Tail recursion

(not on exam)

How is recursion implemented?

When you call function *B* from function *A*, the processor stops executing *A* and starts executing *B* (obviously)

But when *B* returns, how does it know how to go back to *A*?

Answer: the *call stack*

- Before *A* calls *B*, it will push a record of what it was doing: the next instruction to be executed, plus the values of all local variables
- When *B* returns, it will pop that record and see it should return to *A*

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

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

n = 3

Printed: 3

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

Printed: 3

Next line	**Value of n**

Push state onto call stack

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
5	3

n = 2

Printed: 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);
```

Next line	Value of n
5	3

n = 2

Printed: 3 2

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

Printed: 3 2

Next line	Value of n
5	3

Push state onto call stack

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
5	3
5	2

n = 1

Printed: 3 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);
```

Next line	Value of n
5	3
5	2

n = 1

Printed: 3 2 1

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 }
```

Next line	Value of n
5	3
5	2

```
rec(3);
```

n = 1

Printed: 3 2 1

Push state onto call stack

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
5	3
5	2
5	1

n = 0

Printed: 3 2 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`

Printed: 3 2 1

Next line	**Value of n**
5	3
5	2
5	1

Pop state from call stack

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

Printed: 3 2 1

Next line	Value of n
5	3
5	2

Push state onto call stack

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
5	3
5	2
6	1

n = 0

Printed: 3 2 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

Printed: 3 2 1

Next line	Value of n
5	3
5	2
6	1

Pop state from call stack

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

Printed: 3 2 1

Next line	Value of n
5	3
5	2

Pop state from call stack

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

Printed: 3 2 1

Next line	Value of n
5	3

Push state onto call stack

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 }
```

Next line	Value of n
5	3
6	2

```
rec(3);
```

n = 1

Printed: 3 2 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);
```

Next line	Value of n
5	3
6	2

n = 1

Printed: 3 2 1 1

Prints 1

A recursive function

```
void rec(int n) {
  if (n > 0) {
    System.out.println(n);
    rec(n-1);
    rec(n-1);
 }
}
```

How much memory does this function use?

Don't forget to include the call stack!

A recursive function

```
void rec(int n) {
  if (n >
    Syste
    rec(n
    rec(n
  }
}
```

n levels of recursion
n items on call stack
O(n) memory use!

How much memory does this function use?

Don't forget to include the call stack!

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 from call stack:
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?

- Print “hello world” over and over again

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*

Idea: *don't bother pushing anything on the call stack* when making a tail call

- Since the function is going to do nothing afterwards except return again

Tail call optimisation

In languages with *tail call optimisation*:

- Tail calls don't push anything onto the call stack so don't use any stack space
- Hence tail recursion acts just like a loop
- This allows you to choose between using loops or recursion, whichever is more natural for the problem at hand

Most functional languages have TCO, since you're supposed to use tail recursion instead of looping:

- e.g. Haskell, ML, Scala, Erlang, Scheme
- but also some other civilised languages e.g. Lua

Unfortunately many languages (e.g. Java) don't :(

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 return;
   }
}
```

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

Explicitly return
when the recursion
is finished

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!

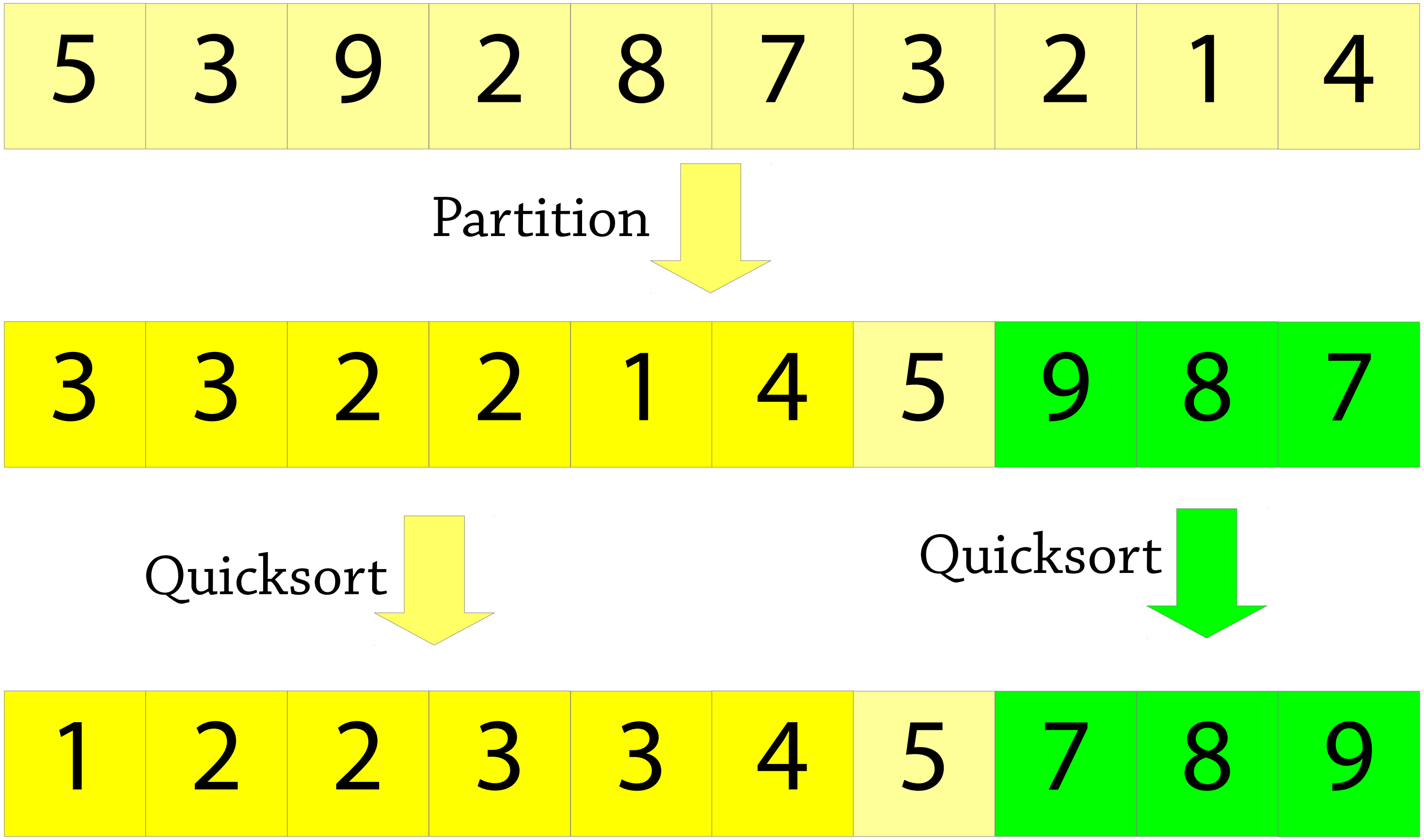
Tail calls

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



5	3	9	2	8	7	3	2	1	4

Partition

3	3	2	2	1	4	5	9	8	7

Quicksort

Quicksort

1	2	2	3	3	4	5	7	8	9

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) return;
   int pivot = partition(a, low, high);
   sort(a, low, pivot-1);
   sort(a, pivot+1, high);
}
```

How much memory does this use in the worst case, including the call stack?

Quicksort

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

```
void sort(int[] a, int  ow     t high) {
    if (low >= hi    re
    int pivot = p                  high);
    sort(a,
    sort(a, pi
}
```

How much m   is use in the worst case, inc   g the call stack?

O(n), including the call stack!

Quicksort

Let's make a version of quicksort that uses O(log n) stack 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;
   }
   }
}
```