Questions

1. (a) Suppose you wish to "count down" the numbers from a given `n` down to 1. You can use a `while` loop to do this:

```
countdown(n){
   while (n > 0) {
       print n
       n-- }
}
```

   Write a recursive version of this algorithm.

   (b) Now write a recursive `countUp(n)` algorithm which counts up from 1 to `n`.

2. Here is an alternative reverse method for the `SLinkedList<E>` class which reverses the order of nodes in a singly linked list. (See online code from linked list exercises). The method calls a recursive helper method `reverseRecursiveHelper` which does most of the work. Write this helper method. If your solution is different from the given one, then you should add your method to the SLinkedList class and test it!

```
public void reverseRecursive(){
   SNode<E>  oldHead = head;
   reverseRecursiveHelper(head);
   head = tail;              //  swap head and tail
   tail = oldHead;           //
}
```

3. Consider a simplified version of the game "20 questions", in which one person has a number in mind from 0 to $2^{20} - 1$. You can easily find that number with your twenty questions e.g. by asking for the ith bit of the number. (This is essentially binary search.)

   Here is a slightly different game. Suppose I am thinking of a positive integer n but it can be *any* positive integer. It is easy for you figure out the number using n questions, namely question i is "is it the number i ?". Give a faster algorithm, namely one that runs in time proportional to $\log n$.

   Hint: you may have the intuition that binary search would be good here. The trouble is that you don't know in advance how large the number *might* be.

4. Suppose that you are given an array of n different numbers that strictly increase from index 0 to index m, and strictly decrease from index m to index $n - 1$, where n is known but m is unknown. Note that there is a unique largest number in such a list, and it is at index m. Here are a few examples.


```
                                  m       n
                                 ---     ---
 [ 3, 5, 17, 18, 21, 6 ]          4       6
 [ -3, 5 ]                        1       2
 [ 12, 7, 4, 2, -5 ]              0       5
```

Provide the missing pseudocode below of a recursive algorithm that returns the index m of the largest number in the array, in time proportional to log n.

The algorithm is initially called with `low = 0, high = n-1`.

```
findM( a, low, high ){   // array is a[ ],  assume low <= high
   if (low == high)
      return low
   else{
    //   ADD YOUR CODE HERE (AND ONLY HERE)
   }
}
```

5. Suppose you sort a list of numbers using the mergesort algorithm.

   Show the order of the list elements after all merges of lists of size 1 to 2 have been completed, and then after all merges of lists of size 2 to lists of size 4 have been completed:

```
( 6 , 5 , 2 , 8 , 4 , 3 , 7 , 1 )       original list, n=8

(   ,   ,   ,   ,   ,   ,   ,   )       merges from n=1 to n=2 completed

(   ,   ,   ,   ,   ,   ,   ,   )       merges from n=2 to n=4 completed

( 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 )       final sorted list
```

6. Recall the Tower of Hanoi algorithm.

```
tower(n, start, finish, other){
   if n==1
       move from start to finish
   else {
       tower(n-1, start, other, finish)
       move from start to finish
       tower(n-1, other, finish, start)
   }
}
```

   Suppose we call `tower(5, 1, 3, 2)`. What are the parameters of the function the first time a `tower()` call *returns* to its caller?

   Hint: consider the call stack which keeps track of the sequence of recursive calls. When the call stack is first popped, what were the parameters of `tower()` ?

Answers

1. (a)
```
countdown(n){
    if (n > 0) {
        print n
        countdown(n - 1)
    }
}
```

   (b)
```
countUp(n){
    if (n > 0) {
        countUp(n - 1)
        print n
    }
}
```
   Did you get the order of the instructions correct?

2. The following recursively reverses the list from `head.next`, and then cleans up the references that involve the original `head`. Also note the base case that the list has just one element.

```
private void reverseRecursiveHelper(SNode<E> head){
   if (head.next != null){
      reverseRecursiveHelper(head.next);
      head.next.next = head;
      head.next = null;
   }
}
```

3. The idea is to guess increasing powers of 2 until the power of 2 is bigger than the number. Then do a binary search (backwards). For example, suppose $n = 51$. We would guess "Is $n < 1$? Is $n < 2$? Is $n < 4$? Is $n < 8$? .... Is $n < 64$? The answers would be no, no, no, no, ... yes. It would take about $\log n$ guesses to reach that point.

   Then, we could do a binary search for n. We know $32 <= n < 64$ and so we could ask "Is $n \leq mid$? " where $mid = (32 + 64)/2$ and keep going, shrinking the interval [low,high] from [32,64] until low == high. Then we're done. In general, it would take another $\log_2 n$ guesses to narrow down to the exact value of n.

4. There are several solutions.

```
//  --------  SOLUTION 1   ------

    if low == high
       return a[low]
    else {
       mid = (low + high)/2
       if (a[mid] < a[mid+1]){
          return findM(a, mid+1, high)
       else
          return findM(a, low, mid)
}

//  --------  SOLUTION 2   ------

         if (high-low == 1){
            if (a[low] < a[high])
               return high
            else
               return low
         }
         else{
            mid = (low + high)/2
            if (a[mid-1] < a[mid]){
               return findM(a, mid, high)
            else
               return findM(a, low, mid)
         }

//  --------  SOLUTION 3   ------
         if (high-low == 1){
            if (a[low] < a[high])
               return high
            else
               return low
         }
         else{
            mid = (low + high)/2
            if (a[mid] < a[mid+1]){
               return findM(a, mid+1, high)
            else
               return findM(a, low, mid)
        }
```

5. Below I have grouped into four lists of size two, and then these four lists of size two are merged into two lists of size four.

```
( 5,6,    2,8,   3,4,    1,7 )    after merging lists of size 1 to lists of size 2
( 2,5,6,8,    1,3,4,7 )           after merging lists of size 2 to lists of size 4
```

6. Here is the sequence of calls. Think of the call stack growing downwards here. The last call below will be the first call that returns, which is what the question was asking.

```
tower(5, 1, 3, 2)
tower(4, 1, 2, 3)
tower(3, 1, 3, 2)
tower(2, 1, 2, 3)
tower(1, 1, 3, 2)  <-   top of stack  (and the first to return i.e. be popped)
```

