First Come First serve with different Arrival Times

```
import java.util.Arrays;

class Process {
   String name;
   int arrivalTime;
   int burstTime;

   public Process(String name, int arrivalTime, int burstTime) {
      this.name = name;
      this.arrivalTime = arrivalTime;
      this.burstTime = burstTime;
   }
}

public class FCFSScheduler {
   public static void main(String[] args) {
      // Create an array of processes with different arrival times
      Process[] processes = {
            new Process("P1", 0, 5),
            new Process("P2", 2, 3),
            new Process("P3", 4, 6),
            new Process("P4", 6, 2)
      };

      fcfsScheduling(processes);
   }

   public static void fcfsScheduling(Process[] processes) {
      int currentTime = 0;
      int totalProcesses = processes.length;
      int[] completionTimes = new int[totalProcesses];
      int[] waitingTimes = new int[totalProcesses];
      int[] turnaroundTimes = new int[totalProcesses];

      // Sort the processes array based on arrival time
      Arrays.sort(processes, (p1, p2) -> p1.arrivalTime - p2.arrivalTime);

      for (int i = 0; i < totalProcesses; i++) {
         Process currentProcess = processes[i];

         // Calculate waiting time
         waitingTimes[i] = currentTime - currentProcess.arrivalTime;
```

```
            // Calculate completion time
            completionTimes[i] = currentTime + currentProcess.burstTime;

            // Calculate turnaround time
            turnaroundTimes[i] = completionTimes[i] - currentProcess.arrivalTime;

            // Move the current time forward
            currentTime += currentProcess.burstTime;
        }

        // Print process details
        System.out.println("Process\tArrival Time\tBurst Time\tCompletion Time\tWaiting
Time\tTurnaround Time");
        for (int i = 0; i < totalProcesses; i++) {
            Process currentProcess = processes[i];
            System.out.println(currentProcess.name + "\t\t" +
                    currentProcess.arrivalTime + "\t\t" +
                    currentProcess.burstTime + "\t\t" +
                    completionTimes[i] + "\t\t" +
                    waitingTimes[i] + "\t\t" +
                    turnaroundTimes[i]);
        }
    }
}
```

Sample Output

```
Process Arrival Time    Burst Time      Completion Time     Waiting Time    Turnaround Time
P1              0               5               5               0               5
P2              2               3               8               3               6
P3              4               6               14              4               10
P4              6               2               16              8               10
```

Preemptive SJF

```
import java.util.*;

class Process {
    int processId;
    int arrivalTime;
    int burstTime;
    int remainingTime;

    public Process(int processId, int arrivalTime, int burstTime) {
        this.processId = processId;
        this.arrivalTime = arrivalTime;
        this.burstTime = burstTime;
        this.remainingTime = burstTime;
    }
}

public class PreemptiveSJF {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of processes: ");
        int numProcesses = scanner.nextInt();

        // Create an array to store the processes
        Process[] processes = new Process[numProcesses];

        // Input process details
        for (int i = 0; i < numProcesses; i++) {
            System.out.println("Enter details for Process " + (i + 1) + ":");
            System.out.print("Arrival Time: ");
            int arrivalTime = scanner.nextInt();
            System.out.print("Burst Time: ");
            int burstTime = scanner.nextInt();
            processes[i] = new Process(i + 1, arrivalTime, burstTime);
        }

        // Sort the processes based on arrival time
        Arrays.sort(processes, Comparator.comparingInt(p -> p.arrivalTime));

        // Execute processes
        executeProcesses(processes);

        scanner.close();
    }

    public static void executeProcesses(Process[] processes) {
        int numProcesses = processes.length;
        int currentTime = 0;
        int completedProcesses = 0;
        ArrayList<Integer> executionOrder = new ArrayList<>();

        while (completedProcesses < numProcesses) {
            int shortestJobIndex = -1;
            int shortestJobRemainingTime = Integer.MAX_VALUE;

            // Find the process with the shortest remaining time
            for (int i = 0; i < numProcesses; i++) {
                Process process = processes[i];

                if (process.arrivalTime <= currentTime && process.remainingTime < shortestJobRemainingTime
&& process.remainingTime > 0) {
                    shortestJobIndex = i;
                    shortestJobRemainingTime = process.remainingTime;
                }
            }

            if (shortestJobIndex == -1) {
                currentTime++;
                continue;
            }

            Process shortestJob = processes[shortestJobIndex];
            shortestJob.remainingTime--;
            currentTime++;

            // Check if the process has completed execution
            if (shortestJob.remainingTime == 0) {
                completedProcesses++;
                executionOrder.add(shortestJob.processId);
            }
        }

        System.out.println("Execution Order: " + executionOrder);
    }
}
```

Sample Output

```
Enter the number of processes: 3
Enter details for Process 1:
Arrival Time: 3
Burst Time: 2
Enter details for Process 2:
Arrival Time: 1
Burst Time: 4
Enter details for Process 3:
Arrival Time: 2
Burst Time: 5
Execution Order: [2, 1, 3]
```

Non-Preemptive SJF

```
import java.util.*;

class Process {
   int pid;
   int arrivalTime;
   int burstTime;

   Process(int pid, int arrivalTime, int burstTime) {
      this.pid = pid;
      this.arrivalTime = arrivalTime;
      this.burstTime = burstTime;
   }
}

public class SJF {
   public static void main(String[] args) {
      // Create a list of processes
      List<Process> processes = new ArrayList<>();
      processes.add(new Process(1, 0, 5));
      processes.add(new Process(2, 1, 3));
      processes.add(new Process(3, 2, 8));
      processes.add(new Process(4, 3, 6));

      // Sort the processes based on arrival time
      Collections.sort(processes, Comparator.comparingInt(p -> p.arrivalTime));

      // Execute the processes
      int currentTime = 0;
      for (Process p : processes) {
         // Wait for the process to arrive if necessary
         if (p.arrivalTime > currentTime) {
            currentTime = p.arrivalTime;
         }

         // Execute the process
         System.out.println("Executing process " + p.pid + " at time " + currentTime);
         currentTime += p.burstTime;
      }
   }
}
```

Sample Output

```
Executing process 1 at time 0
Executing process 2 at time 5
Executing process 3 at time 8
Executing process 4 at time 16
```

Preemptive Priority

```
import java.util.*;

class Process {
    int processId;
    int arrivalTime;
    int priority;
    int burstTime;

    public Process(int processId, int arrivalTime, int priority, int burstTime) {
        this.processId = processId;
        this.arrivalTime = arrivalTime;
        this.priority = priority;
        this.burstTime = burstTime;
    }
}

public class PreemptivePriorityScheduling {
    public static void main(String[] args) {
        // Create a list of processes
        List<Process> processes = new ArrayList<>();
        processes.add(new Process(1, 0, 3, 6));
        processes.add(new Process(2, 2, 1, 4));
        processes.add(new Process(3, 3, 4, 2));
        processes.add(new Process(4, 5, 2, 8));

        // Sort processes based on arrival time
        Collections.sort(processes, Comparator.comparingInt(p -> p.arrivalTime));

        // Create a priority queue to store the ready processes
        PriorityQueue<Process> readyQueue = new PriorityQueue<>(Comparator.comparingInt(p ->
p.priority));

        int currentTime = 0;

        while (!processes.isEmpty() || !readyQueue.isEmpty()) {
            // Check if there are any arriving processes
            while (!processes.isEmpty() && processes.get(0).arrivalTime <= currentTime) {
                readyQueue.add(processes.remove(0));
            }

            if (readyQueue.isEmpty()) {
                currentTime++;
                continue;
            }

            Process currentProcess = readyQueue.poll();
            System.out.println("Executing process " + currentProcess.processId + " at time " + currentTime);

            currentProcess.burstTime--;

            if (currentProcess.burstTime > 0) {
                // Add the process back to the ready queue
                readyQueue.add(currentProcess);
            }

            currentTime++;
        }
    }
}
```

Sample Output:

```
Executing process 1 at time 0
Executing process 1 at time 1
Executing process 2 at time 2
Executing process 2 at time 3
Executing process 2 at time 4
Executing process 2 at time 5
Executing process 4 at time 6
Executing process 4 at time 7
Executing process 4 at time 8
Executing process 4 at time 9
Executing process 4 at time 10
Executing process 4 at time 11
Executing process 4 at time 12
Executing process 4 at time 13
Executing process 1 at time 14
Executing process 1 at time 15
Executing process 1 at time 16
Executing process 1 at time 17
Executing process 3 at time 18
Executing process 3 at time 19
```

Non-Premptive Priority

```
import java.util.ArrayList;
import java.util.Collections;

class Process implements Comparable<Process> {
    private int id;
    private int arrivalTime;
    private int burstTime;
    private int priority;

    public Process(int id, int arrivalTime, int burstTime, int priority) {
        this.id = id;
        this.arrivalTime = arrivalTime;
        this.burstTime = burstTime;
        this.priority = priority;
    }

    public int getId() {
        return id;
    }

    public int getArrivalTime() {
        return arrivalTime;
    }

    public int getBurstTime() {
        return burstTime;
    }

    public int getPriority() {
        return priority;
    }

    @Override
    public int compareTo(Process other) {
        if (this.priority == other.priority)
            return this.arrivalTime - other.arrivalTime;
        return this.priority - other.priority;
    }
}

public class PriorityScheduling {
    public static void main(String[] args) {
        // Create processes
        ArrayList<Process> processes = new ArrayList<>();
        processes.add(new Process(1, 0, 8, 3));
        processes.add(new Process(2, 1, 4, 1));
        processes.add(new Process(3, 2, 9, 2));
        processes.add(new Process(4, 3, 5, 4));
        processes.add(new Process(5, 4, 2, 5));

        // Sort processes based on arrival time
        Collections.sort(processes);

        // Execute processes
        int currentTime = 0;
        for (Process process : processes) {
            // Wait if the process has not arrived yet
            if (currentTime < process.getArrivalTime())
                currentTime = process.getArrivalTime();

            System.out.println("Executing process " + process.getId() + " at time " + currentTime);

            // Update current time and execute the process
            currentTime += process.getBurstTime();
        }
    }
}
```

Sample Output

```
Executing process 2 at time 1
Executing process 3 at time 5
Executing process 1 at time 14
Executing process 4 at time 22
Executing process 5 at time 27
```

Round Robin

```
import java.util.LinkedList;
import java.util.Queue;

class Process {
   String name;
   int arrivalTime;
   int burstTime;
   int remainingTime;

   public Process(String name, int arrivalTime, int burstTime) {
      this.name = name;
      this.arrivalTime = arrivalTime;
      this.burstTime = burstTime;
      this.remainingTime = burstTime;
   }
}

public class RoundRobinScheduler {
   public static void main(String[] args) {
      // Create an array of processes with different arrival times
      Process[] processes = {
            new Process("P1", 0, 5),
            new Process("P2", 2, 3),
            new Process("P3", 4, 6),
            new Process("P4", 6, 2)
      };

      int quantum = 2; // Time quantum for round robin

      roundRobinScheduling(processes, quantum);
   }

   public static void roundRobinScheduling(Process[] processes, int quantum) {
      Queue<Process> readyQueue = new LinkedList<>();
      int currentTime = 0;
      int totalProcesses = processes.length;
      int completedProcesses = 0;

      while (completedProcesses < totalProcesses) {
         // Add processes to the ready queue when their arrival time is reached
         for (Process process : processes) {
            if (process.arrivalTime <= currentTime && !readyQueue.contains(process)) {
               readyQueue.add(process);
            }
          }

          if (readyQueue.isEmpty()) {
            currentTime++; // No process in the ready queue, move to the next time unit
            continue;
          }

          Process currentProcess = readyQueue.poll();

          System.out.println("Executing process " + currentProcess.name +
                " from time " + currentTime + " to " + (currentTime +
Math.min(currentProcess.remainingTime, quantum)));

          if (currentProcess.remainingTime <= quantum) {
            currentTime += currentProcess.remainingTime;
            currentProcess.remainingTime = 0;
            completedProcesses++;
          } else {
            currentTime += quantum;
            currentProcess.remainingTime -= quantum;
            readyQueue.add(currentProcess);
          }
        }
    }
}
```

Sample Output:

```
Executing process P1 from time 0 to 2
Executing process P1 from time 2 to 4
Executing process P2 from time 4 to 6
Executing process P1 from time 6 to 7
Executing process P3 from time 7 to 9
Executing process P2 from time 9 to 10
Executing process P4 from time 10 to 12
Executing process P1 from time 12 to 12
```