Write a method `evens` that accepts an integer parameter *n* and that returns the integer formed by removing the odd digits from *n*. The following table shows several calls and their expected return values:

Call	**Valued Returned**
`evens(8342116);`	`8426`
`evens(4109);`	`40`
`evens(8);`	`8`
`evens(-34512);`	`-42`
`evens(-163505);`	`-60`
`evens(3052);`	`2`
`evens(7010496);`	`46`
`evens(35179);`	`0`
`evens(5307);`	`0`
`evens(7);`	`0`

If a negative number with even digits other than 0 is passed to the method, the result should also be negative, as shown above when -34512 is passed. Leading zeros in the result should be ignored and if there are no even digits other than 0 in the number, the method should return 0, as shown in the last three outputs.

```
public int evens(int n) {
      if (n==0) return 0;
    int lastdigit = n%10; // 4109 % 10 = 9
    if (lastdigit%2 != 0){
        return evens(n/10);
    }
    else {
        return 10 * evens(n / 10) + lastdigit;
    }
}
```

Write a method `writeSquares` that accepts an integer parameter *n* and prints the first *n* squares separated by commas, with the odd squares in descending order followed by the even squares in ascending order. The following table shows several calls to the method and their expected output:

Call	Valued Returned
`writeSquares(5);`	`25, 9, 1, 4, 16`
`writeSquares(1);`	`1`
`writeSquares(8);`	`49, 25, 9, 1, 4, 16, 36, 64`

Your method should throw an `IllegalArgumentException` if passed a value less than 1. Note that the output does not advance onto the next line.

```
public void writeSquares(int n) {
      if (n<1) throw new IllegalArgumentException();
      else if (n == 1) System.out.print(1);
      else if (n%2 ==1 ){
            System.out.print((int)(Math.pow((double) n, 2.0)));
            System.out.print(", ");
            writeSquares(n-1);
    }
      else {
            writeSquares(n-1);
            System.out.print(", ");
            System.out.print((int)(Math.pow((double) n, 2.0)));
    }
}
```

Write a method `writeSequence` that accepts an integer *n* as a parameter and prints a symmetric sequence of *n* numbers with descending integers ending in 1 followed by ascending integers beginning with 1, as in the table below:

Call	**Output**
`writeSequence(1);`	`1`
`writeSequence(2);`	`1 1`
`writeSequence(3);`	`2 1 2`
`writeSequence(4);`	`2 1 1 2`
`writeSequence(5);`	`3 2 1 2 3`
`writeSequence(6);`	`3 2 1 1 2 3`
`writeSequence(7);`	`4 3 2 1 2 3 4`
`writeSequence(8);`	`4 3 2 1 1 2 3 4`
`writeSequence(9);`	`5 4 3 2 1 2 3 4 5`
`writeSequence(10);`	`5 4 3 2 1 1 2 3 4 5`

Notice that for odd numbers the sequence has a single 1 in the middle while for even values it has two 1s in the middle. Your method should throw an `IllegalArgumentException` if passed a value less than 1.

```
public void writeSequence(int n) {

      if (n<1) throw new IllegalArgumentException();

    else if (n==1) System.out.print("1 ");

    else {

        System.out.print((n+1)/2);

        System.out.print(" ");

        if (n !=2) {

            writeSequence(n-2);

        }

        System.out.print(((n+1)/2) + " ");

    }

}
```

Given a non-negative int n, return the sum of its digits recursively (no loops). Note that mod (%) by 10 yields the rightmost digit (126 % 10 is 6), while divide (/) by 10 removes the rightmost digit (126 / 10 is 12).

sumDigits(126) → 9
sumDigits(49) → 13
sumDigits(12) → 3

```
public int sumDigits(int n) {
  if (n/10 < 1) return n;
  else {
    return (n%10) + sumDigits(n/10);
  }
}
```

Given a non-negative int n, return the count of the occurrences of 7 as a digit, so for example 717 yields 2. (no loops). Note that mod (%) by 10 yields the rightmost digit (126 % 10 is 6), while divide (/) by 10 removes the rightmost digit (126 / 10 is 12).

count7(717) → 2
count7(7) → 1
count7(123) → 0

```
public int count7(int n) {
  if (n/10 <1){     // if its only one digit - check if its 7
    if (n != 7) return 0;
    else if (n == 7) return 1;
  }
  else {
    if ((n%10) == 7) {    // if rightmost digit is 7
      return 1 + count7(n/10); //count+1 then count7 remaining digits
    }
    else {return count7(n/10);}
  }
  return 0;
}
```

Given a string, compute recursively (no loops) the number of lowercase 'x' chars in the string.

countX("xxhixx") → 4
countX("xhixhix") → 3
countX("hi") → 0

```java
public static int countX(String str) {
            //Base Case
            if (str.length()==0) return 0;

            //Recursive Case:
            // Just start at the first letter then
            // recurse on the remaining string
            else if (str.charAt(0)=='x')
                  return 1 + countX(str.substring(1, str.length()));
            else
                  return countX(str.substring(1, str.length()));
      }
```

Given a string, compute recursively (no loops) the number of times lowercase "hi" appears in the string.

countHi("xxhixx") → 1
countHi("xhixhix") → 2
countHi("hi") → 1

```
public int countHi(String str) {
  // if length is 0 or 1, return 0
  if (str.length()==0 || str.length()==1) return 0;
  // if found hi, remove these 2 char from string and recurse
  else if (str.charAt(0)=='h' && str.charAt(1)=='i') {
    return 1 + countHi(str.substring(2));
  }
  // if not, remove first char from string and recurse
  else {
    return countHi(str.substring(1));
  }
}
```

Given a string, compute recursively (no loops) a new string where all the lowercase 'x' chars have been changed to 'y' chars.

changeXY("codex") → "codey"
changeXY("xxhixx") → "yyhiyy"
changeXY("xhixhix") → "yhiyhiy"

```
public static String changeXY (String str) {
            //Base Case:
            if (str.length()==0)
            return str;

      //Recursive Case:
            else if (str.charAt(0) == ('x'))
            return 'y' + changeXY(str.substring(1));
            else
            return str.charAt(0) + changeXY(str.substring(1));
      }
```

Given a string, return true if it is a nesting of zero or more pairs of parenthesis, like "(())" or "((()))". Suggestion: check the first and last chars, and then recur on what's inside them.

nestParen("(())") → true
nestParen("((()))") → true
nestParen("(((x))") → false

```
public boolean nestParen(String str) {
  if (str.length()==0) return true;
  else if (str.charAt(0)=='(' && str.charAt(str.length()-1)==')'){
    return nestParen(str.substring(1,str.length()-1));
  }
  else{
    return false;
  }
}
```

Given a string and a non-empty substring **sub**, compute recursively the number of times that sub appears in the string, without the sub strings overlapping.

strCount("catcowcat", "cat") → 2
strCount("catcowcat", "cow") → 1
strCount("catcowcat", "dog") → 0

```
public int strCount(String str, String sub) {
              if(str.length() < sub.length()) return 0;


              int len = sub.length();


              if(str.substring(0,len).equals(sub))
                return 1 + strCount(str.substring(len), sub);


              return strCount(str.substring(1), sub);
            }
```

Given a string and a non-empty substring **sub**, compute recursively the largest substring which starts and ends with sub and return its length.

strDist("catcowcat", "cat") → 9
strDist("catcowcat", "cow") → 3
strDist("cccatcowcatxx", "cat") → 9

```
public int strDist(String str, String sub) {
  if (str.length() < sub.length())
                  return 0;
if (str.startsWith(sub)) {
                  if (str.lastIndexOf(sub) > 0)
                        return str.lastIndexOf(sub) + sub.length();
                  else
                        return sub.length();
            }
            return strDist(str.substring(1), sub);


}
```

Given an array of size n, generate and print all possible combinations of r elements in array. For example, if input array is {1, 2, 3, 4} and r is 2, then output should be {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4} and {3, 4}.

Method 2 (Include and Exclude every element)

We one by one consider every element of input array, and recur for two cases:

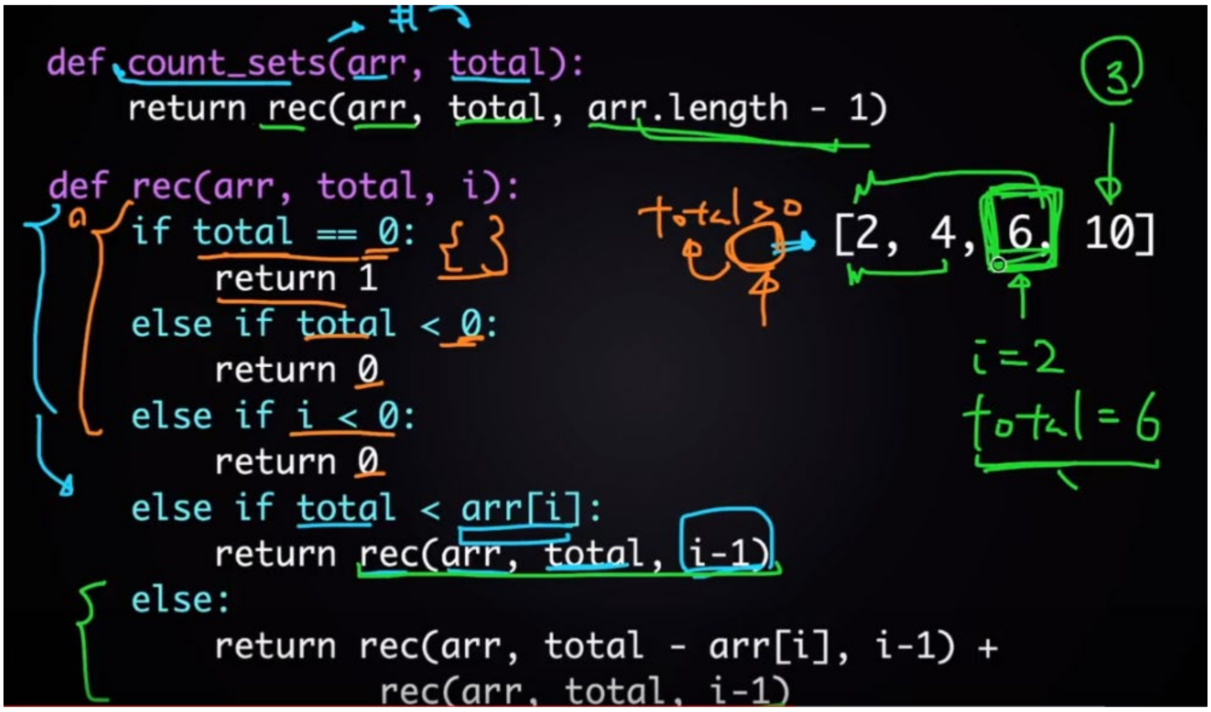
1) The element is included in current combination (We put the element in data[] and increment next available index in data[])
2) The element is excluded in current combination (We do not put the element and do not change index)

When number of elements in data[] become equal to r (size of a combination), we print it.

Refer to url

Given array of integer, arr, find the number of subsets of this array that adds up to a certain number, total.

arr=[2,4,6,10] total=16
Number of subsets=2
Subsets = {6,10} & {2,4,10}



```
def count_sets(arr, total):
    return rec(arr, total, arr.length - 1)

def rec(arr, total, i):
    if total == 0:
        return 1
    else if total < 0:
        return 0
    else if i < 0:
        return 0
    else if total < arr[i]:
        return rec(arr, total, i-1)
    else:
        return rec(arr, total - arr[i], i-1) +
               rec(arr, total, i-1)
```

Given a string and a non-empty substring **sub**, compute recursively the largest substring which starts and ends with sub and return its length.

strDist("catcowcat", "cat") → 9
strDist("catcowcat", "cow") → 3
strDist("cccatcowcatxx", "cat") → 9

```java
public int strDist(String str, String sub) {
    if (sub.isEmpty()) {
        throw new IllegalArgumentException("sub mustn’t be empty");
    }
    if (str.length() <= sub.length()) {
        if (str.equals(sub)) {
            return str.length();
        } else { // sub cannot be in str
            return 0;
        }
    }
    if (str.startsWith(sub)) {
        if (str.endsWith(sub)) {
            return str.length();
        } else {
            return strDist(str.substring(0, str.length() - 1), sub);
        }
    } else {
        return strDist(str.substring(1), sub);
    }
}
```