3. An electric car that runs on batteries must be periodically recharged for a certain number of hours. The battery technology in the car requires that the charge time not be interrupted.

The cost for charging is based on the hour(s) during which the charging occurs. A rate table lists the 24 one-hour periods, numbered from 0 to 23, and the corresponding hourly cost for each period. The same rate table is used for each day. Each hourly cost is a positive integer. A sample rate table is given below.

Hour	Cost	Hour	Cost	Hour	Cost
0	50	8	150	16	200
1	60	9	150	17	200
2	160	10	150	18	180
3	60	11	200	19	180
4	80	12	40	20	140
5	100	13	240	21	100
6	100	14	220	22	80
7	120	15	220	23	60

GO ON TO THE NEXT PAGE.

The class `BatteryCharger` below uses a rate table to determine the most economic time to charge the battery. You will write two of the methods for the `BatteryCharger` class.

```
public class BatteryCharger
{
  /** rateTable has 24 entries representing the charging costs for hours 0 through 23. */
  private int[] rateTable;

  /** Determines the total cost to charge the battery starting at the beginning of startHour.
   *  @param startHour the hour at which the charge period begins
   *           Precondition: 0 ≤ startHour ≤ 23
   *  @param chargeTime the number of hours the battery needs to be charged
   *           Precondition: chargeTime > 0
   *  @return the total cost to charge the battery
   */
  private int getChargingCost(int startHour, int chargeTime)
  {  /* to be implemented in part (a) */  }

  /** Determines start time to charge the battery at the lowest cost for the given charge time.
   *  @param chargeTime the number of hours the battery needs to be charged
   *           Precondition: chargeTime > 0
   *  @return an optimal start time, with 0 ≤ returned value ≤ 23
   */
  public int getChargeStartTime(int chargeTime)
  {  /* to be implemented in part (b) */  }

  // There may be instance variables, constructors, and methods that are not shown.
}
```

(a) Write the `BatteryCharger` method `getChargingCost` that returns the total cost to charge a battery given the hour at which the charging process will start and the number of hours the battery needs to be charged.

For example, using the rate table given at the beginning of the question, the following table shows the resulting costs of several possible charges.

Start Hour of Charge	Hours of Charge Time	Last Hour of Charge	Total Cost
12	1	12	40
0	2	1	110
22	7	4 (the next day)	550
22	30	3 (two days later)	3,710

Note that a charge period consists of consecutive hours that may extend over more than one day.

Complete method `getChargingCost` below.

```
/** Determines the total cost to charge the battery starting at the beginning of startHour.
 *  @param startHour the hour at which the charge period begins
 *          Precondition: 0 ≤ startHour ≤ 23
 *  @param chargeTime the number of hours the battery needs to be charged
 *          Precondition: chargeTime > 0
 *  @return the total cost to charge the battery
 */
private int getChargingCost(int startHour, int chargeTime)
```

(b) Write the `BatteryCharger` method `getChargeStartTime` that returns the start time that will allow the battery to be charged at minimal cost. If there is more than one possible start time that produces the minimal cost, any of those start times can be returned.

For example, using the rate table given at the beginning of the question, the following table shows the resulting minimal costs and optimal starting hour of several possible charges.

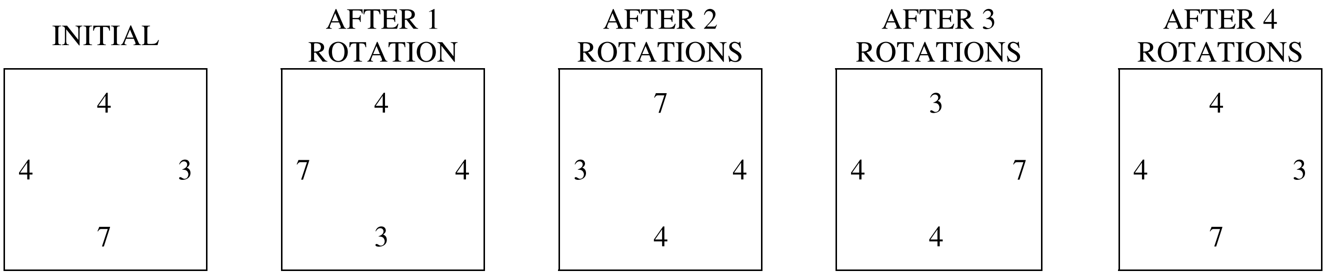
Hours of Charge Time	Minimum Cost	Start Hour of Charge	Last Hour of Charge
1	40	12	12
2	110	0 **or** 23	1 0 (the next day)
7	550	22	4 (the next day)
30	3,710	22	3 (two days later)

Assume that `getChargingCost` works as specified, regardless of what you wrote in part (a).

Complete method `getChargeStartTime` below.

```
/** Determines start time to charge the battery at the lowest cost for the given charge time.
 *  @param chargeTime  the number of hours the battery needs to be charged
 *            Precondition: chargeTime > 0
 *  @return  an optimal start time, with 0 ≤ returned value ≤ 23
 */
public int getChargeStartTime(int chargeTime)
```

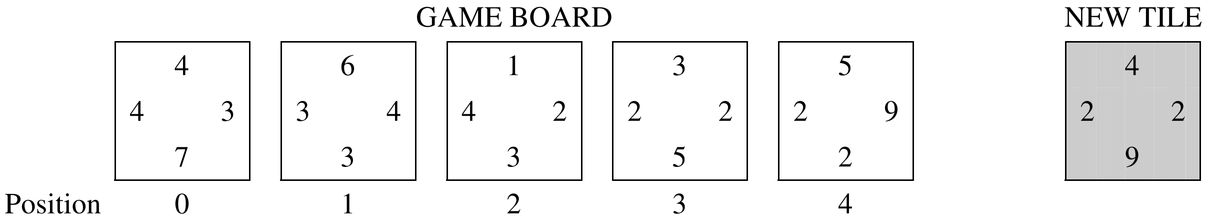
4. A game uses square tiles that have numbers on their sides. Each tile is labeled with a number on each of its four sides and may be rotated clockwise, as illustrated below.



The tiles are represented by the `NumberTile` class, as given below.

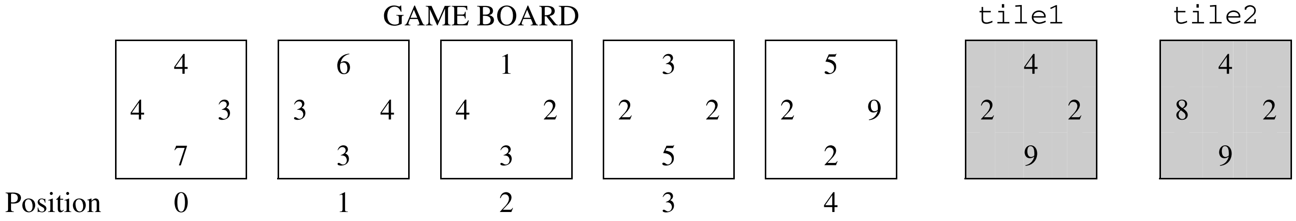
```
public class NumberTile
{
   /** Rotates the tile 90 degrees clockwise
    */
   public void rotate()
   {   /* implementation not shown */   }


   /** @return value at left edge of tile
    */
   public int getLeft()
   {   /* implementation not shown */   }


   /** @return value at right edge of tile
    */
   public int getRight()
   {   /* implementation not shown */   }

   // There may be instance variables, constructors, and methods that are not shown.
}
```

Tiles are placed on a game board so that the adjoining sides of adjacent tiles have the same number. The following figure illustrates an arrangement of tiles and shows a new tile that is to be placed on the game board.



(a) Write the `TileGame` method `getIndexForFit` that determines where a given tile, in its current orientation, fits on the game board. A tile can be inserted at either end of a game board or between two existing tiles if the side(s) of the new tile match the adjacent side(s) of the tile(s) currently on the game board. If there are no tiles on the game board, the position for the insert is 0. The method returns the position that the new tile will occupy on the game board after it has been inserted. If there are multiple possible positions for the tile, the method will return any one of them. If the given tile does not fit anywhere on the game board, the method returns `-1`.

For example, the following diagram shows a game board and two potential tiles to be placed. The call `getIndexForFit(tile1)` can return either 3 or 4 because `tile1` can be inserted between the tiles at positions 2 and 3, or between the tiles at positions 3 and 4. The call `getIndexForFit(tile2)` returns `-1` because `tile2`, in its current orientation, does not fit anywhere on the game board.



Complete method `getIndexForFit` below.

```
/** Determines where to insert tile, in its current orientation, into game board
 *  @param tile the tile to be placed on the game board
 *  @return the position of tile where tile is to be inserted:
 *            0 if the board is empty;
 *           -1 if tile does not fit in front, at end, or between any existing tiles;
 *           otherwise, 0 ≤ position returned ≤ board.size()
 */
private int getIndexForFit(NumberTile tile)
```

GO ON TO THE NEXT PAGE.

Question 3: Battery Charger

Part (a) `getChargingCost` 5 points

- **+1 1/2** accesses array elements
    - **+1/2** accesses any element of `rateTable`
    - **+1/2** accesses an element of `rateTable` using an index derived from `startHour`
    - **+1/2** accesses multiple elements of `rateTable` with no out-of-bounds access potential

- **+2 1/2** accumulates values
    - **+1/2** declares and initializes an accumulator
    - **+1/2** accumulates values from elements of `rateTable`
    - **+1/2** selects values from `rateTable` using an index derived from `startHour` and `chargeTime`
    - **+1** determines correct sum of values from `rateTable` based on `startHour` and `chargeTime`

- **+1** value returned
    - **+1/2** returns any nonconstant (derived) value
    - **+1/2** returns accumulated value

Part (b) `getChargeStartTime` 4 points

- **+1/2** invokes `getChargingCost` or replicates functionality with no errors

- **+1** determines charging cost
    - **+1/2** considers **all** potential start times; must include at least 0 … 23
    - **+1/2** determines charging cost for potential start times

  Note: No penalty here for parameter passed to `getChargingCost` *that violates its preconditions (e.g., 24)*

- **+1** compares charging costs for two different start times

- **+1** determines minimum charging cost based on potential start times

  Note: Penalty here for using result of call to `getChargingCost` *that violates its preconditions (e.g., 24)*

- **+1/2** returns start time for minimum charging cost