Questions

1. Use mathematical induction to prove that, for any $n \geq 1$

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}.$$

(Recall that I proved it in a different way back in lecture 2.)

2. Use mathematical induction to prove that, for all $n \geq 1$,

$$1 + 3 + 5 + \cdots + (2n - 1) = n^2.$$

3. (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`.

4. Write a recursive algorithm that prints out a consecutive sequence of elements in an array. The method has three parameters: the array name, and the first and last indices to print from i.e. `displayArray( a, first, last)`

This should be easy, but there is one little "gotcha" there so see if you can avoid it.

5. 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;           //
}
```

6. 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 $\log n$.

7. 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)
   }
}
```

8. Recall the mergesort algorithm which recursively partitions a list of size n into two sub-lists of half the size, sorts the two sublists, and then merges them

   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
```

9. I claimed in the lecture that the recursive method for `power(x,n)` takes about the same amount of time to run as the iterative method, even though the recursive method uses $O(\log_2 n)$ multiplies and the iterative method uses $O(n)$ multiplies. The reason is that one cannot just just consider the number of multiplications, but one also needs to consider the number of digits being multiplied.

   Calculate a "ballpark" estimate on the amount of time it takes to compute x^n using the recursive versus iterative method. Assume that x is represented with M digits, and assume that the time it takes to multiple two numbers with M_1 and M_2 digits is $cM_1\ M_2$ where c is some constant.

Answers

1. The base case is easy. Substitute $n = 0$ and we get $0 = 0$ which is true.

   For the induction step, we *hypothesize* that

   $$\sum_{i=0}^{k-1} x^i = \frac{x^k - 1}{x - 1}$$

   for $k \geq 0$, and we want to show it follows from this hypothesis that

   $$\sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1}.$$

   Take the left side of the last equation, and rewrite it:

   $$\begin{aligned}
   \sum_{i=0}^{k} x^i &= \sum_{i=0}^{k-1} x^i + x^k \\
   &= \frac{x^k - 1}{x - 1} + x^k, \quad \text{by induction hypothesis} \\
   &= \frac{x^k - 1}{x - 1} + x^k(\frac{x - 1}{x - 1}) \\
   &= \frac{x^{k+1} - 1}{x - 1}
   \end{aligned}$$

   which is what we wanted to show.

2. The base case of $n_0 = 1$ is obvious, since there is only a single term on the left hand side, i.e. $1 = 1^2$. The induction hypothesis is the statement $P(k)$:

   $$P(k) \equiv \text{" } 1 + 3 + 5 + \cdots + (2k - 1) = k^2 \text{ "}$$

   To prove the induction step, we show that if $P(k)$ is true, then $P(k + 1)$ must also be true. As usual, we take the left side of the equation of $P(k + 1)$:

   $$\begin{aligned}
   \sum_{i=1}^{k+1}(2i - 1) &= 2(k + 1) - 1 + \sum_{i=1}^{k}(2i - 1) \\
   &= 2(k + 1) - 1 + k^2, \quad \text{by the induction hypothesis} \\
   &= 2k + 1 + k^2 \\
   &= (k + 1)^2.
   \end{aligned}$$

   Thus, the induction step is also proved, and so we're done.

3. (a)
```
countdown(n){
    if (n > 0) {
        print n
        countdown(n - 1)
    }
}
```
Did you remember the base case?

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

4.
```
displayArray( a,  first,  last){
   print  a[first])
   if (first < last)              //  Did you include this condition?
      displayArray(a, first+1, last)
}
```
Alternatively, you could do it like this:
```
displayArray( a,  first,  last){
   if (first < last)
      displayArray(a, first, last-1);
   print  a[last])
}
```

5. 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;
   }
}
```

6. The idea of the algorithm is to guess increasing powers of 2 until the power of 2 is bigger than the number. Then do a binary search (backwards).

```
guess = 1
answer = false
while (answer == false){
   answer = (guess > n)   //  i.e. evaluate boolean expression
   guess = guess*2
}
//  To reach here, we must have guess/2 <=  n < guess
binarySearch(n,  [guess/2, guess] )
```

   Both the while loop and the binary search take time proportional to $\log n$.

7. Here are two different solutions.

```
//  SOLUTION 1

      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 2   ------

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

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

9. First consider the iterative method. It computes $x*x$ (giving a result with $2M$ digits) and then multiplies the result by x (giving a result with $3M$ digits), and then multiplies the result by x etc. The first multiply takes time cM^2. The second multiply $(x * x^2)$ takes time $cM * (2M)$. The third multiply $(x * x^3)$ takes time $cM * (3M)$. The last or $(n-1)^{th}$ multiply $(x * x^{n-1})$ takes time $cM * ((n-1)M)$. So the total time taken is:

$$cM * M + cM * (2M) + cM * (3M) + ... + cM * ((n-1)M)$$

which is

$$cM^2(1 + 2 + 3 + ... + (n-1))$$

or

$$c\ M^2\ \frac{n(n-1)}{2}$$

Next consider the recursive method. The main point to make here is that

$$x^n = x^{\frac{n}{2}}\ x^{\frac{n}{2}}$$

and notice that $x^{\frac{n}{2}}$ has $\frac{n}{2}M$ digits. So multiplying $x^{\frac{n}{2}}$ by itself takes time $c(\frac{n}{2}M)(\frac{n}{2}M)$ which is roughly half the time it takes for the iterature method. But this ignores all the work that was needed to compute $x^n 2$ itself.

In terms of $O(\ )$ ideas, you can already see that the recursive method for computing x^n is no faster than the iterative method. And the recursive one might even be worse. (In fact, the two are the same. But I won't torture you with the proof of that. I'll consider myself satisfied if you get to here!)