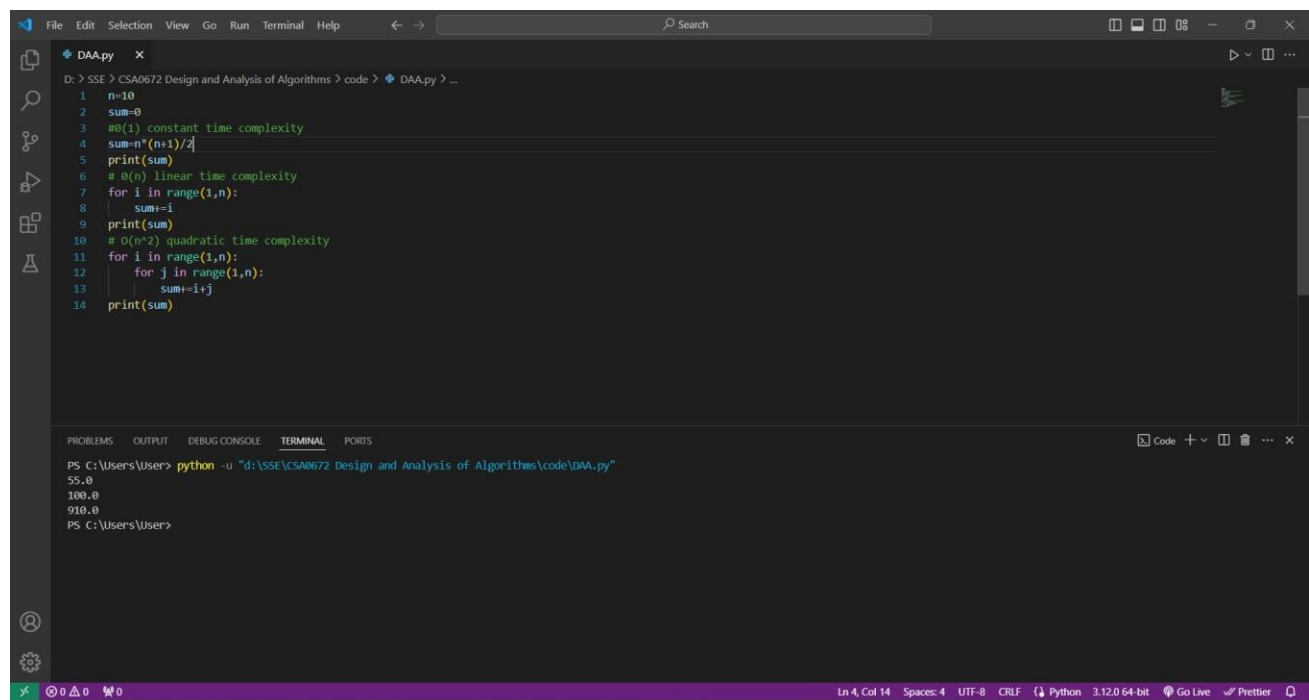
Time Complexity: $O(n)$

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

Code:

```
n=10
sum=0
#O(1) constant time complexity
sum=n*(n+1)/2
print(sum)
# O(n) linear time complexity
for i in range(1,n):
    sum+=i
print(sum)
# O(n^2) quadratic time complexity
for i in range(1,n):
    for j in range(1,n):
        sum+=i+j
print(sum)
```

Screenshot for I/O:



The screenshot shows a code editor with a file named `DAA.py` open. The code in the editor is as follows:

```
1 n=10
2 sum=0
3 #O(1) constant time complexity
4 sum=n*(n+1)/2
5 print(sum)
6 # O(n) linear time complexity
7 for i in range(1,n):
8     sum+=i
9 print(sum)
10 # O(n^2) quadratic time complexity
11 for i in range(1,n):
12     for j in range(1,n):
13         sum+=i+j
14 print(sum)
```

Below the code editor, the terminal output is displayed, showing the execution of the program:

```
PS C:\Users\User> python -u "d:\SSE\CSA0672 Design and Analysis of Algorithms\code\DAA.py"
55.0
100.0
910.0
PS C:\Users\User>
```

The status bar at the bottom of the editor indicates the current line and column (Ln 4, Col 14), the encoding (UTF-8), the line ending (CRLF), the Python version (3.12.0 64-bit), and the active theme (Go Live, Prettier).

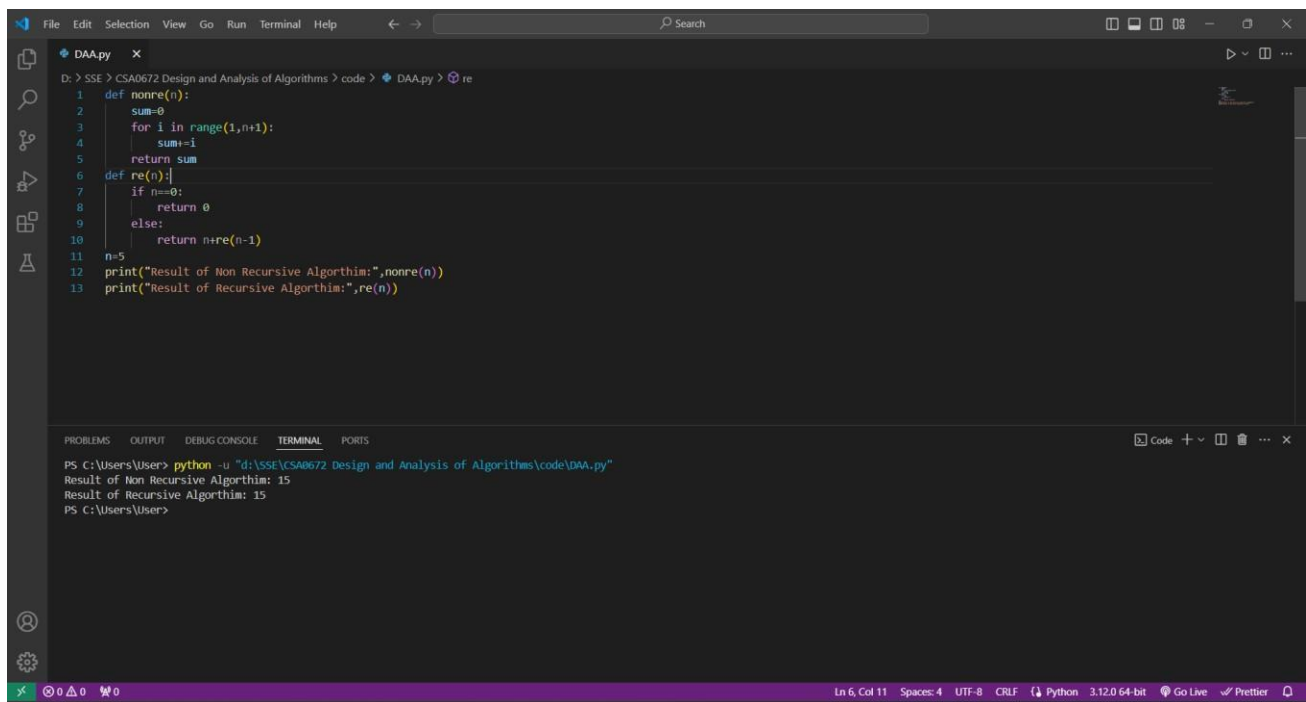
Time Complexity: $O(n^2)$

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

Code:

```
def nonre(n):
    sum=0
    for i in range(1,n+1):
        sum+=i
    return sum
def re(n):
    if n==0:
        return 0
    else:
        return n+re(n-1)
n=5
print("Result of Non Recursive Algorithim:",nonre(n))
print("Result of Recursive Algorithim:",re(n))
```

Screenshot for I/O:

A screenshot of a code editor window with a dark theme. The editor shows a Python file named 'DAA.py' with the following code:

```
1 def nonre(n):
2     sum=0
3     for i in range(1,n+1):
4         sum+=i
5     return sum
6 def re(n):
7     if n==0:
8         return 0
9     else:
10        return n+re(n-1)
11 n=5
12 print("Result of Non Recursive Algorithim:",nonre(n))
13 print("Result of Recursive Algorithim:",re(n))
```

 Below the code editor is a terminal window showing the command `python -u "d:\SSE\CSA0672 Design and Analysis of Algorithms\code\DAA.py"` and its output: `Result of Non Recursive Algorithm: 15` and `Result of Recursive Algorithm: 15`. The status bar at the bottom indicates 'Ln 6, Col 11', 'Spaces: 4', 'UTF-8', 'CRLF', 'Python', '3.12.0 64-bit', 'Go Live', and 'Prettier'.

Time Complexity: $O(n)$

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.

Code:

```
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

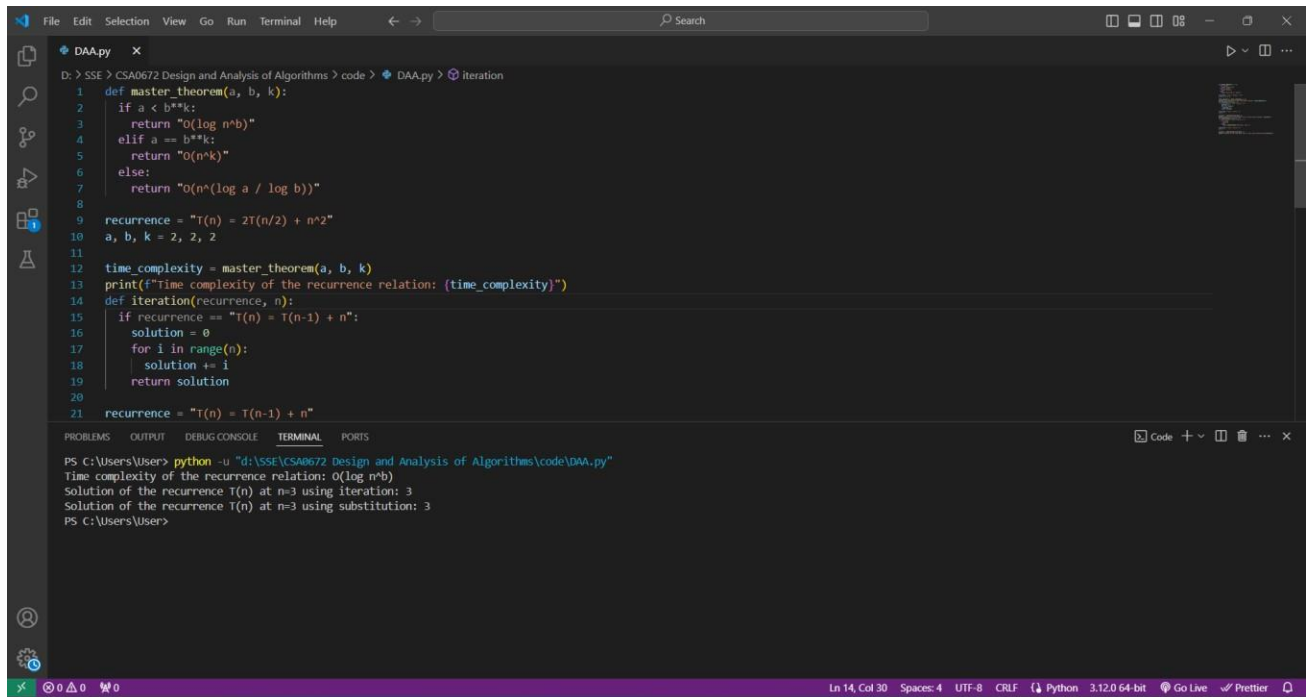
solution = iteration(recurrence, n)
print(f"Solution of the recurrence T(n) at n={n} using iteration: {solution}")

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}")
```

Screenshot for I/O:



The screenshot displays a Visual Studio Code editor window with a Python file named `DAA.py`. The code defines a `master_theorem` function to calculate time complexity based on the form of a recurrence relation. It also includes an `iteration` function to solve a specific recurrence relation $T(n) = 2T(n/2) + n^2$ for $n=3$ using an iterative approach. The terminal output shows the execution results, including the time complexity $O(\log n^b)$ and the solution value of 3.

```
D:\> SSE > CSA0672 Design and Analysis of Algorithms > code > DAA.py > iteration
1 def master_theorem(a, b, k):
2     if a < b**k:
3         return "O(log n^b)"
4     elif a == b**k:
5         return "O(n^k)"
6     else:
7         return "O(n^(log a / log b))"
8
9 recurrence = "T(n) = 2T(n/2) + n^2"
10 a, b, k = 2, 2, 2
11
12 time_complexity = master_theorem(a, b, k)
13 print(f"Time complexity of the recurrence relation: {time_complexity}")
14 def iteration(recurrence, n):
15     if recurrence == "T(n) = T(n-1) + n":
16         solution = 0
17         for i in range(n):
18             solution += 1
19         return solution
20
21 recurrence = "T(n) = T(n-1) + n"
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\User> python -u "d:\SSE\CSA0672 Design and Analysis of Algorithms\code\DAA.py"
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
PS C:\Users\User>
```

Ln 14, Col 30 Spaces: 4 UTF-8 CRLF Python 3.12.0 64-bit Go Live Prettier

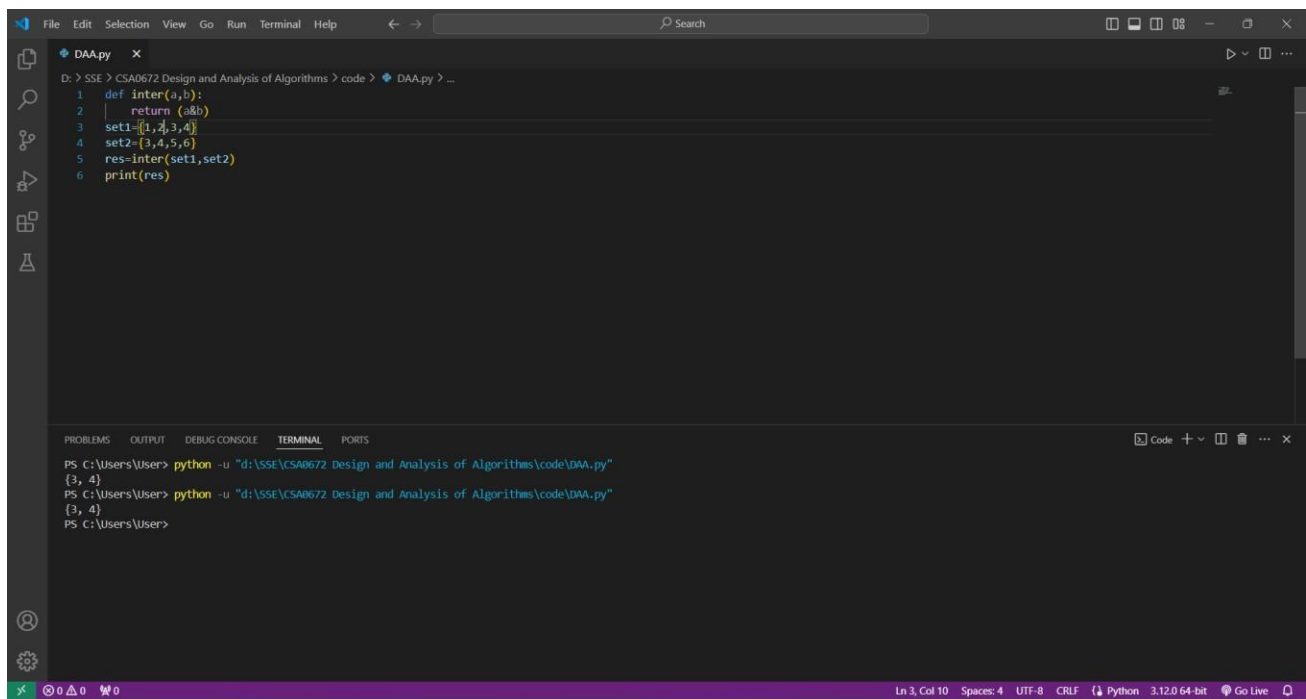
Time Complexity: $O(n)$

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.

Code:

```
def inter(a,b):  
    return (a&b)  
set1={1,2,3,4}  
set2={3,4,5,6}  
res=inter(set1,set2)  
print(res)
```

Screenshot for I/O:



The screenshot shows a code editor with a file named 'DAA.py'. The code defines a function 'inter(a,b)' that returns the bitwise AND of two sets. It then creates two sets, 'set1' and 'set2', and calls the 'inter' function on them, printing the result. The terminal output shows the command 'python -u "d:\SSE\CSA0672 Design and Analysis of Algorithms\code\DAA.py"' being executed, and the output is '{3, 4}'.

```
D:\SSE\CSA0672 Design and Analysis of Algorithms\code> python -u "d:\SSE\CSA0672 Design and Analysis of Algorithms\code\DAA.py"  
{3, 4}
```

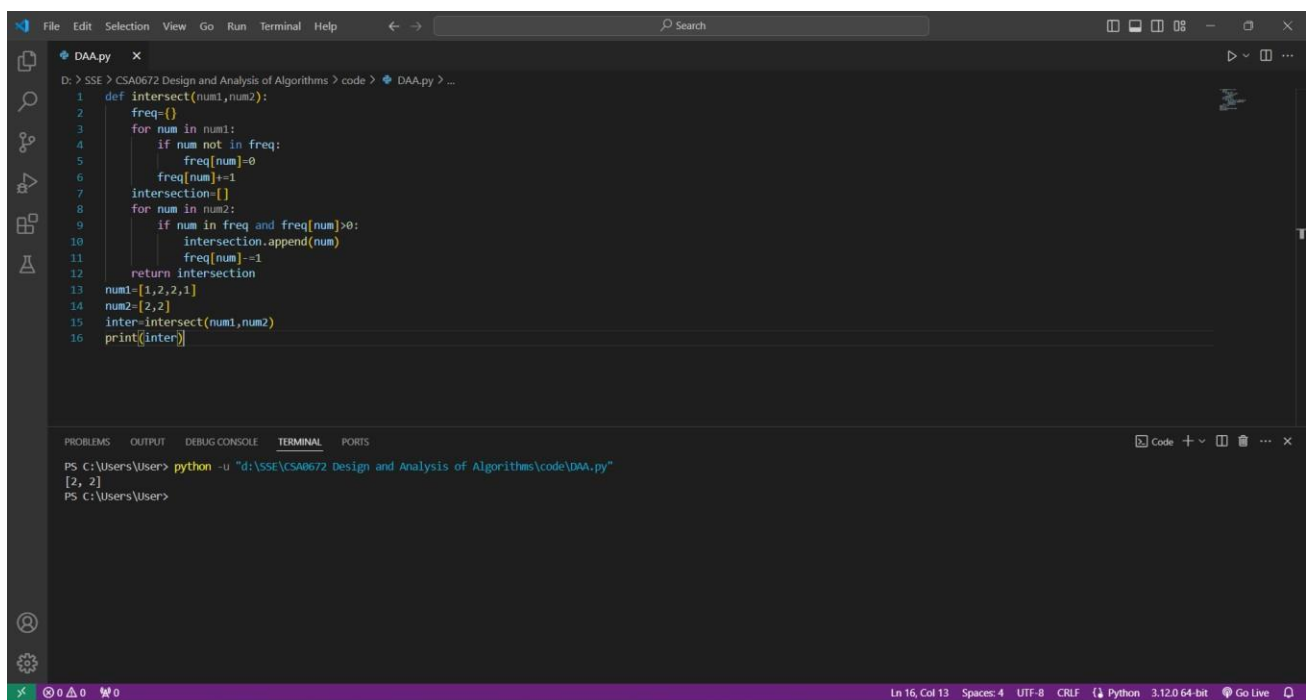
Time Complexity: $O(n)$

7. Given two integer arrays num1 and num2, 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.

Code:

```
def intersect(num1,num2):
    freq={}
    for num in num1:
        if num not in freq:
            freq[num]=0
        freq[num]+=1
    intersection=[]
    for num in num2:
        if num in freq and freq[num]>0:
            intersection.append(num)
            freq[num]-=1
    return intersection
num1=[1,2,2,1]
num2=[2,2]
inter=intersect(num1,num2)
print(inter)
```

Screenshot for I/O:

A screenshot of a code editor window titled 'DAA.py'. The editor shows the following Python code:

```
1 def intersect(num1,num2):
2     freq={}
3     for num in num1:
4         if num not in freq:
5             freq[num]=0
6         freq[num]+=1
7     intersection=[]
8     for num in num2:
9         if num in freq and freq[num]>0:
10            intersection.append(num)
11            freq[num]-=1
12    return intersection
13 num1=[1,2,2,1]
14 num2=[2,2]
15 inter=intersect(num1,num2)
16 print(inter)
```

The bottom of the window features a terminal pane with the following output:

```
PS C:\Users\User> python -u "d:\SSE\CSA0672 Design and Analysis of Algorithms\code\DAA.py"
[2, 2]
PS C:\Users\User>
```

The status bar at the bottom indicates 'Ln 16, Col 13', 'Spaces: 4', 'UTF-8', 'CRLF', 'Python', '3.12.0 64-bit', and 'Go Live'.

Time Complexity: $O(n*m)$

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.

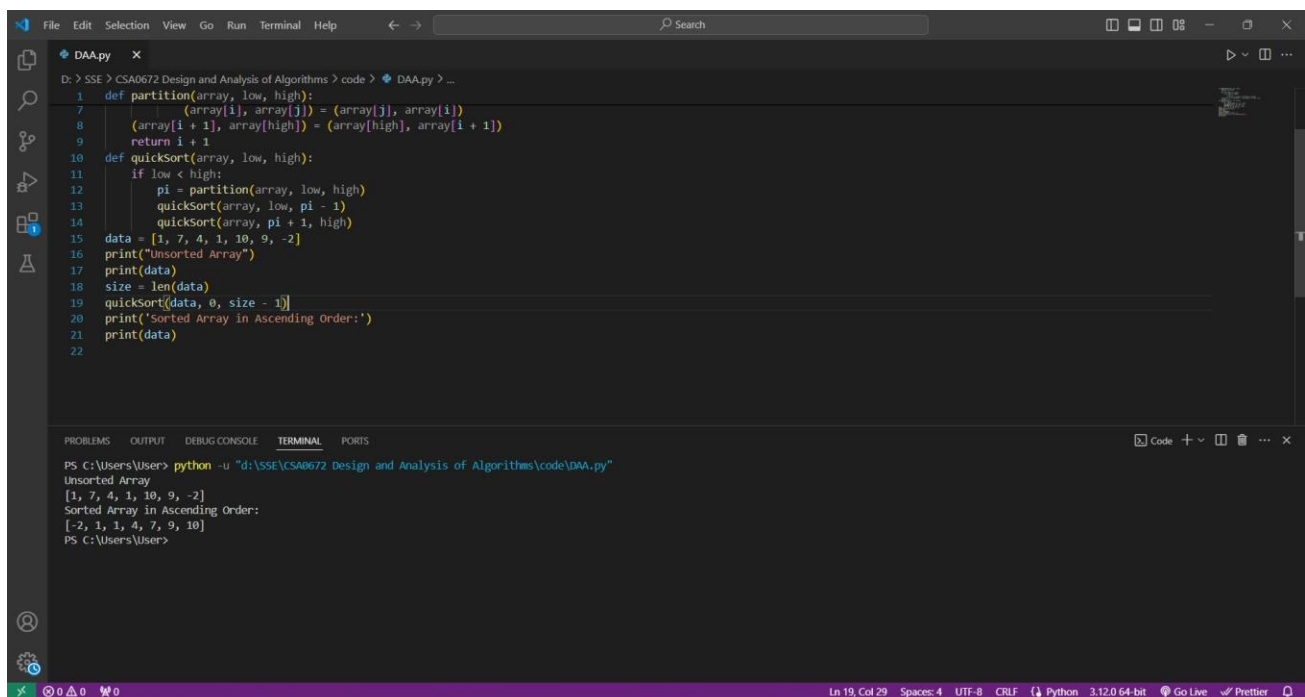
Code:

```
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)
```

Screenshot for I/O:



The screenshot displays a code editor with a Python script for Quick Sort. The script defines a `partition` function to rearrange elements around a pivot and a `quickSort` function to recursively sort the array. The main execution block initializes an array `data = [1, 7, 4, 1, 10, 9, -2]`, prints it as 'Unsorted Array', and then sorts it using `quickSort`. The final output is 'Sorted Array in Ascending Order:' followed by the sorted array `[-2, 1, 1, 4, 7, 9, 10]`.

```
DAA.py X
D:\SSE\CSA0672 Design and Analysis of Algorithms > code > DAA.py > ...
1 def partition(array, low, high):
2     pivot = array[high]
3     i = low - 1
4     for j in range(low, high):
5         if array[j] <= pivot:
6             i = i + 1
7             (array[i], array[j]) = (array[j], array[i])
8     (array[i + 1], array[high]) = (array[high], array[i + 1])
9     return i + 1
10 def quickSort(array, low, high):
11     if low < high:
12         pi = partition(array, low, high)
13         quickSort(array, low, pi - 1)
14         quickSort(array, pi + 1, high)
15 data = [1, 7, 4, 1, 10, 9, -2]
16 print("Unsorted Array")
17 print(data)
18 size = len(data)
19 quickSort(data, 0, size - 1)
20 print('Sorted Array in Ascending Order:')
21 print(data)
22
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\User> python -u "d:\SSE\CSA0672 Design and Analysis of Algorithms\code\DAA.py"
Unsorted Array
[1, 7, 4, 1, 10, 9, -2]
Sorted Array in Ascending Order:
[-2, 1, 1, 4, 7, 9, 10]
PS C:\Users\User>
```

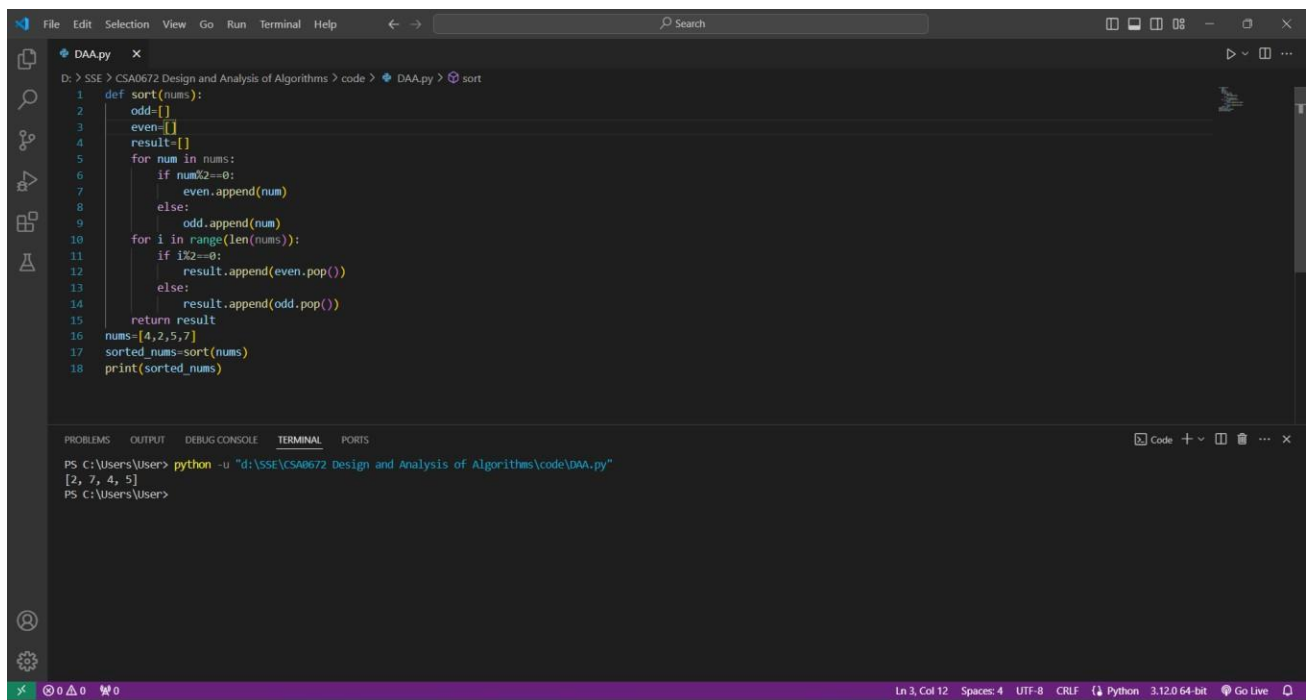
Ln 19, Col 29 Spaces: 4 UTF-8 CRLF Python 3.12.0 64-bit Go Live Prettier

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

Code:

```
def sort(nums):
    odd=[]
    even=[]
    result=[]
    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:
            result.append(even.pop())
        else:
            result.append(odd.pop())
    return result
nums=[4,2,5,7]
sorted_nums=sort(nums)
print(sorted_nums)
```

Screenshot for I/O:

A screenshot of a code editor window. The editor has a dark theme. The top part shows the code for the 'sort' function and its execution. The bottom part shows the output of the program in the terminal. The code is as follows:

```
1 def sort(nums):
2     odd=[]
3     even=[]
4     result=[]
5     for num in nums:
6         if num%2==0:
7             even.append(num)
8         else:
9             odd.append(num)
10    for i in range(len(nums)):
11        if i%2==0:
12            result.append(even.pop())
13        else:
14            result.append(odd.pop())
15    return result
16 nums=[4,2,5,7]
17 sorted_nums=sort(nums)
18 print(sorted_nums)
```

The terminal output shows:

```
PS C:\Users\User> python -u "d:\SSE\CSA0672 Design and Analysis of Algorithms\code\DAA.py"
[2, 7, 4, 5]
PS C:\Users\User>
```

Time Complexity: $O(n*m)$