Recursion, tail recursion

How is recursion implemented?

The processor natively supports *jumping* from one instruction to another

Function calls are implemented using a *call stack*

To call a procedure:

- Push the next instruction and the values of local variables on the call stack, then jump to the beginning of the procedure

To return from a procedure:

- Pop the instruction and the values of local variables from the call stack, and jump to the instruction, restoring the values of the local variables

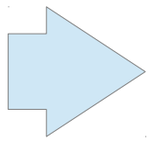
A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

Next line	Value of n

A recursive function



```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 3

Next line	Value of n

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 3

Next line	Value of n

Prints 3

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 3

Next line	Value of n

Prints 3

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 2

Next line	Value of n
5	3

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 2

Next line	Value of n
5	3

Prints 2

A recursive function

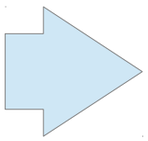
```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 2

Next line	Value of n
5	3

A recursive function



```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 1

Next line	Value of n
5	3
5	2

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 1

Next line	Value of n
5	3
5	2

Prints 1

A recursive function

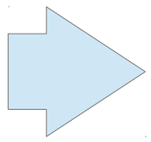
```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 1

Next line	Value of n
5	3
5	2

A recursive function



```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 0

Next line	Value of n
5	3
5	2
5	1

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 0

Next line	Value of n
5	3
5	2
5	1

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 1

Next line	Value of n
5	3
5	2

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 0

Next line	Value of n
5	3
5	2
6	1

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 0

Next line	Value of n
5	3
5	2
6	1

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 1

Next line	Value of n
5	3
5	2

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 2

Next line	Value of n
5	3

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 1

Next line	Value of n
5	3
6	2

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 1

Next line	Value of n
5	3
6	2

Prints 1

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }
```

How much memory does this use?

Don't forget to include the stack!

O(n)
memory use!

Memory use of recursive functions

Calling a function pushes information on the call stack

Hence recursive functions use memory in the form of the call stack!

Total memory use: **O(maximum recursion depth)**

Another recursive function

```
void hello() {
    System.out.println(“hello world”);
    hello();
}
```

What is this program supposed to do?

What does it actually do?

Another recursive function

```
void hello() {
    System.out.println(“hello world”);
    hello();
}
```

What is this program supposed to do?

What does it actually do?

```
Exception in thread "main"
java.lang.StackOverflowError
```

The recursive call to *hello* fills the call stack!

Tail calls

```
void hello() {
   System.out.println(“hello world”);
   hello();
}
```

The recursive call is the last thing *hello* does before it returns

This is called a *tail call*, and *hello* is *tail recursive*

There's no need to push anything on the call stack for a tail call – but most languages (e.g. Java) do it anyway :(

Languages with *tail call optimisation* or *TCO* (e.g. Haskell, ML, Scala, Erlang, Scheme) don't bother to push anything on the call stack when it's a tail call – this way tail recursion is as efficient as a loop

Is this a tail call?

```
void hello(int n) {
    if (n > 0) {
      System.out.println(“hello world”);
      hello(n-1);
    }
}
```

Is this a tail call?

```
void hello(int n) {
    if (n > 0) {
      System.out.println(“hello world”);
      hello(n-1);
    }
}
```

Yes! - nothing more happens after the recursive call to *hello*

Is this a tail call?

```
int fac(int n) {
    if (n == 0) return 1;
    else return n * fac(n-1);
}
```

Is this a tail call?

```
int fac(int n) {
    if (n == 0) return 1;
    else return n * fac(n-1);
}
```

No! - after the recursive call *fac(n-1)* returns, you have to multiply by *n*

Tail recursion using a loop

You can always write a tail-recursive function using a *while(true)*-loop instead:

```
void hello(int n) {
    while(true) {
      if (n > 0) {
        System.out.println("hello world");
        hello(n-1); n = n-1;
      } else break;
      }
}
```

Instead of returning,
exit the loop

Instead of making
a recursive call,
go through the loop again

Tail recursion using a loop

Tidied up a bit:

```
void hello(int n) {
    while (n > 0) {
        System.out.println(“hello world”);
        n = n-1;
    }
}
```

Searching in a binary tree

```
Node<E> search(Node<E> node, int value) {
    if (node == null) return null;
    if (value == node.value) return node;
    else if (value < node.value)
      return search(node.left);
    else
      return search(node.right);
}
```

The same, tail-recursive

```
Node<E> search(Node<E> node, int value) {
    while(true) {
    if (node == null) return null;
    if (value == node.value) return node;
    else if (value < node.value)
      node = node.left;
    else
      node = node.right;
    }
}
```

When programming in languages like Java that don't have TCO, you might need to do this transformation yourself!

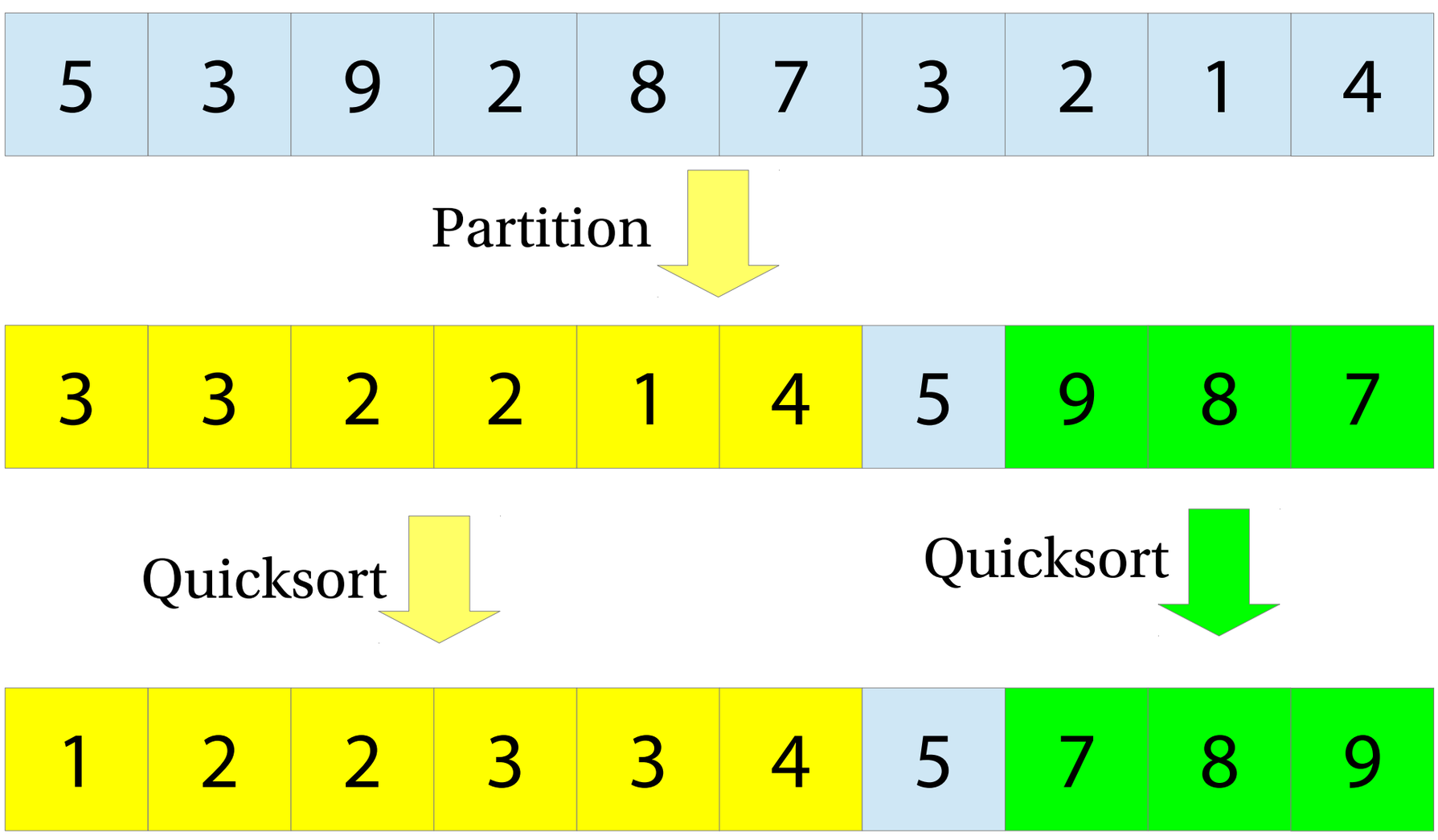
Tail calls

Remember that the total amount of extra memory used by a recursive function is **O(maximum recursion depth)**

If the language supports TCO, the amount is instead **O(maximum depth of non-tail recursive calls)** – better!

In languages without TCO, you can transform tail recursion into a loop to save stack space (memory)

A bigger example: quicksort



Quicksort

We said that quicksort was in-place, but it makes two recursive calls!

```
void sort(int[] a, int low, int high) {
    if (low >= high)
    int pivot =                    high);
    sort(a,
    sort(a
}
```

How much memory does this use in the worst case?

O(n), including the call stack!

Quicksort

Let's make a version of quicksort that uses O(log n) space.

```
void sort(int[] a, int low, int high) {
    if (low >= high) return;
    int pivot = partition(a, low, high);
    sort(a, low, pivot-1);
    sort(a, pivot+1, high);
}
```

Quicksort in O(log n) space

Idea: if we are using a language with TCO, the *second* recursive call uses no stack space (it's a tail call)!

Hence, the total memory use is O(recursion depth of *first* recursive call)

So: sort the *smaller* partition with the first recursive call, and the bigger one with the second recursive call

If the array has size *n*, the smaller partition has size at most *n/2*, so the recursion depth is at most O(log n).

Sorting the smaller partition first

In languages with TCO (i.e. not Java), this uses O(log n) space.

```
void sort(int[] a, int low, int high) {
    if (low >= high) return;
    int pivot = partition(a, low, high);
    if (pivot – low < high – pivot) {
      sort(a, low, pivot-1);
      sort(a, pivot+1, high);
    } else {
      sort(a, pivot+1, high);
      sort(a, low, pivot-1);
    }
}
```

Sort the smaller partition first

Sorting the smaller partition first

In Java, we must transform the tail recursion into a *while(true)*-loop.

```
• void sort(int[] a, int low, int high) {
    while(true) {
    if (low >= high) return;
    int pivot = partition(a, low, high);
    if (pivot – low < high – pivot) {
      sort(a, low, pivot-1);
      sort(a, pivot+1, high); low = pivot+1;
    } else {
      sort(a, pivot+1, high);
      sort(a, low, pivot-1); high = pivot-1;
    }
    }
}
```