Expt no:1(a)

Aim: To simulate the FCFS CPU scheduling algorithm

Description:

First Come First Serve CPU Scheduling Algorithm shortly known as FCFS is the first algorithm of CPU Process Scheduling Algorithm. In First Come First Serve Algorithm what we do is to allow the process to execute in linear manner.

This means that whichever process enters process enters the ready queue first is executed first. This shows that First Come First Serve Algorithm follows First In First Out (FIFO) principle.

The First Come First Serve Algorithm can be executed in Pre Emptive and Non Pre Emptive manner.

Program:

```
def FCFS(p):
    ct=[0]*len(p)
    tat=[0]*len(p)
    wt=[0]*len(p)
    print(f"pid\t Arrival time\t Burst time\t Completion time\t TAT \t waiting time")
    for i in range(len(p)):
        ct[i]=ct[i-1]+p[i][2]
        tat[i]=ct[i-1]-p[i][1]+p[i][2]
        wt[i]=tat[i]-p[i][2]
        print(f"{p[i][0]}\t{p[i][1]}\t\t{p[i][2]}\t\t{ct[i]}\t\t\t{tat[i]}\t\t{wt[i]}\t")
    print(f"avg TAT: {sum(tat)/len(p)}")
    print(f"avg WT: {sum(wt)/len(p)}")
n=int(input("enter no. of processes:"))
p=[0]*n
for i in range(n):
    p[i]=tuple(map(int,input("enter pid,arrival time,burst time:").split()))
```

FCFS(p)

Output:

enter no. of processes:4

enter pid,arrival time,burst time:2 0 20

enter pid,arrival time,burst time:6 2 40

enter pid,arrival time,burst time:57 4 60

enter pid,arrival time,burst time:15 8 80

pid	Arrival time	Burst time	Completion time	TAT	waiting time
2	0	20	20	20	0
6	2	40	60	58	18
57	4	60	120	116	56
15	8	80	200	192	112

avg TAT: 96.5

avg WT: 46.5

Expt no: 1(b)

Aim: To simulate the SJF(non preemptive) CPU scheduling algorithm

Program:

```
n=int(input("enter no. of processes"))

print("enter the process id and burst time:/n")

l=[]

for i in range(n):

    l.append(list(map(int,input().split())))

for i in range(n):

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

```
        if l[i][1]>l[j][1]:

            l[i],l[j]=l[j],l[i]
ct=0
for i in range(n):
    ct+=l[i][1]
    l[i].append(ct)
print("pid bt tat wt")
tat,wt=0,0
for i in range(n):
    print(l[i][0]," ",l[i][1]," ",l[i][2]," ",l[i][2]-l[i][1])
    tat+=l[i][2]
    wt+=l[i][2]-l[i][1]
print("avg tat:",tat/n)
print("avg wt:",wt/n)
```

Output:

```
enter no. of processes4
enter the process id and burst time:/n
1  8
2  5
3  3
4  6

pid bt tat wt
3   3   3   0
```

2   5   8   3

4   6   14   8

1   8   22   14

avg tat: 11.75

avg wt: 6.25

Expt no: 1(b)

Aim: To simulate SJF preemptive scheduling algorithm

Program:

```
def findWaitingTime(processes, n, wt):

    rt = [0] * n

    for i in range(n):

        rt[i] = processes[i][1]

    complete = 0

    t = 0

    minm =999999999

    short = 0

    check = False

    while (complete != n):

        for j in range(n):

            if ((processes[j][2] <= t) and

                (rt[j] < minm) and rt[j] > 0):

                minm = rt[j]

                short = j

                check = True
```

```
        if (check == False):
            t += 1
            continue
        rt[short] -= 1
        minm = rt[short]
        if (minm == 0):
            minm = 999999999
        if (rt[short] == 0):
            complete += 1
            check = False
            fint = t + 1
            wt[short] = (fint - proc[short][1] -proc[short][2])
            if (wt[short] < 0):
                wt[short] = 0

        t += 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("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(" ", processes[i][0], "\t\t",
                processes[i][1], "\t\t",
                wt[i], "\t\t", tat[i])

    print("\nAverage waiting time = %.5f "%(total_wt /n) )
    print("Average turn around time = ", total_tat / n)
n = int(input("enter no of process:"))
proc=[]
print("enter process id,burst time,arrivaltime")
for i in range(n):
    proc.append(list(map(int,input().split())))
findavgTime(proc, n)
```

Output:

```
enter no of process:4
enter process id,burst time,arrivaltime
1  8  0
2  5  1
3  3  2
4  6  4
```

Processes	Burst Time	Waiting Time	Turn-Around Time
1	8	14	22
2	5	3	8
3	3	0	3
4	6	5	11

Average waiting time = 5.50000

Average turn around time = 11.0

Expt no: 1(c)

Aim: To simulate priority scheduling algorithm (non preemptive)

Program:

```
n=int(input("enter no of processes:"))
print("enter processid bt and priority:")
l=[]
for i in range(n):
    l.append(list(map(int,input().split())))
for i in range(n):
    for j in range(i+1,n):
        if l[i][2]>l[j][2]:
            l[i],l[j]=l[j],l[i]
ct=0
for i in range(n):
    ct+=l[i][1]
    l[i].append(ct)
```

```
tat,wt=0,0
print("pid bt p ct tat wt")
for i in range(n):
    print(l[i][0]," ",l[i][1]," ",l[i][2]," ",l[i][3]," ",l[i][3]," ",l[i][3]-l[i][1])
ttat=0
twt=0
for i in range(n):
    ttat+=l[i][3]
    twt+=l[i][3]-l[i][1]
print("average tat:",ttat/n)
print("average wt:",twt/n)
```

Output:

```
enter no of processes:5
enter processid bt and priority:
1 10 3
2 1 1
3 2 4
4 1 5
5 5 2
pid bt p ct tat wt
2   1   1   1   1   0
5   5   2   6   6   1
1   10   3   16   16   6
3   2   4   18   18   16
4   1   5   19   19   18
```

average tat: 12.0

average wt: 8.2

Expt no: 1(c)

Aim: To simulate priority scheduling preemptive algorithm

Program:

```
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 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(" ", processes[i][0], "\t\t",processes[i][1], "\t\t",wt[i], "\t\t", tat[i])
```

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

def priorityScheduling(proc, n):
    proc = sorted(proc, key = lambda proc:proc[2],reverse = True);
    findavgTime(proc, n)
proc = []
n = int(input("enter no. of processes:"))
print("enter process id,burst time,priority:")
for i in range(n):
    proc.append(list(map(int,input().split())))
priorityScheduling(proc, n)
```

Output:

```
enter no. of processes:5
enter process id,burst time,priority:
1 10 3
2 20 2
3 30 5
4 40 1
5 50 4

Processes Burst Time Waiting Time Turn-Around Time
 3          30          0           30
 5          50          30          80
```

1	10	80	90
2	20	90	110
4	40	110	150

Average waiting time = 62.00000

Average turn around time = 92.0

Expt no: 1(d)

Aim: To simulate round robin scheduling algorithm

Program:

```
def findWaitingTime(processes, n, bt,wt, quantum):
    rem_bt = [0] * n
    for i in range(n):
        rem_bt[i] = bt[i]
    t = 0
    while(1):
        done = True
        for i in range(n):
            if (rem_bt[i] > 0) :
                done = False # There is a pending process
                if (rem_bt[i] > quantum):
                    t += quantum
                    rem_bt[i] -= quantum
                else:
                    t = t + rem_bt[i]
```

```
                wt[i] = t - bt[i]
                rem_bt[i] = 0
        if (done == True):
            break
def findTurnAroundTime(processes, n, bt, wt, tat):
    for i in range(n):
        tat[i] = bt[i] + wt[i]
def findavgTime(processes, n, bt, quantum):
    wt = [0] * n
    tat = [0] * n
    findWaitingTime(processes, n, wt, quantum)
    findTurnAroundTime(processes, n, bt,wt, tat)
    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))
proc = []
n = int(input("enter no. of processes:"))
burst_time = []
for i in range(n):
```

```
    p=int(input("enter process id:"))
    proc.append(p)
    bt=int(input("enter burst time:"))
    burst_time.append(bt)
quantum =int(input("enter time quantum:"))
findavgTime(proc, n, burst_time, quantum)
```

Output:

```
Enter no. of processes:4
Enter process id:1
Enter burst time:10
Enter process id:2
Enter burst time:20
Enter process id:3
Enter burst time:30
Enter process id:4
Enter burst time:40
Enter time quantum:4
Process   burst time     waiting time     turn around time
1          10             24               34
2          20             42               62
3          30             58               88
4          40             60               100
Average waiting time=46.0
Average turn around time=71.0
```

Expt no: 2(a)

Aim: To implement Bounded buffer problem (producer consumer)

Program:

```
import threading

import time

CAPACITY = 10

buffer = [-1 for i in range(CAPACITY)]

in_index = 0

out_index = 0

mutex = threading.Semaphore()

empty = threading.Semaphore(CAPACITY)

full = threading.Semaphore(0)

class Producer(threading.Thread):

  def run(self):

    global CAPACITY, buffer, in_index, out_index

    global mutex, empty, full

    items_produced = 0

    counter = 0

    while items_produced < 20:

      empty.acquire()

      mutex.acquire()

      counter += 1

      buffer[in_index] = counter

      in_index = (in_index + 1)%CAPACITY
```

```
      print("Producer produced : ", counter)
      mutex.release()
      full.release()
      time.sleep(1)
      items_produced += 1
class Consumer(threading.Thread):
  def run(self):
    global CAPACITY, buffer, in_index, out_index, counter
    global mutex, empty, full
    items_consumed = 0
    while items_consumed < 20:
      full.acquire()
      mutex.acquire()
      item = buffer[out_index]
      out_index = (out_index + 1)%CAPACITY
      print("Consumer consumed item : ", item)
      mutex.release()
      empty.release()
      time.sleep(2.5)
      items_consumed += 1
producer = Producer()
consumer = Consumer()
consumer.start()
producer.start()
producer.join()
```

```
consumer.join()
```

Output:

```
Producer produced :  1
Consumer consumed item :  1
Producer produced :  2
Producer produced :  3
Consumer consumed item :  2
Producer produced :  4
Producer produced :  5
Consumer consumed item :  3
Producer produced :  6
Producer produced :  7
Producer produced :  8
Consumer consumed item :  4
Producer produced :  9
Producer produced :  10
Consumer consumed item :  5
Producer produced :  11
Producer produced :  12
Producer produced :  13
Consumer consumed item :  6
Producer produced :  14
Producer produced :  15
Consumer consumed item :  7
Producer produced :  16
Producer produced :  17
Consumer consumed item :  8
Producer produced :  18
Consumer consumed item :  9
Producer produced :  19
Consumer consumed item :  10
Producer produced :  20
Consumer consumed item :  11
Consumer consumed item :  12
Consumer consumed item :  13
Consumer consumed item :  14
Consumer consumed item :  15
Consumer consumed item :  16
Consumer consumed item :  17
Consumer consumed item :  18
Consumer consumed item :  19
Consumer consumed item :  20
```

Expt no: 2(b)
Aim: To implement Reader-writer problem
Program:

```
import threading as thread
import random
```

```
global x               #Shared Data
x = 0
lock = thread.Lock()    #Lock for synchronising access

def Reader():
    global x
    print('Reader is Reading!')
    lock.acquire()      #Acquire the lock before Reading (mutex approach)
    print('Shared Data:', x)
    lock.release()      #Release the lock after Reading
    print()

def Writer():
    global x
    print('Writer is Writing!')
    lock.acquire()      #Acquire the lock before Writing
    x += 1              #Write on the shared memory
    print('Writer is Releasing the lock!')
    lock.release()      #Release the lock after Writing
    print()

if __name__ == '__main__':
    for i in range(0, 10):
        randomNumber = random.randint(0, 100)   #Generate a Random number between 0 to
100
        if(randomNumber > 50):
            Thread1 = thread.Thread(target = Reader)
            Thread1.start()
        else:
            Thread2 = thread.Thread(target = Writer)
            Thread2.start()

Thread1.join()
Thread2.join()
Output:
```

Writer is Writing!

Writer is Releasing the lock!

Reader is Reading!

Shared Data:Writer is Writing!

Writer is Writing!

Writer is Writing!

1

Writer is Releasing the lock!

Writer is Releasing the lock!

Writer is Releasing the lock!Writer is Writing!

Writer is Releasing the lock!

Reader is Reading!

Shared Data: 5

Writer is Writing!

Writer is Releasing the lock!

Writer is Writing!

Writer is Releasing the lock!

Writer is Writing!

Writer is Releasing the lock!

Expt no: 2(c)

Aim: To implement dining philosophers problem using semaphores

Program:

```
import threading

import random

import time

class Philosopher(threading.Thread):

    running = True

    def __init__(self, index, forkOnLeft, forkOnRight):

        threading.Thread.__init__(self)

        self.index = index

        self.forkOnLeft = forkOnLeft

        self.forkOnRight = forkOnRight

    def run(self):

        while(self.running):

            time.sleep(30)

            print ('Philosopher %s is hungry.' % self.index)

            self.dine()

    def dine(self):

        fork1, fork2 = self.forkOnLeft, self.forkOnRight

        while self.running:

            fork1.acquire() # wait operation on left fork

            locked = fork2.acquire(False)

            if locked: break #if right fork is not available leave left fork

            fork1.release()

            print ('Philosopher %s swaps forks.' % self.index)

            fork1, fork2 = fork2, fork1
```

```
        else:
            return
        self.dining()
        fork2.release()
        fork1.release()
    def dining(self):
        print ('Philosopher %s starts eating. '% self.index)
        time.sleep(30)
        print ('Philosopher %s finishes eating and leaves to think.' % self.index)
def main():
    forks = [threading.Semaphore() for n in range(5)]
    philosophers= [Philosopher(i, forks[i%5], forks[(i+1)%5])
            for i in range(5)]
    Philosopher.running = True
    for p in philosophers: p.start()
    time.sleep(100)
    Philosopher.running = False
    print ("Now we're finishing.")
if __name__ == "__main__":
    main()
```

Output:

Philosopher 4 is hungry.

Philosopher 4 starts eating.

Philosopher 3 is hungry.

Philosopher 3 swaps forks.

Philosopher 2 is hungry.

Philosopher 2 starts eating.

Philosopher 1 is hungry.

Philosopher 1 swaps forks.

Philosopher 0 is hungry.

Philosopher 4 finishes eating and leaves to think.

Philosopher 0 starts eating.

Philosopher 3 swaps forks.

Philosopher 2 finishes eating and leaves to think.

Philosopher 3 starts eating.

Philosopher 1 swaps forks.

Philosopher 4 is hungry.Philosopher 0 finishes eating and leaves to think.

Philosopher 1 starts eating.

Philosopher 3 finishes eating and leaves to think.Philosopher 2 is hungry.

Philosopher 4 starts eating.

Now we're finishing.

Expt no: 2(d)

Aim: To implement dining philosopher problem using monitors

Program:

```
import threading

class Chopstick:

    def __init__(self):

        self.lock = threading.Lock()
```

```
    def pick_up(self):
        self.lock.acquire()
    def put_down(self):
        self.lock.release()
class DiningPhilosophers:
    def __init__(self, num_philosophers):
        self.philosophers = []
        self.chopsticks = [Chopstick() for _ in range(num_philosophers)]
        for i in range(num_philosophers):
            philosopher = threading.Thread(target=self.dine, args=(i,))
            self.philosophers.append(philosopher)
    def start_dining(self):
        for philosopher in self.philosophers:
            philosopher.start()
        for philosopher in self.philosophers:
            philosopher.join()
    def dine(self, philosopher_index):
        left_chopstick = self.chopsticks[philosopher_index]
        right_chopstick = self.chopsticks[(philosopher_index + 1) % len(self.chopsticks)]
        while True:
            self.think(philosopher_index)
            left_chopstick.pick_up()
            right_chopstick.pick_up()
            self.eat(philosopher_index)
            left_chopstick.put_down()
```

```
            right_chopstick.put_down()

    def think(self, philosopher_index):
        print(f"Philosopher {philosopher_index} is thinking.")

    def eat(self, philosopher_index):
        print(f"Philosopher {philosopher_index} is eating.")
dining_philosophers = DiningPhilosophers(5)
dining_philosophers.start_dining()
```

Output:

Philosopher 0 is thinking.Philosopher 1 is thinking.

Philosopher 0 is eating.

Philosopher 2 is thinking.

Philosopher 0 is thinking.

Philosopher 1 is eating.

Philosopher 3 is thinking.Philosopher 4 is thinking.

Philosopher 1 is thinking.

Philosopher 2 is eating.

Philosopher 0 is eating.Philosopher 2 is thinking.

Philosopher 2 is eating.

Philosopher 2 is thinking.

Philosopher 2 is eating.

Philosopher 2 is thinking.Philosopher 0 is thinking.Philosopher 4 is eating.

Philosopher 4 is thinking.

Philosopher 1 is eating.

Philosopher 1 is thinking.Philosopher 3 is eating.

Philosopher 0 is eating.

Philosopher 0 is thinking.

Philosopher 3 is thinking.

Philosopher 2 is eating.

Philosopher 2 is thinking.Philosopher 1 is eating.

Philosopher 1 is thinking.Philosopher 0 is eating.

Philosopher 0 is thinking.

Philosopher 4 is eating.

Philosopher 4 is thinking.

Philosopher 3 is eating.

Philosopher 3 is thinking.

Expt no: 3(A)

Aim: To simulate banker’s algorithm for dead lock avoidance

Program:

```
P = 5

R = 3

def calculateNeed(need, maxm, allot):

    for i in range(P):

        for j in range(R):

            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 is not in safe state")
        return False
    print("System is in safe state.","\nSafe sequence is: ", end = " ")
    print(*safeSeq)
    return True


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

Expt no: 3(b)

Aim: To simulate bankers algorithm for dead lock prevention

Program:

```
if __name__=="__main__":

    n = 5

    m = 3

    alloc = [[0, 1, 0 ],[ 2, 0, 0 ],[3, 0, 2 ],[2, 1, 1] ,[ 0, 0, 2]]

    max = [[7, 5, 3 ],[3, 2, 2 ],[ 9, 0, 2 ],[2, 2, 2],[4, 3, 3]]

    avail = [3, 3, 2]

    f = [0]*n

    ans = [0]*n

    ind = 0

    for k in range(n):

        f[k] = 0

    need = [[ 0 for i in range(m)]for i in range(n)]

    for i in range(n):

        for j in range(m):

            need[i][j] = max[i][j] - alloc[i][j]

    y = 0

    for k in range(5):

        for i in range(n):

            if (f[i] == 0):

                flag = 0

                for j in range(m):

                    if (need[i][j] > avail[j]):
```

```
                        flag = 1
                        break
                if (flag == 0):
                    ans[ind] = i
                    ind += 1
                    for y in range(m):
                        avail[y] += alloc[i][y]
                f[i] = 1
    print("Following is the SAFE Sequence")
    for i in range(n - 1):
        print(" P", ans[i], " ->", sep="", end="")
    print(" P", ans[n - 1], sep="")
```

Output:

```
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P0
```

Expt no: 4(a)

Aim: To simulate FIFO page replacement algorithm

Program:

```
def FIFO(pages,capacity):
    memory=[]
    page_faults=0
    for i in pages:
        if i not in memory:
            page_faults+=1
            if len(memory)<capacity:
```

```
                memory.append(i)
            else:
                memory.pop(0)
                memory.append(i)
    return page_faults
pages=list(map(int,input("enter the input sequence").strip().split()))
capacity=int(input("enter the max no. of pages"))
print("no. of page faults:",FIFO(pages,capacity))
```

Output:

```
enter the input sequence7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
enter the max no. of pages3
no. of page faults: 15
```

Expt no: 4(b)

Aim: To simulate LRU page replacement algorithm

Program:

```
def lru(pages,capacity):
    memory=[]
    lru=[]
    page_faults=0
    for i in pages:
        if i not in memory:
            if len(memory)<capacity:
                page_faults+=1
                memory.append(i)
                lru.append(i)
```

```
            else:
                page_faults+=1
                index=memory.index(lru[0])
                memory[index]=i
                lru.pop(0)
                lru.append(i)
        else:
            index=lru.index(i)
            lru.pop(index)
            lru.append(i)
    return page_faults
pages=list(map(int,input("enter the input sequence:").strip().split()))
capacity=int(input("enter max no.of pages:"))
print("no.of page faults:",lru(pages,capacity))
```

Output:

enter the input sequence:7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

enter max no.of pages:3

no.of page faults: 12

Expt no: 4(c)

Aim: TO simulate LFU page replacement algorithm

Program:

```
from collections import defaultdict
def lfu(pages,capacity):
    memory=[]
    pagefaults=0
```

```
    freq=defaultdict(int)

    for page in pages:

        if page not in memory:

            pagefaults+=1

            if len(memory)<capacity:

                memory.append(page)

                freq[page]=1

            else:

                lfupage=min(memory,key=lambda p:freq[p])

                memory.remove(lfupage)

                memory.append(page)

                freq[page]=1

        else:

            freq[page]+=1

    return pagefaults

pages=list(map(int,input("enter page sequence:").strip().split()))

capacity=int(input("enter capacity:"))

print("pagefaults:",lfu(pages,capacity))
```

Output:

enter page sequence:5 0 1 3 2 4 1 0 5

enter capacity:4

pagefaults: 8

Expt no: 4(d)

Aim: To simulate MFU page replacement algorithm

Program:

```
from collections import defaultdict

def mfu(pages,capacity):

    memory=[]

    pagefaults=0

    freq=defaultdict(int)

    for page in pages:

        if page not in memory:

            pagefaults+=1

            if len(memory)<capacity:

                memory.append(page)

                freq[page]=1

            else:

                lfupage=max(memory,key=lambda p:freq[p])

                memory.remove(lfupage)

                memory.append(page)

                freq[page]=1

        else:

            freq[page]+=1

    return pagefaults

pages=list(map(int,input("enter page sequence:").strip().split()))

capacity=int(input("enter capacity:"))

print("pagefaults:",mfu(pages,capacity))
```

Output:

enter page sequence:1 3 3 2 5 2 1 4 2 2 5

enter capacity:3

pagefaults: 5

Expt no: 5(a)

Aim: To simulate multiprogramming with fixed no. of tasks

Program:

```
totalmem=int(input("enter total ammount of memory"))

blocksize=int(input("enter the block size"))

noofblocks=totalmem//blocksize

print(noofblocks)

ef=totalmem-(noofblocks*blocksize)

#print(ef)

nop=int(input("enter the number of the processes"))

processlist=[0]*nop

for i in range(nop):

    processlist[i]=int(input("enter the ammount of memory required for the process"))

print(processlist)

print("PROCESS\tMEMORYREQUIRED\tALLOCATED\tINTERNALFRAGMENTATIO
N")

totalif=0

blockallocated=0

for i in range(nop):

    if(blockallocated<noofblocks):#chexking for blocks are available

            print("p",i,end="\t\t")#po

            print(processlist[i],end="\t\t")

            if(processlist[i]>blocksize):#process size<block size

                print("NO")
```

```
            else:
                print("YES",end="\t\t")
                print(blocksize-processlist[i])
                totalif=totalif+(blocksize-processlist[i])
                blockallocated=blockallocated+1
    else:
        print("\nMemory is Full, Remaining Processes cannot be accomodated");
print("\n\nTotal Internal Fragmentation is",totalif)
print("\nTotal External Fragmentation is",ef)
```

Output:

```
enter total ammount of memory256
enter the block size40
6
enter the number of the processes4
enter the ammount of memory required for the process50
enter the ammount of memory required for the process35
enter the ammount of memory required for the process20
enter the ammount of memory required for the process40
[50, 35, 20, 40]
PROCESS     MEMORYREQUIRED      ALLOCATED
        INTERNALFRAGMENTATION
p 0         50          NO
p 1         35          YES         5
p 2         20          YES         20
p 3         40          YES         0
```

Total Internal Fragmentation is 25

Total External Fragmentation is 16

Expt no: 5(b)

Aim: To simulate multiprogramming with variable no. of tasks

Program:

```
totalmem=int(input("enter total ammount of memory"))

temp=totalmem

choice="Y"

while(choice=='Y'):

    processmem=int(input("enter memory required for user process"))

    if(processmem<temp):

        print("memory allocated for user process")

        temp=temp-processmem

    else:

        print("memory is full.so memory not allocated for user process")

        break

    choice=input("do you want to continue(Y/N)")


print("total available memory is",totalmem)

print("total external fragmentation is",temp)
```

Output:

enter total ammount of memory256

enter memory required for user process128 memory allocated for user process

do you want to continue(Y/N)Y

enter memory required for user process200

memory is full.so memory not allocated for user process

total available memory is 256

total external fragmentation is 128

Expt no: 6(a)

Aim: To simulate contiguous file allocation strategy

Program:

```
disk_blocks = [False] * 100


file_allocation = {}

def allocate_file(filename, size):

    start_block = -1

    consecutive_blocks = 0

    for i in range(len(disk_blocks)):

        if not disk_blocks[i]:

            consecutive_blocks += 1

            if consecutive_blocks == size:

                start_block = i - size + 1

                break

        else:

            consecutive_blocks = 0
```

```
    if start_block != -1:
        for i in range(start_block, start_block + size):
            disk_blocks[i] = True
        file_allocation[filename] = start_block
        print(f"File '{filename}' allocated starting from block {start_block}")
    else:
        print(f"Not enough contiguous blocks to allocate file '{filename}'")


def deallocate_file(filename):
    if filename in file_allocation:
        start_block = file_allocation[filename]
        size = len(filename)
        for i in range(start_block, start_block + size):
            disk_blocks[i] = False
        del file_allocation[filename]
        print(f"File '{filename}' deallocated")
    else:
        print(f"File '{filename}' not found")


# Example usage
allocate_file("file1.txt", 5)
allocate_file("file2.txt", 3)
allocate_file("file3.txt", 4)
deallocate_file("file2.txt")
```

Output:

File 'file1.txt' allocated starting from block 0

File 'file2.txt' allocated starting from block 5

File 'file3.txt' allocated starting from block 8

File 'file2.txt' deallocated

Expt no: 6(b)

Aim: To simulate the Linked file allocation strategy

Program:

```
class DiskBlock:
    def __init__(self, data=None, next_block=None):
        self.data = data
        self.next_block = next_block
class FileSystem:
    def __init__(self):
        self.disk_blocks = {}
        self.file_allocation = {}
    def allocate_file(self, filename, size):
        if filename in self.file_allocation:
            print(f"File '{filename}' already exists")
            return
        first_block = None
        prev_block = None
        for i in range(size):
            block = DiskBlock()
            self.disk_blocks[i] = block
            if prev_block:
```

```
            prev_block.next_block = block
        if not first_block:
            first_block = block
        prev_block = block
    self.file_allocation[filename] = first_block
    print(f"File '{filename}' allocated with {size} blocks")
  def deallocate_file(self, filename):
    if filename not in self.file_allocation:
        print(f"File '{filename}' not found")
        return
    current_block = self.file_allocation[filename]
    while current_block:
        next_block = current_block.next_block
        del self.disk_blocks[current_block.data]
        current_block = next_block
    del self.file_allocation[filename]
    print(f"File '{filename}' deallocated")
file_system = FileSystem()
file_system.allocate_file("file1.txt", 5)
file_system.allocate_file("file2.txt", 3)
file_system.allocate_file("file3.txt", 4)
```

Output:

File 'file1.txt' allocated with 5 blocks

File 'file2.txt' allocated with 3 blocks

File 'file3.txt' allocated with 4 blocks

Expt no: 6(c)

Aim: To simulate indexed file allocation strategy

Program:

```
class IndexedFileAllocation:
    def __init__(self):
        self.file_index = {}
        self.data_blocks = []

    def create_file(self, filename, data):
        if filename in self.file_index:
            print(f"File '{filename}' already exists.")
        else:
            if len(data) <= len(self.data_blocks):
                file_blocks = []
                for i in range(len(data)):
                    block_index = self.data_blocks.index(None)
                    self.data_blocks[block_index] = data[i]
                    file_blocks.append(block_index)
                self.file_index[filename] = file_blocks
```

```
            print(f"File '{filename}' created successfully.")
        else:
            print(f"Not enough space to create file '{filename}'.")

    def delete_file(self, filename):
        if filename in self.file_index:
            file_blocks = self.file_index[filename]
            for block_index in file_blocks:
                self.data_blocks[block_index] = None
            del self.file_index[filename]
            print(f"File '{filename}' deleted successfully.")
        else:
            print(f"File '{filename}' does not exist.")

    def read_file(self, filename):
        if filename in self.file_index:
            file_blocks = self.file_index[filename]
            data = [self.data_blocks[block_index] for block_index in file_blocks]
            print(f"Data in file '{filename}': {''.join(data)}")
        else:
            print(f"File '{filename}' does not exist.")

    def display_file_system(self):
        print("File System:")
        print("File Index:")
```

```
        for filename, blocks in self.file_index.items():
            print(f"{filename}: {blocks}")
        print("Data Blocks:")
        for i, block in enumerate(self.data_blocks):
            if block is None:
                status = "Free"
            else:
                status = "Occupied"
            print(f"Block {i}: {status}")
fs = IndexedFileAllocation()
fs.create_file("file1", "Hello")
fs.create_file("file2", "World")
fs.display_file_system()
fs.read_file("file1")
fs.read_file("file2")
fs.delete_file("file1")
fs.display_file_system()
```

Output:

```
Not enough space to create file 'file1'.
Not enough space to create file 'file2'.
File System:
File Index:
Data Blocks:
File 'file1' does not exist.
File 'file2' does not exist.
```

File 'file1' does not exist.

File System:

File Index:

Data Blocks:

Expt no: 7(a)

Aim: To simulate FCFS disk scheduling algorithm

Program:

```
from matplotlib import pyplot as plt

def fcfs(sequence,head):

    sequence1=sequence.copy()

    sequence1.insert(0,head)

    plt.rcParams['xtick.bottom']=plt.rcParams['xtick.labelbottom']=False

    plt.rcParams['xtick.top']=plt.rcParams['xtick.labeltop']=True

    temp=sequence.copy()

    temp.insert(0,head)

    size=len(temp)

    x=temp

    y=[]

    headmovement=0

    for i in range(0,size):

        y.append(-i)

        if i != size-1:

            headmovement+=abs(temp[i]-temp[i+1])


plt.plot(x,y,color="green",markerfacecolor="blue",marker="o",markersize=6,linewidth=2,lab
el="fcfs")
```

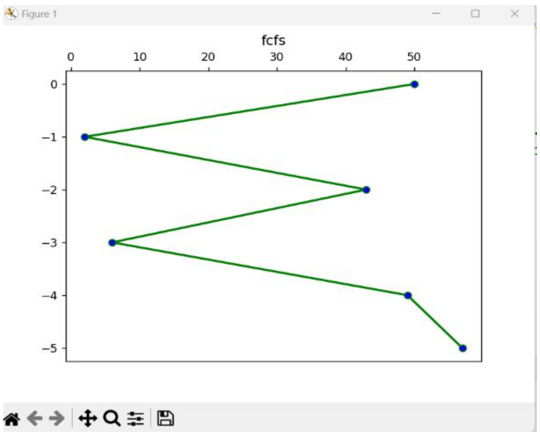
```
    plt.ylim=(0,size)

    plt.yticks=([])

    plt.xlim=(0,199)

    plt.title("fcfs")

    plt.show()

    seektime=0

    for i in range(len(sequence)):

        seektime+=abs(sequence1[i]-sequence1[i+1])

    return seektime

if __name__=="__main__":


    sequence=list(map(int,input("enter the sequence:").strip().split()))

    head=int(input("enter the current position of head:"))

    print("seek time:",fcfs(sequence,head))
```

Output:

enter the sequence:2 43 6 49 57

enter the current position of head:50

seek time: 177



Expt no: 7(b)

Aim: To simulate SSTF disk scheduling algorithm

Program:

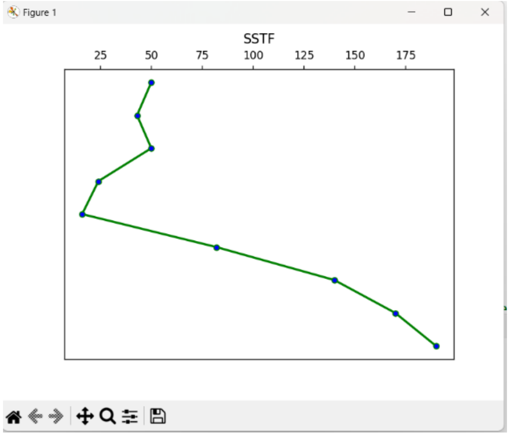
```
import math
import matplotlib.pyplot as plt
def next_in_sequence(seq,val):
        diff = 0
        mindiff = math.inf
        nextval = 0
        #print(seq)
        for i in range(0,len(seq)):
            if(seq[i]!=val):
                diff = abs(seq[i]-val)
                if(diff<mindiff ):
                    mindiff=diff
                    nextval = seq[i]
        return nextval
def sstfDiskScheduling(sequence, head):
    plt.rcParams['xtick.bottom']=plt.rcParams['xtick.labelbottom']=False
    plt.rcParams['xtick.top']=plt.rcParams['xtick.labeltop']=True
    temp = sequence.copy()
    temp.insert(0,head)
    val = head
    x = []
    y = []
```

```
    size = 0
    x.append(head)
    headmovement = 0
    while(len(temp)):
        val = next_in_sequence(temp,val)
        x.append(val)
        temp.remove(val)
    size = len(x)
    for i in range(0,size):
        y.append(-i)
        if i!=(size-1):
            headmovement = headmovement + abs(x[i]-x[i+1])
    ##string2 = str(x)


    plt.plot(x,y, color="green", markerfacecolor = 'blue', marker='o', markersize = 5, linewidth = 2, label="SSTF")
    plt.ylim = (0,size)
    plt.xlim = (0,199)
    plt.yticks([])
    plt.title("SSTF")
    plt.show()
    seek_time = 0
    tracks=sequence.copy()
    while len(tracks) != 0:
        min_track = tracks[0]
        # finding track with the lowest seek time
```

```
        for j in range(len(tracks)):

            track_seek_time = abs(tracks[j] - head)
            if min_track - head > track_seek_time:
                min_track = tracks[j]

        seek_time += abs(head - min_track)
        head = min_track
        tracks.remove(min_track)

    return seek_time



if __name__ == '__main__':
    tracks = [82,170,43,140,24,16,190]
    head = 50
    print(sstfDiskScheduling(tracks, head))
```

Output:

208



Expt no: 7(c)

Aim: To simulate SCAN disk scheduling algorithm

Program:

```
from matplotlib import pyplot as plt



def scanDiskScheduling(tracks, head, direction):
    left = list()
    right = list()
    #left.insert(0,head)
    right.insert(0,head)
    seek_time = 0

    for i in range(len(tracks)):
        if tracks[i] <= head:
            left.append(tracks[i])
        else:
            right.append(tracks[i])
```

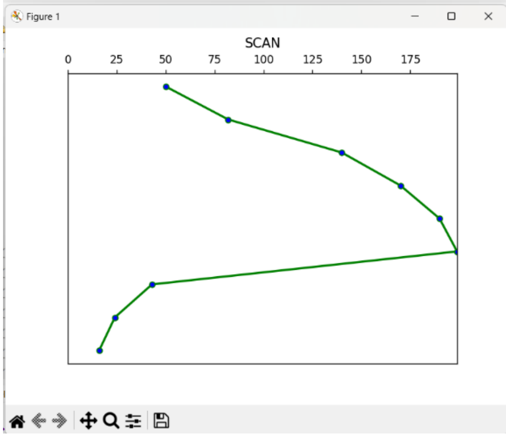
```
left.sort()
right.sort()

if not direction:
    for i in range(len(left)-1, -1, -1):
        seek_time += abs(head - left[i])
        head = left[i]
    seek_time += abs(head - 0)
    head = 0

    for i in range(len(right)):
        seek_time += abs(head - right[i])
        head = right[i]
    x = sorted(left, reverse=True) + [0] + right

else:
    for i in range(len(right)):
        seek_time += abs(head - right[i])
        head = right[i]
    seek_time += abs(head - 199)
    head = 199

    for i in range(len(left)-1, -1, -1):
        seek_time += abs(head - left[i])
        head = left[i]
```

```
        x =right + [199] + sorted(left, reverse=True)
    print(seek_time)
    plt.rcParams['xtick.bottom'] = plt.rcParams['xtick.labelbottom'] = False
    plt.rcParams['xtick.top'] = plt.rcParams['xtick.labeltop'] = True
    y = []
    for i in range(len(x)):
        y.append(-i)

    #print(list(zip(x, y)))
    plt.plot(x, y, color="green",
             markerfacecolor="blue",
             marker="o",
             markersize=5,
             linewidth=2,
             label="SCAN")
    plt.ylim = (0, len(x))
    plt.xlim(0, 199)
    plt.yticks([])
    plt.title('SCAN')
    plt.show()


if __name__ == '__main__':
    tracks = [82,170,43,140,24,16,190]
```

```
    head =50
    scanDiskScheduling(tracks, head, 1)
```

Output:

332



Expt no: 7(d)

Aim: To simulate CSCAN disk scheduling algorithm

Program:

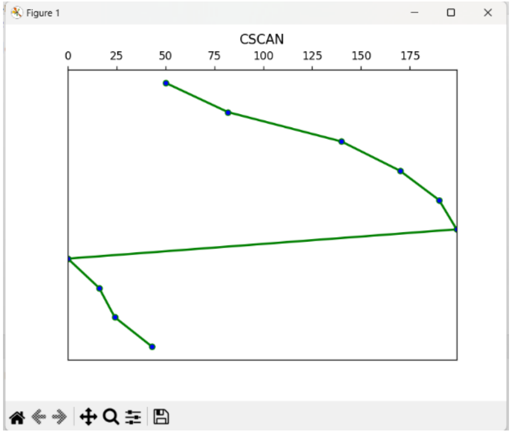
```
from matplotlib import pyplot as plt
def cscanDiskScheduling(tracks, head):
    left = list()
    right = list()
    #left.insert(0,head)
    right.insert(0,head)
    seek_time = 0
    #x=tracks.copy()
    for i in range(len(tracks)):
        if tracks[i] <= head:
            left.append(tracks[i])
```

```
    else:
        right.append(tracks[i])

left.sort()
right.sort()

for i in range(len(right)):
    seek_time += abs(head - right[i])
    head = right[i]
seek_time += abs(head - 199)
seek_time += 199
head = 0
x = sorted(left, reverse=False) + [0] + right

for i in range(len(left)):
    seek_time += abs(head - left[i])
    head = left[i]
x =right + [199] + [0]+sorted(left, reverse=False)
print(seek_time)
plt.rcParams['xtick.bottom'] = plt.rcParams['xtick.labelbottom'] = False
plt.rcParams['xtick.top'] = plt.rcParams['xtick.labeltop'] = True
y = []
for i in range(len(x)):
    y.append(-i)
```

```
    #print(list(zip(x, y)))
    plt.plot(x, y, color="green",
             markerfacecolor="blue",
             marker="o",
             markersize=5,
             linewidth=2,
             label="CSCAN")
    plt.ylim = (0, len(x))
    plt.xlim(0, 199)
    plt.yticks([])
    plt.title('CSCAN')
    plt.show()
tracks = [82,170,43,140,24,16,190]
head = 50
cscanDiskScheduling(tracks, head)
```

Output:

391



Expt no: 7(e)

Aim: To simulate LOOK disk scheduling algorithm

Program:

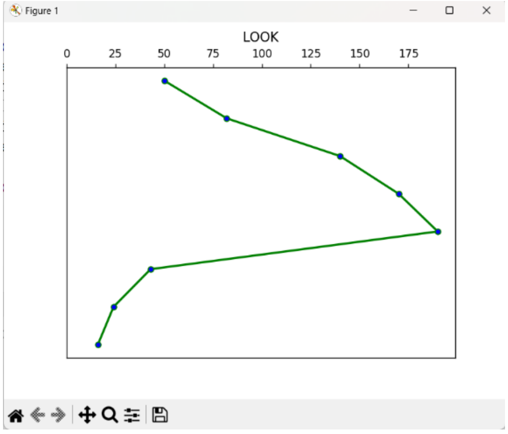
```
from matplotlib import pyplot as plt

def lookDiskScheduling(tracks, head, direction):

    left = list()

    right = list()

    #left.insert(0,head)

    right.insert(0,head)

    seek_time = 0


    for i in range(len(tracks)):

        if tracks[i] <= head:

            left.append(tracks[i])

        else:

            right.append(tracks[i])


    left.sort()

    right.sort()


    if not direction:

        for i in range(len(left)-1, -1, -1):

            seek_time += abs(head - left[i])

            head = left[i]

        seek_time += abs(head - 0)

        head = 0
```

```
    for i in range(len(right)):
        seek_time += abs(head - right[i])
        head = right[i]
    x = sorted(left, reverse=True) + [0] + right

else:
    for i in range(len(right)):
        seek_time += abs(head - right[i])
        head = right[i]
    seek_time += abs(head - right[-1])
    head = right[-1]

    for i in range(len(left)-1, -1, -1):
        seek_time += abs(head - left[i])
        head = left[i]

    x = right  + sorted(left, reverse=True)
print(seek_time)
plt.rcParams['xtick.bottom'] = plt.rcParams['xtick.labelbottom'] = False
plt.rcParams['xtick.top'] = plt.rcParams['xtick.labeltop'] = True
y = []

for i in range(len(x)):
    y.append(-i)
```

```
    #print(list(zip(x, y)))
    plt.plot(x, y, color="green",
             markerfacecolor="blue",
             marker="o",
             markersize=5,
             linewidth=2,
             label="LOOK")
    plt.ylim = (0, len(x))
    plt.xlim(0, 199)
    plt.yticks([])
    plt.title('LOOK')
    plt.show()


if __name__ == '__main__':
    tracks = [82,170,43,140,24,16,190]

    head = 50
    lookDiskScheduling(tracks, head, 1)
```

Output:

314



Expt no: 7(f)

Aim: To simulate CLOOK disk scheduling algorithm

Program:

```
from matplotlib import pyplot as plt

def clookDiskScheduling(tracks, head):

    left = list()

    right = list()

    #left.insert(0,head)

    right.insert(0,head)

    seek_time = 0

    #x=tracks.copy()

    for i in range(len(tracks)):

        if tracks[i] <= head:

            left.append(tracks[i])

        else:

            right.append(tracks[i])


    left.sort()

    right.sort()
```

```
for i in range(len(right)):
    seek_time += abs(head - right[i])
    head = right[i]
x = sorted(left, reverse=False) + [0] + right

for i in range(len(left)):
    seek_time += abs(head - left[i])
    head = left[i]
x =right+sorted(left, reverse=False)
print(seek_time)
plt.rcParams['xtick.bottom'] = plt.rcParams['xtick.labelbottom'] = False
plt.rcParams['xtick.top'] = plt.rcParams['xtick.labeltop'] = True
y = []
for i in range(len(x)):
    y.append(-i)

#print(list(zip(x, y)))
plt.plot(x, y, color="green",
         markerfacecolor="blue",
         marker="o",
         markersize=5,
         linewidth=2,
         label="CLOOK")
plt.ylim = (0, len(x))
```

```
    plt.xlim(0, 199)
    plt.yticks([])
    plt.title('CLOOK')
    plt.show()
tracks = [82,170,43,140,24,16,190]
head = 50
clookDiskScheduling(tracks, head)
```

Output:

341