Write your answers on paper. Then, compare with your partner and see if you can improve on each other's work.

1. Write the following method:

```java
/** Copies an array into a new array of the given size.
 * If the new size is smaller than the array's current
 * size, the elements at the end are dropped. If the new
 * size is larger than the current size, 0s are appended
 * to the end of the existing data.
 * The original array is not modified.
 *
 * @param arr The source array
 * @param size The size of the returned array
 * @return The resized array
 */
public static int[] resizeArray(int[] arr, int size)
```

2. Say we want to be able to find the first number less than 10 in an ArrayList. We could use a loop to do this, but that's a bit frustrating because ArrayList gives is a number of methods that do searching. Instead, we want to write this:

```java
/** Finds the index of the first element in nums that is
 * less than n, or -1 if no such element exists.
 *
 * @param nums The list to search in
 * @param n The number to compare against
 * @return The index of the first element less than n,
 *         or -1 if no such element exists
 */
public static int firstLessThan(ArrayList<Integer> nums, int n)
{
    LessThan lt = new LessThan(n);
    return nums.indexOf(lt);
}
```

Write the class LessThan such that the method above has the desired behavior. Here is the documentation for ArrayList's indexOf method (which you do not have to write):

```java
/** Returns the index of the first occurrence of the
 * specified element in this list, or -1 if this list does
 * not contain the element. More formally, returns the
 * lowest index i such that
 * (o==null ? get(i)==null : o.equals(get(i))),
 * or -1 if there is no such index.
 *
 * @param o element to search for
 * @return the index of the first occurrence of the
 * specified element in this list, or -1 if this
 * list does not contain the element
 */
public int indexOf(Object o)
```

3. Write the following method:

```java
/** Returns the head of a linked list containing up to the
 * last two elements of the linked list headed by the node
 * provided as a parameter. In other words, if the input
 * list has 0 or 1 elements, returns that list. Otherwise,
 * removes the first (n-2) elements from the list, where
 * n is the length of the original list.
 */
```
This method does not touch the data field of any node.

@param head The head of the input list
@return The head of the list with at most 2 elements */
public static Node<E> lastTwo(Node<E> head)

Here is the Node class to work with:

public class Node<E> {
    public E data; // the data stored at this node
    public Node<E> next; // the next node of data

    /** Creates a new node with a null next field
     * @param dataItem The data stored
     */
    private Node(E dataItem) {
        data = dataItem;
        next = null;
    }

    /** Creates a new node that references another node.
     * @param dataItem The data stored
     * @param nodeRef The node referenced by new node
     */
    private Node(E dataItem, Node<E> nodeRef) {
        data = dataItem;
        next = nodeRef;
    }
}

4. What is the big-O running time of the following method? State your answer in terms of \(n\), the length of the input array. Note that you do not need to figure out what \texttt{frob} does.

    public static void frob(int[] nums) {
        for(int i = 0; i < nums.length; i++) {
            int minIndex = i;
            for(int j = i + 1; j < nums.length; j++) {
                if(nums[j] < nums[minIndex]) {
                    minIndex = j;
                }
            }
            int temp = nums[i];
            nums[i] = nums[minIndex];
            nums[minIndex] = temp;
        }
    }

5. Write the following method:

    /** Takes the back element from this queue and puts it
     * at the front of the queue. This method runs in \(O(n)\)
     * time where \(n\) is the length of the queue.
     * @param q The q to rearrange.
     */
public void cutQueue(Queue<String> q)

Here is the Queue interface:

public interface Queue<E> extends Collection<E> {
/**
 * Inserts the specified element into this queue if it is possible to do so
 * immediately without violating capacity restrictions, returning
 * true upon success and throwing an IllegalArgumentException
 * if no space is currently available.
 * @param e the element to add
 * @return true
 * @throws NullPointerException if the specified element is null and
 *         this queue does not permit null elements
 * @throws IllegalArgumentException if some property of this element
 *         prevents it from being added to this queue
 */
boolean add(E e);

/**
 * Inserts the specified element into this queue if it is possible to do
 * so immediately without violating capacity restrictions.
 * When using a capacity-restricted queue, this method is generally
 * preferable to add, which can fail to insert an element only
 * by throwing an exception.
 * @param e the element to add
 * @return true if the element was added to this queue, else
 *         false
 * @throws NullPointerException if the specified element is null and
 *         this queue does not permit null elements
 * @throws IllegalArgumentException if some property of this element
 *         prevents it from being added to this queue
 */
boolean offer(E e);

/**
 * Retrieves and removes the head of this queue. This method differs
 * from poll() only in that it throws an exception if
 * this queue is empty.
 * @return the head of this queue
 * @throws NoSuchElementException if this queue is empty
 */
E remove();

/**
 * Retrieves and removes the head of this queue,
 * or returns null if this queue is empty.
 * @return the head of this queue, or null if this queue is empty
 */
E poll();

/**
 * Retrieves, but does not remove, the head of this queue. This method

* differs from peek only in that it throws an exception
* if this queue is empty.
* * @return the head of this queue
* * @throws NoSuchElementException if this queue is empty
* /
E element();

/**
 * Retrieves, but does not remove, the head of this queue,
 * or returns null if this queue is empty.
 * * @return the head of this queue, or (null) if this queue is empty
 * /
E peek();
}

The Queue interface extends Collection, which contains this relevant method:

/**
 * Returns the number of elements in this collection. If this collection
 * contains more than Integer.MAX_VALUE elements, returns
 * Integer.MAX_VALUE.
 * * @return the number of elements in this collection
 */
int size();