(a) Basic I/O Programming file Operations

```
file1 = open('myfile.txt','w')
L = ["This is Delhi \n", "This is Paris \n", "This is London \n"]

file1.write("Hello \n")
file1.writelines(L)
file1.close()

file1 = open("myfile.txt","r+")
print("Output of Read function is ")
print(file1.read())
print()

file1.seek(0)

print("Output of ReadLine function is ")
print(file1.readline())
print()

file1.seek(0)

print("Output of Read(9) function is ")
print(file1.read(9))
print()

file1.seek(0)

print("Output of Readline(9) function is ")
print(file1.readline(9))

file1.seek(0)

# readlines function
print("Output of Readlines function is ")
```

```
print(file1.readlines())
print()
file1.close()
```

OUTPUT:

Output of Read function is

Hello

This is Delhi

This is Paris

This is London

Output of ReadLine function is

Hello

Output of Read(9) function is

Hello

Th

Output of Readline(9) function is

Hello

Output of Readlines function is

['Hello \n', 'This is Delhi \n', 'This is Paris \n', 'This is London \n']

2. Shortest Job First Algorithm

```
# Shortest Job First Non- Preemptive
print("Shortest Job First (Non- Preemptive) programming!".center(105, "~"),"\n")

# process to list
P = ['p1','p2','p3','p4']
p = P.copy()

# set a AT to list
AT = [0,1,2,3]
at = AT.copy()

# set a BT to list
BT = [8,4,9,5]
bt = BT.copy()

GC = [] # Create a Gantt chart
for i in range(len(P)):
    miv = bt.index(min(bt)) # min index value for bt !
    if i == 0:
        miv = at.index(min(at))
        GC.append([at[miv], p[miv], bt[miv]])
    else:
        GC.append([at[miv], p[miv], GC[i - 1][2] + bt[miv]])
    at.pop(miv);p.pop(miv);bt.pop(miv)
#print(GC)
CT = [i for i in range(1,6)]
TAT = [i for i in range(1,6)]
WT = [i for i in range(1,6)]

for i in range(len(P)):
    index = P.index(GC[i][1])
    CT[index] = GC[i][2]
    TAT[index] = CT[index] - AT[index]
```

```
    WT[index] = TAT[index] - BT[index]


print("*" * 105)
print("process : Arrival Time : Burst Time : Completion Time : Turn Around Time : Waiting Time ")
for i in range(len(P)):
    print(P[i]," " * 4,":",AT[i]," " * 10,":",BT[i]," " * 8,":",CT[i]," " * 14,":",TAT[i],
          " " * 14,":",WT[i])
print("*" * 105)
print("Average time of Turn Around time :", sum(TAT) / len(P))
print("Average time of Waiting time :", sum(WT) / len(P))
```

OUTPUT:

```
~~~~~~~~~~~~~~~~~~Shortest Job First (Non-Preemptive) programming!~~~~~~~~~~~~~~~~

**********************************************************************************
process : Arrival Time : Burst Time : Completion Time : Turn Around Time : Waiting Time
p1       : 0            : 8          : 8               : 8                : 0
p2       : 1            : 4          : 12              : 11               : 7
p3       : 2            : 9          : 26              : 24               : 15
p4       : 3            : 5          : 17              : 14               : 9
**********************************************************************************
Average time of Turn Around time : 14.25
Average time of Waiting time : 7.75
```

3. First Come First Served Algorithm.

```
print("FIRST COME FIRST SERVE SCHEDULLING")
n= int(input("Enter number of processes : "))
d = dict()

for i in range(n):
    key = "P"+str(i+1)
    a = int(input("Enter arrival time of process"+str(i+1)+": "))
    b = int(input("Enter burst time of process"+str(i+1)+": "))
    l = []
    l.append(a)
    l.append(b)
    d[key] = l

d = sorted(d.items(), key=lambda item: item[1][0])

ET = []
for i in range(len(d)):
    # first process
    if(i==0):
        ET.append(d[i][1][1])
    elif (d[i][1][0] > d[i-1][1][1]):
        ET.append(d[i][1][0] + d[i][1][1])
    # get prevET + newBT
    else:
        ET.append(ET[i-1] + d[i][1][1])

TAT = []
for i in range(len(d)):
    TAT.append(ET[i] - d[i][1][0])

WT = []
for i in range(len(d)):
    WT.append(TAT[i] - d[i][1][1])
```

```
avg_WT = 0
for i in WT:
    avg_WT +=i
avg_WT = (avg_WT/n)

print("Process | Arrival | Burst | Exit | Turn Around | Wait |")
for i in range(n):
      print("   ",d[i][0],"   |   ",d[i][1][0]," |    ",d[i][1][1]," |    ",ET[i],"  |    ",TAT[i],"  |   ",WT[i],"   |  ")
print("Average Waiting Time: ",avg_WT)
```

OUTPUT:

~~~~~~~~First Come First Served Algorithm~~~~~~~~~

processes	Burst time	Waiting time	Turn around time
1	24	0	24
2	3	24	27
3	3	27	30

Average Waiting time = 17.0

Average turn around time = 27.0
~~~~~~~~

4. Round Robin CPU Scheduling Algorithm.

```
def findWaitingTime(processes, n, bt, wt, quantum):
    rem_bt = [0] * n

    # Copy the burst time into rt[]
    for i in range(n):
        rem_bt[i] = bt[i]
    t = 0 # Current time

    # Keep traversing processes in round
    # robin manner until all of them are
    # not done.
    while(1):
        done = True

        # Traverse all processes one by
        # one repeatedly
        for i in range(n):

            # If burst time of a process is greater
            # than 0 then only need to process further
            if (rem_bt[i] > 0) :
                done = False # There is a pending process

                if (rem_bt[i] > quantum) :

                    # Increase the value of t i.e. shows
                    # how much time a process has been processed
                    t += quantum

                    # Decrease the burst_time of current
                    # process by quantum
                    rem_bt[i] -= quantum
```

```
                # If burst time is smaller than or equal
                # to quantum. Last cycle for this process
                else:

                    # Increase the value of t i.e. shows
                    # how much time a process has been processed
                    t = t + rem_bt[i]

                    # Waiting time is current time minus
                    # time used by this process
                    wt[i] = t - bt[i]

                    # As the process gets fully executed
                    # make its remaining burst time = 0
                    rem_bt[i] = 0

        # If all processes are done
        if (done == True):
            break

# Function to calculate turn around time
def findTurnAroundTime(processes, n, bt, wt, tat):

    # Calculating turnaround time
    for i in range(n):
        tat[i] = bt[i] + wt[i]

# Function to calculate average waiting
# and turn-around times.
def findavgTime(processes, n, bt, quantum):
    wt = [0] * n
    tat = [0] * n

    # Function to find waiting time
    # of all processes
```

```
    findWaitingTime(processes, n, bt, wt, quantum)

    # Function to find turn around time
    # for all processes
    findTurnAroundTime(processes, n, bt, wt, tat)

    # Display processes along with all details
    print("Processes Burst Time    Waiting","Time Turn-Around Time")
    total_wt = 0
    total_tat = 0
    for i in range(n):
        total_wt = total_wt + wt[i]
        total_tat = total_tat + tat[i]
        print(" ", i + 1, "\t\t", bt[i],"\t\t", wt[i], "\t\t", tat[i])

    print("\nAverage waiting time = %.5f "%(total_wt /n) )
    print("Average turn around time = %.5f "% (total_tat / n))

# Driver code
if __name__ =="__main__":

    # Process id's
    proc = [1, 2, 3]
    n = 3

    # Burst time of all processes
    burst_time = [24, 3, 3]

    # Time quantum
    quantum = 4;
    findavgTime(proc, n, burst_time, quantum)
```

OUTPUT:

---------Round robin Scheduling Algorithm---------

Processes	Burst Time	Waiting Time	Turn-Around Time
1	24	6	30
2	3	4	7
3	3	7	10

Average waiting time = 5.66667
Average turn around time = 15.66667

.

5.Priority Scheduling

```
def findWaitingTime(processes, n, wt):
    wt[0] = 0
    for i in range(1, n):
        wt[i] = processes[i - 1][1] + wt[i - 1]

def findTurnAroundTime(processes, n, wt, tat):
    for i in range(n):
        tat[i] = processes[i][1] + wt[i]

def findavgTime(processes, n):
    wt = [0] * n
    tat = [0] * n

    findWaitingTime(processes,n,wt)
    findTurnAroundTime(processes, n, wt, tat)
    print("\nprocesses \tBurst time \tWaiting Time \tTurn-Around Time")
    total_wt = 0
    total_tat = 0

    for i in range(n):
        total_tat += wt[i]
        total_wt += tat[i]
        print(" ", processes[i][0], "\t\t",
            processes[i][1], "\t\t", wt[i],"\t\t",tat[i])



    print("Average turn around time = " + str(total_tat/n))
    print("Average Waiting time = " + str(total_wt /n))

def priorityScheduling(proc, n):
    proc = sorted(proc, key = lambda proc:proc[2], reverse = False)
    print("Order in which processes gets executed")
```

```
    for i in proc:
        print(i[0], end = " ")
    findavgTime(proc, n)

if __name__ == "__main__":

    # Process id's
    proc = [
        [1, 10, 3],
        [2, 1, 1],
        [3, 2, 3],
        [4, 1, 4],
        [5, 5, 2]
        ]
    n = 5
    priorityScheduling(proc, n)
```

OUTPUT:

Order in which processes gets executed
2 5 1 3 4

Processes	Burst Time	Waiting Time	Turn-Around Time
2	1	0	1
5	5	1	6
1	10	6	16
3	2	16	18
4	1	18	19

Average waiting time = 8.20000
Average turn around time = 12.0

6. To implement reader/writer problem using semaphore.

```
# implement  reader write problem using semaphore

import threading
import time

class ReaderWriterProblem():
    def __init__(self):
        self.mutex = threading.Semaphore()
        self.wrt = threading.Semaphore()
        self.r_c = 0

    def reader(self):
        while True:
            self.mutex.acquire()
            self.r_c += 1

            if self.r_c == 1:
                self.wrt.acquire()

            self.mutex.release()
            print(f"\nReader {self.r_c} is reading")

            self.mutex.acquire()
            self.r_c -=1

            if self.r_c == 0:
                self.wrt.release()

            self.mutex.release()
            time.sleep(3)

    def writer(self):
        while True:
```

```
            self.wrt.acquire()
            print("Writing data .......")
            print("-"* 20)
            self.wrt.release()
            time.sleep(3)

    def main(self):
        t1 = threading.Thread(target = self.reader)
        t1.start()

        t2 = threading.Thread(target = self.writer)
        t2.start()

        t3 = threading.Thread(target = self.reader)
        t3.start()

        t4 = threading.Thread(target = self.reader)
        t4.start()

        t6 = threading.Thread(target = self.writer)
        t6.start()

        t5 = threading.Thread(target = self.reader)
        t5.start()

if __name__ == "__main__":
    c = ReaderWriterProblem()
    c.main()
```

OUTPUT:

```
Reader 1 is reading
Writing data .......
--------------------
Writing data .......
--------------------

Reader 1 is reading

Reader 1 is reading

Reader 1 is reading
Reader 2 is reading

Reader 2 is reading

Reader 2 is reading

Writing data .......
--------------------
Writing data .......
--------------------

Reader 1 is reading
Reader 2 is reading
Reader 3 is reading


Writing data .......
--------------------
Writing data .......
--------------------
```

7. To implement Banker's algorithm for Deadlock avoidance.

Program for page replacement algorithms:

```
# implement Banker's algorithm for Deadlock avoidance
P = 5
R = 3
def calculateNeed(need, maxm, allot):
    for i in range(P):
        for j in range(R):
            #print(need)
            #print("i:",i, "j:",j)
            need[i][j] = maxm[i][j] - allot[i][j]

def isSafe(processes,avail, maxm, allot):
    need = []
    for i in range(P):
        l = []
        for j in range(R):
            l.append(0)
        need.append(l)

    calculateNeed(need, maxm, allot)
    finish = [0] * P
    safeSeq = [0] * P

    work = [0] * R
    for i in range(R):
        work[i] = avail[i]
    count = 0
    while(count < P):
        found = False
        for p in range(P):
            if (finish[p] == 0):
                for j in range(R):
                    if (need[p][j] > work[j]):
```

```
                break
            if (j == R - 1):
                for k in range(R):
                    work[k] += allot[p][k]
                safeSeq[count] = p
                count += 1
                finish[p] = 1
                found = True

        if (found == False):
            print("System i not in safe state")
            return False

    print("System is in safe state.\nsafe sequence is: ", end = " ")
    print(*safeSeq)
    return True

if __name__ == "__main__":
    processes = [0,1,2,3,4]

    avail = [3,3,2]

    maxm = [ [7,5,3],
            [3,2,2],
            [9,0,2],
            [2,2,2],
            [4,3,3]
        ]
      allot = [[0,1,0],
            [2,0,0],
            [3,0,2],
            [2,1,1],
            [0,0,2]]

    isSafe(processes, avail, maxm, allot)
```

OUTPUT:

System is in safe state.

safe sequence is: 1 3 4 0 2

8. First In First Out Algorithm.

```
# implement of FIFO page replacement in Operating system.

from queue import Queue
# Fucntion to find page faults using FIFO
def pageFaults(pages, n, capacity):
    s = set()
    indexes = Queue()
    page_faults = 0
    for i in range(n):
        if (len(s) < capacity):


            if (pages[i] not in s):
                s.add(pages[i])
                page_faults += 1
                indexes.put(pages[i])
        else:
            if (pages[i] not in s):
                val =  indexes.queue[0]
                indexes.get()
                s.remove(val)
                s.add(pages[i])
                indexes.put(pages[i])
                page_faults += 1
        print(s,end = " ")
        print("Page Fault Count", page_faults)
    return page_faults

if __name__ == "__main__":
    pages = [3,2,1,0,3,2,4,3,2,1,0,4]
    n = len(pages)
    capacity = 3
    print("Total page fault count",pageFaults(pages, n, capacity))
```

OUTPUT:

```
{3} Page Fault Count 1
{2, 3} Page Fault Count 2
{1, 2, 3} Page Fault Count 3
{0, 1, 2} Page Fault Count 4
{0, 1, 3} Page Fault Count 5
{0, 2, 3} Page Fault Count 6
{2, 3, 4} Page Fault Count 7
{2, 3, 4} Page Fault Count 7
{2, 3, 4} Page Fault Count 7
{1, 2, 4} Page Fault Count 8
{0, 1, 4} Page Fault Count 9
{0, 1, 4} Page Fault Count 9
Total page fault count 9
```

9. Least Recently Used Algorithm.

```
# python3 program for LRU page replacement algorithm
size = 3
reference_string = [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 1, 2, 0]

# creating a list to store the current pages in memory
pages = []

# page faults
faults = 0

# page hits
hits = 0

# iterating the reference string
for ref_page in reference_string:

  # if a ref_page already exists in pages list, remove it and append it at the end
  if ref_page in pages:
    pages.remove(ref_page)

    pages.append(ref_page)

    # incrementing the page hits
    hits += 1

  # if ref_page is not in the pages list
  else:
    # incrementing the page faults
    faults +=1

    # check length of the pages list. If length of pages list
    # is less than memory size, append ref_page into pages list
    if(len(pages) < size):
```

```
        pages.append(ref_page)

    # if length of pages list is greater than or equal to memory size,
    # remove first page of pages list and append new page to pages
    else:
        pages.remove(pages[0])

        pages.append(ref_page)

# printing the number of page hits and page faults
print("Total number of Page Hits:", hits)
print("Total number of Page Faults:", faults)
```

OUTPUT:

Total number of Page Hits: 3
Total number of Page Faults: 12

10. To implement first fit, best fit and worst fit algorithm for memory management.

```
# python3 program for first fit memory management algorithm
# 9th a
def FirstFit(block_size, blocks, process_size, processes):
    allocate = [-1] * processes
    occupied = [False] * blocks

    for i in range(processes):
        for j in range(blocks):
            if not occupied[j] and (block_size[j] >= process_size[i]):

                allocate[i] = j
                occupied[j] = True
                break
    print("Process No.\t process size \t\t Block no.")

    for i in range(processes):
        print(i+1,"\t\t\t",process_size[i], "\t\t\t", end = "   ")

        if allocate[i] != -1:
            print(allocate[i] + 1)
        else:
            print("Not Allocated")

block_size = [20,100,40,200,10]
process_size = [90,50,30,40]
m = len(block_size)
n = len(process_size)

FirstFit(block_size, m, process_size, n)
```

OUTPUT:

Process No.	**process size**	**Block no.**
1	90	2
2	50	4
3	30	3
4	40	Not Allocated

11. simulate the Best- Fit contiguous memory allocation technique.

```
# Python3 implementation of Best - Fit algorithm
# Function to allocate memory to blocks
# as per Best fit algorithm
def bestFit(blockSize, m, processSize, n):

    # Stores block id of the block
    # allocated to a process
    allocation = [-1] * n

    # pick each process and find suitable
    # blocks according to its size ad
    # assign to it
    for i in range(n):

        # Find the best fit block for
        # current process
        bestIdx = -1
        for j in range(m):
            if blockSize[j] >= processSize[i]:
                if bestIdx == -1:
                    bestIdx = j
                elif blockSize[bestIdx] > blockSize[j]:
                    bestIdx = j

        # If we could find a block for
        # current process
        if bestIdx != -1:

            # allocate block j to p[i] process
            allocation[i] = bestIdx

            # Reduce available memory in this block.
            blockSize[bestIdx] -= processSize[i]
```

```
    print("Process No. Process Size      Block no.")
    for i in range(n):
        print(i + 1, "          ", processSize[i],
                                    end = "          ")
        if allocation[i] != -1:
            print(allocation[i] + 1)
        else:
            print("Not Allocated")

# Driver code
if __name__ == '__main__':
    blockSize = [100, 500, 200, 300, 600]
    processSize = [212, 417, 112, 426]
    m = len(blockSize)
    n = len(processSize)

    bestFit(blockSize, m, processSize, n)
```

OUTPUT:

Process No.	Process Size	Block no.
1	212	4
2	417	2
3	112	3
4	426	5

12. implement worst fit algorithm for memory management

```
def worstFit(blockSize, m, processSize, n):

    # Stores block id of the block
    # allocated to a process

    # Initially no block is assigned
    # to any process
    allocation = [-1] * n

    # pick each process and find suitable blocks
    # according to its size ad assign to it
    for i in range(n):

        # Find the best fit block for
        # current process
        wstIdx = -1
        for j in range(m):
            if blockSize[j] >= processSize[i]:
                if wstIdx == -1:
                    wstIdx = j
                elif blockSize[wstIdx] < blockSize[j]:
                    wstIdx = j

        # If we could find a block for
        # current process
        if wstIdx != -1:

            # allocate block j to p[i] process
            allocation[i] = wstIdx

            # Reduce available memory in this block.
            blockSize[wstIdx] -= processSize[i]
```

```
    print("Process No. Process Size Block no.")
    for i in range(n):
        print(i + 1, "                ",
            processSize[i], end = "            ")
        if allocation[i] != -1:
            print(allocation[i] + 1)
        else:
            print("Not Allocated")

# Driver code
if __name__ == '__main__':
    blockSize = [100, 500, 200, 300, 600]
    processSize = [212, 417, 112, 426]
    m = len(blockSize)
    n = len(processSize)

    worstFit(blockSize, m, processSize, n)
```

OUTPUT:

Worst Fit Algorithm

Process No.	Process Size	Block no.
1	212	5
2	417	2
3	112	5
4	426	Not Allocated

13. Program for Inter-process Communication.

```
# python3 program for multiprogramming shared memory
import multiprocessing
def sender(conn, msgs):
	"""
	function to send messages to other end of pipe
	"""
	for msg in msgs:
		conn.send(msg)
		print("Sent the message: {}".format(msg))
	conn.close()

def receiver(conn):
	"""
	function to print the messages received from other
	end of pipe
	"""
	while 1:
		msg = conn.recv()
		if msg == "END":
			break
		print("Received the message: {}".format(msg))

if __name__ == "__main__":
	# messages to be sent
	msgs = ["hello", "hey", "hru?", "END"]

	# creating a pipe
	parent_conn, child_conn = multiprocessing.Pipe()

	# creating new processes
	p1 = multiprocessing.Process(target=sender, args=(parent_conn,msgs))
	p2 = multiprocessing.Process(target=receiver, args=(child_conn,))
```

```
# running processes
p1.start()
p2.start()

# wait until processes finish
p1.join()
p2.join()
```

OUTPUT:

Sent the message: hello
Sent the message: hey
Sent the message: hru?
Sent the message: END
Received the message: hello
Received the message: hey
Received the message: hru?