Recursion is a powerful technique that is used in mathematics and computer science to solve a problem by repeatedly breaking it down into smaller instances of the same problem. The sign of a recursive function is that it calls itself. A couple of famous examples are: the Fibonacci sequence and Tower of Hanoi problem. But there are many everyday problems that lend themselves to recursive solutions.

Imagine we wanted to, given an input integer like 10, sum up all the numbers from 1 until the input, i.e. sum up [1,…,10].

We know this is solved by doing 1 + 2 + 3 + … + 10, so we can first solve it iteratively using a for loop:

```
public static int sumUp(int n) {
     int sum = 0;

     for (int i = 1; i <= n; i++) {
          sum += i;
     }
     return sum;
}
```

We note that the main operation here is the addition of a set of numbers. We can imagine the problem by visualizing a stack of numbers, where 10 is at the top, and 1 is at the bottom. It's like we are adding up all the numbers in the stack (the 'stack' here does not refer to the Java Collection, just a vertical stack of numbers).

But how do we get from the item at the top (i.e. 10), to each item underneath it? That's easy – we subtract 1 from each layer to get the layer underneath it. So we start with 10 at the top, then subtract 1 to get 9, then subtract 1 again to get 8, etc.

This subtraction by 1 is a repetitive operation, so we can have the function call itself:

```
public static int sumUp(int n) {

     return n + sumUp(n - 1); // Recursive call or smaller case
}
```

So, this recursive function is doing: 10 + (9 + (8 + (7 + ...))). The solution has two parts: At the beginning the first part adds 10, and then calls 'itself' ie *sumUp* on 10-1 (the smaller case) ie 9. So the call is *sumUp(9). sumUp(9*) will in turn return 9 + *sumUp(8).* This subsequent call will include 8 + *sumUp(7).* These series of successive recursive calls will continue until we decide when to stop. This last execution is called the *base case* or *exit case.*

But the way *sumUp()* is written right now, it will get to *sumUp(1*), where it will just do: 1 + *sumUp(0)*, which will do 0 + *sumUp(-1).*
In fact, this will keep going infinitely! Eventually we will get 10 + 9 + 8 + ... + 0 + ... -100 + ...

We definitely want to halt the function call somewhere in the function body. So, we create the 'base case' which gives the recursive function a condition at which it can finally stop and return a definite value instead of another recursive call.

```
public static int sumUp(int n) {

     //base case: return definite value to prevent recursive
call
     if (n <= 1) {
          return 1;
     }

     return n + Sum(n - 1); // Recursive call, smaller case
}
```

Now, sumUp(1) will just return 1, instead of 1 + sumUp(0). Perfect.

Try a similar problem, and visualize the input being broken into multiple layers, where a certain repetitive operation can break down each layer into the one underneath. Also think the halt condition for the lowest layer.

To sum up, recursive problems involve repetition, a base case (or exit case) and a smaller case. (the recursive call to the solve the same problem, but with a smaller "value" so it is moving a step closer to the exit case).

Many recursive problems can be solved using iteration, but often the recursive solution is more elegant – sometimes there is no other way to solve the problem except by recursion.

Think about the mechanics and efficiency of recursive solutions. Would they be more or less expensive in terms of resources than their iterative counterparts?

When solving problems using recursion, it is a good idea to identify the base case (exit case) and the smaller case as part of the algorithmic problem - solving process. This will make the code writing easier.

Ex1) Convert the following iterative function *countDigits()* into a recursive one.

Note: *countDigits()* will take an input like 12345, and repeatedly separates the input into layers. In the loops, **n** will 1234, then 123, then 12, then 1, then 0. Each time, it whittles away the modified input. The number of loops required to remove all the digits, is itself the total number of digits.

What is halting condition for the base case?

```
public static int countDigits(int n) {

     int count = 0;
     while (n != 0) {
          n = n / 10;
          count++;
     }
     return count;
}
```

Ex2) Try a similar problem. Write recursive function sumUpDigits(integer n), which takes in a value like 12345 and returns 1+2+3+4+5=15. Use the modulus operator to separate each layer, with the layers being 12345, 1234, 123, 12, and 1. Note that 12345 % 10 = 1234, and 1234 % 10 = 123, etc.

Ex3) Here's another useful one. Write recursive function printEvens(integer n) to print all the even numbers from **n...0**. For example, printEvens(15) prints 14, 12, 10, 8, 6, 4, 2, 0.

Again, think about how to get from the input to the layer underneath it, and also what the halt condition would be (hint: it should stop at the smallest even number, zero).

Let's extend our thinking to not only integers, but also strings and characters. The concept will remain the same – given an input string, figure out which operation to use to repeatedly break the top string into a slighter smaller string, while keeping in mind this will typically stop when the string is empty.

Here is a simple example to count the length of a string, or the number of characters.

```
public static int strLen(String str) {

     if (str.isEmpty()) {
          return 0; // base case
     }

     // splitting operation takes away first char
     char first = str.charAt(0);
     String rest = str.substring(1);

     // recursive call with rest of string
     // add 1 for each recursive call
     return 1 + strLen(rest);
}
```

We first identified how to get from the top layer (the input string) to the next layer down – by splitting the string right after the first character. We then do a recursive call with the rest of the string, and we keep adding 1 each time we do this. This counts up the total number of recursive calls.

The key is that number of recursive calls equals the number of times we split the string, equals the number of elements in the string!

P4) Write a recursive function called reverse(), which takes in an input string and returns it reversed. For example, if the input is "hello", the function should return "olleh".

Hint: keep splitting the string until you reach the base case of empty string, then build back from the last character to the first. Think about reaching the lowest layer, then building back to the top layer.

Ex5) Using similar concepts from Ex4, write a recursive function called *palindrome(),* which takes in an input string and returns a *boolean* to indicate if it's palindrome or not. For example, "racecar" would return true, while "building" would return false.

Hint: think about splitting the string at the ends. Checking a string for being a palindrome involves checking the first and last character for equivalence, then the second and second-last, and so on until you reach the middle.

Now let's try solving some other string problems with recursion.

Ex6) Write a function called *findIndex*(String str, char target) which takes in a string and target character, and returns the index of the character in the string. For example, *findIndex*("hello", "o") should return 4.

Ex7) Write a recursive function called *countOccur*(String str, char target) which counts the number of times a specified character appears in an input string. For example, *countOccur("hello", "l")* should return 2.

Hint: One approach is to keep splitting the string until you get to the base case, where the string is empty. Then, as you return from the lowest layer to the top layer, check each layer for a character match and increment count if there is a match.

Ex8) Modify *countOccur*() to instead count the number of vowels in the input string. The function will be called *countVowels*(String str). The vowels are the set (a, e, i, o, u).

Recursion is a powerful tool that can be used to process recursively defined data structures. Recall that a *ListNode* class has a recursive definition. A *ListNode* is chained to another *ListNode* in the LinkedList Class. We can use recursion to retrieve elements from and to process LinkedList objects. The base case for the recursion would be the empty list (or null) and the smaller case is one less than the current size of the list and would be the obtained by moving "down" the list to the next node.

Redo the *LinkedIntList* problems in Practice-it Chapter 16 by using recursion instead of iteration.