```
public class Test_Static
{
   int a;
   static int b;

   public Test_Static(int av, int bv)
   {
        a= av; b= bv;
   }
   public void print()
   {
        System.out.println
          ("a= " + a + " b= " + b);
   }
   public static void main (String [] args)
   {
          Test_Static x, y;
          x= new Test_Static(3, 4);
          System.out.print("x has ");
          x.print();

          y= new Test_Static(5, 6);
          System.out.print("y has ");
          y.print();

          System.out.print("x has ");
          x.print();
   }
}
```

What does this print?

If the type of a field of an object is not static, every object has its own value for that field.

If the type is static, the program only has ONE variable for that field, not one per object.

Every LinkedList has a value for n, start and rear.

So should n, start and rear be static or not?

Hints for designing recursive routines:

At the top of the class use

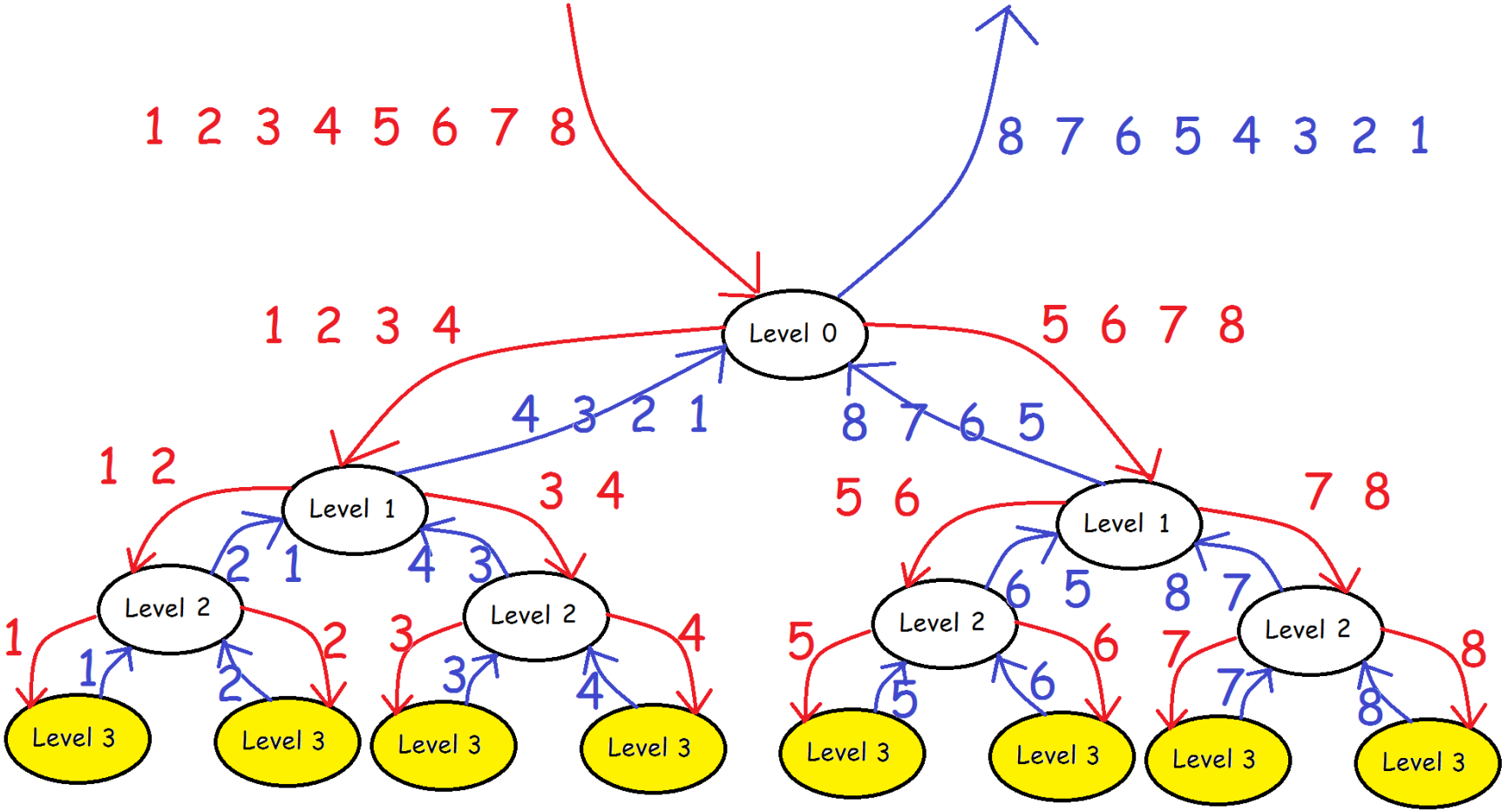
```
static boolean debug = true;
```

Do lots of debug printing.
For example, for reverse:

Print the list when entering the routine.
Print list1 and list2 before and after reversing.
Print the final reversed list.

For all print statements, include the level.

Recursive call structure for reverse:



To ensure that level has the correct value at each recursive call, you call reverse initially like this:

```
reverse(0);
```

The recursive calls for list1 and list2 should be:

```
list1.reverse(level+1);
list2.reverse(level+1);
```

DO NOT increment or decrement reverse.
Some bad code:

```
level++;
```

or

```
list1.reverse(++level);
```

```
Level 0: The original linked list:1 2 3 4 5 6
Level 0: List 1 before reversing: 1 2 3
Level 0: List 2 before reversing: 4 5 6
Level 1: The original linked list:1 2 3
Level 1: List 1 before reversing: 1
Level 1: List 2 before reversing: 2 3
Level 2: The original linked list:1
Level 2: The original linked list:2 3
Level 2: List 1 before reversing: 2
Level 2: List 2 before reversing: 3
Level 3: The original linked list:2
Level 3: The original linked list:3
Level 2: List 1 after  reversing: 2
Level 2: List 2 after  reversing: 3
Level 2: The list after reversing:3 2
Level 1: List 1 after  reversing: 1
Level 1: List 2 after  reversing: 3 2
Level 1: The list after reversing:3 2 1
```

```
Level 1: The original linked list:4 5 6
Level 1: List 1 before reversing: 4
Level 1: List 2 before reversing: 5 6
Level 2: The original linked list:4
Level 2: The original linked list:5 6
Level 2: List 1 before reversing: 5
Level 2: List 2 before reversing: 6
Level 3: The original linked list:5
Level 3: The original linked list:6
Level 2: List 1 after  reversing: 5
Level 2: List 2 after  reversing: 6
Level 2: The list after reversing:6 5
Level 1: List 1 after  reversing: 4
Level 1: List 2 after  reversing: 6 5
Level 1: The list after reversing:6 5 4
Level 0: List 1 after  reversing: 3 2 1
Level 0: List 2 after  reversing: 6 5 4
Level 0: The list after reversing:6 5 4 3 2 1
```

In a class such as LinkedList, a non-static method has an object associated with the method. For example: `public void reverse(int level)`

Inside reverse you can refer to n, start and rear. To call a method like this you use
`object_name.reverse(level);`

Inside printBigInteger (which is not static) you can call reverse like this: `reverse(0);`
which is semantically equivalent to:
`this.reverse(0);` So the object name is "this".

If you want to call it on list1, use
`list1.reverse(level+1);`

A method that is static has no object associated with the method.

For example: `public static LinkedList readBigInteger(Scanner in)`

When you call this you do not yet have a LinkedList object. It creates one and returns it to you.

You cannot refer to n, start, rear in a static method in the LinkedList class.

But if we have
`LinkedList x= new LinkedList();`
You can refer to x.n, x.start and x.rear.

To call a static method such as :

```
public static LinkedList
                readBigInteger(Scanner in)
```

Suppose we have declared x:

```
LinkedList x;
```

If you are calling it from a method inside the LinkedList class you can just use:

```
X= readBigInteger(in);
```

To call a static method such as :

```
public static LinkedList
readBigInteger(Scanner in)
```

Suppose we have declared x:

```
LinkedList x;
```

From outside the class such as in Test.java you can use (preferred):

```
x= LinkedList.readBigInteger(in);
```

Or (not as nice but legal):

```
x=null; // To avoid an error with x not being initialized
x= x.readBigInteger(in)
```

Note: calling like this does not give you access to n, start or rear inside the readBigInteger method.

To do reverse recursively:

0. If n is 1, return. // Base case
1. Divide the problem into two subproblems.
2. Solve the subproblems. (Conquer)
3. Marry the solutions.

Some people are trying to do the marriage step before the conquer step.

The recurrence relation from last class:

Consider this recurrence which is only defined for values of $n = 2^k$ for some integer $k \geq 0$:

$T(1) = 1$, $T(n) = 1 + 2T(n/2)$.

The solution we obtained was:

$T(2^k) = 1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1$

To show your work when using repeated substitution, number your steps:
Step 0: The original recurrence for T(n).
Step 1: The formula for T(n) after one substitution into the RHS.
Step 2: The formula for T(n) after two substitutions into the RHS.
…

DO not oversimplify by grouping terms together.

At step i, we want to be able to see what the ith term is and what the term is involving T.

Determine the general pattern for Step i:

Step i: The formula for T(n) after i substitutions into the RHS.

Determine at which step i the base case appears on the RHS of the formula, say at some step f.

Set i=f and then plug in the base case to get a formula for the recurrence relation that no longer has T in it.

The recurrence relation from last class:

$n = 2^k$ for some integer $k \geq 0$:

$T(1) = 1$, $T(n) = 1 + 2T(n/2)$.

Replace n by 2^k (this often makes the math easier): $T(2^0) = 1$, $T(2^k) = 1 + 2T(2^{k-1})$.

Step 0: $T(2^k) = 1 + 2T(2^{k-1})$.

Step 1: $T(2^k) = 1 + 2 * [\, 1 + 2T(2^{k-2})]$

$= 1 + 2 + 4 * T(2^{k-2})$

Step 2: $T(2^k) = 1 + 2 + 4 * [1 + 2T(2^{k-3})]$

$= 1 + 2 + 4 + 8 * T(2^{k-3})$

Step i: $T(2^k) = 1 + 2 + 4 + + \ldots + 2^i + 2^{i+1} * T(2^{k-i-1})$

The base case is: $T(2^0) = 1$

Step i: $T(2^k) = 1 + 2 + 4 + + \ldots + 2^i + 2^{i+1} * T(2^{k-i-1})$

At which step i is the base case on the RHS?

$T(2^0) = T(2^{k-i-1}) \Rightarrow 2^0 = 2^{k-i-1} \Rightarrow 0 = k - i - 1$

$\Rightarrow i = k-1$

Step k-1: $T(2^k) = 1 + 2 + 4 + + \ldots + 2^{k-1} + 2^{k-1+1} * T(2^{k-(k-1)-1})$

Since $T(2^{k-(k-1)-1}) = T(2^0) = 1$,

Step k-1: $T(2^k) = 1 + 2 + 4 + + \ldots + 2^{k-1} + 2^k$

$S = 1 + 2 + 4 + + \ldots + 2^{k-1} + 2^{k}$

A trick for remembering the formula for S:

Write S as a binary number

Column	2^{k+1}	2^{k}	2^{k-1}	…	2^{3}	2^{2}	2^{1}	2^{0}
S=		1	1		1	1	1	1
Add 1								1
S+1=	1	0	0		0	0	0	0

Since $S+1 = 2^{k+1}$, $S = 2^{k+1} - 1$.

The recurrence relation from last class:

Consider this recurrence which is only defined for values of $n = 2^k$ for some integer $k \geq 0$:

$T(1) = 1$, $T(n) = 1 + 2T(n/2)$.

The solution we obtained was:

$T(2^k) = 1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1$

Suppose instead we did the math incorrectly And had the solution $T(2^k) = 2^k$.

Consider this recurrence which is only defined for values of $n = 2^k$ for some integer $k \geq 0$:
$T(1) = 1$, $T(n) = 1 + 2T(n/2)$.
We will try to prove by induction that $T(2^k) = 2^k$.
[Base case]
The base case for the recurrence relation is that $T(1) = T(2^0) = 1$. The formula gives that $T(2^0) = 2^0 = 1$ as required.
[Induction step]
Assume that $T(2^k) = 2^k$.
We want to prove that $T(2^{k+1}) = 2^{k+1}$.
From the recurrence relation,
$T(2^{k+1}) = 1 + 2T(2^k)$. By induction, $T(2^k) = 2^k$.
Therefore, $T(2^{k+1}) = 1 + 2 * 2^k = 2^{k+1} + 1$
$\neq 2^{k+1}$. Therefore the proof fails.

Consider this recurrence which is only defined for values of $n = 2^k$ for some integer $k \geq 0$:
$T(1) = 1$, $T(n) = 1 + 2T(n/2)$.
Prove by induction that $T(2^k) = 2^{k+1} - 1$.

[Base case]

The base case for the recurrence relation is that $T(1) = T(2^0) = 1$. The formula gives that $T(2^0) = 2^1 - 1 = 1$ as required.

[Induction step]

Assume that $T(2^k) = 2^{k+1} - 1$.
We want to prove that $T(2^{k+1}) = 2^{k+2} - 1$.
From the recurrence relation,
$T(2^{k+1}) = 1 + 2T(2^k)$. By induction, $T(2^k) = 2^{k+1} - 1$.
Therefore, $T(2^{k+1}) = 1 + 2 * [2^{k+1} - 1] = 2^{k+2} - 1$
as required.

What is wrong with my induction proof?

In a drunken haze I decided that the solution to the recurrence $T(1)=1$, $T(n)= 1 + T(n-1)$ is

$1 + 2 + 3 + \ldots + n$.

Theorem: The solution to the recurrence is $n(n+1)/2$.

Proof. [Basis] $T(1)=1$ and $1 *(1+1)/2 = 1$ as required.

[Induction step] Assume that $1 + 2 + \ldots + n-1 + n = n(n+1)/2$.

We want to prove that $1 + 2 + \ldots + n-1 + n + (n+1) = (n+1)(n+2)/2 = (n^2 +3n +2)/2$.

By induction, $1 + 2 + \ldots + n= n(n+1)/2$.

So $1 + 2 + .. + n + (n+1)= n(n+1)/2 + (n+1)$.

Simplifying: $(n^2 + n + 2n + 2)/2= (n^2 +3n +2)/2$ as required.