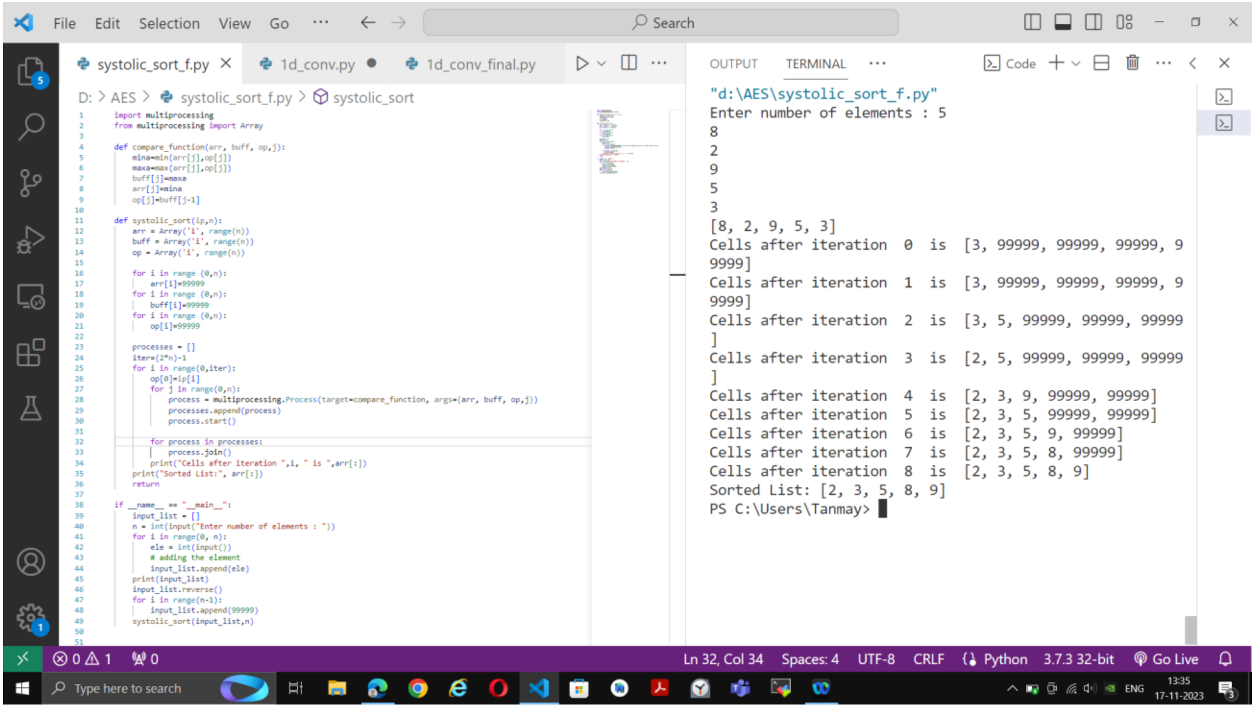
Simulating Systolic Arrays

1. Systolic Sort-

   Code:

```python
import multiprocessing
from multiprocessing import Array

def compare_function(arr, buff, op,j):
    mina=min(arr[j],op[j])
    maxa=max(arr[j],op[j])
    buff[j]=maxa
    arr[j]=mina
    op[j]=buff[j-1]

def systolic_sort(ip,n):
    arr = Array('i', range(n))
    buff = Array('i', range(n))
    op = Array('i', range(n))

    for i in range (0,n):
        arr[i]=99999
    for i in range (0,n):
        buff[i]=99999
    for i in range (0,n):
        op[i]=99999

    processes = []
    iter=(2*n)-1
    for i in range(0,iter):
        op[0]=ip[i]
        for j in range(0,n):
            process = multiprocessing.Process(target=compare_function, args=(arr, buff, op,j))
            processes.append(process)
            process.start()

        for process in processes:
            process.join()
        print("Cells after iteration ",i, " is ",arr[:])
    print("Sorted List:", arr[:])
    return

if __name__ == "__main__":
    input_list = []
    n = int(input("Enter number of elements : "))
    for i in range(0, n):
        ele = int(input())
        # adding the element
        input_list.append(ele)
    print(input_list)
    input_list.reverse()
    for i in range(n-1):
        input_list.append(99999)
    systolic_sort(input_list,n)
```

Output-



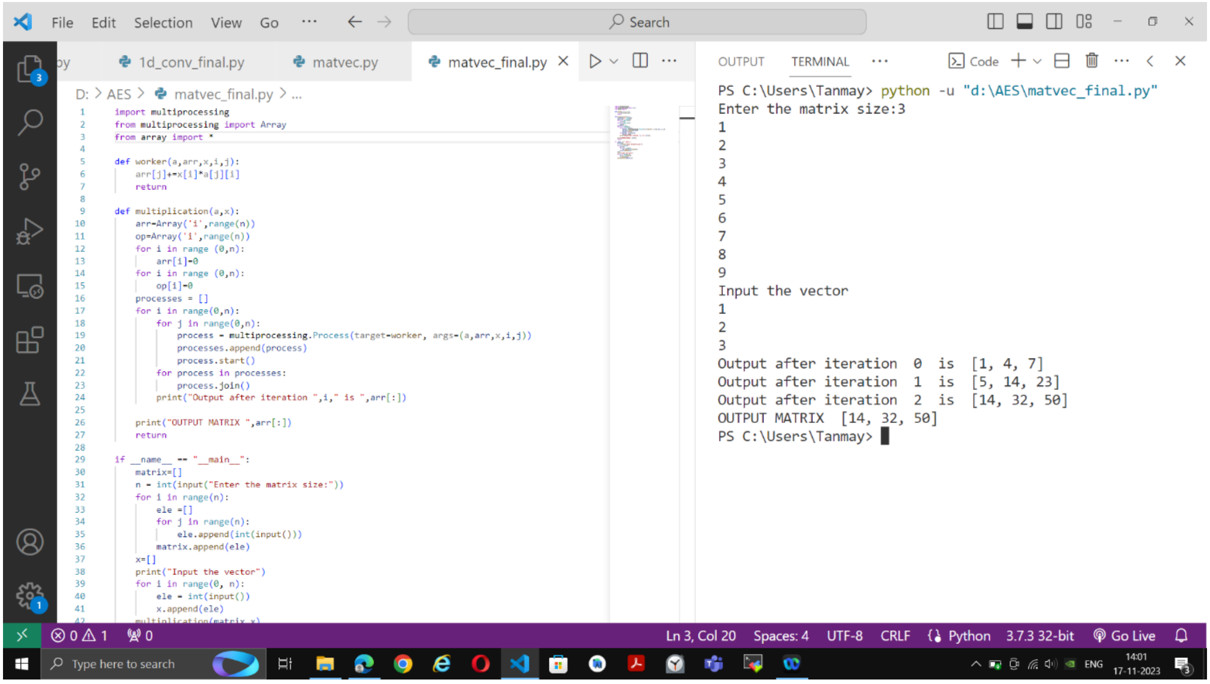
In this implementation of sorting using systolic arrays, we are using multiprocessing library of Python for parallel computation. We are keeping an array arr which gives us the output and two arrays buff, op.
buff stores the results in the current iteration and op holds the result of previous iteration. I've made two different arrays because we don't want results of both the iterations to be mixed.
In an iteration, a cell arr[i] takes an input from op[i] and compares with its present value. Greater of those values is passed to buff[i] and smaller one is stored in arr[i]. After an iteration, values of buff is passed to op.
Arrays are initialized with a large value so that they don't interfere with comparison and are forwarded further.

2. Matrix Vector Multiplication

Code-

```python
import multiprocessing
from multiprocessing import Array
from array import *

def worker(a,arr,x,i,j):
    arr[j]+=x[i]*a[j][i]
    return

def multiplication(a,x):
    arr=Array('i',range(n))
    op=Array('i',range(n))
    for i in range (0,n):
        arr[i]=0
    for i in range (0,n):
        op[i]=0
    processes = []
    for i in range(0,n):
        for j in range(0,n):
            process = multiprocessing.Process(target=worker, args=(a,arr,x,i,j))
            processes.append(process)
            process.start()
        for process in processes:
            process.join()
        print("Output after iteration ",i," is ",arr[:])

    print("OUTPUT MATRIX ",arr[:])
    return

if __name__ == "__main__":
    matrix=[]
    n = int(input("Enter the matrix size:"))
    for i in range(n):
        ele =[]
        for j in range(n):
            ele.append(int(input()))
        matrix.append(ele)
    x=[]
    print("Input the vector")
    for i in range(0, n):
        ele = int(input())
        x.append(ele)
    multiplication(matrix,x)
```

Output-



In this implementation of matrix-vector multiplication, we have calculated first terms of Yi.

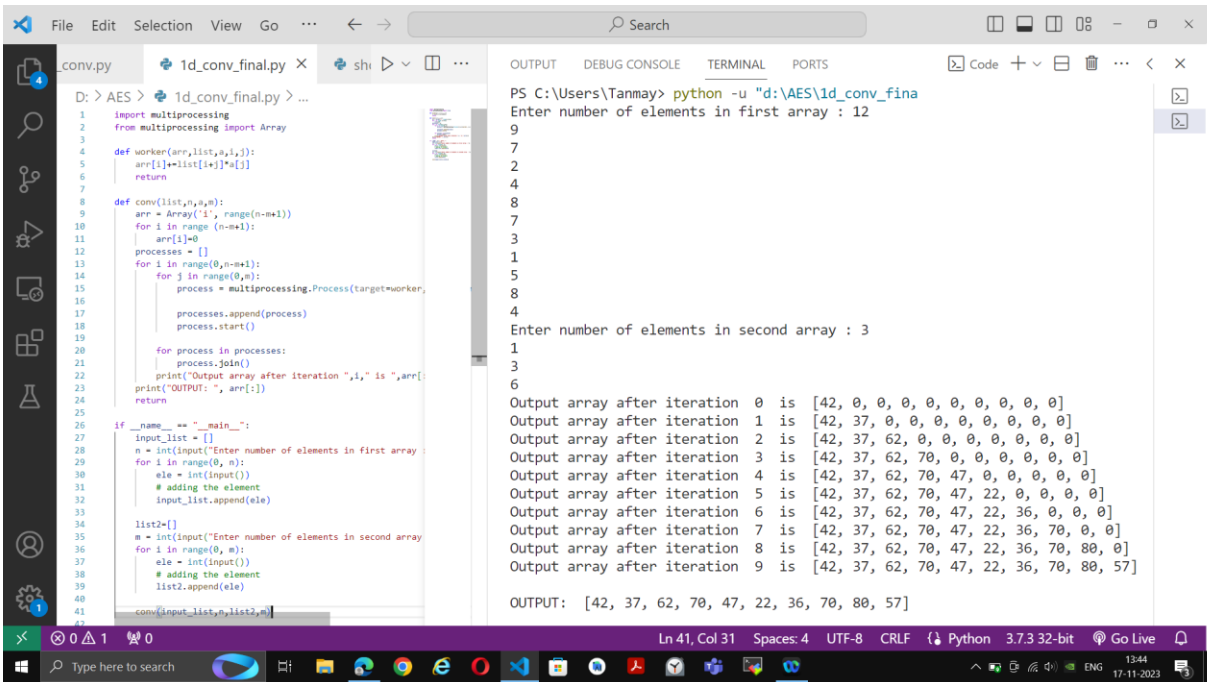
Let's say we have 4X4 matrix A and vector X.
Y1=A11X1 + A12X2 + A13X3 + A14X4
Y2=A21X1 + A22X2 + A23X3 + A24X4
.... and so on
We are computing terms Ai1X1(ith term) parallely for all Yi

3. 1-D Convolution

Code-

```python
import multiprocessing
from multiprocessing import Array

def worker(arr,list,a,i,j):
    arr[i]+=list[i+j]*a[j]
    return

def conv(list,n,a,m):
    arr = Array('i', range(n-m+1))
    for i in range (n-m+1):
        arr[i]=0
    processes = []
    for i in range(0,n-m+1):
        for j in range(0,m):
            process = multiprocessing.Process(target=worker, args=(arr,list,a,i,j))

            processes.append(process)
            process.start()

        for process in processes:
            process.join()
        print("Output array after iteration ",i," is ",arr[:])
    print("OUTPUT: ", arr[:])
    return

if __name__ == "__main__":
    input_list = []
    n = int(input("Enter number of elements in first array : "))
    for i in range(0, n):
        ele = int(input())
        # adding the element
        input_list.append(ele)

    list2=[]
    m = int(input("Enter number of elements in second array : "))
    for i in range(0, m):
        ele = int(input())
        # adding the element
        list2.append(ele)

    conv(input_list,n,list2,m)
```

Output-



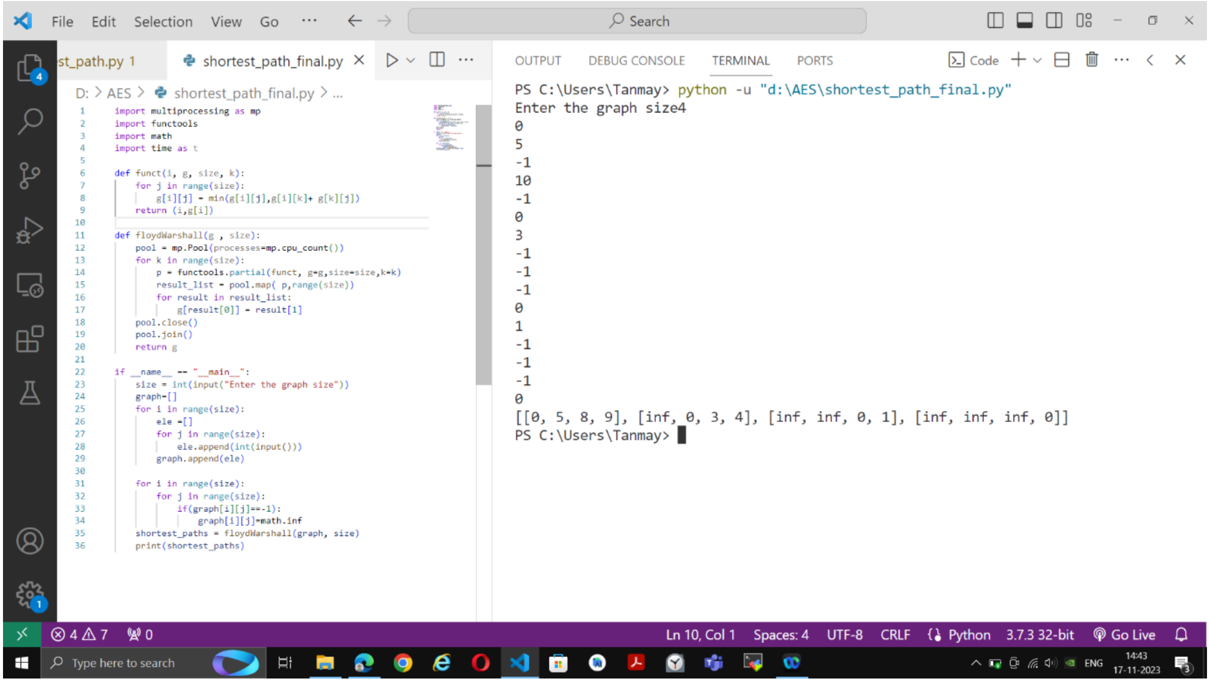
1D convolution is a mathematical operation that involves multiplying and adding an array with another array. The second array is called a kernel, filter, or weights. In one iteration, we are parallelly multiplying second array on a given index(es). We have used similar implementation of systolic arrays as previous programs.

4. Shortest Path

Code-

```
import multiprocessing as mp
import functools
import math
import time as t

def funct(i, g, size, k):
    for j in range(size):
        g[i][j] = min(g[i][j],g[i][k]+ g[k][j])
    return (i,g[i])

def floydWarshall(g , size):
    pool = mp.Pool(processes=mp.cpu_count())
    for k in range(size):
        p = functools.partial(funct, g=g,size=size,k=k)
        result_list = pool.map( p,range(size))
        for result in result_list:
            g[result[0]] = result[1]
    pool.close()
    pool.join()
    return g

if __name__ == "__main__":
    size = int(input("Enter the graph size"))
    graph=[]
    for i in range(size):
        ele =[]
        for j in range(size):
            ele.append(int(input()))
        graph.append(ele)

    for i in range(size):
        for j in range(size):
            if(graph[i][j]==-1):
                graph[i][j]=math.inf
    shortest_paths = floydWarshall(graph, size)
    print(shortest_paths)
```

Output-



This is a parallel implementation of Floyd Warshall algorithm. Its single processor runtime is O(n^3). Other implementations like All Pair Shortest path takes worse time. And it was not possible to parallelize Dijkstra's algorithm. In multiprocessing.Process, passing 2D array is not possible. Hence, I used multiprocessing.Pool. I was unaware of its implementation. Hence, used resources online for implementing Floyd Warshall and learning Pool multiprocessing.