Question 1. [10 MARKS]

Part (a) [2 MARKS]

`list[s...m]` and `list[m+1...e]` are sorted in non-descending order and list != null and list contents are mutually Comparable.

Part (b) [5 MARKS]

```
private static void merge(Comparable[] list, int s, int m, int e) {

  _F_  // index of current candidate in list[s...m]
  _G_  // index of current candidate in list[m+1...e]

  _K_  // Temporary storage to accumulate the merged result
  int p=0;  // index to put next element into merged

  // merged[0...p] contains list[s...p1-1] merged with list[m+1...p2-1]
  _J_
  _E_
  _H_
  _B_
  } else {
  _I_
  _C_
  _M_
  _A_
  }
  _D_
  System.arraycopy(list, p1, list, s+p, m+1-p1);
  _N_
  _L_
```

Part (c) [3 MARKS]

Answer:

```
+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 4 | 5 | 8 | 2 | 3 | 6 | 7 | 9 |
+---+---+---+---+---+---+---+---+---+---+
```

Question 2. [10 MARKS]

```
public class ArrayBST {

  /** REPRESENTATION INVARIANT:

     The left and right children of keys[i] are in keys[2i+1] and keys[2i+2], and for any node
     the children in its left subtree are less than that node and the children in the right subtree are
     greater.

   */
  private Comparable[] keys;

  /** An empty ArrayBST that will always represent a tree of height <= maxHeight.
    * Requires: maxHeight >= 0. */
  public ArrayBST(int maxHeight) {

    keys= new Comparable[((int) Math.pow(2, maxHeight)) - 1];

  /** Return the root of a BST made of BSTNodes that represents the
    * same tree as this tree does. */
  public BSTNode linkedRepresentation() {

    return linkedHelper(0);
  }

  private BSTNode linkedHelper(int i) {
    if (i >= keys.length) {
      return null;
    }

    if (keys[i] == null) {
      return null;
    }

    BSTNode temp= new BSTNode();
    temp.left= linkedHelper(i * 2 + 1);
    temp.right= linkedHelper(i * 2 + 2);
    temp.key= keys[index];

    return temp;
  }

}
```

Question 3. [10 MARKS]

Rewritten comment:

```
/**
* Return the first position of name in this list if name appears
* within the range beginning one position after start and ending at
* end (inclusive); return -1 otherwise.
* Requires: -1 <= start && end < size() && name != null
* @param name the name to search for
* @param start the position in OrderedList after which search starts
* @param end last position in OrderedList at which search stops
* @return position of name in OrderedList.
*/
```

Question 4. [10 MARKS]

Part (a) [3 MARKS]

```
private static void checkDigits(String s) throws WrongDigitException {
  for (int i= 0; i != s.length(); i++) {
    if (s.charAt(i) > '7') {
      throw new WrongDigitException("" + s.charAt(i));
    }
  }
}
```

Part (b) [3 MARKS]

```
public class WrongDigitException extends Exception {
  public WrongDigitException(String m) {
    super(m);
  }
}
```

Part (c) [4 MARKS]

```
public static String getInput(BufferedReader br) throws IOException {
  String result= br.readLine();
  try {
    checkDigits(result);
  } catch (WrongDigitException e) {
    result= e.getMessage() + " is not a proper digit in base 8";
  }

  return result;
}
```

Question 5. [10 MARKS]

Part (a) [4 MARKS]

Solution: The second, third and fourth statements should be circled.

Part (b) [6 MARKS]

Consider the following theorem:

> 2^n is always less than $n!$ for integer values of n greater than or equal to 4.

1. Fill in the blank with the smallest correct value.
2. Use induction to prove this theorem.

Let $S(n) = 2^n < n!$.

Prove: $S(n) \forall n \geq 4$.

Base case: Prove $S(4)$

When $n = 4$, $2^n = 16 < 24 = n!$. Thus, $S(n)$ holds for the base case.

Let $k \geq 4$ be an arbitrary integer.

Induction Hypothesis: Assume $S(k-1)$: $2^{k-1} < (k-1)!$.

Induction Step: Prove $S(k)$: $2^k < k!$.

$2^k = 2 * 2^{k-1}$. From the induction hypothesis, $2^{k-1} < (k-1)!$. So $2^k < 2 * (k-1)!$.

Because $k \geq 4$, $k \geq 2$. So $2^k < 2 * (k-1)! < k * (k-1)! = k!$.

Conclusion: $S(n)$ for all $n >= 4$.

Question 6. [10 MARKS]

Part (a) [6 MARKS]

What is the running time for the following operations, in `O(..)` notation? Give as small and simple a bound as possible.

1. Finding out whether a linked list with n nodes contains any duplicate elements: `O(n^2)`
2. Printing the nth element of a sorted array: `O(1)`
3. Calculating the height of a binary tree with n nodes: `O(n)`
4. Printing the contents of a binary tree with n nodes in ascending order, without using any kind of list or another tree: `O(n^2)`
5. Printing the contents of a binary search tree with n nodes in ascending order: `O(n)`
6. Finding the smallest element in a full binary search tree with n nodes: `O(log n)`

Part (b) [4 MARKS]

A method used to find all prime numbers that are less than n runs as follows:

1. Make a table of integers from 2 to n.
2. For every value of i from 2 to $\sqrt{n}$, cross out the entries $2 * i$, $3 * i$, $4 * i$ up to n.
3. Once $i > \sqrt{n}$, print out all entries that have not been crossed out.

Circle the tightest bound for the questions below.

A) What is the running time for Step 1?
O(n)

B) What is the running time for Step 2?
O(n*sqrt(n))

C) What is the running time for Step 3?
O(n)

D) What is the total running time for this method? O(n*sqrt(n))

Question 7. [10 MARKS]

Consider this code.

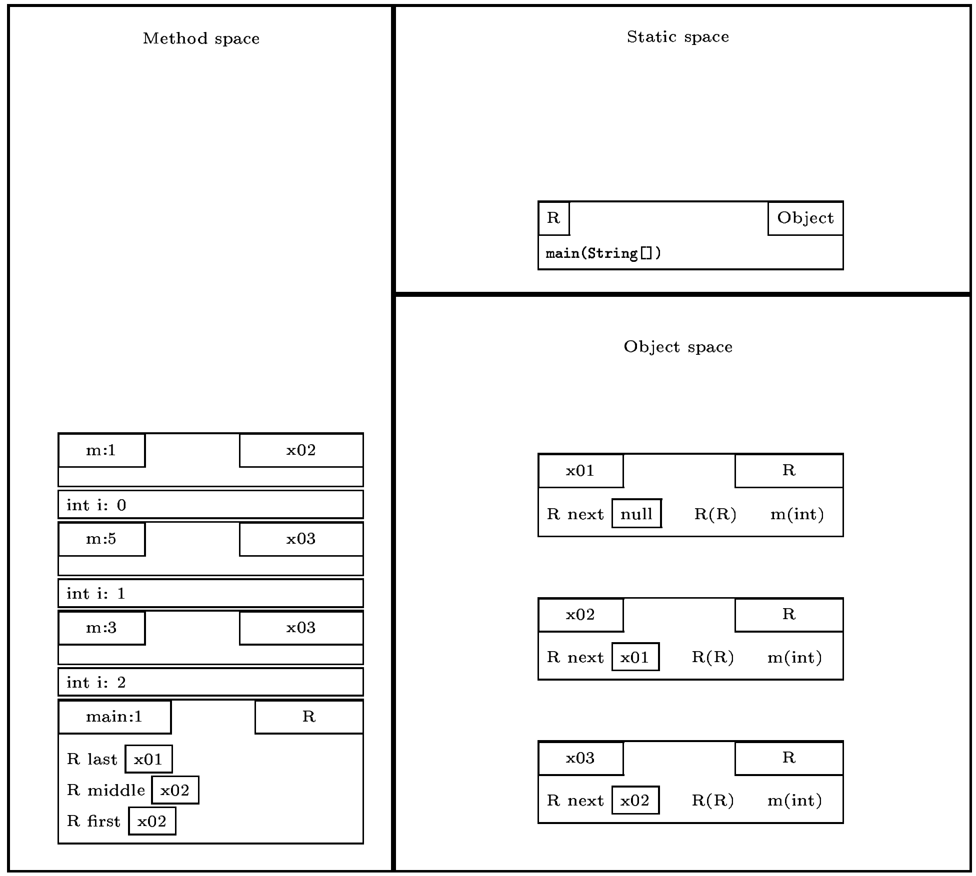
```
public class R {
  private R next;

  public R(R n) {
    next= n;                   // 1
  }

  public void m(int i) {
    if (i != 0) {              // 1
      if (i % 2 == 0) {        // 2
        m(i - 1);              // 3
      } else {                 // 4
        next.m(i - 1);         // 5
      }
    }
  }

  public static void main(String[] args) {
    R last= new R(null);       // 1
    R middle= new R(last);     // 2
    R first= new R(middle);    // 3
    first.m(2);                // 4
  }
}
```

Below is a memory model trace paused just after `main` has been called. Show the state of memory when line 1 of method `m` has been reached and parameter `i` is `0`.



Question 8. [10 MARKS]

Part (a) [5 MARKS]

```
/** Reverse the order of the nodes in the linked list <code>list</code> and return the list.
  * @param  list  first node of a linked list, null if the list is empty.
  * @return the first node of the modified list. */
public static Node reverse(Node list) {

  Node previous= null;

  while (list != null) {
    Node temp= list.next;
    list.next= previous;
    previous= list;
    list= temp;
  }

  return previous;
}
```

Part (b) [5 MARKS]

```
/** Reverse the order of the nodes in the linked list <code>list</code> and return the list.
  * @param  list  first node of a linked list, null if the list is empty.
  * @return the first node of the modified list. */
public static DoubleNode reverse(DoubleNode list) {
```

Here are two solutions:

```
  DoubleNode previous= null;

  while (list != null) {
    list.previous= list.next;
    list.next= previous;
    previous= list;
    list= list.previous;
  }

  return previous;
```

```
  DoubleNode t1= list;

  while (t1 != null){
    DoubleNode t2= temp.next;
    t1.next= temp.previous;
    t1.previous= t2;

    list= t1;
    temp= t1.previous;
  }

  return list;
```

Total Marks = 80