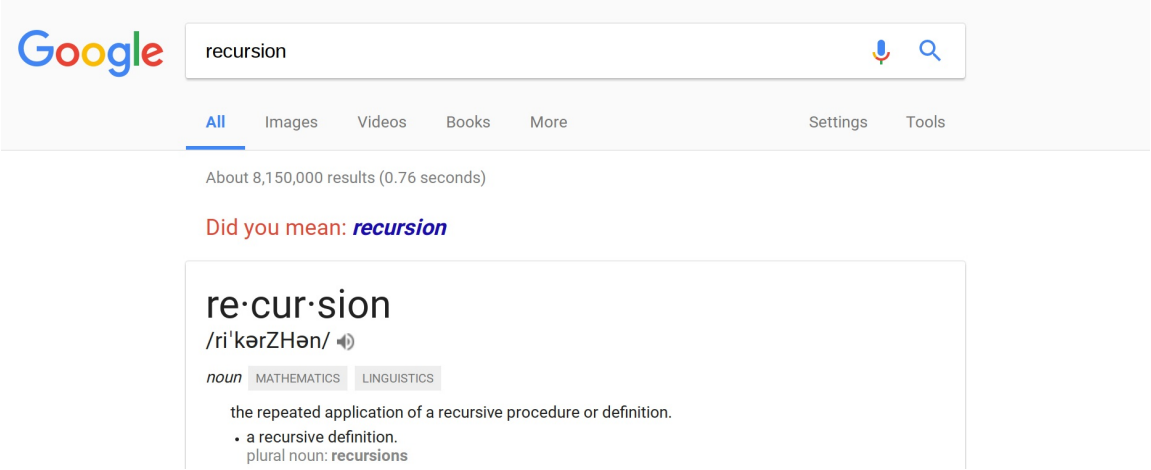
Recursion



Enable native scrolling

Recursion

Solving a problem using recursion means sucessively reducing the problem to a simpler form of itself until the solutions are trivial

Factorial

Factorial `n (n!)` is the product of all numbers from `1..n`

`5! = 1·2·3·4·5 = 120`

It can also be defined using recursion `n! = (n-1)!·n`

`5! = (5-1)!·5 = 120`

`(n-1)!` is a simpler form of the same problem. The base case is `0!` which is by definition `1`.

factorial()

```
1 uint32_t factorial(uint32_t n)
2 {
3   if(/*TODO*/) return(1);
4
5   return(factorial(n));
6 }
```

...

Download the code above and fix the broken `factorial` function

factorial()

```
1 uint32_t factorial(uint32_t n)
2 {
3   if(n==0) return(1);
4
5   return(n*factorial(n-1));
6 }
```

...

Observation:

The code on the previous slide neither specified the base case nor did it reduce the problem towards it

arr_max()

Many problems can be described in a recursive way

```
1 uint32_t arr_max(uint32_t *arr, size_t length)
2 {
3   …
4   uint32_t max_l= arr_max(&arr[0], length_l);
5   uint32_t max_r= arr_max(&arr[length_l], length_r);
6   return(max_l > max_r ? max_l : max_r);
7 }
```

...

The program above does not work. Download it, find out what `arr_max` should do, why it does not work and try to fix it.

arr_max()

What does `arr_max` do?

`arr_max` should find the largest number in an array

It splits the array in a right and a left half, calls itself on both halves and returns the largest of the two return values

```
 arr_max(1, 2, 3, 4)
=max_2(arr_max(1, 2), arr_max(3, 4))
=max_2(max_2(arr_max(1), arr_max(2)), max_2(arr_max(3), …))
…
```


arr_max()

Why doesn't it work?

```
1 uint32_t arr_max(uint32_t *arr, size_t length)
2 {
3   if(length == 1) return(arr[0]);
4   …
5 }
```

...

The original code was missing a check for the base case

The largest item in an array with only one item is the item itself

tree_max()

```
1 uint32_t tree_max(struct node_t *root)
2 {
3   uint32_t max_val= root->value;
4   …
5   return(max_val);
6 }
```

...

The `tree_max` function should return the largest value in a binary tree using recursion

It does not work. Find the errors and fix them

tree_max() - Error #1

```
1 if(root->child_l) {
2   uint32_t max_cl= tree_max(root->child_l);
3
4   if(max_cl > max_val) max_val= max_cl;
5 }
```

To make the function recursive `tree_max` has to be called on a subset of the previous problem (the child nodes)

tree_max() - Error #2

```
1 if(root->child_r) {
2   uint32_t max_cr= tree_max(root->child_r);
3
4   if(max_cr > max_val) max_val= max_cr;
5 }
```

...

To reach every node in the tree the left *and* right branch must be taken

tree_max() - 1337

```
1 uint32_t tree_max(struct node_t *root)
2 {
3   uint32_t max_val= root->value;
4   …
5   return(max_val);
6 }
```

...

The code above contains a more difficult to find error

Try to fix it

Appendix

The following slides are annotations that are intended to help you understand computers a bit better

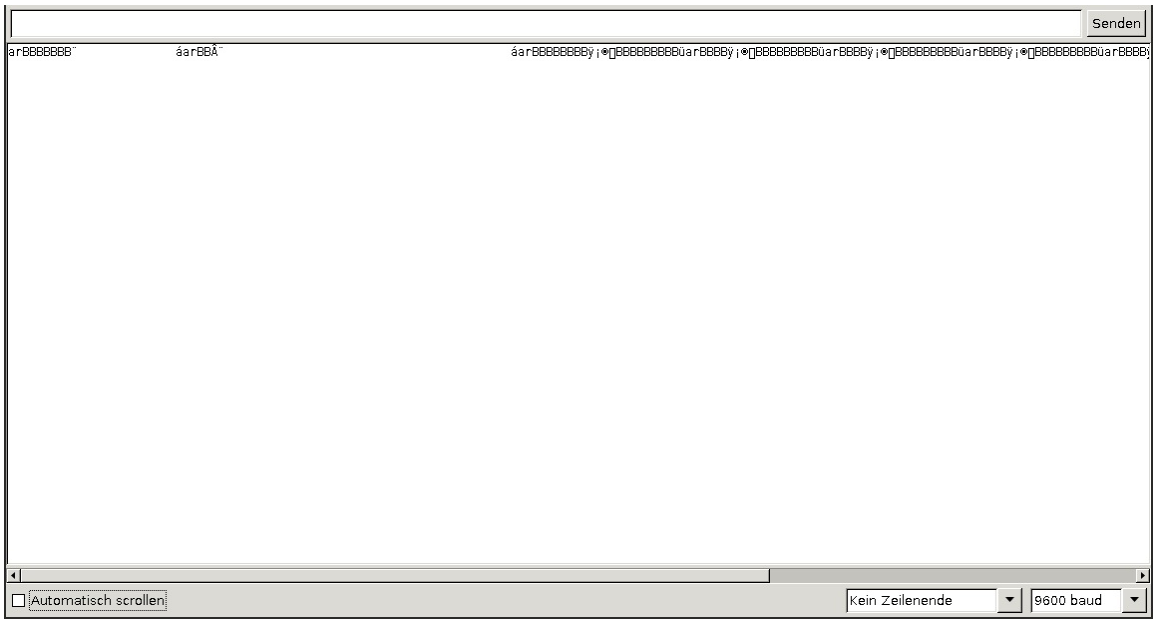
They are not part of the tutorial or the lecture. You are not required to read or understand them

Please direct any questions regarding them directly to the autor of the slides

They are also oversimplified to make them easier to grasp

Stack overflow

When the broken arr_max function runs on the Arduino the Serial monitor displays gibberish



Why is that?

Stack overflow

```
 0x08FF
-0x08bf main() Variables
…
 0x01ff
-0x0100 global Variables
…
 0x00ff
-0x0000 "Stuff"
```

Local variables like `arr` are stored on the stack

The stack is located at the end of RAM

Global variables, Arduino internal variables, io-Registers and the heap are located at the beginning of the RAM

Stack overflow

```
 0x08FF
-0x08bf main() Variables
 0x08be
-0x089e arr_max() Variables (called from main)
…
 0x01ff
-0x0100 global Variables
```

The stack grows for every function call to contain the local variables of the function and an address to jump to when the function is done


```
 0x08FF
-0x08bf main() Variables
 0x08be
-0x089e arr_max() Variables (called from main)
 0x089d
-0x087d arr_max() Variables (called from arr_max)
…
 0x01ff
-0x0100 global Variables
```

As there is no base case checking every instance of `arr_max` will call a new instance of `arr_max`


```
 0x08FF
-0x08bf main() Variables
 0x08be
-0x089e arr_max() Variables (called from main)
 0x089d
-0x087d arr_max() Variables (called from arr_max)
…
 0x0170
-0x0150 arr_max() Variables / global Variables
-0x0100 global Variables
```

After a lot of nested `arr_max` calls the stack will overflow into the global variables area. The behavior of the program is now undefined

Stack overflow

To prevent stack overflows you should only ever use recursive functions when you know that the calls will not be nested too deeply

Most modern operating systems will prevent the stack from running into the heap and will instead produce an error

Weird programming languages - Brainfuck

In the lecture you got a glimpse at a few "weird" programming languages

One of them is brainfuck. A language designed to be as minimalistic as possible.

A brainfuck program consists only of the following characters: `><+-.,[]`

All other characters are ignored

Brainfuck

Brainfuck operates on an infinite tape like the original turing machine

The characters the program consists of have the following meanings:

```
>/< - move to the next/previous tape position
+/- - increment/decrement the value at the tape position
./, - print/read a character into the tape position
[/] - loop if the value at the tape position is !=0
```


Brainfuck

```
+++++[>++<-]>[>+++++
++>>>>+++>+++++<<<<<
<-]>[>+>+>+<<<-]>+>-
Informatik ist super
->+++>++>[>+>+>+>+<<
<<-]>-->->+++++>[<<<
<+>>>>-]<<<<<<<<<[.>]
```

...

Download the code above, implement the missing operations, disable the debug mode and observe the output of the program