SJF PREEMPTIVE

```
package scheduling_algos;


// Java program to implement Shortest Remaining Time First
// Shortest Remaining Time First (SRTF)

class Process
{
    int pid; // Process ID
    int bt; // Burst Time
    int art; // Arrival Time

    public Process(int pid, int bt, int art)
    {
        this.pid = pid;
        this.bt = bt;
        this.art = art;
    }
}

public class SJF_PREEMPTIVE
{
    // Method to find the waiting time for all
    // processes
    static void findWaitingTime(Process proc[], int n, int wt[])
    {
        int rt[] = new int[n];

        // Copy the burst time into rt[]
        for (int i = 0; i < n; i++)
            rt[i] = proc[i].bt;

        int complete = 0, t = 0, minm = Integer.MAX_VALUE;
        int shortest = 0, finish_time;
        boolean check = false;

        // Process until all processes gets
        // completed
        while (complete != n) {

            // Find process with minimum
            // remaining time among the
            // processes that arrives till the
            // current time`
            for (int j = 0; j < n; j++)
            {
                if ((proc[j].art <= t) &&
                        (rt[j] < minm) && rt[j] > 0) {
                    minm = rt[j];
                    shortest = j;
                    check = true;
                }
            }

            if (check == false) {
                t++;
                continue;
```

```
            }

            // Reduce remaining time by one
            rt[shortest]--;

            // Update minimum
            minm = rt[shortest];
            if (minm == 0)
                minm = Integer.MAX_VALUE;

            // If a process gets completely
            // executed
            if (rt[shortest] == 0) {

                // Increment complete
                complete++;
                check = false;

                // Find finish time of current
                // process
                finish_time = t + 1;

                // Calculate waiting time
                wt[shortest] = finish_time -
                        proc[shortest].bt -
                        proc[shortest].art;

                if (wt[shortest] < 0)
                    wt[shortest] = 0;
            }
            // Increment time
            t++;
        }
    }

    // Method to calculate turn around time
    static void findTurnAroundTime(Process proc[], int n,
                                   int wt[], int tat[])
    {
        // calculating turnaround time by adding
        // bt[i] + wt[i]
        for (int i = 0; i < n; i++)
            tat[i] = proc[i].bt + wt[i];
    }

    // Method to calculate average time
    static void findavgTime(Process proc[], int n)
    {
        int wt[] = new int[n], tat[] = new int[n];
        int total_wt = 0, total_tat = 0;

        // Function to find waiting time of all
        // processes
        findWaitingTime(proc, n, wt);

        // Function to find turn around time for
        // all processes
        findTurnAroundTime(proc, n, wt, tat);

        // Display processes along with all
        // details
```

```
        System.out.println("Processes " +
                "  Burst time " +
                "  Waiting time " +
                "  Turn around time");

        // Calculate total waiting time and
        // total turnaround time
        for (int i = 0; i < n; i++) {
            total_wt = total_wt + wt[i];
            total_tat = total_tat + tat[i];
            System.out.println("  " + proc[i].pid + "\t\t"
                    + proc[i].bt + "\t\t " + wt[i]
                    + "\t\t" + tat[i]);
        }

        System.out.println("Average waiting time = " +
                (float)total_wt / (float)n);
        System.out.println("Average turn around time = " +
                (float)total_tat / (float)n);
    }

    // Driver Method
    public static void main(String[] args)
    {
        Process proc[] = { new Process(1, 6, 1),
                new Process(2, 8, 1),
                new Process(3, 7, 2),
                new Process(4, 3, 3)};

        findavgTime(proc, proc.length);
    }
}
```

Round Robin with zeroth AT

```
package scheduling_algos;
//With zeroth arrival time
import java.util.Scanner;
public class RoundRobin
{
    public static void main(String args[])
    {
        int n,i,qt,count=0,temp,sq=0,bt[],wt[],tat[],rem_bt[];
        float awt=0,atat=0;
        bt = new int[10];
        wt = new int[10];
        tat = new int[10];
        rem_bt = new int[10];
        Scanner s=new Scanner(System.in);
        System.out.print("Enter the number of process = ");
        n = s.nextInt();
        System.out.print("Enter the burst time of the process\n");
        for (i=0;i<n;i++)
        {
            System.out.print("P"+i+" = ");
            bt[i] = s.nextInt();
            rem_bt[i] = bt[i];
        }
        System.out.print("Enter the quantum time: ");
        qt = s.nextInt();
```

```
        while(true)
        {
            for (i=0,count=0;i<n;i++)
            {
                temp = qt;
                if(rem_bt[i] == 0)
                {
                    count++;
                    continue;
                }
                if(rem_bt[i]>qt)
                    rem_bt[i]= rem_bt[i] - qt;
                else
                if(rem_bt[i]>=0)
                {
                    temp = rem_bt[i];
                    rem_bt[i] = 0;
                }
                sq = sq + temp;
                tat[i] = sq;
            }
            if(n == count)
                break;
        }
        System.out.print("------------------------------------------------
------------------------------");
        System.out.print("\nProcess\t      Burst Time\t       Turnaround
Time\t          Waiting Time\n");
        System.out.print("------------------------------------------------
------------------------------");
        for(i=0;i<n;i++)
        {
            wt[i]=tat[i]-bt[i];
            awt=awt+wt[i];
            atat=atat+tat[i];
            System.out.print("\n "+(i+1)+"\t "+bt[i]+"\t\t "+tat[i]+"\t\t
"+wt[i]+"\n");
        }
        awt=awt/n;
        atat=atat/n;
        System.out.println("\nAverage waiting Time = "+awt+"\n");
        System.out.println("Average turnaround time = "+atat);
    }
}
```

ROUND ROBIN DIFF AT

How to implement in a programming language

1. Declare arrival[], burst[], wait[], turn[] arrays and initialize them. Also declare a timer variable and initialize it to zero. To sustain the original burst array create another array (temp_burst[]) and copy all the values of burst array in it.

2. To keep a check we create another array of bool type which keeps the record of whether a process is completed or not. we also need to maintain a queue array which contains the process

indices (initially the array is filled with 0).

3. Now we increment the timer variable until the first process arrives and when it does, we add the process index to the queue array

4. Now we execute the first process until the time quanta and during that time quanta, we check whether any other process has arrived or not and if it has then we add the index in the queue (by calling the fxn. queueUpdation()).

5. Now, after doing the above steps if a process has finished, we store its exit time and execute the next process in the queue array. Else, we move the currently executed process at the end of the queue (by calling another fxn. queueMaintainence()) when the time slice expires.

6. The above steps are then repeated until all the processes have been completely executed. If a scenario arises where there are some processes left but they have not arrived yet, then we shall wait and the CPU will remain idle during this interval.

```
package scheduling_algos;

//JAVA Program for implementing
//Round Robin Algorithm
// code by Sparsh_cbs
import java.util.*;

public class RoundRobin_diff_AT{
    private static Scanner inp = new Scanner(System.in);
    //Driver Code
    public static void main(String[] args){
        int n,tq, timer = 0, maxProccessIndex = 0;
        float avgWait = 0, avgTT = 0;
        System.out.print("\nEnter the time quanta : ");
        tq = inp.nextInt();
        System.out.print("\nEnter the number of processes : ");
        n = inp.nextInt();
        int arrival[] = new int[n];
        int burst[] = new int[n];
        int wait[] = new int[n];
        int turn[] = new int[n];
        int queue[] = new int[n];
        int temp_burst[] = new int[n];
        boolean complete[] = new boolean[n];

        System.out.print("\nEnter the arrival time of the processes : ");
        for(int i = 0; i < n; i++)
            arrival[i] = inp.nextInt();

        System.out.print("\nEnter the burst time of the processes : ");
        for(int i = 0; i < n; i++){
            burst[i] = inp.nextInt();
            temp_burst[i] = burst[i];
        }

        for(int i = 0; i < n; i++){ //Initializing the queue and complete
array
            complete[i] = false;
            queue[i] = 0;
        }
```

```
        while(timer < arrival[0]) //Incrementing Timer until the first
process arrives
            timer++;
        queue[0] = 1;

        while(true){
            boolean flag = true;
            for(int i = 0; i < n; i++){
                if(temp_burst[i] != 0){
                    flag = false;
                    break;
                }
            }
            if(flag)
                break;

            for(int i = 0; (i < n) && (queue[i] != 0); i++){
                int ctr = 0;
                while((ctr < tq) && (temp_burst[queue[0]-1] > 0)){
                    temp_burst[queue[0]-1] -= 1;
                    timer += 1;
                    ctr++;

                    //Updating the ready queue until all the processes
arrive
                    checkNewArrival(timer, arrival, n, maxProccessIndex,
queue);
                }
                if((temp_burst[queue[0]-1] == 0) && (complete[queue[0]-1]
== false)){
                    turn[queue[0]-1] = timer;   //turn currently stores
exit times
                    complete[queue[0]-1] = true;
                }

                //checks whether or not CPU is idle
                boolean idle = true;
                if(queue[n-1] == 0){
                    for(int k = 0; k < n && queue[k] != 0; k++){
                        if(complete[queue[k]-1] == false){
                            idle = false;
                        }
                    }
                }
                else
                    idle = false;

                if(idle){
                    timer++;
                    checkNewArrival(timer, arrival, n, maxProccessIndex,
queue);
                }

                //Maintaining the entries of processes after each premption
in the ready Queue
                queueMaintainence(queue,n);
            }
        }

        for(int i = 0; i < n; i++){
            turn[i] = turn[i] - arrival[i];
```

```
            wait[i] = turn[i] - burst[i];
        }

        System.out.print("\nProgram No.\tArrival Time\tBurst Time\tWait
Time\tTurnAround Time"
                + "\n");
        for(int i = 0; i < n; i++){
            System.out.print(i+1+"\t\t"+arrival[i]+"\t\t"+burst[i]
                    +"\t\t"+wait[i]+"\t\t"+turn[i]+ "\n");
        }
        for(int i =0; i< n; i++){
            avgWait += wait[i];
            avgTT += turn[i];
        }
        System.out.print("\nAverage wait time : "+(avgWait/n)
                +"\nAverage Turn Around Time : "+(avgTT/n));
    }
    public static void queueUpdation(int queue[],int timer,int
arrival[],int n, int maxProccessIndex){
        int zeroIndex = -1;
        for(int i = 0; i < n; i++){
            if(queue[i] == 0){
                zeroIndex = i;
                break;
            }
        }
        if(zeroIndex == -1)
            return;
        queue[zeroIndex] = maxProccessIndex + 1;
    }

    public static void checkNewArrival(int timer, int arrival[], int n, int
maxProccessIndex,int queue[]){
        if(timer <= arrival[n-1]){
            boolean newArrival = false;
            for(int j = (maxProccessIndex+1); j < n; j++){
                if(arrival[j] <= timer){
                    if(maxProccessIndex < j){
                        maxProccessIndex = j;
                        newArrival = true;
                    }
                }
            }
            if(newArrival) //adds the index of the arriving process(if any)
                queueUpdation(queue,timer,arrival,n, maxProccessIndex);
        }
    }

    public static void queueMaintainence(int queue[], int n){

        for(int i = 0; (i < n-1) && (queue[i+1] != 0) ; i++){
            int temp = queue[i];
            queue[i] = queue[i+1];
            queue[i+1] = temp;
        }
    }
}
```

PRIORITY ZEROTH ARRIVAL

```
// Java program for implementation of FCFS
// scheduling
import java.util.*;

class Pro
{
    int pid; // Pro ID
    int bt; // CPU Burst time required
    int priority; // Priority of this Pro
    Pro(int pid, int bt, int priority)
    {
        this.pid = pid;
        this.bt = bt;
        this.priority = priority;
    }
    public int prior() {
        return priority;
    }
}


public class Priority
{
    public void findWaitingTime(Pro proc[], int n,
                                int wt[])
    {

        // waiting time for first Pro is 0
        wt[0] = 0;

        // calculating waiting time
        for (int i = 1; i < n ; i++ )
            wt[i] = proc[i - 1].bt + wt[i - 1] ;
    }

    // Function to calculate turn around time
    public void findTurnAroundTime( Pro proc[], int n, int wt[], int tat[])
    {
        // calculating turnaround time by adding
        // bt[i] + wt[i]
        for (int i = 0; i < n ; i++)
            tat[i] = proc[i].bt + wt[i];
    }

    // Function to calculate average time
    public void findavgTime(Pro proc[], int n)
    {
        int wt[] = new int[n], tat[] = new int[n], total_wt = 0, total_tat
= 0;

        // Function to find waiting time of all Processes
        findWaitingTime(proc, n, wt);

        // Function to find turn around time for all Processes
        findTurnAroundTime(proc, n, wt, tat);

        // Display Processes along with all details
        System.out.print("\nProes Burst time Waiting time Turn around
time\n");
```

```
        // Calculate total waiting time and total turn
        // around time
        for (int i = 0; i < n; i++)
        {
            total_wt = total_wt + wt[i];
            total_tat = total_tat + tat[i];
            System.out.print(" " + proc[i].pid + "\t\t" + proc[i].bt + "\t
" + wt[i] + "\t\t " + tat[i] + "\n");
        }

        System.out.print("\nAverage waiting time = "
                +(float)total_wt / (float)n);
        System.out.print("\nAverage turn around time = "+(float)total_tat /
(float)n);
    }

    public void priorityScheduling(Pro proc[], int n)
    {

        // Sort Processes by priority
        Arrays.sort(proc, new Comparator<Pro>() {
            @Override
            public int compare(Pro a, Pro b) {
                return b.prior() - a.prior();
            }
        });
        System.out.print("Order in which Processes gets executed \n");
        for (int i = 0 ; i < n; i++)
            System.out.print(proc[i].pid + " ") ;

        findavgTime(proc, n);
    }

    // Driver code
    public static void main(String[] args)
    {
        Priority ob=new Priority();
        int n = 3;
        Pro proc[] = new Pro[n];
        proc[0] = new Pro(1, 10, 2);
        proc[1] = new Pro(2, 5, 0);
        proc[2] = new Pro(3, 8, 1);
        ob.priorityScheduling(proc, n);
    }
}

// This code is contributed by rahulpatil07109.
```

PRIORITY DIFF ARRIVAL

```
// Java implementation for Priority Scheduling with
//Different Arrival Time priority scheduling
import java.util.*;

/// Data Structure
class Process {
    int at, bt, pri, pno;
    Process(int pno, int at, int bt, int pri)
    {
```

```
        this.pno = pno;
        this.pri = pri;
        this.at = at;
        this.bt = bt;
    }
}

/// Gantt chart structure
class GChart {
    // process number, start time, complete time,
    // turn around time, waiting time
    int pno, stime, ctime, wtime, ttime;
}

// user define comparative method (first arrival first serve,
// if arrival time same then heigh priority first)
class MyComparator implements Comparator {

    public int compare(Object o1, Object o2)
    {

        Process p1 = (Process)o1;
        Process p2 = (Process)o2;
        if (p1.at < p2.at)
            return (-1);

        else if (p1.at == p2.at && p1.pri > p2.pri)
            return (-1);

        else
            return (1);
    }
}


// class to find Gantt chart
class FindGantChart {
    void findGc(LinkedList queue)
    {

        // initial time = 0
        int time = 0;

        // priority Queue sort data according
        // to arrival time or priority (ready queue)
        TreeSet prique = new TreeSet(new MyComparator());

        // link list for store processes data
        LinkedList result = new LinkedList();

        // process in ready queue from new state queue
        while (queue.size() > 0)
            prique.add((Process)queue.removeFirst());

        Iterator it = prique.iterator();

        // time set to according to first process
        time = ((Process)prique.first()).at;

        // scheduling process
        while (it.hasNext()) {
```

```
            // dispatcher dispatch the
            // process ready to running state
            Process obj = (Process)it.next();

            GChart gc1 = new GChart();
            gc1.pno = obj.pno;
            gc1.stime = time;
            time += obj.bt;
            gc1.ctime = time;
            gc1.ttime = gc1.ctime - obj.at;
            gc1.wtime = gc1.ttime - obj.bt;

            /// store the exxtreted process
            result.add(gc1);
        }

        // create object of output class and call method
        new ResultOutput(result);
    }
}
```

BANKERS ALGO

```
// Java program to illustrate Banker's Algorithm
import java.util.*;

class bankersalgo
{

    // Number of processes
    static int P = 5;

    // Number of resources
    static int R = 3;

    // Function to find the need of each process
    static void calculateNeed(int need[][], int maxm[][],
                              int allot[][])
    {
        // Calculating Need of each P
        for (int i = 0 ; i < P ; i++)
            for (int j = 0 ; j < R ; j++)

                // Need of instance = maxm instance -
                //           allocated instance
                need[i][j] = maxm[i][j] - allot[i][j];
    }

    // Function to find the system is in safe state or not
    static boolean isSafe(int processes[], int avail[], int maxm[][],
                          int allot[][])
    {
        int [][]need = new int[P][R];

        // Function to calculate need matrix
        calculateNeed(need, maxm, allot);

        // Mark all processes as infinish
        boolean []finish = new boolean[P];
```

```
        // To store safe sequence
        int []safeSeq = new int[P];

        // Make a copy of available resources
        int []work = new int[R];
        for (int i = 0; i < R ; i++)
            work[i] = avail[i];

        // While all processes are not finished
        // or system is not in safe state.
        int count = 0;
        while (count < P)
        {
            // Find a process which is not finish and
            // whose needs can be satisfied with current
            // work[] resources.
            boolean found = false;
            for (int p = 0; p < P; p++)
            {
                // First check if a process is finished,
                // if no, go for next condition
                if (finish[p] == false)
                {
                    // Check if for all resources of
                    // current P need is less
                    // than work
                    int j;
                    for (j = 0; j < R; j++)
                        if (need[p][j] > work[j])
                            break;

                    // If all needs of p were satisfied.
                    if (j == R)
                    {
                        // Add the allocated resources of
                        // current P to the available/work
                        // resources i.e.free the resources
                        for (int k = 0 ; k < R ; k++)
                            work[k] += allot[p][k];

                        // Add this process to safe sequence.
                        safeSeq[count++] = p;

                        // Mark this p as finished
                        finish[p] = true;

                        found = true;
                    }
                }
            }

            // If we could not find a next process in safe
            // sequence.
            if (found == false)
            {
                System.out.print("System is not in safe state");
                return false;
            }
        }
```

```
        // If system is in safe state then
        // safe sequence will be as below
        System.out.print("System is in safe state.\nSafe"
                +" sequence is: ");
        for (int i = 0; i < P ; i++)
            System.out.print(safeSeq[i] + " ");

        return true;
    }

    // Driver code
    public static void main(String[] args)
    {
        int processes[] = {0, 1, 2, 3, 4};

        // Available instances of resources
        int avail[] = {3, 3, 2};

        // Maximum R that can be allocated
        // to processes
        int maxm[][] = {{7, 5, 3},
                {3, 2, 2},
                {9, 0, 2},
                {4, 2, 2},
                {5, 3, 3}};

        // Resources allocated to processes
        int allot[][] = {{0, 1, 0},
                {2, 0, 0},
                {3, 0, 2},
                {2, 1, 1},
                {0, 0, 2}};

        // Check system is in safe state or not
        isSafe(processes, avail, maxm, allot);
    }
}

// This code has been contributed by 29AjayKumar
```