**EXPERIMENT - 1**

**Aim:** Write a program to implement CPU scheduling for first come first serve.

**Theory:** In First Come First Serve (FCFS) CPU scheduling, the process that arrives first is the one that gets executed first. It's like standing in line at a grocery store — the person who arrived first is the one who gets to check out first. By default, it assumes non-pre-emptive processes entering the scheduler.

Algorithm:

1. Arrival of Processes: When processes arrive at the CPU, they are added to the end of the ready queue. The ready queue is essentially a line of processes waiting for their turn to be executed.
2. Execution: The CPU executes the processes in the order they arrive. The first process that entered the queue is the first one to be selected for execution.
3. Completion: Once a process finishes its execution, the next process in the ready queue is picked and executed.

This scheduling algorithm is simple to understand and implement, but it has some drawbacks. For example, it may lead to the "Convoy Effect.".

The Convoy Effect in FCFS (First-Come-First-Serve) scheduling occurs when a long process holds up the CPU, causing shorter processes to wait, even if they arrive later. This phenomenon can lead to inefficient resource utilization and increased response times. It's analogous to a slow-moving vehicle on a one-lane road, causing a line of cars to form behind it. To mitigate the convoy effect, scheduling algorithms like Shortest Job First (SJF) prioritize shorter processes, minimizing wait times and improving system efficiency.

**Source Code:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Process {string p\_id; int arrival\_time, burst\_time; };

bool strt\_time\_cmp(Process p1, Process p2) {return p1.arrival\_time < p2.arrival\_time;}

void FCFSScheduling(vector<Process> table) {

double avg\_wait = 0, avg\_turn\_around = 0;

int n = table.size(), completion\_time = table[0].burst\_time;

vector<int> wait(n,0), turn\_around(n,0);

turn\_around[0] = table[0].burst\_time;

for (int i = 1; i < n; i++) {

Process p = table[i];

wait[i] = completion\_time - table[i].arrival\_time;

turn\_around[i] = wait[i] + table[i].arrival\_time;

avg\_wait += wait[i];

avg\_turn\_around += turn\_around[i];

completion\_time += table[i].burst\_time;

}

cout << "Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time" << endl;

for (int i = 0; i < n; i++)

cout << table[i].p\_id << "\t\t" << table[i].arrival\_time << "\t\t"

<< table[i].burst\_time << "\t\t" << wait[i] << "\t\t" << turn\_around[i] << endl;

cout << "Average Waiting Time: " << avg\_wait / n << "\n"

<< "Average Turnaround Time: " << avg\_turn\_around / n;

}

int main() {

vector<Process> table = {{"P1", 0, 4},{"P2", 10, 33},{"P3", 4, 32},{"P4", 3, 5},{"P5", 3, 5}};

sort(table.begin(), table.end(), strt\_time\_cmp);

FCFSScheduling(table);

return 0;

}

**Output:**

Process Arrival Time Burst Time Waiting Time Turnaround Time

P1 0 4 0 4

P4 3 5 1 6

P5 3 5 6 11

P3 4 32 10 42

P2 10 33 36 69

Average Waiting Time: 10.6

Average Turnaround Time: 25.6

**EXPERIMENT - 2**

**Aim:** Write a program to implement CPU scheduling for shortest job first.

**Theory:**

Shortest Job First (SJF) CPU scheduling is a non-pre-emptive scheduling algorithm where the process with the shortest burst time is selected for execution first. The burst time is the time required by a process to complete its execution.

Algorithm:

1. Arrival of Processes: When processes arrive at the CPU, they are added to the ready queue. The ready queue is ordered based on the burst time of the processes, with the shortest burst time at the front.
2. Selection of Process: The process with the shortest burst time in the ready queue is selected for execution. If two processes have the same burst time, FCFS (First Come First Serve) order is used to break the tie.
3. Execution: The selected process is executed until completion. Since SJF is non-pre-emptive, once a process starts its execution, it continues until it finishes.
4. Completion: After the current process completes its execution, the process with the next shortest burst time in the ready queue is selected for execution.
5. Repeat: Steps 2-4 are repeated until all processes are completed.

Shortest Job First (SJF) Non-Preemptive: In non-preemptive SJF scheduling, the CPU is allocated to the waiting process with the shortest execution time. This approach minimizes average waiting times but may lead to process starvation for longer jobs, as they must wait for shorter processes to complete.

Shortest Job First (SJF) Preemptive: Preemptive SJF allows the CPU to be taken away from a running process if a shorter one becomes available. This reduces waiting times and ensures fair execution. However, it adds scheduling overhead due to frequent context switches

**Source Code:**

**Pre-emptive:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Process {string p\_id; int start\_time, burst\_time, wait = 0, turn\_around = 0,

run\_time = 0;

Process(string p\_id, int start\_time, int burst\_time)

: p\_id(p\_id), start\_time(start\_time), burst\_time(burst\_time) {}

};

bool strt\_time\_cmp(Process p1, Process p2) {

if (p1.start\_time == p2.start\_time) return p1.burst\_time < p2.burst\_time;

return p1.start\_time < p2.start\_time;

}

void SJFScheduling(vector<Process> table) {

double avg\_wait = 0, avg\_turn\_around = 0;

int n = table.size(), completion\_time = 0;

vector<Process> res\_table;

while (table.size() > 0) {

Process p = table[0];

int index = 0;

for (int j = 0; j < table.size(); j++) {

if (table[j].start\_time > completion\_time) break;

else if (table[j].burst\_time < p.burst\_time) {

p = table[j];

index = j;

}

}

table[index].run\_time += 1;

if (p.burst\_time == table[index].run\_time) {

p.wait = max(0,completion\_time - p.start\_time - p.burst\_time + 1);

p.turn\_around = p.wait + p.burst\_time;

avg\_wait += p.wait;

avg\_turn\_around += p.turn\_around;

table.erase(table.begin() + index);

res\_table.push\_back(p);

}

completion\_time += 1;

}

cout << "Process\tStart Time\tBurst Time\tWaiting Time\tTurnaround Time" << endl;

for (int i = 0; i < n; i++)

cout << res\_table[i].p\_id << "\t\t" << res\_table[i].start\_time << "\t\t"

<< res\_table[i].burst\_time << "\t\t"

<< res\_table[i].wait << "\t\t" << res\_table[i].turn\_around << endl;

cout << "Average Waiting Time: " << avg\_wait / n << "\n"

<< "Average Turnaround Time: " << avg\_turn\_around / n;

}

int main() {

vector<Process> table = {{"P1", 0, 7}, {"P2", 2, 5}, {"P3", 4, 1},

{"P4", 5, 4}, {"P5", 6, 2}, {"P6", 2, 3}};

sort(table.begin(), table.end(), strt\_time\_cmp);

SJFScheduling(table);

return 0;

}

**Non-pre-emptive:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Process {string p\_id; int start\_time, burst\_time; };

bool strt\_time\_cmp(Process p1, Process p2) {

if (p1.start\_time == p2.start\_time) return p1.burst\_time < p2.burst\_time;

return p1.start\_time < p2.start\_time;

}

void SJFScheduling(vector<Process> table) {

double avg\_wait = 0, avg\_turn\_around = 0;

int n = table.size(), completion\_time = 0;

vector<int> wait(n,0), turn\_around(n,0);

vector<Process> res\_table;

int i = 0;

while (table.size() > 0) {

Process p = table[0];

int index = 0;

for (int j = 0; j < table.size(); j++) {

if (table[j].start\_time > completion\_time) break;

else if (table[j].burst\_time < p.burst\_time) {

p = table[j];

index = j;

}

}

wait[i] = completion\_time - p.start\_time;

turn\_around[i] = wait[i] + p.burst\_time;

avg\_wait += wait[i];

avg\_turn\_around += turn\_around[i];

completion\_time += p.burst\_time;

table.erase(table.begin() + index);

res\_table.push\_back(p);

i++;

}

cout << "Process\tStart Time\tBurst Time\tWaiting Time\tTurnaround Time" << endl;

for (int i = 0; i < n; i++)

cout << res\_table[i].p\_id << "\t\t" << res\_table[i].start\_time << "\t\t"

<< res\_table[i].burst\_time << "\t\t" << wait[i] << "\t\t" << turn\_around[i] << endl;

cout << "Average Waiting Time: " << avg\_wait / n << "\n"

<< "Average Turnaround Time: " << avg\_turn\_around / n;

}

int main() {

vector<Process> table = {{"P1", 0, 7}, {"P2", 2, 5}, {"P3", 4, 1},

{"P4", 5, 4}, {"P5", 6, 2}, {"P6", 2, 3}};

sort(table.begin(), table.end(), strt\_time\_cmp);

SJFScheduling(table);

return 0;

}

**Output:**

**Pre-emptive:**

Process Start Time Burst Time Waiting Time Turnaround Time

P3 4 1 0 1

P6 2 3 1 4

P5 6 2 0 2

P4 5 4 3 7

P2 2 5 10 15

P1 0 7 15 22

Average Waiting Time: 4.83333

Average Turnaround Time: 8.5

**Non-pre-emptive:**

Process Start Time Burst Time Waiting Time Turnaround Time

P1 0 7 0 7

P3 4 1 3 4

P5 6 2 2 4

P6 2 3 8 11

P4 5 4 8 12

P2 2 5 15 20

Average Waiting Time: 6

Average Turnaround Time: 9.66667

**EXPERIMENT - 3**

**Aim:** Write a program to perform priority scheduling.

**Theory:**

Priority scheduling is a non-preemptive or preemptive CPU scheduling algorithm that assigns priorities to processes. Processes with higher priority are executed first. It allows for the efficient execution of high-priority tasks but can lead to starvation if lower-priority tasks are ignored. Preemptive priority scheduling can address this issue by temporarily suspending lower-priority tasks. Priority values can be static or dynamic, depending on system requirements, and they dictate the order in which processes are allocated CPU time.

**Preemptive Priority Scheduling:** In this approach, the CPU scheduler can interrupt the execution of a lower-priority process to allow a higher-priority process to run. It offers responsiveness but may lead to frequent context switches.

**Non-Preemptive Priority Scheduling:** In the non-preemptive variant, a process runs to completion without interruption, based on its assigned priority. While it reduces context switches, it can lead to lower-priority tasks waiting for extended periods, potentially causing issues like starvation.

**Source Code:**

**Preemptive:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Process {

string p\_id;

int start\_time, burst\_time, priority, wait = 0, turn\_around = 0, run\_time = 0;

Process(string p\_id, int start\_time, int burst\_time, int priority)

: p\_id(p\_id), start\_time(start\_time), burst\_time(burst\_time), priority(priority) {}

};

bool strt\_time\_cmp(Process p1, Process p2) {

if (p1.start\_time == p2.start\_time)

return p1.priority < p2.priority;

return p1.start\_time < p2.start\_time;

}

void PriorityScheduling(vector<Process> table) {

double avg\_wait = 0, avg\_turn\_around = 0;

int n = table.size(), completion\_time = 0;

vector<Process> res\_table;

while (table.size() > 0) {

Process p = table[0];

int index = 0;

for (int j = 0; j < table.size(); j++) {

if (table[j].start\_time > completion\_time) break;

else if (table[j].priority > p.priority) {

p = table[j];

index = j;

}

}

table[index].run\_time += 1;

if (p.burst\_time == table[index].run\_time) {

p.wait = max(0,completion\_time - p.start\_time - p.burst\_time + 1);

p.turn\_around = p.wait + p.burst\_time;

avg\_wait += p.wait;

avg\_turn\_around += p.turn\_around;

table.erase(table.begin() + index);

res\_table.push\_back(p);

}

completion\_time += 1;

}

cout << "Process\tStart Time\tBurst Time\tPriority\tWaiting Time\tTurnaround Time" << endl;

for (int i = 0; i < n; i++)

cout << res\_table[i].p\_id << "\t\t" << res\_table[i].start\_time << "\t\t"

<< res\_table[i].burst\_time << "\t\t" << res\_table[i].priority << "\t\t"

<< res\_table[i].wait << "\t\t" << res\_table[i].turn\_around << endl;

cout << "Average Waiting Time: " << avg\_wait / n << "\n"

<< "Average Turnaround Time: " << avg\_turn\_around / n;

}

int main()

{

vector<Process> table = {{"P1", 0, 7, 3}, {"P2", 2, 5, 2}, {"P3", 4, 1, 1},

{"P4", 5, 4, 4}, {"P5", 6, 2, 2}, {"P6", 2, 3, 1}};

sort(table.begin(), table.end(), strt\_time\_cmp);

PriorityScheduling(table);

return 0;

}

**Non-preemptive:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Process {string p\_id; int start\_time, burst\_time, priority; };

bool strt\_time\_cmp(Process p1, Process p2) {

if (p1.start\_time == p2.start\_time) return p1.priority < p2.priority;

return p1.start\_time < p2.start\_time;

}

void PriorityScheduling(vector<Process> table) {

double avg\_wait = 0, avg\_turn\_around = 0;

int n = table.size(), completion\_time = 0;

vector<int> wait(n,0), turn\_around(n,0);

vector<Process> res\_table;

int i = 0;

while (table.size() > 0) {

Process p = table[0];

int index = 0;

for (int j = 0; j < table.size(); j++) {

if (table[j].start\_time > completion\_time) break;

else if (table[j].priority > p.priority) {

p = table[j];

index = j;

}

}

wait[i] = completion\_time - p.start\_time;

turn\_around[i] = wait[i] + p.burst\_time;

avg\_wait += wait[i];

avg\_turn\_around += turn\_around[i];

completion\_time += p.burst\_time;

table.erase(table.begin() + index);

res\_table.push\_back(p);

i++;

}

cout << "Process\tStart Time\tBurst Time\tPriority\tWaiting Time\tTurnaround Time" << endl;

for (int i = 0; i < n; i++)

cout << res\_table[i].p\_id << "\t\t" << res\_table[i].start\_time << "\t\t"

<< res\_table[i].burst\_time << "\t\t" << res\_table[i].priority << "\t\t"

<< wait[i] << "\t\t" << turn\_around[i] << endl;

cout << "Average Waiting Time: " << avg\_wait / n << "\n"

<< "Average Turnaround Time: " << avg\_turn\_around / n;

}

int main()

{

vector<Process> table = {{"P1", 0, 7, 3}, {"P2", 2, 5, 2}, {"P3", 4, 1, 1},

{"P4", 5, 4, 4}, {"P5", 6, 2, 2}, {"P6", 2, 3, 1}};

sort(table.begin(), table.end(), strt\_time\_cmp);

PriorityScheduling(table);

return 0;

}

**Output:**

**Preemptive:**

Process Start Time Burst Time Priority Waiting Time Turnaround Time

P4 5 4 4 0 4

P1 0 7 3 4 11

P2 2 5 2 9 14

P5 6 2 2 10 12

P6 2 3 1 16 19

P3 4 1 1 17 18

Average Waiting Time: 9.33333

Average Turnaround Time: 13

**Non-preemptive:**

Process Start Time Burst Time Priority Waiting Time Turnaround Time

P1 0 7 3 0 7

P4 5 4 4 2 6

P2 2 5 2 9 14

P5 6 2 2 10 12

P6 2 3 1 16 19

P3 4 1 1 17 18

Average Waiting Time: 9

Average Turnaround Time: 12.6667

**EXPERIMENT - 4**

**Aim:** Write a program to implement CPU scheduling for Round Robin.

**Theory:**

Round Robin is a time-sharing scheduling algorithm used in operating systems. It allocates each process a fixed time slice, and if a process doesn't complete its execution within that time, it's moved to the back of the queue. This ensures fair and equal CPU time for all processes and is suitable for multi-programmed environments where processes have varying burst times. It prevents any single process from monopolizing system resources and is commonly used in interactive systems..

**Preemptive Round Robin:-**In the preemptive case, a process can be interrupted and moved to the back of the queue after consuming its time slice. If there are higher-priority processes, the CPU can switch to them, allowing for better responsiveness and adaptability.

**Non-Preemptive Round Robin:-**In the non-preemptive case, a process runs until it completes or gives up the CPU voluntarily. Only then is the next process in the queue selected. This approach may lead to longer response times for high-priority tasks.

**Source Code:-**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Process {

string p\_id;

int start\_time, burst\_time, wait = 0, turn\_around = 0, run\_time = 0;

Process(string p\_id, int start\_time, int burst\_time)

: p\_id(p\_id), start\_time(start\_time), burst\_time(burst\_time) {}

};

bool strt\_time\_cmp(Process p1, Process p2) {return p1.start\_time < p2.start\_time;}

void RRScheduling(vector<Process> table, int time\_quantum) {

cout << "Time Quantum:" << time\_quantum << "\n";

double avg\_wait = 0, avg\_turn\_around = 0;

int completion\_time = 0;

vector<Process> res\_table;

vector<Process> q;

bool notEmpty = false;

while (table.size() > 0 || q.size() > 0) {

int index = 0;

if (q.size() != 0) notEmpty = true;

int j = 0;

while (j < table.size()) {

if (table[j++].start\_time > completion\_time) break;

else {q.push\_back(table[--j]); table.erase(table.begin());}

}

if (notEmpty) {

q.push\_back(q[0]);

q.erase(q.begin());

}

if (q.size() == 0) continue;

Process p = q[0];

q[0].run\_time += time\_quantum;

if (p.burst\_time <= q[0].run\_time) {

p.wait = max(0,completion\_time - p.start\_time - p.burst\_time + time\_quantum);

p.turn\_around = p.wait + p.burst\_time;

avg\_wait += p.wait;

avg\_turn\_around += p.turn\_around;

q.erase(q.begin() + index);

res\_table.push\_back(p);

}

completion\_time += time\_quantum;

}

cout << "Process\tStart Time\tBurst Time\tWaiting Time\tTurnaround Time" << endl;

for (auto& i: res\_table)

cout << i.p\_id << "\t\t" << i.start\_time << "\t\t"

<< i.burst\_time << "\t\t"

<< i.wait << "\t\t" << i.turn\_around << endl;

int n = res\_table.size();

cout << "Average Waiting Time: " << avg\_wait / n << "\n"

<< "Average Turnaround Time: " << avg\_turn\_around / n;

}

int main() {

vector<Process> table = {{"P1", 0, 7}, {"P2", 2, 5}, {"P3", 4, 1},

{"P4", 5, 4}, {"P5", 6, 2}, {"P6", 2, 3}};

sort(table.begin(), table.end(), strt\_time\_cmp);

RRScheduling(table,2);

cout << "\n\n";

RRScheduling(table,3);

cout << "\n\n";

RRScheduling(table,4);

cout << "\n\n";

RRScheduling(table,5);

return 0;

}

**Output:**

Time Quantum:2

Process Start Time Burst Time Waiting Time Turnaround Time

P3 4 1 5 6

P5 6 2 6 8

P4 5 4 11 15

P1 0 7 15 22

P6 2 3 19 22

P2 2 5 19 24

Average Waiting Time: 12.5

Average Turnaround Time: 16.1667

Time Quantum:3

Process Start Time Burst Time Waiting Time Turnaround Time

P6 2 3 4 7

P3 4 1 7 8

P5 6 2 7 9

P2 2 5 17 22

P4 5 4 18 22

P1 0 7 23 30

Average Waiting Time: 12.6667

Average Turnaround Time: 16.3333

Time Quantum:4

Process Start Time Burst Time Waiting Time Turnaround Time

P6 2 3 7 10

P1 0 7 9 16

P5 6 2 12 14

P3 4 1 19 20

P2 2 5 21 26

P4 5 4 23 27

Average Waiting Time: 15.1667

Average Turnaround Time: 18.8333

Time Quantum:5

Process Start Time Burst Time Waiting Time Turnaround Time

P2 2 5 3 8

P3 4 1 10 11

P1 0 7 13 20

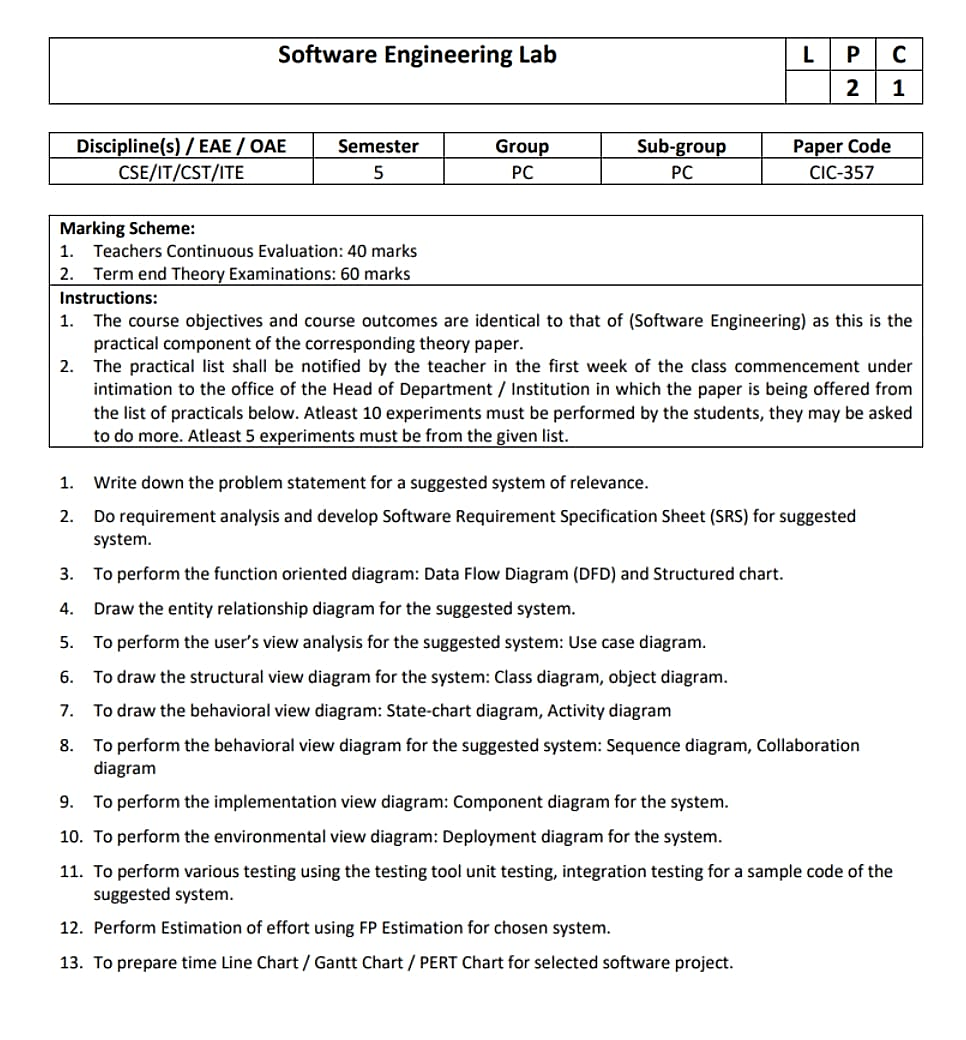
P6 2 3 20 23

P5 6 2 22 24

P4 5 4 26 30

Average Waiting Time: 15.6667

Average Turnaround Time: 19.3333



**SOFTWARE ENGINEERING LAB**

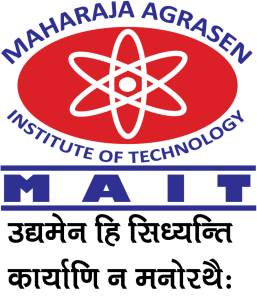
**(CIC-357)**

**Faculty Name:** Dr. Jyoti Sharma **Student Name:** Lakshay Sharma

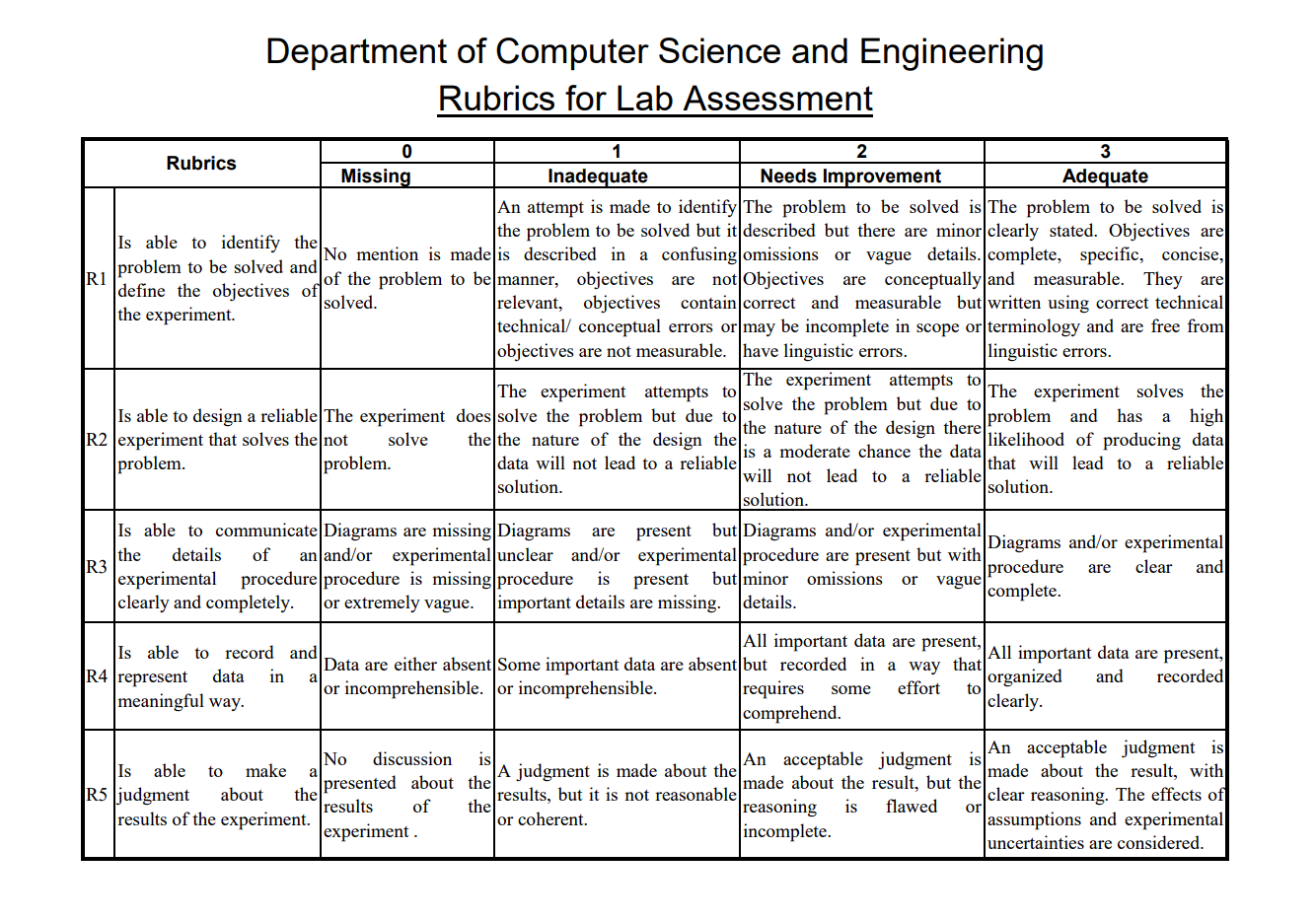
(Associate Professor, Department of CSE) **Roll No.:** 02396402721

**Semester:** 5

**Batch:** C12



**Maharaja Agrasen Institute of technology, PSP area, Sector – 22, Rohini, New Delhi – 110085**



**Software Engineering Lab (CIC-357)**

**Lab Assessment Sheet**

Student Enrolment No: 02396402721 Student Name: Lakshay Sharma

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Exp. no** | **Experiment Name** | **Date** | **Marks** | | | | | **Total Marks** | **Remarks if any** | **Signature** |
| **R1** | **R2** | **R3** | **R4** | **R5** |
| 1 | Write down the problem statement for a Library Management System. |  |  |  |  |  |  |  |  |  |
| 2 | Perform requirement analysis and develop Software Requirement Specification Sheet (SRS) for Library Management System. |  |  |  |  |  |  |  |  |  |
| 3 | To draw the Use Case Diagram for Library Management System. |  |  |  |  |  |  |  |  |  |
| 4 | To draw the Data Flow Diagram and Structured Chart for Library Management System. |  |  |  |  |  |  |  |  |  |
| 5 | To draw the Entity Relationship diagram for Library Management System. |  |  |  |  |  |  |  |  |  |
| 6 | To draw the behavioural view diagram: State‐chart diagram, Activity diagram for Library Management System. |  |  |  |  |  |  |  |  |  |
| 7 | To perform the behavioural view diagram: Sequence diagram, Collaboration diagram for Library Management System. |  |  |  |  |  |  |  |  |  |
| 8 | To perform the implementation view diagram: Component diagram for Library Management System. |  |  |  |  |  |  |  |  |  |
| 9 | To perform the environmental view diagram: Deployment diagram for Library Management System. |  |  |  |  |  |  |  |  |  |
| 10 | To perform various testing using the testing tool unit testing, integration testing for a sample code of for Library Management System . |  |  |  |  |  |  |  |  |  |
| 11 | Perform Estimation of effort using FP Estimation for Library Management System . |  |  |  |  |  |  |  |  |  |
| 12 | To prepare time Line Chart / Gantt Chart / PERT Chart for Library Management System . |  |  |  |  |  |  |  |  |  |

