Some solutions to Practice Questions. If the solution is not provided here, it is likely in the lecture slides or Midterm Review questions.

Algorithm Design (iterative and recursive)

Design algorithms for the following problems and represent them with flowcharts and pseudocode…

1. Design an algorithm that will search a list for a given number. The algorithm should takes a list of numbers, the length and the number you are looking for. If the number is found, the index it was found at should be returned, otherwise it will return a -1. Design both an iterative and recursive solution. Compare the runtime efficiency of the two.
   Examples:
   in list [1, 3, 5, 2] looking for 3 would return 1 since 3 is at index 1 of the list
   in list [1, 3, 5, 2] looking for 7 would return -1 since 7 is not found in the list

   iterative solution in lecture slides and midterm practice

   recursive solution (O(n)):

   ```
   find(array, length, val)
         if(length == 0)
               return -1
         else
               if (val == array[length-1])
                     return length-1
               else
                     return find(array, length-1, val)
   ```

2. Design an algorithm that takes a list of numbers and the length of that list as input. The algorithm should count the number of values in that list that are odd. The algorithm should return the number of odd values in the list. Design both an iterative and recursive solution. Compare the runtime efficiency of the two.
   NOTE: You can assume there is function called **odd** that take a number and returns true if the number is odd and false if it isn’t. You may call this function in your algorithm.

   iterative solution (O(n)):

   ```
   sumOdd (array, length)
         count = 0
         totalOdds = 0
         while(count<length)
               if (odd? (array[count])
                     totalOdds = totalOdds + 1
               count = count+1
         return totalOdds
   ```

   recursive solution (O(n)):

   ```
   sumOdd(array, length)
         if(length == 0)
               return 0
         else
               if (odd (array[length-1]))
                     return 1 + summOdd(length-1)
               else
                     return summOdd(length-1)
   ```

3. Design an algorithm that takes a number between 1 and 6 as input and rolls a dice over and over until the number rolled is the same as the input number. The algorithm should count and return the number of dice rolls it took to roll the input number.
   NOTE: You can assume there is a function called rollDice that doesn't take any input but returns a random number between 1 and 6. You may call this function in your algorithm.

4. Design an algorithm that takes a list of numbers and the length of that list as input. The algorithm should find and return the biggest number in the list. Design both an iterative and recursive solution. Compare the runtime efficiency of the two.
   NOTE: You can assume the list has at least one number in it.

   iterative solution (O(n)):

```
biggest (array, length)
      biggest = array[0]
      count = 1
      while(count<length)
            if (array[count]> biggest)
                  biggest = array[count]
            count = count+1
      return biggest
```

   recursive solution (O(n)):

```
biggest (array, length)
      if(length == 0)
            return the smallest possible number
      else
            biggestInRestOfList = biggest (length-1)
            if  (array[length-1]) > biggestInRestOfList
                  return array[length-1])
            else
                  return biggestInRestOfList
```

5. Design a recursive algorithm that takes a binary tree and doubles every value in the tree.

```
doubleValues (tree)
      if(tree is empty)
            return
      else
            tree's node value = 2 * tree's node value
            doubleValues(tree's left)
            doubleValues(tree's right)
            return
```

6. Design a recursive algorithm that takes a binary tree and a number (n) and counts the number of nodes in the tree with values greater than n.

```
countGreater (tree, n)
      if(tree is empty)
            return 0
      else
            if (tree's node value > n)
                  return ( 1 + doubleValues(tree's left)
                             + doubleValues(tree's right) )
            else
                  return ( doubleValues(tree's left)
                           + doubleValues(tree's right) )
```