Answers to multiple choice questions:

Answers:

1. d
2. a
3. e
4. e
5. b
6. c
7. d
8. a
9. c
10. c

Shell Sort:

The following solution cleverly uses 3 nested loops instead of a more naïve (but equally good) implementation that would use 4 nested loops (or that is broken down to an insertion sort method that runs within a nested loop).

Part a:

```
/**
 * shellSort( ) performs a shell sort on an array of integers.
 * This method is very similar to the insertion sort algorithm
 * presented in class in slide 14-61 (and the variable names here
 * match that presentation). The main differences are: 1) the use
 * of a while loop around the for loop, to run the interior loop
 * multiple times (for each different increment value); and 2) the
 * use of the increment variable in several places in the interior
 * for loop, where, in the insertion sort in the lecture slides,
 * we used the constant "1".
 */
static public void shellSort(int[ ] data)  {
    int newest, current, increment, newItem, size;

    size = data.length;
    increment = size/2;
    while (increment > 0) {
        for (newest = increment; newest < size; newest++) {
            current = newest;
            newItem = data[newest];
            while ((current >= increment) &&
                    (data[current - increment] > newItem)) {
                data[current] = data[current - increment];
                current -= increment;
            }
            data[current] = newItem;
        }
        increment /= 2;
    }
```

```
}
```

Part b:
Worst case for the shellSort( ) method is $O(n^2)$, as it is for regular insertion sort (actually, using a different series of *k*'s, this complexity can be improved); we will also accept $O(n^2 \log n)$ as an intuitive description of the algorithm's complexity. However, shellSort( ) usually performs fewer comparisons and swaps than insertion sort, because shellSort( ) moves values closer towards their final positions in its first *k*-sorts (and with bigger jumps than normal insertion sort). Then, the final insertion sort is very efficient, since the input is "almost sorted" at that last stage. Moreover, the elements in the array remain *k*-sorted even as *k* gets smaller (e.g., if an array is 5-sorted then 3-sorted, it is not only 3-sorted, but has *remained* 5-sorted) – so previous work is not undone.

Palindrome:

The idea is to compare pairs of items at place i and length-1-i in the list. Since the list has no links leading back, the complexity cannot be lower than O(n^2).

```
public static boolean isPalindrome(Node head) {
        int length = listLength(head);
        for (int i = 0; i < length / 2; i++) {
                if (!identical(i, head, length)) {
                        return false;
                }
        }
        return true;
}

/**
 * Check to see if the value in the given location of the list is identical
 * to the value at location length-location-1. (If it is not then we don't
 * have a palindrome)
 */
private static boolean identical(int location, Node head, int length) {
        int i; // our current location in the list.
        int value; // will hold the value at location i.
        // travel up to the given location in the list:
        for (i = 0; i < location; i++) {
                head = head.getNext();
        }
        /*
         * (previous part could be made more efficient if we don't travel from
         * the beginning of the list every time)
         */
        value = head.getData();
        for (; i < length - location - 1; i++) {
                head = head.getNext();
        }
```

```
        return head.getData() == value;
    }

    /**
     * Find the length of the given list.
     */
    public static int listLength(Node head) {
        int length = 0;
        while (head != null) {
            head = head.getNext();
            length++;
        }
        return length;
    }
```

Part two:

The basic idea: disconnect the list into two separate ones, reverse one of the lists using the setNext method and then compare the two lists.
Reversing the list can be done in linear time, and this allows us to achieve a complexity of O(n) for the entire procedure.

```
    public static boolean isPalindrome(Node head) {
        int length = listLength(head);
        if (length==1){
            return true;
        }
        Node current = head;
        //have current travel to before the mid point:
        for(int i=0; i<length/2-1; i++){
            current = current.getNext();
        }
        //get the second list
        Node secondList = current.getNext();
        // cut off the first list from the second part
        current.setNext(null);
        //reverse the second part
        Node reversedList = reverseList(secondList);
        //and compare the two parts.
        return compareLists(head,reversedList);
    }

    /**
     *Compares two lists until the first list (which is assumed to be shorter) ends.
     * returns true if they are the same.
     */
    private static boolean compareLists(Node firstList, Node secondList) {
        while(firstList != null){
            if (firstList.getData()!= secondList.getData()){
                return false;
            }
```

```
                firstList = firstList.getNext();
                secondList = secondList.getNext();
        }
        return true;
}

/**
 *      Reverses the given list, and returns its new head.
 */
private static Node reverseList(Node head) {
        Node current = head;
        Node newTail = null;
        while (current.getNext()!=null){
                Node temp = current.getNext();
                current.setNext(newTail);
                newTail = current;
                current = temp;
        }
        current.setNext(newTail);
        return current;
}
```

Recursion question (Azrieli):

There are several ways to solve this problem. The basic idea behind the solution presented below (Which is not the most efficient one) is to go over all permutations of the array of blocks, and to consider each permutation as a potential distribution of blocks to buildings. For each permutation, the blocks are read in order, and when they sum to the required height, that section of the array is considered as one building. If at some point blocks sum to something that is higher than height than all the blocks from that point on are discarded.
With this solution, the recursion is very simple (just the permutations problem we already saw in one of our exercises) and the rest of the work is done using simpler methods that look at each permutation.

```
public class Azrieli {

    /** The array to hold blocks from which towers are built */
    private int[] _towers;

    /** Maximal amount of towers */
    private int _numOfTowers;

    /** Height of towers to build */
    private int _height;


    /**
```

```
 * The function counts how many towers can be legally build if
 * blocks are taken sequencially from their list.
 */
private int countTowers(int[] data){
	int currentHeight=0;
	int towersCounter = 0;
	for(int i = 0; i<data.length; i++) {
	    currentHeight += data[i];
	    if(currentHeight == _height){
		towersCounter++;
		currentHeight = 0;
	    }
	    if(currentHeight > _height)
		break;
	}
	return towersCounter;
}

/**
 * The recursive function passes over all possible permutations of
 * the given set of block sizes.
 *
 * For each permutation the function counts the number of towers
 * that can be build from it. If the number is greater than the
 * one previously recorded, the function updated the tracking
 * data.
 */
private void permutations(int[] data, int start){
	if (start >= data.length-1) {
	    int numOfTowers = countTowers(data);
	    if (numOfTowers > _numOfTowers) {
		_numOfTowers = numOfTowers;
		System.arraycopy(data,0,_towers,0,data.length);
	    }
	}
	for (int j=start; j<data.length; j++){
	swap(data, start, j);
	permutations(data, start+1);
	swap(data, j, start);
    }
}

/**
 * Swap two elements within an array
 */
private void swap(int[] data, int i, int j){
	int temp;
	temp = data[i];
	data[i]=data[j];
	data[j]=temp;
```

```
    }

    /**
     * Pretty print the best block-to-tower assignment
     */
    private void printTowers(){
        if (_numOfTowers == 0)
            System.out.println("Can build no towers of height "+_height);
        else {
            int index = 0;
            int currentHeight = 0;
            for(int i = 0;i<_numOfTowers;i++){
                System.out.print("Tower number "+i+":");
                while(currentHeight < _height){
                    System.out.print(" "+_towers[index]);
                    currentHeight +=_towers[index++];
                }
                System.out.print("\n");
                currentHeight = 0;
            }
        }
    }

    /**
     * The function discovers the best possible allocation of blocks
     * to build towers, and then prints the assigned set.
     */
    public int buildTowers(int[] blocks, int height){
        if( (blocks == null) || (blocks.length == 0))
            return 0;
        _towers = new int[blocks.length];
        _numOfTowers = 0;
        _height = height;

        permutations(blocks, 0);
        printTowers();
        return _numOfTowers;
    }
}
```