DELHI TECHNOLOGICAL UNIVERSITY

Operating Systems Lab File

MC – 301

Anish Sachdeva
DTU/2K16/MC/013

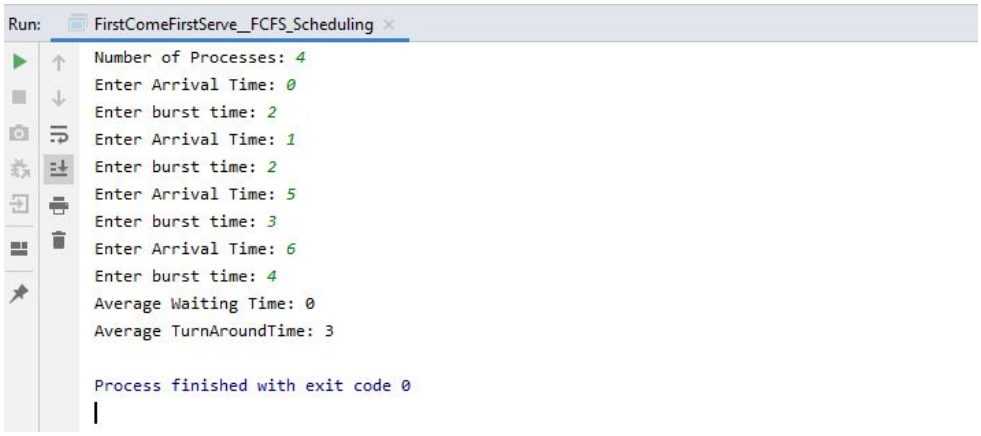
First Come First Serve Scheduling

Code

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class FirstComeFirstServe__FCFS_Scheduling {
   private static Scanner scanner = new Scanner(System.in);

   public static void main(String[] args) {
       System.out.print("Number of Processes: ");
       int numberOfProcesses = scanner.nextInt();
       List<JobProcess> processes = getProcesses(numberOfProcesses);
       computeTelemetry(processes);
       System.out.println("Average Waiting Time: " + averageWaitingTime(processes));
       System.out.println("Average TurnAroundTime: " + averageTurnAroundTime(processes));
   }

   private static void computeTelemetry(List<JobProcess> processes) {
       processes.sort(JobProcess::compareTo);
       processes.get(0).commencementTime = processes.get(0).arrivalTime;
       computeJobMetrics(processes.get(0));

       for (int index = 1 ; index < processes.size() ; index++) {
           JobProcess previousJob = processes.get(index - 1);
           JobProcess currentJob = processes.get(index);
           currentJob.commencementTime = Math.max(currentJob.arrivalTime,
previousJob.completionTime);
           computeJobMetrics(currentJob);
       }
   }

   private static void computeJobMetrics(JobProcess process) {
       process.completionTime = process.commencementTime + process.burstTime;
       process.turnAroundTime = process.completionTime - process.arrivalTime;
       process.waitingTime = process.completionTime - process.arrivalTime - process.burstTime;
   }

   private static long averageWaitingTime(List<JobProcess> processes) {
       long answer = 0;
       for (JobProcess process : processes) {
           answer += process.waitingTime;
       }
       return answer / processes.size() ;
   }

   private static long averageTurnAroundTime(List<JobProcess> processes) {
       long answer = 0;
       for (JobProcess process : processes) {
           answer += process.turnAroundTime;
       }
```

```
        return answer / processes.size() ;
    }

    private static List<JobProcess> getProcesses(int numberOfProcesses) {
        List<JobProcess> processes = new ArrayList<>();
        for (int index = 0 ; index < numberOfProcesses ; index++) {
            System.out.print("Enter Arrival Time: ");
            int arrivalTime = scanner.nextInt();
            System.out.print("Enter burst time: ");
            int burstTime = scanner.nextInt();
            processes.add(new JobProcess(arrivalTime, burstTime));
        }
        return processes;
    }

    private static class JobProcess implements Comparable<JobProcess> {
        final int arrivalTime;
        final int burstTime;
        int commencementTime;
        int completionTime;
        int turnAroundTime;
        int waitingTime;

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

        @Override
        public int compareTo(JobProcess jobProcess) {
            return Integer.compare(this.arrivalTime, jobProcess.arrivalTime);
        }

        @Override
        public String toString() {
            return "AT:{" + arrivalTime + "} BT:{" + burstTime + "} ComT:{" + commencementTime +
"} ComplT:{"
                    + completionTime + "} TAT:{" + turnAroundTime +"} WAT:{" + waitingTime +"}";
        }
    }
}
```

Snapshot


Run:
FirstComeFirstServe_FCFS_Scheduling
Number of Processes: 4
Enter Arrival Time: 0
Enter burst time: 2
Enter Arrival Time: 1
Enter burst time: 2
Enter Arrival Time: 5
Enter burst time: 3
Enter Arrival Time: 6
Enter burst time: 4
Average Waiting Time: 0
Average TurnAroundTime: 3
Process finished with exit code 0

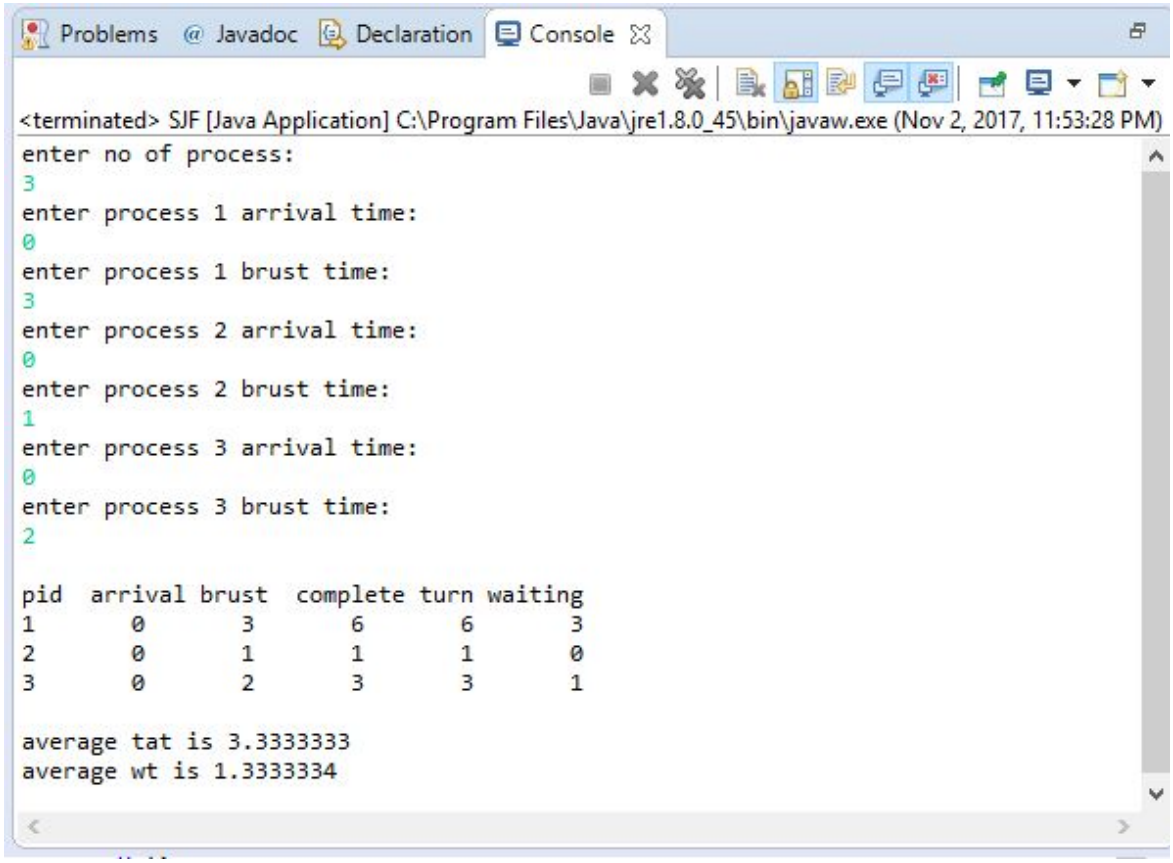
Shortest Job First Scheduling

Code

```
import java.util.*;

public class SJF {
      public static void main(String args[])
      {
            Scanner sc = new Scanner(System.in);
            System.out.println ("enter no of process:");
            int n = sc.nextInt();
            int pid[] = new int[n];
            int at[] = new int[n]; // at means arrival time
            int bt[] = new int[n]; // bt means burst time
            int ct[] = new int[n]; // ct means complete time
            int ta[] = new int[n]; // ta means turn around time
            int wt[] = new int[n];  //wt means waiting time
            int f[] = new int[n];  // f means it is flag it checks process is
completed or not
            int st=0, tot=0;
            float avgwt=0, avgta=0;

            for(int i=0;i<n;i++)
            {
                  System.out.println ("enter process " + (i+1) + " arrival
time:");
                  at[i] = sc.nextInt();
                  System.out.println ("enter process " + (i+1) + " brust
time:");
                  bt[i] = sc.nextInt();
                  pid[i] = i+1;
                  f[i] = 0;
            }

            boolean a = true;
            while(true)
            {
                  int c=n, min=999;
                  if (tot == n) // total no of process = completed process loop
will be terminated
                        break;

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

```
                        /*
                         * If i'th process arrival time <= system time and its
flag=0 and burst<min
                         * That process will be executed first
                         */
                        if ((at[i] <= st) && (f[i] == 0) && (bt[i]<min))
                        {
                              min=bt[i];
                              c=i;
                        }
                  }

                  /* If c==n means c value can not updated because no process
arrival time< system time so we increase the system time */
                  if (c==n)
                        st++;
                  else
                  {
                        ct[c]=st+bt[c];
                        st+=bt[c];
                        ta[c]=ct[c]-at[c];
                        wt[c]=ta[c]-bt[c];
                        f[c]=1;
                        tot++;
                  }
            }

            System.out.println("\npid  arrival brust  complete turn waiting");
            for(int i=0;i<n;i++)
            {
                  avgwt+= wt[i];
                  avgta+= ta[i];

System.out.println(pid[i]+"\t"+at[i]+"\t"+bt[i]+"\t"+ct[i]+"\t"+ta[i]+"\t"+wt[i
]);
            }
            System.out.println ("\naverage tat is "+ (float)(avgta/n));
            System.out.println ("average wt is "+ (float)(avgwt/n));
            sc.close();
      }
}
```

Snapshot


Problems
Javadoc
Declaration
Console
<terminated> SJF [Java Application] C:\Program Files\Java\jre1.8.0_45\bin\javaw.exe (Nov 2, 2017, 11:53:28 PM)
enter no of process:
3
enter process 1 arrival time:
0
enter process 1 brust time:
3
enter process 2 arrival time:
0
enter process 2 brust time:
1
enter process 3 arrival time:
0
enter process 3 brust time:
2
pid arrival brust complete turn waiting
1 0 3 6 6 3
2 0 1 1 1 0
3 0 2 3 3 1
average tat is 3.3333333
average wt is 1.3333334

Round Robin

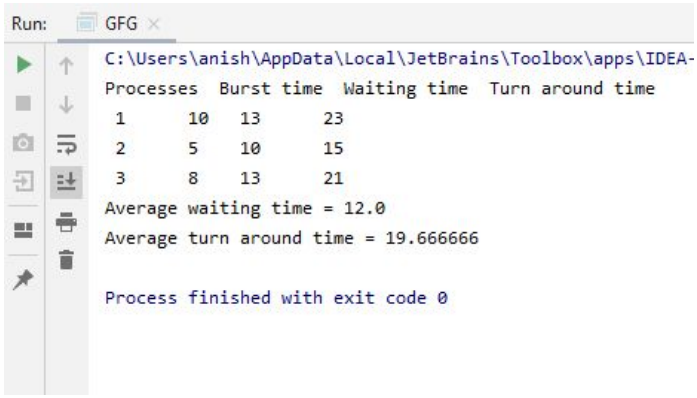
Code

```
// Java program for implementation of RR scheduling

public class GFG
{
	// Method to find the waiting time for all
	// processes
	static void findWaitingTime(int processes[], int n,
				int bt[], int wt[], int quantum)
	{
		// Make a copy of burst times bt[] to store remaining
		// burst times.
		int rem_bt[] = new int[n];
		for (int i = 0 ; i < n ; i++)
			rem_bt[i] = bt[i];

		int t = 0; // Current time

		// Keep traversing processes in round robin manner
		// until all of them are not done.
		while(true)
		{
			boolean done = true;

			// Traverse all processes one by one repeatedly
			for (int i = 0 ; i < n; i++)
			{
				// If burst time of a process is greater than 0
				// then only need to process further
				if (rem_bt[i] > 0)
				{
					done = false; // There is a pending process

					if (rem_bt[i] > quantum)
					{
						// Increase the value of t i.e. shows
						// how much time a process has been processed
						t += quantum;

						// Decrease the burst_time of current process
						// by quantum
						rem_bt[i] -= quantum;
					}

					// If burst time is smaller than or equal to
```

```
                        // quantum. Last cycle for this process
                        else
                        {
                               // Increase the value of t i.e. shows
                               // how much time a process has been processed
                               t = t + rem_bt[i];

                               // Waiting time is current time minus time
                               // used by this process
                               wt[i] = t - bt[i];

                               // As the process gets fully executed
                               // make its remaining burst time = 0
                               rem_bt[i] = 0;
                        }
                    }
                }

                // If all processes are done
                if (done == true)
                break;
        }
}

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

// Method to calculate average time
static void findavgTime(int processes[], int n, int bt[],
                                                                int quantum)
{
        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(processes, n, bt, wt, quantum);

        // Function to find turn around time for all processes
        findTurnAroundTime(processes, n, bt, 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 turn
        // around time
        for (int i=0; i<n; i++)
        {
               total_wt = total_wt + wt[i];
               total_tat = total_tat + tat[i];
               System.out.println(" " + (i+1) + "\t\t" + bt[i] +"\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 id's
        int processes[] = { 1, 2, 3};
        int n = processes.length;

        // Burst time of all processes
        int burst_time[] = {10, 5, 8};

        // Time quantum
        int quantum = 2;
        findavgTime(processes, n, burst_time, quantum);
    }
}
```

Snapshot



Priority Scheduling

Code

```
// 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);
	}
}
```

Snapshot

```
Process_no Start_time Complete_time Trun_Around_Time Wating_Time
1           1          4             3              0
2           4          9             7              2
3           9          10            7              6
4          10          17            13             6
5          17          21            16             12
Average Wating Time is : 5.2
Average Trun Around time is : 9.2
```

Banker’s Algorithm

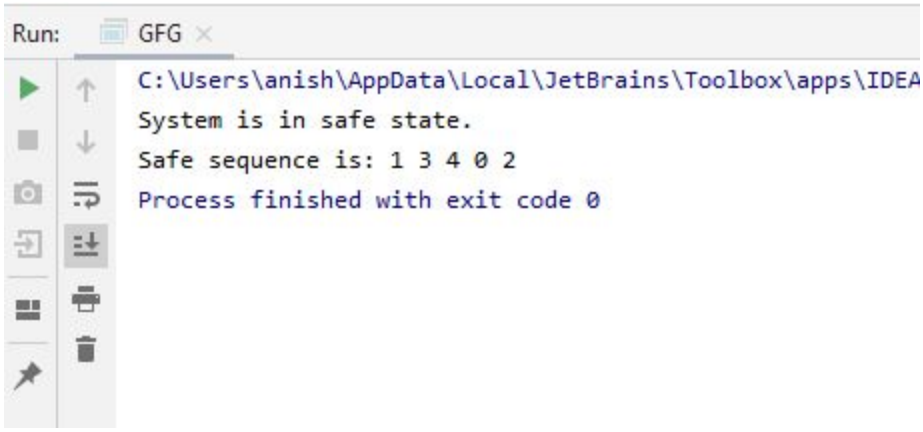
Code

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

class GFG
{

// 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},
                              {2, 2, 2},
                              {4, 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);
}
}
```

Snapshot


Run:
GFG
C:\Users\anish\AppData\Local\JetBrains\Toolbox\apps\IDEA
System is in safe state.
Safe sequence is: 1 3 4 0 2
Process finished with exit code 0

Shortest Seek Time First (SSTF) Disk Scheduling

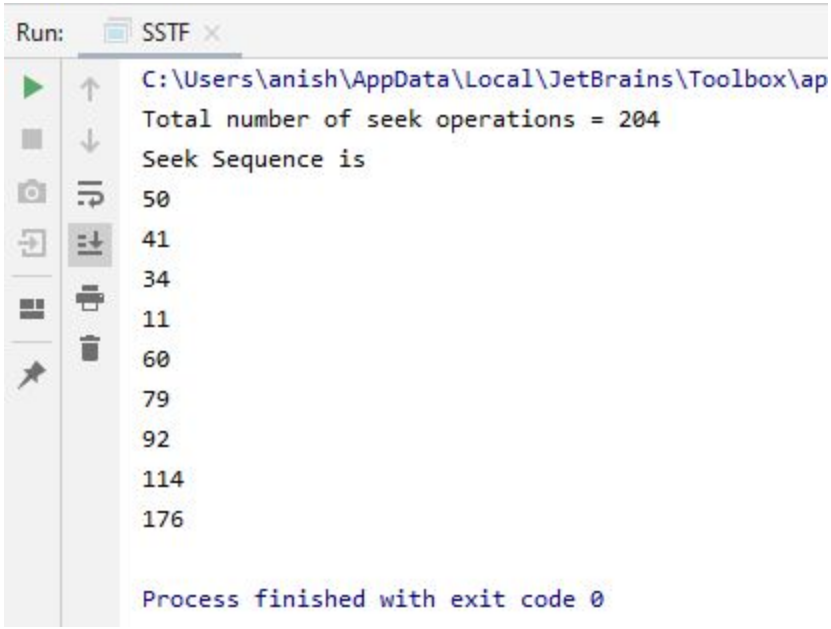
Code

```
class node {

	// represent difference between
	// head position and track number
	int distance = 0;

	// true if track has been accessed
	boolean accessed = false;
}

public class SSTF {

	// Calculates difference of each
	// track number with the head position
	public static void calculateDifference(int queue[],
                                               int head, node
diff[])

	{
		for (int i = 0; i < diff.length; i++)
			diff[i].distance = Math.abs(queue[i] - head);
	}

	// find unaccessed track
	// which is at minimum distance from head
	public static int findMin(node diff[])
	{
		int index = -1, minimum = Integer.MAX_VALUE;

		for (int i = 0; i < diff.length; i++) {
			if (!diff[i].accessed && minimum > diff[i].distance) {

				minimum = diff[i].distance;
				index = i;
			}
		}
		return index;
	}

	public static void shortestSeekTimeFirst(int request[],
```

```
int head)

	{
		if (request.length == 0)
			return;

		// create array of objects of class node
		node diff[] = new node[request.length];

		// initialize array
		for (int i = 0; i < diff.length; i++)

			diff[i] = new node();

		// count total number of seek operation
		int seek_count = 0;

		// stores sequence in which disk access is done
		int[] seek_sequence = new int[request.length + 1];

		for (int i = 0; i < request.length; i++) {

			seek_sequence[i] = head;
			calculateDifference(request, head, diff);

			int index = findMin(diff);

			diff[index].accessed = true;

			// increase the total count
			seek_count += diff[index].distance;

			// accessed track is now new head
			head = request[index];
		}

		// for last accessed track
		seek_sequence[seek_sequence.length - 1] = head;

		System.out.println("Total number of seek operations = "

+ seek_count);

		System.out.println("Seek Sequence is");

		// print the sequence
		for (int i = 0; i < seek_sequence.length; i++)
			System.out.println(seek_sequence[i]);
	}
```

```
	public static void main(String[] args)
	{
		// request array
		int arr[] = { 176, 79, 34, 60, 92, 11, 41, 114 };
		shortestSeekTimeFirst(arr, 50);
	}
}
```

Snapshot



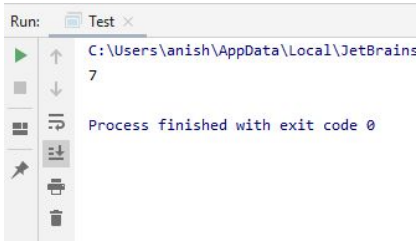
First In First Out (FIFO) Paging

Code

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


class Test
{
	// Method to find page faults using FIFO
	static int pageFaults(int pages[], int n, int capacity)
	{
		// To represent set of current pages. We use
		// an unordered_set so that we quickly check
		// if a page is present in set or not
		HashSet<Integer> s = new HashSet<>(capacity);

		// To store the pages in FIFO manner
		Queue<Integer> indexes = new LinkedList<>() ;

		// Start from initial page
		int page_faults = 0;
		for (int i=0; i<n; i++)
		{
			// Check if the set can hold more pages
			if (s.size() < capacity)
			{
				// Insert it into set if not present
				// already which represents page fault
				if (!s.contains(pages[i]))
				{
					s.add(pages[i]);

					// increment page fault
					page_faults++;

					// Push the current page into the queue
					indexes.add(pages[i]);
				}
			}

			// If the set is full then need to perform FIFO
			// i.e. remove the first page of the queue from
			// set and queue both and insert the current page
			else
			{
```

```
                        // Check if current page is not already
                        // present in the set
                        if (!s.contains(pages[i]))
                        {
                                //Pop the first page from the queue
                                int val = indexes.peek();

                                indexes.poll();

                                // Remove the indexes page
                                s.remove(val);

                                // insert the current page
                                s.add(pages[i]);

                                // push the current page into
                                // the queue
                                indexes.add(pages[i]);

                                // Increment page faults
                                page_faults++;
                        }
                }
        }

        return page_faults;
    }

    // Driver method
    public static void main(String args[])
    {
        int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,
                                    2, 3, 0, 3, 2};

        int capacity = 4;
        System.out.println(pageFaults(pages, pages.length, capacity));
    }
}
```

Snapshot



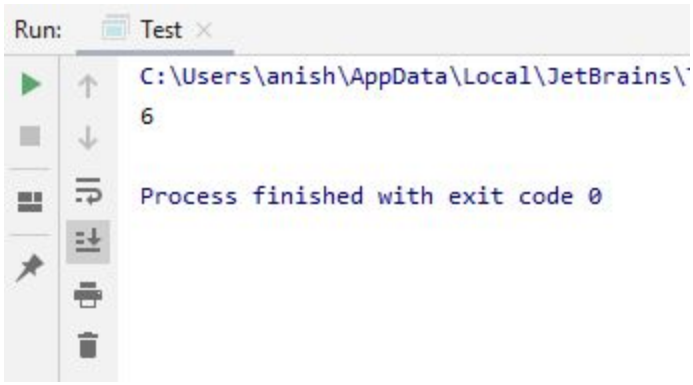
Least Recently Used (LRU) Paging Algorithm

Code

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;

class Test
{
	// Method to find page faults using indexes
	static int pageFaults(int pages[], int n, int capacity)
	{
		// To represent set of current pages. We use
		// an unordered_set so that we quickly check
		// if a page is present in set or not
		HashSet<Integer> s = new HashSet<>(capacity);

		// To store least recently used indexes
		// of pages.
		HashMap<Integer, Integer> indexes = new HashMap<>();

		// Start from initial page
		int page_faults = 0;
		for (int i=0; i<n; i++)
		{
			// Check if the set can hold more pages
			if (s.size() < capacity)
			{
				// Insert it into set if not present
				// already which represents page fault
				if (!s.contains(pages[i]))
				{
					s.add(pages[i]);

					// increment page fault
					page_faults++;
				}

				// Store the recently used index of
				// each page
				indexes.put(pages[i], i);
			}

			// If the set is full then need to perform lru
```

```
                // i.e. remove the least recently used page
                // and insert the current page
                else
                {
                        // Check if current page is not already
                        // present in the set
                        if (!s.contains(pages[i]))
                        {
                                // Find the least recently used pages
                                // that is present in the set
                                int lru = Integer.MAX_VALUE, val=Integer.MIN_VALUE;

                                Iterator<Integer> itr = s.iterator();

                                while (itr.hasNext()) {
                                        int temp = itr.next();
                                        if (indexes.get(temp) < lru)
                                        {
                                                lru = indexes.get(temp);
                                                val = temp;
                                        }
                                }

                                // Remove the indexes page
                                s.remove(val);
                        //remove lru from hashmap
                        indexes.remove(val);
                                // insert the current page
                                s.add(pages[i]);

                                // Increment page faults
                                page_faults++;
                        }

                        // Update the current page index
                        indexes.put(pages[i], i);
                }
        }

        return page_faults;
    }

    // Driver method
    public static void main(String args[])
    {
        int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
        int capacity = 4;
        System.out.println(pageFaults(pages, pages.length, capacity));
    }
}
```

Snapshot


Run:
Test
C:\Users\anish\AppData\Local\JetBrains\
6
Process finished with exit code 0