Recursion

A structure or function that is defined in terms of itself. We will study functions that call themselves!

Example: factorial function

```
3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1
5! = 5 * 4 * 3 * 2 * 1 = 120
         ^-----------^
              4!
   = 5 * 4!

n! = n * (n-1) * ... * 2 * 1
   = n * (n-1)!
```

Let's code this recursively:

```
int fact(int n) {
     return n * fact(n - 1);
}
```

This function is actually infinite. In an program this will cause a StackOverflowException. "Thrown when a stack overflow occurs because an application recurses too deeply."

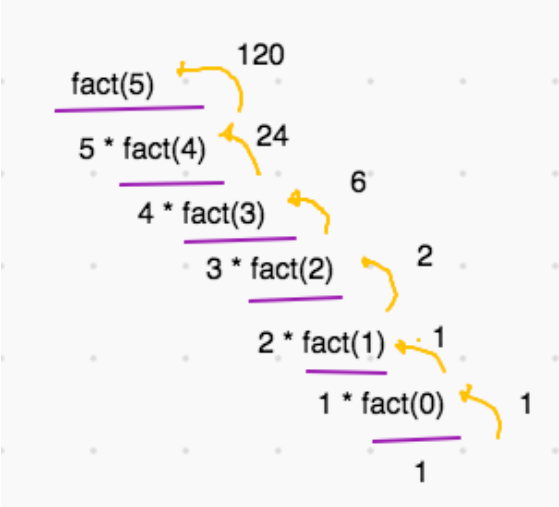
We have to add to the definition

```
0! = 1
```

and update our code

```
int fact(int n) {
     if(n == 0)    // base case
         return 1;
     else          // recursive case
         return n * fact(n - 1);
                //  ^----------^  a recursive call
}
```

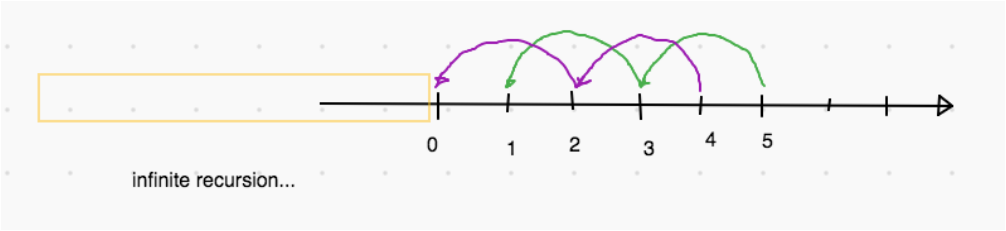
Here is a trace of a call to `fact(5)`.



Example: an ”odd” function

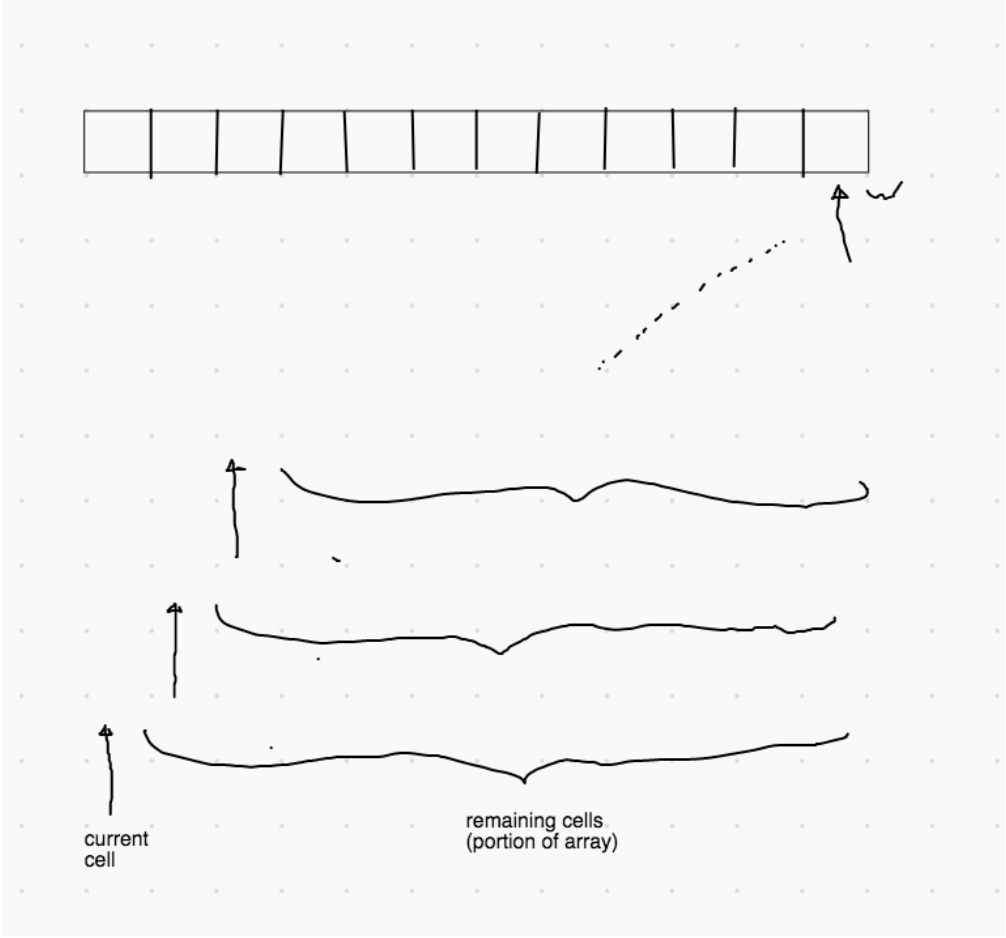
```
public boolean isOdd(int x) {
    if(x == 1)      // base case!
       return true;
    else if(x == 0)
       return false;
    else            // recursive case
       return isOdd(x - 2);
}
```

We need to reason about the recursive call: it should ”approach” one of the base cases:



Recursive methods using arrays.

There are a few ways we can think of arrays as recursive:



Challenge: compute the sum of an array, recursively!

```
// This method will sum up the values in arr from pos to the end of the
// array.
private int sum(int[] arr, int pos) {
    if(pos == arr.length)
       return 0;
    else
       return arr[pos] + sum(arr, pos + 1);
}
```

```
// call the recursive "helper" method.
public int sum(int[] arr) {
    return sum(arr, 0);
}
```

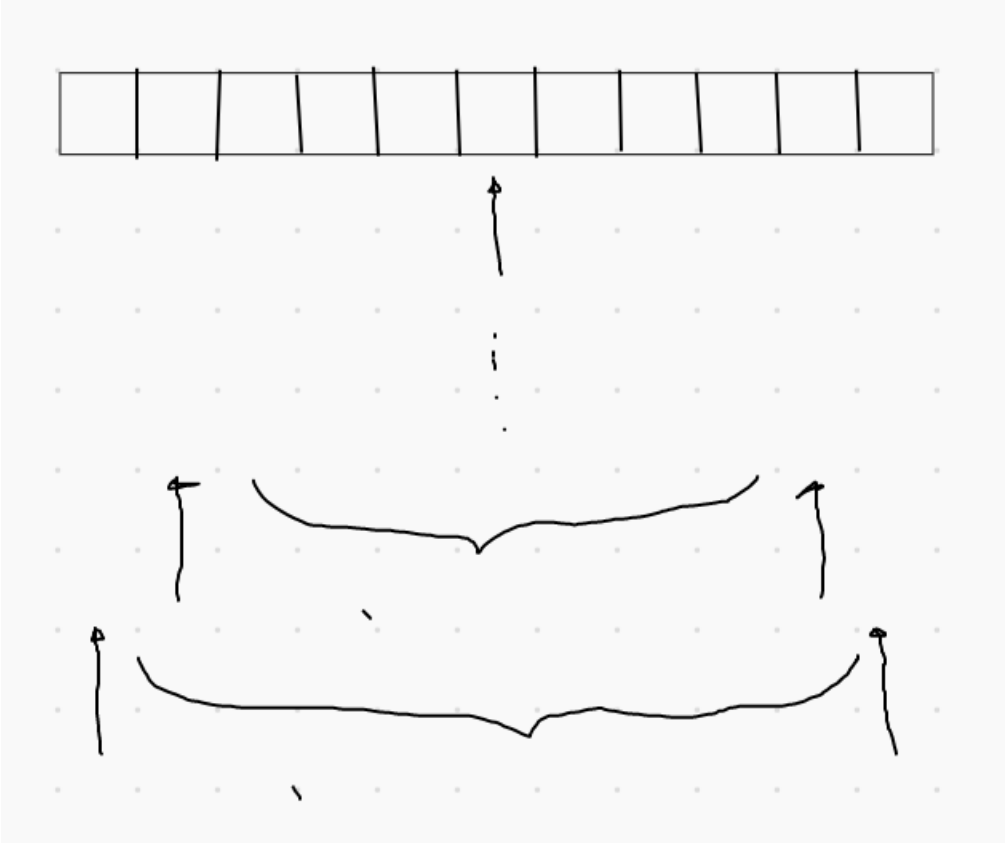
Challenge: compute the max of an integer array, recursively!

```
public int max(int[] arr) {
    if(arr.length == 0) throw new RuntimeException();
    return max(arr, 0);
}

private int max(int[] arr, int pos) {
   if(pos == arr.length - 1)
      return arr[pos];
   else {
      int tmp = max(arr, pos + 1);
      if(arr[pos] > tmp)
          return arr[pos];
      else
          return tmp;
   }
}
```

Challenge: code a palindrome detector, recursively.

Palindrome: a words spelled the same forwards and backwards. Ex: ”racecar”, ”Anna”, ”Nitin”, ”taco cat”.

Idea:



Version 1 with extra parameters:

```
public static boolean isPalindrome(String text) {
     return isPalindrome(text, 0);
}

private static boolean isPalindrome(String text, int pos) {
     if(pos ≥ text.size()/2)
         return true;
     if(text.charAt(pos) ≠ text.charAt(text.size() - pos - 1)
         return false;
     return isPalindrome(text, pos + 1);
}
```

Version 2 with only `String`s:

```
public static boolean isPalindrome(String text) {
```

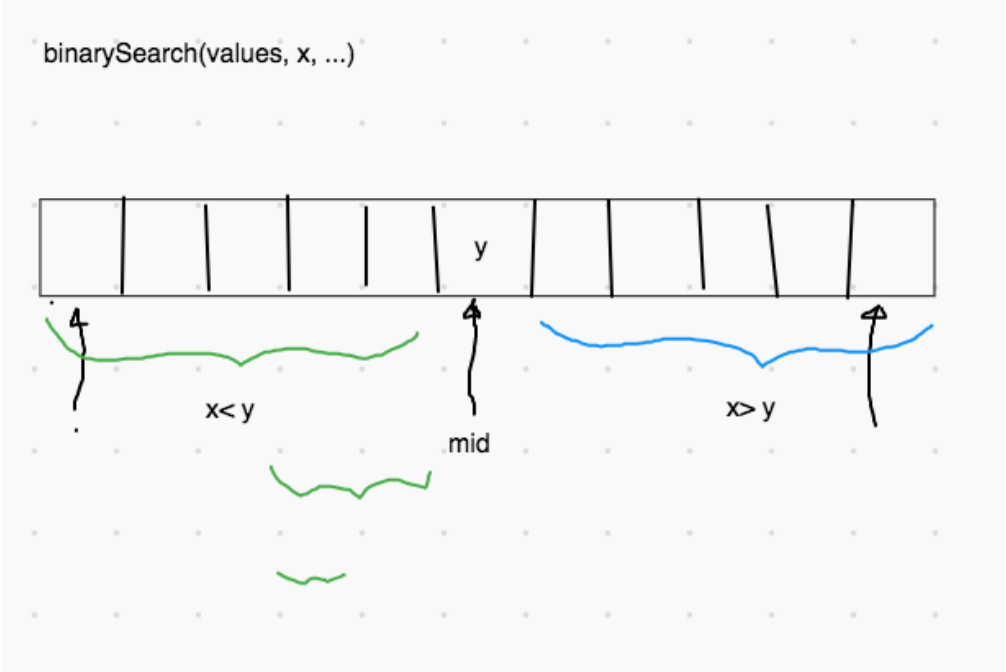
```
    if(text.size() == 1) return true;
    if(text.charAt(0) != text.charAt(text.size() - 1)
         return false;
    return isPalindrome(text.substring(1, text.size() - 2));
}
```

Version 2 is less efficient since the `substring()` creates a new `String` object per call.

Challenge: write binary search, recursively.

Returns the position of the value in the array, or -1 if not found.

Idea:

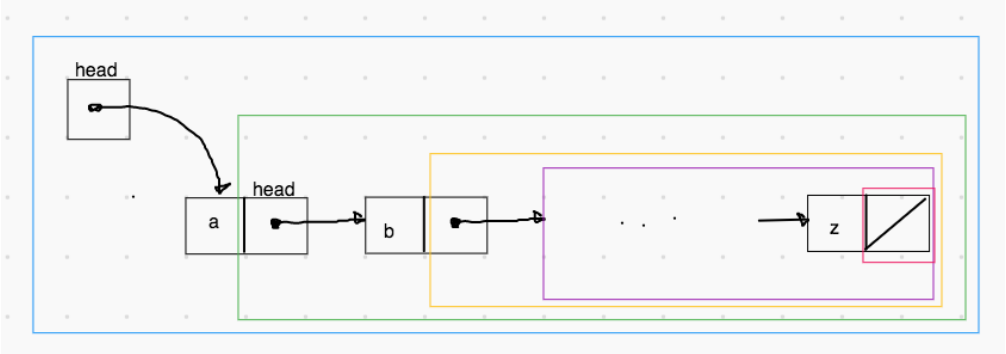



```
public static <T extends Comparable<T>> int binarySearch(T[] values, T value) {
   return binarySearch(values, value, 0, values.length-1);
}

public static <T extends Comparable<T>> int binarySearch(T[] values, T value, int low, int

     if(low > high)
        return -1;
```

```
    int mid = (low + high) / 2;
    int cmp = value.compareTo(values[mid]);
    if(cmp == 0)
        return mid;
    else if(cmp < 0)
        return binarySearch(values, value, low, mid - 1);
    else
        return binarySearch(values, value, mid + 1, high);
}
```

Recursive methods using link chains.

Recursive structure: each chain contains a smaller chain!



Example: recursive chain print

```
public static void print(Link<T> head) {
    if(head == null) return;
    sout(head.element);
    print(head.next);
}
```

We can write check `head.next` before the recursive call, but it does introduce a precondition:

```
// precondition: head is not null ...  downside to this version.
public static void print(Link<T> head) {
    sout(head.element);
    if(head.next ≠ null)
```

```
        print(head.next);
}
```

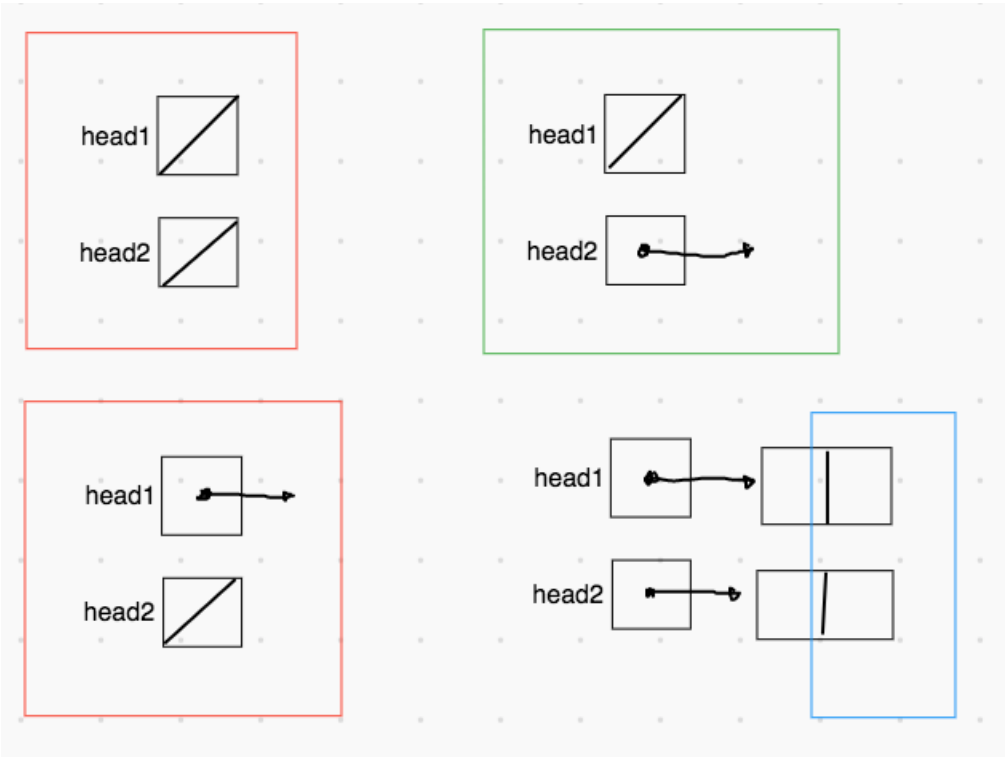
Challenge: recursive chain reverse print.

To print the elements in reverse order, you just need swap the recursive call with the print statement:

```
public static void reversePrint(Link<T> head) {
     if(head == null) return;
     reversePrint(head.next);
     sout(head.element);
}
```

Challenge: implement a method that determines if one chain is shorter than another, recursively!

Returns true of the number of elements in `head1` is less than the number of elements in `head2`, false otherwise.



```
public static boolean isShorter(Link<T> head1, Link<T> head2) {
     if(head1 == null && head2 == null)
          return false;
     if(head1 ≠ null && head2 == null)
          return false;
     if(head1 == null && head2 ≠ null)
          return true;
     return isShorter(head1.next, head2.next);
}
```

We can simplify the base case to:

```
public static boolean isShorter(Link<T> head1, Link<T> head2) {
     if(head2 == null)
          return false;
     if(head1 == null)
          return true;
     return isShorter(head1.next, head2.next);
}
```