#include <iostream>

#include <string>

#include <iomanip>

using namespace std;

class Process;

class Round\_Robin;

struct node

{

int data;

node\* next;

};

struct Process\_Type // To store average times for all the processes.

{

string process\_name;

float avg\_turnaround\_time;

float avg\_waiting\_time;

};

class Queue

{

node\* front;

node\* rear;

public:

Queue();

bool isEmpty();

void dequeue();

void enqueue(int &);

int peek();

};

class Process

{

int id;

static int count; // Count will maintain the record of total processes created.

int burst\_time;

int stored\_burst\_time; // To keep the original value of burst time intact.

int arrival\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

int priority;

public:

friend class Priority\_Scheduling;

Process() :id(count) { completion\_time = turnaround\_time = waiting\_time = 0; count++; priority = 0; }

// Every process will have a unique Process ID as assigned by count...

void input();

// Set and Get Functions.

int get\_t() { return turnaround\_time; }

int get\_w() { return waiting\_time; }

int getid() { return id; }

int get\_bt() { return burst\_time; }

int get\_ct() { return completion\_time; }

int get\_at() { return arrival\_time; }

int get\_sbt() { return stored\_burst\_time; }

void set\_bt(int &a) { burst\_time = a; }

void set\_at(int &a) { arrival\_time = a; }

void set\_ct(int &a) { completion\_time = a; }

void set\_t(int &a) { turnaround\_time = a; }

void set\_w(int &a) { waiting\_time = a; }

void set\_p(int &a) { priority = a; }

int get\_priority() { return priority; }

void Display();

void Display\_Before();

};

class Round\_Robin

{

public:

void Calculate(Queue &);

void algorithm();

};

int total\_time = 0;

int Process::count = 0;

int time\_slice, num;

Process\_Type p\_type[4];

Process \*obj = NULL;

Queue Ready\_Queue;

float avg\_turnaround\_time;

float avg\_waiting\_time;

void input();

int main()

{

input();

Round\_Robin rr;

// Round Robin Algorithm is used as a driver algorithm for the implementation of question.

rr.algorithm();

// Printing Before Calculation

cout << endl << endl << setw(15) << "Process ID" << setw(15) << "Arrival Time" << setw(15) << "Burst Time" << endl << "------------------------------------------------------------------" << endl;

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

obj[i].Display\_Before();

// Printing After Calculation

cout << endl << endl << setw(15) << "Process ID" << setw(15) << "Arrival Time" << setw(15) << "Burst Time" << setw(15) << "Turnaround" << setw(15) << "Waiting Time" << endl

<< "------------------------------------------------------------------" << endl;

for (int i = 1; i < num; ++i)

{

for (int j = 0; j < num - i; ++j)

if (obj[j].getid() > obj[j + 1].getid())

swap(obj[j], obj[j + 1]);

}

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

obj[i].Display();

cout << endl << endl;

cout << "Average Turnaround Time: " << avg\_turnaround\_time << endl

<< "Average Waiting Time: " << avg\_waiting\_time << endl;

cout << endl;

system("pause");

return 0;

}

void input()

{

do

{

cout << "Enter Number of Processes: ";

cin >> num;

} while (num < 1);

obj = new Process[num];

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

obj[i].input();

}

void Process::Display()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << setw(15) << turnaround\_time << setw(15) << waiting\_time;

cout << endl;

}

void Process::Display\_Before()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << endl;

}

void Process::input()

{

cout << "Process: " << id << endl

<< "Enter Arrival Time: "; cin >> arrival\_time;

cout << "Enter Burst Time: "; cin >> burst\_time;

stored\_burst\_time = burst\_time;

// cout << "Enter Priority: "; cin >> priority;

}

// --------------------ROUND ROBIN CLASS ------------------------------------

void Round\_Robin::Calculate(Queue &Ready\_Queue)

{

int total\_time = 0;

while (!Ready\_Queue.isEmpty())

{

int temp;

temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

if (obj[temp].get\_at() > total\_time)

Ready\_Queue.enqueue(temp);

else

{

if (obj[temp].get\_bt() <= time\_slice)

{

total\_time = total\_time + obj[temp].get\_bt();

obj[temp].set\_ct(total\_time);

int b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

//obj[temp].waiting\_time = obj[temp].turnaround\_time - obj[temp].burst\_time;

}

else

{

total\_time = total\_time + time\_slice;

int b = obj[temp].get\_bt() - time\_slice;

obj[temp].set\_bt(b);

Ready\_Queue.enqueue(temp);

}

}

}

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

{

int b = (obj[i].get\_t() - obj[i].get\_sbt());

obj[i].set\_w(b);

}

}

void Round\_Robin::algorithm()

{

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

(Ready\_Queue).enqueue(i); // All the processes pushed into the Ready Queue...

cout << "Enter Time Slice for Round Robin: "; cin >> time\_slice;

Calculate(Ready\_Queue);

cout << endl << left << setw(30) << " " << "ROUND ROBIN ALGORITHM" << endl << endl;

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

{

avg\_turnaround\_time += obj[i].get\_t();

avg\_waiting\_time += obj[i].get\_w();

}

avg\_turnaround\_time = avg\_turnaround\_time / num;

avg\_waiting\_time = avg\_waiting\_time / num;

}

// ------------------- QUEUE CLASS IMPLEMENTATION ---------------------------

Queue::Queue() { front = NULL; rear = NULL; }

bool Queue::isEmpty()

{

if (!front)

return true;

return false;

}

int Queue::peek() { return front->data; }

void Queue::enqueue(int &d)

{

node\* temp = new node;

temp->data = d;

temp->next = NULL;

if (isEmpty())

{

front = temp;

rear = temp;

return;

}

rear->next = temp;

rear = temp;

}

void Queue::dequeue()

{

if (isEmpty())

{

cout << "Nothing to delete";

return;

}

node\* temp = front;

front = front->next;

free(temp);

}

**Sample Input Used:**

Enter Number of Processes: 3

Process: 0

Enter Arrival Time: 2

Enter Burst Time: 5

Process: 1

Enter Arrival Time: 0

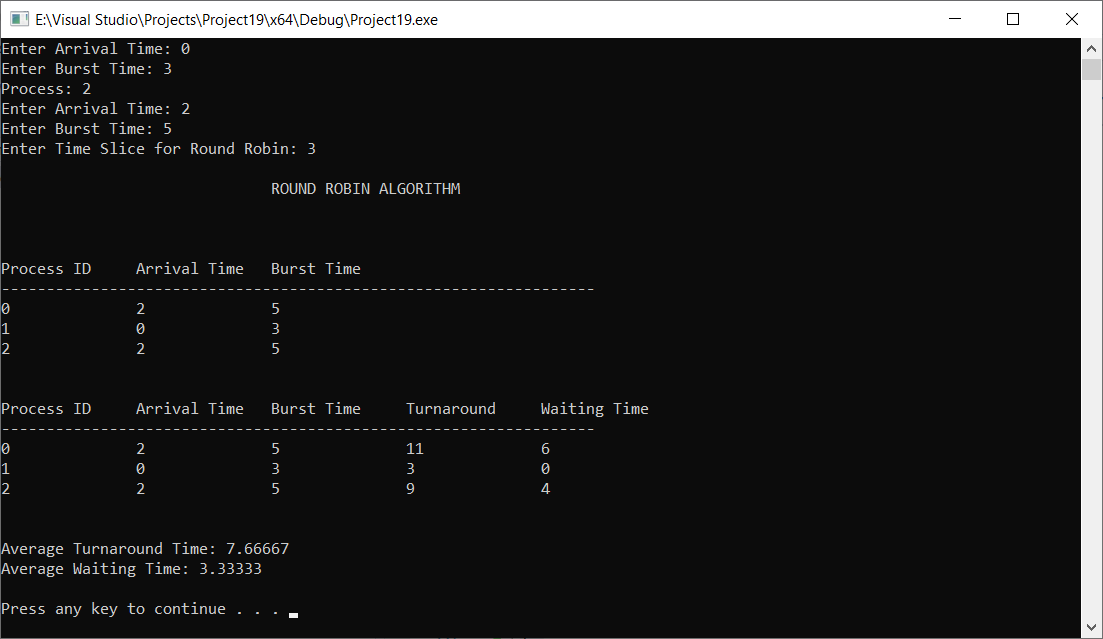
Enter Burst Time: 3

Process: 2

Enter Arrival Time: 2

Enter Burst Time: 5

Enter Time Slice for Round Robin: 3



**Question 2**

#include <iostream>

#include <string>

#include <iomanip>

using namespace std;

class Process;

class Round\_Robin;

class FCFS;

class SJF;

class Priority\_Scheduling;

struct node

{

int data;

node\* next;

};

struct Process\_Type // To store average times for all the processes.

{

string process\_name;

float avg\_turnaround\_time;

float avg\_waiting\_time;

};

class Queue

{

node\* front;

node\* rear;

public:

Queue();

bool isEmpty();

void dequeue();

void enqueue(int &);

int peek();

};

class Process

{

int id;

static int count; // Count will maintain the record of total processes created.

int burst\_time;

int stored\_burst\_time; // To keep the original value of burst time intact.

int arrival\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

int priority;

bool has\_finished; // To check whether process has finished execution or not

public:

friend class Priority\_Scheduling;

Process() :id(count) { completion\_time = turnaround\_time = waiting\_time = 0; count++; has\_finished = false; priority = 0; }

// Every process will have a unique Process ID as assigned by count...

void input();

// Set and Get Functions.

int get\_t() { return turnaround\_time; }

int get\_w() { return waiting\_time; }

int getid() { return id; }

int get\_bt() { return burst\_time; }

int get\_ct() { return completion\_time; }

int get\_at() { return arrival\_time; }

int get\_sbt() { return stored\_burst\_time; }

void set\_bt(int &a) { burst\_time = a; }

void set\_at(int &a) { arrival\_time = a; }

void set\_ct(int &a) { completion\_time = a; }

void set\_t(int &a) { turnaround\_time = a; }

void set\_w(int &a) { waiting\_time = a; }

bool get\_finished() { return has\_finished; }

void set\_finished(bool a) { has\_finished = a; }

int get\_priority() { return priority; }

void Display();

void Display\_Before();

Process(const Process &);

};

class Round\_Robin

{

public:

void Calculate(Queue &);

void algorithm();

};

class FCFS

{

public:

void Calculate(Queue &);

void algorithm();

void sort();

};

class SJF

{

int total\_time;

public:

void Calculate(Queue &);

void algorithm();

void sort();

bool check();

};

class Priority\_Scheduling

{

int total\_time;

public:

void sort();

bool check();

void priority\_sort();

void Calculate(Queue &);

void algorithm();

};

int Process::count = 0;

int time\_slice, num;

Process\_Type p\_type[4];

Process \*obj = NULL;

Queue Ready\_Queue[4];

void input();

int main()

{

input();

Round\_Robin rr;

Priority\_Scheduling ps;

SJF sjf;

FCFS fcfs;

cout << "Enter Time Slice for Round Robin: "; cin >> time\_slice;

cout << endl << endl << setw(15) << "Process ID" << setw(15) << "Arrival Time" << setw(15) << "Burst Time" << setw(15) << "Priority" << endl

<< "------------------------------------------------------------------" << endl;

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

obj[i].Display\_Before();

fcfs.algorithm();

sjf.algorithm();

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

obj[i].set\_finished(true);

ps.algorithm();

rr.algorithm();

int min = 0;

for (int i = 0; i < 4; i++)

{

if (p\_type[min].avg\_waiting\_time > p\_type[i].avg\_waiting\_time)

min = i;

}

cout << endl << endl;

cout << "Best Algorithm for given Input is: " << p\_type[min].process\_name << endl;

cout << "Minimum Average Waiting Time: " << p\_type[min].avg\_waiting\_time << endl;

cout << endl;

system("pause");

return 0;

}

void input()

{

do

{

cout << "Enter Number of Processes: ";

cin >> num;

} while (num < 1);

obj = new Process[num];

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

obj[i].input();

}

void Process::Display()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << setw(15) << turnaround\_time << setw(15) << waiting\_time;

cout << endl;

}

void Process::Display\_Before()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << setw(15) << priority << endl;

}

void Process::input()

{

cout << "Process: " << id << endl

<< "Enter Arrival Time: "; cin >> arrival\_time;

cout << "Enter Burst Time: "; cin >> burst\_time;

stored\_burst\_time = burst\_time;

cout << "Enter Priority: "; cin >> priority;

}

// --------------------FCFS CLASS ------------------------------------

void FCFS::sort()

{

int j;

for (int i = 1; i < num; ++i)

{

for (j = 0; j < num - i; ++j)

if (obj[j].get\_at() > obj[j + 1].get\_at())

swap(obj[j], obj[j + 1]);

}

}

void FCFS::algorithm()

{

sort();

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

Ready\_Queue[0].enqueue(i); // Inserting Process IDs into the Queue for FCFS

Calculate(Ready\_Queue[0]);

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

{

p\_type[0].avg\_turnaround\_time += obj[i].get\_t();

p\_type[0].avg\_waiting\_time += obj[i].get\_w();

}

cout << endl << left << setw(30) << " " << "FIRST COME FIRST SERVE ALGORITHM" << endl << endl;

p\_type[0].process\_name = "FIRST COME FIRST SERVE ALGORITHM";

p\_type[0].avg\_turnaround\_time = p\_type[0].avg\_turnaround\_time / num;

p\_type[0].avg\_waiting\_time = p\_type[0].avg\_waiting\_time / num;

cout << "Average Turnaround Time: " << p\_type[0].avg\_turnaround\_time << endl

<< "Average Waiting Time: " << p\_type[0].avg\_waiting\_time << endl;

}

void FCFS::Calculate(Queue &Ready\_Queue)

{

int total\_time = 0;

while (!(Ready\_Queue.isEmpty()))

{

int temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

int b = total\_time + obj[temp].get\_sbt();

obj[temp].set\_ct(b);

total\_time += obj[temp].get\_sbt();

b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

b = obj[temp].get\_t() - obj[temp].get\_sbt();

obj->set\_w(b);

}

}

// --------------------SJF CLASS ------------------------------------

void SJF::sort()

{

int j;

for (int i = 1; i < num; ++i)

{

for (j = 0; j < num - i; ++j)

if (obj[j].get\_sbt() > obj[j + 1].get\_sbt())

swap(obj[j], obj[j + 1]);

}

}

bool SJF::check()

{

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

if (obj[i].get\_finished() == false)

return false;

return true;

}

void SJF::algorithm()

{

total\_time = 0;

sort();

while (!check())

{

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

{

if (obj[i].get\_at() <= total\_time) // Initially, only those processes are inserted whose arrival time is zero.

if (obj[i].get\_finished() == false)

Ready\_Queue[1].enqueue(i); // Inserting Process IDs into the Queue for SJF

}

if (!(Ready\_Queue[1].isEmpty()))

Calculate(Ready\_Queue[1]);

}

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

{

p\_type[1].avg\_turnaround\_time += obj[i].get\_t();

p\_type[1].avg\_waiting\_time += obj[i].get\_w();

}

cout << endl << left << setw(30) << " " << "SHORT JOB FIRST ALGORITHM" << endl << endl;

p\_type[1].process\_name = "SHORT JOB FIRST ALGORITHM";

p\_type[1].avg\_turnaround\_time = p\_type[1].avg\_turnaround\_time / num;

p\_type[1].avg\_waiting\_time = p\_type[1].avg\_waiting\_time / num;

cout << "Average Turnaround Time: " << p\_type[1].avg\_turnaround\_time << endl

<< "Average Waiting Time: " << p\_type[1].avg\_waiting\_time << endl;

}

void SJF::Calculate(Queue &Ready\_Queue)

{

int temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

int b = total\_time + obj[temp].get\_sbt();

obj[temp].set\_ct(b);

total\_time += obj[temp].get\_sbt();

b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

b = obj[temp].get\_t() - obj[temp].get\_sbt();

obj[temp].set\_w(b);

obj[temp].set\_finished(true);

while (!(Ready\_Queue.isEmpty()))

Ready\_Queue.dequeue();

}

// --------------------Priority CLASS ------------------------------------

void Priority\_Scheduling::sort()

{

int j;

for (int i = 1; i < num; ++i)

{

for (j = 0; j < num - i; ++j)

if (obj[j].get\_at() > obj[j + 1].get\_at())

swap(obj[j], obj[j + 1]);

}

}

void Priority\_Scheduling::priority\_sort()

{

int j;

for (int i = 1; i < num; ++i)

{

for (j = 0; j < num - 1; ++j)

{

if (obj[j].priority > obj[j + 1].priority)

swap(obj[j], obj[j + 1]);

}

}

}

bool Priority\_Scheduling::check()

{

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

if (obj[i].get\_finished() == false)

return false;

return true;

}

void Priority\_Scheduling::algorithm()

{

total\_time = 0;

cout << endl;

sort();

priority\_sort();

while (!check())

{

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

{

if (obj[i].get\_at() <= total\_time) // Initially, only those processes are inserted whose arrival time is zero.

if (obj[i].get\_finished() == false)

Ready\_Queue[2].enqueue(i); // Inserting Process IDs into the Queue for Priority\_Scheduling

}

if (!(Ready\_Queue[2].isEmpty()))

Calculate(Ready\_Queue[2]);

}

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

{

p\_type[2].avg\_turnaround\_time += obj[i].get\_t();

p\_type[2].avg\_waiting\_time += obj[i].get\_w();

}

p\_type[2].process\_name = "PRIORITY SCHEDULING ALGORITHM";

cout << endl << left << setw(30) << " " << "PRIORITY SCHEDULING ALGORITHM" << endl << endl;

p\_type[2].avg\_turnaround\_time = p\_type[2].avg\_turnaround\_time / num;

p\_type[2].avg\_waiting\_time = p\_type[2].avg\_waiting\_time / num;

cout << "Average Turnaround Time: " << p\_type[2].avg\_turnaround\_time << endl

<< "Average Waiting Time: " << p\_type[2].avg\_waiting\_time << endl;

}

void Priority\_Scheduling::Calculate(Queue &Ready\_Queue)

{

int temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

int b = total\_time + obj[temp].get\_sbt();

obj[temp].set\_ct(b);

total\_time += obj[temp].get\_sbt();

b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

b = obj[temp].get\_t() - obj[temp].get\_sbt();

obj[temp].set\_w(b);

obj[temp].set\_finished(true);

while (!(Ready\_Queue.isEmpty()))

Ready\_Queue.dequeue();

}

// --------------------ROUND ROBIN CLASS ------------------------------------

void Round\_Robin::Calculate(Queue &Ready\_Queue)

{

int total\_time = 0;

while (!Ready\_Queue.isEmpty())

{

int temp;

temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

if (obj[temp].get\_at() > total\_time)

Ready\_Queue.enqueue(temp);

else

{

if (obj[temp].get\_bt() <= time\_slice)

{

total\_time = total\_time + obj[temp].get\_bt();

obj[temp].set\_ct(total\_time);

int b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

//obj[temp].waiting\_time = obj[temp].turnaround\_time - obj[temp].burst\_time;

}

else

{

total\_time = total\_time + time\_slice;

int b = obj[temp].get\_bt() - time\_slice;

obj[temp].set\_bt(b);

Ready\_Queue.enqueue(temp);

}

}

}

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

{

int b = (obj[i].get\_t() - obj[i].get\_sbt());

obj[i].set\_w(b);

}

}

void Round\_Robin::algorithm()

{

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

(Ready\_Queue[3]).enqueue(i); // All the processes pushed into the Ready Queue...

Calculate(Ready\_Queue[3]);

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

{

p\_type[3].avg\_turnaround\_time += obj[i].get\_t();

p\_type[3].avg\_waiting\_time += obj[i].get\_w();

}

cout << endl << left << setw(30) << " " << "ROUND ROBIN ALGORITHM" << endl << endl;

p\_type[3].process\_name = "ROUND ROBIN ALGORITHM";

p\_type[3].avg\_turnaround\_time = p\_type[3].avg\_turnaround\_time / num;

p\_type[3].avg\_waiting\_time = p\_type[3].avg\_waiting\_time / num;

cout << "Average Turnaround Time: " << p\_type[3].avg\_turnaround\_time << endl

<< "Average Waiting Time: " << p\_type[3].avg\_waiting\_time << endl;

}

// ------------------- QUEUE CLASS IMPLEMENTATION ---------------------------

Queue::Queue() { front = NULL; rear = NULL; }

bool Queue::isEmpty()

{

if (!front)

return true;

return false;

}

int Queue::peek() { return front->data; }

void Queue::enqueue(int &d)

{

node\* temp = new node;

temp->data = d;

temp->next = NULL;

if (isEmpty())

{

front = temp;

rear = temp;

return;

}

rear->next = temp;

rear = temp;

}

void Queue::dequeue()

{

if (isEmpty())

{

cout << "Nothing to delete";

return;

}

node\* temp = front;

front = front->next;

free(temp);

}

Process::Process(const Process &Process\_obj)

{

this->arrival\_time = Process\_obj.arrival\_time;

this->burst\_time = Process\_obj.burst\_time;

this->completion\_time = Process\_obj.completion\_time;

this->id = Process\_obj.id;

this->stored\_burst\_time = Process\_obj.stored\_burst\_time;

this->turnaround\_time = Process\_obj.turnaround\_time;

this->waiting\_time = Process\_obj.waiting\_time;

this->has\_finished = Process\_obj.has\_finished;

}

**SAMPLE INPUT**

Enter Number of Processes: 4

Process: 0

Enter Arrival Time: 2

Enter Burst Time: 5

Enter Priority: 5

Process: 1

Enter Arrival Time: 0

Enter Burst Time: 3

Enter Priority: 4

Process: 2

Enter Arrival Time: 3

Enter Burst Time: 5

Enter Priority: 2

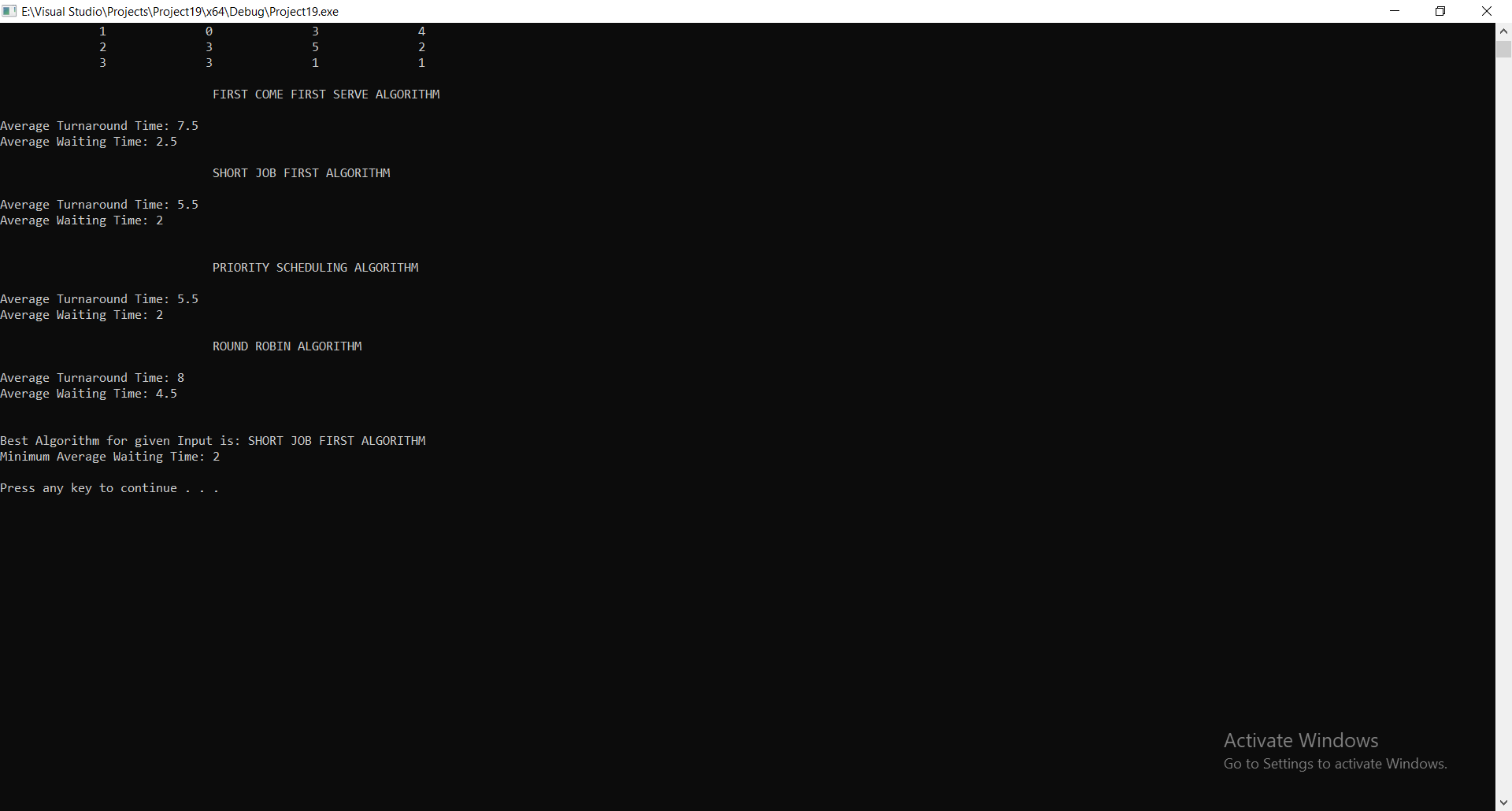
Process: 3

Enter Arrival Time: 3

Enter Burst Time: 1

Enter Priority: 1

Enter Time Slice for Round Robin: 2



**Question 3**

#include <iostream>

#include <string>

#include <iomanip>

using namespace std;

class Process;

class Implementation\_Class;

class FCFS;

struct node

{

int data;

node\* next;

};

class Queue

{

node\* front;

node\* rear;

public:

Queue();

bool isEmpty();

void dequeue();

void enqueue(int &);

int peek();

};

class Process

{

int id;

static int count; // Count will maintain the record of total processes created.

int burst\_time;

int stored\_burst\_time; // To keep the original value of burst time intact.

int arrival\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

public:

Process() :id(count) { completion\_time = turnaround\_time = waiting\_time = 0; count++; }

// Every process will have a unique Process ID as assigned by count...

void input();

// Set and Get Functions.

int get\_t() { return turnaround\_time; }

int get\_w() { return waiting\_time; }

int getid() { return id; }

int get\_bt() { return burst\_time; }

int get\_ct() { return completion\_time; }

int get\_at() { return arrival\_time; }

int get\_sbt() { return stored\_burst\_time; }

void set\_bt(int &a) { burst\_time = a; }

void set\_at(int &a) { arrival\_time = a; }

void set\_ct(int &a) { completion\_time = a; }

void set\_t(int &a) { turnaround\_time = a; }

void set\_w(int &a) { waiting\_time = a; }

void Display();

void Display\_Before();

};

class Implementation\_Class

{

int total\_time = 0;

public:

void Calculate\_1st(Queue &);

void Calculate\_2nd(Queue &);

void FCFS\_Algorithm(Queue &);

void algorithm();

};

int Process::count = 0;

int time\_slice, num, process\_time;

int num1, num2, num3;

Process \*obj = NULL;

Queue Ready\_Queue[3];

void input();

int main()

{

input();

cout << "Enter Time Slice for Round Robin: "; cin >> time\_slice;

Implementation\_Class r1;

cout << endl << endl << setw(15) << "Process ID" << setw(15) << "Arrival Time" << setw(15) << "Burst Time" << setw(15) << "Priority" << endl

<< "------------------------------------------------------------------" << endl;

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

obj[i].Display\_Before();

// Round Robin Algorithm is used as a driver algorithm for the implementation of question.

// Printing Before Calculation

// Printing After Calculation

cout << endl << endl << setw(15) << "Process ID" << setw(15) << "Arrival Time" << setw(15) << "Burst Time" << setw(15) << "Turnaround" << setw(15) << "Waiting Time" << endl

<< "------------------------------------------------------------------" << endl;

for (int i = 1; i < num; ++i)

{

for (int j = 0; j < num - i; ++j)

if (obj[j].getid() > obj[j + 1].getid())

swap(obj[j], obj[j + 1]);

}

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

obj[i].Display();

cout << endl << endl;

cout << endl;

system("pause");

return 0;

}

void input()

{

do

{

cout << "Enter Number of Processes for 1st Queue (Priority 3): ";

cin >> num1;

} while (num1 < 1);

num = 0;

num += num1;

do

{

cout << "Enter Number of Processes for 2nd Queue (Priority 2): ";

cin >> num2;

} while (num2 < 1);

num += num2;

do

{

cout << "Enter Number of Processes for 3rd Queue (Priority 1): ";

cin >> num3;

} while (num3 < 1);

num += num3;

obj = new Process[num];

cout << "\nEnter Processes for Queue 1 (Priority 3), \n";

int i = 0;

for (i = 0; i < num1; i++)

obj[i].input();

cout << "\nEnter Processes for Queue 2 (Priority 2), \n";

for (i; i < num1 + num2; i++)

obj[i].input();

cout << "\nEnter Processes for Queue 3 (Priority 1), \n";

for (i; i < num; i++)

obj[i].input();

}

void Process::Display()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << setw(15) << turnaround\_time << setw(15) << waiting\_time;

cout << endl;

}

void Process::Display\_Before()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << endl;

}

void Process::input()

{

cout << "Process: " << id << endl

<< "Enter Arrival Time: "; cin >> arrival\_time;

cout << "Enter Burst Time: "; cin >> burst\_time;

stored\_burst\_time = burst\_time;

}

// --------------------ROUND ROBIN CLASS ------------------------------------

void Implementation\_Class::Calculate\_1st(Queue &first\_queue)

{

int time\_interval = 0;

while (!first\_queue.isEmpty())

{

int temp;

temp = first\_queue.peek();

first\_queue.dequeue();

if (obj[temp].get\_at() > total\_time)

first\_queue.enqueue(temp);

else

{

if (obj[temp].get\_bt() <= time\_slice)

{

total\_time = total\_time + obj[temp].get\_bt();

process\_time += obj[temp].get\_bt();

time\_interval += obj[temp].get\_bt();

obj[temp].set\_ct(total\_time);

int b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

}

else

{

total\_time = total\_time + time\_slice;

process\_time += time\_slice;

time\_interval += time\_slice;

int b = obj[temp].get\_bt() - time\_slice;

obj[temp].set\_bt(b);

first\_queue.enqueue(temp);

}

}

if (time\_interval >= 10) // If 10 seconds have passed, push the process from 3rd to 2nd queue

{

if (!Ready\_Queue[2].isEmpty())

{

int temp1 = Ready\_Queue[2].peek();

Ready\_Queue[1].enqueue(temp1);

Ready\_Queue[2].dequeue();

time\_interval -= 10;

}

}

if (process\_time > 3 \* time\_slice) // if allocated time is passed then return from function..

return;

}

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

{

int b = (obj[i].get\_t() - obj[i].get\_sbt());

obj[i].set\_w(b);

}

}

void Implementation\_Class::Calculate\_2nd(Queue &second\_queue)

{

int time\_interval = 0;

while (!second\_queue.isEmpty())

{

int temp;

temp = second\_queue.peek();

second\_queue.dequeue();

if (obj[temp].get\_at() > total\_time)

second\_queue.enqueue(temp);

else

{

if (obj[temp].get\_bt() <= time\_slice)

{

total\_time = total\_time + obj[temp].get\_bt();

process\_time += obj[temp].get\_bt();

time\_interval += obj[temp].get\_bt();

obj[temp].set\_ct(total\_time);

int b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

}

else

{

total\_time = total\_time + time\_slice;

process\_time += time\_slice;

time\_interval += time\_slice;

int b = obj[temp].get\_bt() - time\_slice;

obj[temp].set\_bt(b);

second\_queue.enqueue(temp);

}

}

}

if (time\_interval > 10) // If 10 seconds have passed, push the process from 3rd to 2nd queue

{

if (Ready\_Queue[2].isEmpty())

{

int temp1 = Ready\_Queue[2].peek();

Ready\_Queue[1].enqueue(temp1);

Ready\_Queue[2].dequeue();

}

}

if (process\_time > time\_slice) // if allocated time is passed then return from function..

return;

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

{

int b = (obj[i].get\_t() - obj[i].get\_sbt());

obj[i].set\_w(b);

}

}

void Implementation\_Class::FCFS\_Algorithm(Queue &Ready\_Queue)

{

while (!(Ready\_Queue.isEmpty()))

{

int temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

int b = total\_time + obj[temp].get\_sbt();

obj[temp].set\_ct(b);

total\_time += obj[temp].get\_sbt();

b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

b = obj[temp].get\_t() - obj[temp].get\_sbt();

obj[temp].set\_w(b);

}

}

void Implementation\_Class::algorithm()

{

total\_time = 0;

int i = 0;

for (i = 0; i < num1; i++)

(Ready\_Queue[0]).enqueue(i);

for (i; i < num1 + num2; i++)

(Ready\_Queue[1]).enqueue(i);

for (i; i < num; i++)

(Ready\_Queue[2]).enqueue(i);

while ((!Ready\_Queue[0].isEmpty()) || (!Ready\_Queue[1].isEmpty()) || (!Ready\_Queue[2].isEmpty()))

{

process\_time = 0;

Calculate\_1st(Ready\_Queue[0]);

process\_time = 0;

Calculate\_2nd(Ready\_Queue[1]);

if ((Ready\_Queue[0].isEmpty()) && (Ready\_Queue[1].isEmpty()))

break;

}

if (!Ready\_Queue[2].isEmpty())

FCFS\_Algorithm(Ready\_Queue[2]);

}

// ------------------- QUEUE CLASS IMPLEMENTATION ---------------------------

Queue::Queue() { front = NULL; rear = NULL; }

bool Queue::isEmpty()

{

if (!front)

return true;

return false;

}

int Queue::peek() { return front->data; }

void Queue::enqueue(int &d)

{

node\* temp = new node;

temp->data = d;

temp->next = NULL;

if (isEmpty())

{ rear = temp;

return;

}

rear->next = temp;

rear = temp;

}

void Queue::dequeue()

{

if (isEmpty())

{

return;

}

node\* temp = front;

front = front->next;

free(temp);

}

**SAMPLE INPUT:**

front = temp;

Enter Number of Processes for 1st Queue (Priority 3): 3

Enter Number of Processes for 2nd Queue (Priority 2): 3

Enter Number of Processes for 3rd Queue (Priority 1): 2

Enter Processes for Queue 1 (Priority 3),

Process: 0

Enter Arrival Time: 2

Enter Burst Time: 3

Process: 1

Enter Arrival Time: 0

Enter Burst Time: 5

Process: 2

Enter Arrival Time: 0

Enter Burst Time: 6

Enter Processes for Queue 2 (Priority 2),

Process: 3

Enter Arrival Time: 0

Enter Burst Time: 3

Process: 4

Enter Arrival Time: 5

Enter Burst Time: 3

Process: 5

Enter Arrival Time: 5

Enter Burst Time: 2

Enter Processes for Queue 3 (Priority 1),

Process: 6

Enter Arrival Time: 2

Enter Burst Time: 6

Process: 7

Enter Arrival Time: 3

Enter Burst Time: 1

Enter Time Slice for Round Robin: 2

