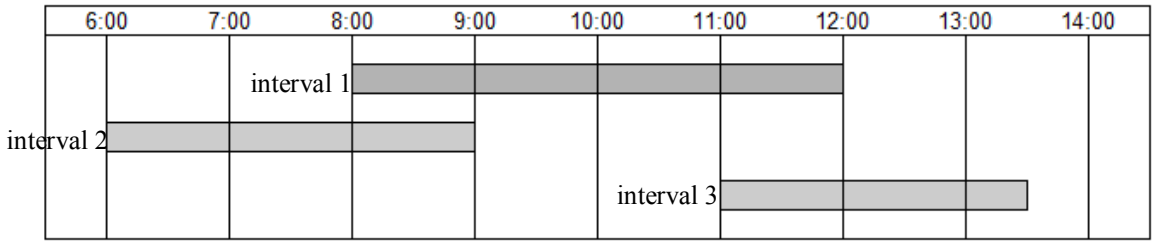
Examination 1
– "Manage time for tasks !" –

Introduction

We cannot do different tasks at the same time.
To manage time for our tasks, we use a "**schedule chart**" such as the following.

Figure 1: An example of **schedule chart**



Outline of this examination

Write programs which check "intervals" to make a proper schedule chart.

What is "interval" ?

"**Interval**" is a time range of each task.
It shows what time each task will start/end and how long it will take.
(e.g.) "Figure 1" has three "**intervals**" such as [08:00,12:00], [06:00,09:00], [11:00,13:30].)

In this examination, "**interval**" is defined as following;

<Definition of "interval">

- **"Interval"** has a "**start time**" and an "**end time**"
- "**Start time**"and "**end time**" have only hour and minute
  - You do not have to consider other information such as year, month, date or second
- The earliest time is "00:00", the latest time is "24:00"
- "24:00" is not equal to "00:00" on **the same day** but equal to "00:00" on **the next day**
- The "**start time**" must be earlier than, or the same as the "**end time**"
- The "**start time**" and "**end time**" are included within "**interval**" (in mathematics, it's called "closed interval")

(e.g.) [08:00,09:00] and [09:00,10:00] are overlapped because both include "09:00".

- You can assume that the given parameters are valid.

Therefore, no invalid interval will be generated in our scoring system.

* **"Interval class"** represents **"interval"**.（See List 3)
* You should use attached code skeleton.

*** Any classes or functions in the skeleton must not be changed.**

(name, variable number, parameters and return value)

* You can add new classes and/or functions if you need.
* Time should be printed in HH:mm format.

Problem 1

Multiple “**intervals**” are to be given as tasks.
Calculate the maximum number of overlapped “**intervals**” at the same time .

<Implementation>

Implement the method "**getMaxIntervalOverlapCount**" in the class "**Problem1**".

int Problem1#getMaxIntervalOverlapCount(List<Interval> intervals)

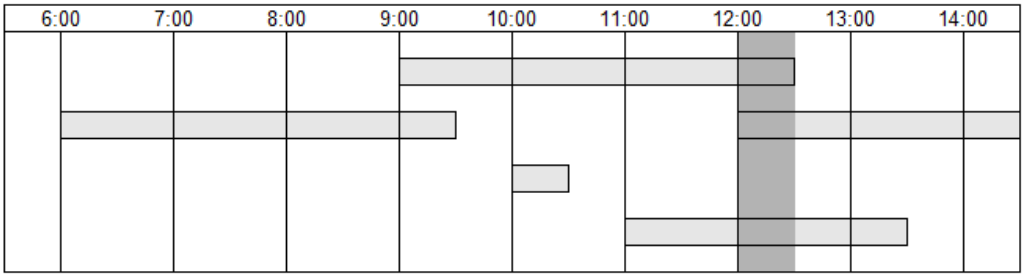
<Specifications>

- Return 0 if the argument is null or an empty list.
- The argument (a list of “**intervals**”) must not contain null.

<Examples>

In Figure 1, two “**intervals**” are overlapped at most. ⇒Return “2” for this case.
In Figure 2, three “**intervals**” are overlapped at most. ⇒Return “3” for this case.

Figure 2: Another example of **schedule chart**.



<Examples of test>
The class “Problem1”, which you have created in this problem, is to be tested as following.

```
@Test
public void testProblem1Usage() {
	Problem1 p = new Problem1();

	// example: Figure 1
	Interval interval1 = new Interval("08:00", "12:00");
	Interval interval2 = new Interval("06:00", "09:00");
	Interval interval3 = new Interval("11:00", "13:30");
	List<Interval> figure1 = Arrays.asList(interval1, interval2, interval3);
	assertThat(p.getMaxIntervalOverlapCount(figure1), is(2));

	// example: Figure 2
	List<Interval> figure2 = Arrays.asList(new Interval("09:00", "12:30"),
			new Interval("06:00", "09:30"), new Interval("12:00", "14:30"),
			new Interval("10:00", "10:30"), new Interval("11:00", "13:30"));
	assertThat(p.getMaxIntervalOverlapCount(figure2), is(3));
}
```

Problem 2

Multiple **“intervals”** are to be given as tasks.
Calculate the maximum work time (minutes) to assign to one worker.

<Implementation>

Implement the method “**int Problem2#getMaxWorkingTime(List<Interval> intervals)**”.

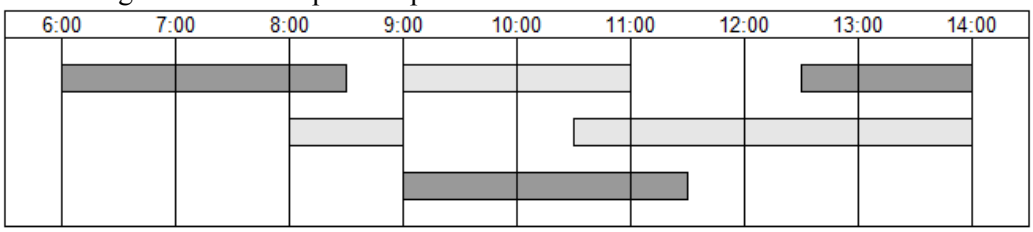
<Specifications>

- Return the maximum time to work on a task when one worker takes it.
- Time assignment unit shall be based on tasks. (i.e. Task assignment shall be either to complete the whole task, or to do nothing for it.)
- The length of the interval, time required to complete the task, is calculated by subtracting "end time" from "start time". (e.g. If [12:00, 13:00] is given, the length of the task is 60 min.)
- Return 0 if the argument is null or an empty list.
- The argument (a list of “**intervals**”) must not contain null.
- The number of “**intervals**” must not exceed 10,000.

<Examples>

In Figure 3 , work time is maximized when three tasks colored in dark grey, [“06:00”, “08:30”], [“09:00”, “11:30”], and [“12:30”, “14:00”], are assigned.
Therefore, the answer is 390 (minutes).

Figure 3: An example of input and answer for Problem2



Program Codes

Examination 1 : Skeleton and Interval Class

List 1: Problem 1 Skeleton Code.

```
package jp.co.wap.exam;

import java.util.List;

import jp.co.wap.exam.lib.Interval;

public class Problem1 {

	public int getMaxIntervalOverlapCount(List<Interval> intervals) {
		// TODO: Implement this method.
		return 0;
	}

}
```

List 2: Problem 2 Skeleton Code.

```
package jp.co.wap.exam;

import java.util.List;

import jp.co.wap.exam.lib.Interval;

public class Problem2 {

	public int getMaxWorkingTime(List<Interval> intervals) {
		// TODO: Implement this method.
		return 0;
	}

}
```

List 3: Interval Class Code.

```
package jp.co.wap.exam.lib;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * This class represents interval.
 * YOU MUST NOT MODIFY THIS CLASS. (Use this as it is)
 * You do not have to submit this class. (Interval class is to be provided in our scoring system.)
 */
public class Interval {

	/**
	 * It represents Time of Hour and Minute.
	 */
	private static class Time {
		final int hour;
		final int minute;

		public Time(int hour, int minute) {
			this.hour = hour;
			this.minute = minute;
		}
		@Override
		public String toString() {
			return String.format("%02d:%02d", hour, minute);
		}
		@Override
		public int hashCode() {
			return toString().hashCode();
		}
		@Override
		public boolean equals(Object obj) {
			if (!(obj instanceof Time)) {
				return false;
			}
			Time other = (Time) obj;
			return (this.hour == other.hour && this.minute == other.minute);
		}
	}









	/** Initial point of interval. */
	private final Time begin;
```

```
	/** Terminal point of interval. */
	private final Time end;

	/** Create interval from begin time (string) and end time (string). */
	public Interval(String begin, String end) {
		this.begin = toTime(begin);
		this.end = toTime(end);
	}
	/** Convert time format (string) to Time structure. */
	private static Time toTime(String timeFormatString) {
		Pattern p = Pattern.compile("(\\d?\\d):([0-5]\\d)");
		Matcher m = p.matcher(timeFormatString);
		if (!m.find()) {
			throw new IllegalArgumentException("invalid time format.");
		}
		int hour = Integer.parseInt(m.group(1));
		int minute = Integer.parseInt(m.group(2));

		return new Time(hour, minute);
	}
	/** Get interval begin string.*/
	public String getBegin() {
		return this.begin.toString();
	}
	/** Get interval end string. */
	public String getEnd() {
		return this.end.toString();
	}
	/** Get interval begin hour. */
	public int getBeginHour() {
		return this.begin.hour;
	}
	/** Get interval begin minute. */
	public int getBeginMinute() {
		return this.begin.minute;
	}
	/** Get interval end hour. */
	public int getEndHour() {
		return this.end.hour;
	}
	/** Get interval end minute. */
	public int getEndMinute() {
		return this.end.minute;
	}
	/**
	 * Get total time (minute) from "00:00" to initial point of interval.
	 * For example, it returns 150 when initial point of interval is "02:30".
	 **/
	public int getBeginMinuteUnit() {
		return getBeginHour() * 60 + getBeginMinute();
	}


	/**
	 * Get total time (minute) from "00:00" to terminal point of interval.
	 * For example, it returns 1440 when terminal point of interval is "24:00".
	 **/
```

```
	public int getEndMinuteUnit() {
		return getEndHour() * 60 + getEndMinute();
	}
	/**
	 * Get total time on interval.
	 * That is, it returns getEndMinuteUnit() minus getBeginMinuteUnit().
	 **/
	public int getIntervalMinute() {
		return getEndMinuteUnit() - getBeginMinuteUnit();
	}
	@Override
	public int hashCode() {
		return toString().hashCode();
	}
	@Override
	public boolean equals(Object obj) {
		if (!(obj instanceof Interval)) {
			return false;
		}
		Interval other = (Interval) obj;
		return (this.begin.equals(other.begin) && this.end.equals(other.end));
	}
	@Override
	public String toString() {
		return String.format("[%s-%s]", begin, end);
	}
}
```

Examination 2 : Implement This Skeleton Code

```
package jp.co.wap.exam;

/**
 * The Queue class represents an immutable first-in-first-out (FIFO) queue of objects.
 * @param <E>
 */
public class PersistentQueue<E> {
	/**
	 * requires default constructor.
	 */
	public PersistentQueue() {
		// TODO: implement if necessary
	}

	/**
	 * Returns the queue that adds an item into the tail of this queue without modifying this queue.
	 * <pre>
	 *   e.g.
	 *     When this queue represents the queue (2, 1, 2, 2, 6) and we enqueue the value 4 into this queue,
	 *     this method returns a new queue (2, 1, 2, 2, 6, 4)
	 *     and this object still represents the queue (2, 1, 2, 2, 6) .
	 * </pre>
	 * If the element e is null, throws IllegalArgumentException.
	 * @param e
	 * @return
	 * @throws IllegalArgumentException
	 */
	public PersistentQueue<E> enqueue(E e) {
		// TODO: implement this method
		return null;
	}

	/**
	 * Returns the queue that removes the object at the head of this queue without modifying this queue.
	 * <pre>
	 *   e.g.
	 *     When this queue represents the queue (7, 1, 3, 3, 5, 1) ,
	 *     this method returns a new queue (1, 3, 3, 5, 1)
	 *     and this object still represents the queue (7, 1, 3, 3, 5, 1) .
	 * </pre>
	 * If this queue is empty, throws java.util.NoSuchElementException.
	 * @return
	 * @throws java.util.NoSuchElementException
	 */
	public PersistentQueue<E> dequeue() {
		// TODO: implement this method
		return null;
	}

	/**
	 * Looks at the object which is the head of this queue without removing it from the queue.
	 * <pre>
	 *   e.g.
	 *     When this queue represents the queue (7, 1, 3, 3, 5, 1),
	 *     this method returns 7 and this object still represents the queue (7, 1, 3, 3, 5, 1)
	 * </pre>
	 * If the queue is empty, throws java.util.NoSuchElementException.
	 * @return
	 * @throws java.util.NoSuchElementException
	 */
	public E peek() {
		// TODO: implement this method
		return null;
	}

	/**
	 * Returns the number of objects in this queue.
	 * @return
	 */
	public int size() {
		// TODO: implement this method
		return 0;
	}
}
```

Examination 2 : Code Sample

```
package jp.co.wap.exam;

import java.util.ArrayList;
import java.util.List;
import java.util.NoSuchElementException;

/**
 * The Queue class represents an immutable first-in-first-out (FIFO) queue of objects.
 * @param <E>
 */
public class PersistentQueue<E> {
	private List<E> queue;

	/**
	 * requires default constructor.
	 */
	public PersistentQueue() {
		// modify this constructor if necessary, but do not remove default constructor
		queue = new ArrayList<E>();
	}
	private PersistentQueue(List<E> queue) {
		// modify or remove this constructor if necessary
		this.queue = queue;
	}
	// add other constructors if necessary

	/**
	 * Returns the queue that adds an item into the tail of this queue without modifying this queue.
	 * <pre>
	 *   e.g.
	 *     When this queue represents the queue (2, 1, 2, 2, 6) and we enqueue the value 4 into this queue,
	 *     this method returns a new queue (2, 1, 2, 2, 6, 4)
	 *     and this object still represents the queue (2, 1, 2, 2, 6) .
	 * </pre>
	 * If the element e is null, throws IllegalArgumentException.
	 * @param e
	 * @return
	 * @throws IllegalArgumentException
	 */
	public PersistentQueue<E> enqueue(E e) {
		// TODO: make this method faster
		if (e == null) {
			throw new IllegalArgumentException();
		}
		List<E> clone = new ArrayList<E>(queue);
		clone.add(e);
		return new PersistentQueue<E>(clone);
	}
```

```
    /**
     * Returns the queue that removes the object at the head of this queue without modifying this queue.
     * <pre>
     *   e.g.
     *      When this queue represents the queue (7, 1, 3, 3, 5, 1) ,
     *      this method returns a new queue (1, 3, 3, 5, 1)
     *      and this object still represents the queue (7, 1, 3, 3, 5, 1) .
     * </pre>
     * If this queue is empty, throws java.util.NoSuchElementException.
     * @return
     * @throws java.util.NoSuchElementException
     */
    public PersistentQueue<E> dequeue() {
        // TODO: make this method faster
        if (queue.isEmpty()) {
            throw new NoSuchElementException();
        }
        List<E> clone = new ArrayList<E>(queue);
        clone.remove(0);
        return new PersistentQueue<E>(clone);
    }

    /**
     * Looks at the object which is the head of this queue without removing it from the queue.
     * <pre>
     *   e.g.
     *      When this queue represents the queue (7, 1, 3, 3, 5, 1),
     *      this method returns 7 and this object still represents the queue (7, 1, 3, 3, 5, 1)
     * </pre>
     * If the queue is empty, throws java.util.NoSuchElementException.
     * @return
     * @throws java.util.NoSuchElementException
     */
    public E peek() {
        // modify this method if needed
        if (queue.isEmpty()) {
            throw new NoSuchElementException();
        }
        return queue.get(0);
    }

    /**
     * Returns the number of objects in this queue.
     * @return
     */
    public int size() {
        // modify this method if necessary
        return queue.size();
    }
}
```