GOOD

Tips for the course

- Attend the labs.
- Practice as much as you can.
- Solve (or try to solve) the exercises multiple times!!
- Don't neglect the course until 2 weeks before the exam (really!).

– The contents of this course are (in parts) hard. It's perfectly normal to struggle. If you don't struggle, you're probably not trying hard enough (or you're a genius).

– The exam will probably be hard, too. Usually you will not need a high percentage of reachable points to pass (though it's relatively easy to reach a low score, unfortunately). To get points you really need to understand stuff. Memorizing things or superficial/cursory knowledge is not sufficient.

The main focus of this course are algorithms and data structures. For passing this course it is crucial that you focus on understanding really them thoroughly. For that, it is imperative that you work on the exercises as well as sample tasks from earlier exams. Just attending the lectures alone will most probably not be sufficient to pass the exam, neither will just looking at the solutions.

In exams, you will be required to be able to provide solutions as C or as pseudo code (but you're probably not able to choose). This year, a stronger emphasis will be put on writing and understanding pseudo code. You need to be able to read and write both C code and pseudo code and both representations might show up in the final exam.

Introduction to C

Every statement in C has to end with a semicolon (`;`);

Operators `&&`, `||` and `!` (instead of `and`, `or` and `not` as in Python)

Blocks are indicated with curly braces (`{`, `}`); additional whitespaces are ignored (but should be properly set either way).

There is no colon (`:`) after function signatures and `if`, `else`, `while`, `for` statements.

Comments are indicated with `/*`, `*/` (and `//`) for comments (instead of `#` as in Python).

Libraries are made available using `#include` (instead of `import` as in Python).

Functions have return types (e.g. `int`) but there is no `def` keyword.

The function `printf` is used for output to the console (and not `print` as in Python).

There is a function `scanf` instead of `input` / `raw_input` as in Python.

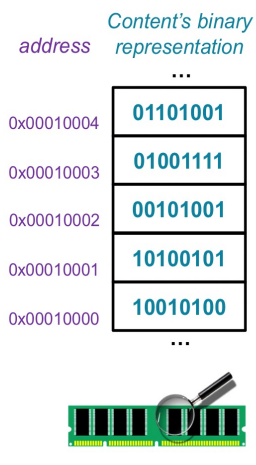
Memory in C

In a strongly simplified model we can consider the memory in a computer to be a sequence of boxes, each having a distinct address and each able to contain certain «things» (represented in the form of the symbols 0 and 1).

Today's computer hardware will almost always have a byte addressable memory architecture. This means that there is a separate address for every byte (8 bit) of memory. (For performance reasons, the hardware will not actually read and write single bytes, though.)

The memory address in a contemporary computer will typically be a 32 bit number or a 64 bit number. They are usually written as hexadecimal numbers (starting with `0x`).

In graphical representations like in the one on the right, memory addresses are mostly increasing from bottom to top.



C vs Python: Loops

	Python	C

Count up from 0 to 99 one by one:
0, 1, 2, …, 99
	`for i in range(0, 100):`
	`  #...`
		`for (int i = 0; i < 100; i++) {`
		`  /* ... */`
		`}`

Count down from 99 to 0 one by one:
99, 98, 97, …, 0
	`for i in range(99, -1 , -1):`
	`  #...`
		`for (int i = 99; i > -1; i--) {`
		`  /* ... */`
		`}`

Count up from 0 to 99 with a step width of 3:
0, 3, 6, …, 99
	`for i in range(0, 100, 3):`
	`  #...`
		`for (int i = 0; i < 100; i += 3) {`
		`  /* ... */`
		`}`

Iterate all elements of a data structure:	`list_of_words = ["hi", "my", "friend"]`
	`for word in list_of_words:`
	`  #...`
		(a bit more complicated; we'll see later how this is done in C; there is no «foreach» like expression)

– **pre-increment operator**: `x++`

– **post-increment operator**: `++x`

These operators have (apart from being shorter) the advantage that they can be applied in places where an assignment like `x = x + 1` could not be used, for example when printing a value: `printf("%d", x++)`. In this case, it would be necessary to perform the increment on a separate line.

These two operators are not working in exactly the same manner. As their names indicate, the

– pre-increment operator will increment the value of the variable first, before it is used within the larger code context, and the

– post-increment operator will use the current, unchanged value of the variable first within the code and only then increment it.

Thus, you can think of pre-incrementation and post-incrementation as two-step processes.

pre-increment
`++x`
increment – use

```
x = ++i;   ≙   i = i + 1;
               x = i;
```

```
int x = 42;
printf("%d, ", ++x);
printf("%d", x);
```
} will print 43, 43

post-increment
`x++`
use – increment

```
x = i++;   ≙   x = i;
               i = i + 1;
```

```
int x = 42;
printf("%d, ", x++);
printf("%d", x);
```
} will print 42, 43

Data Types in C

Keyword	Typical size*	Value range*	Meaning
`char`	1 byte	256 different values / characters	Single character; encoded according to ASCII table
`int`	4 bytes	$-2{,}147{,}483{,}648$ (-2^{31}) to $+2{,}147{,}483{,}647$ ($+2^{31}-1$)	Integer
`float`	4 bytes	ca. $\pm 1.2 \cdot 10^{-38}$ to $\pm 3.4 \cdot 10^{38}$ (about 6 decimal places of precision)	Decimal number
`double`	8 bytes	ca. $\pm 2.2 \cdot 10^{-308}$ to $\pm 1.8 \cdot 10^{308}$ (about 15 decimal places of precision)	Decimal number

Arrays: Comparison to Lists in Python

Arrays are on the first sight quite similar to lists in Python but there are some important differences:

– An array in C cannot grow or shrink, it has a fixed size which is set at compile time and cannot be changed during runtime. Once the array has been declared, its size is fixed and cannot be changed anymore.

– All elements an array have to be of the same type.

– Arrays provide no access control. Beware of out of bounds errors with arrays!

– There is no generally applicable method for retrieving the length of an array once it has been declared. (There is a workaround using the `sizeof` operator which cannot always be applied and / or there is a possibility to have a special value signaling the end of an array. The most famous example of the latter are strings as we will see later.)

– There is no equivalent to the negative indices used for string slicing in Python (where -1 is the last position, -2 the second last and so forth).

Functions

In C, functions have a return type (type which the return value needs to have). If a function should not return a value, there is the special type `void` (meaning nothing is returned).

In C, functions have to be declared before they can be used at another point in the code («forward declaration»). Note: Functions only need to be declared before used, but not actually defined (implemented); this declaration is called a function prototype.

Example of a function declaration:

```
int myFunction(int a, int b);
```

return type
type of the value which is returned by the function (may be void if no value is returned)

name
used to call the function elsewhere

parameters
input for the function: types and names of variables within function scope (can be empty)

The name, return type, together with the ordered list of parameter types is called the signature of the function. Note that C does *not* support function overloading, i.e. it is *not* possible that there are two functions with the same name but different signatures (a typical feature provided by object-oriented languages).

Operators in C: Precedence

The operator precedence in C is more or less identical to operator precedence in Python.

C Operator	Type	Associativity
`()`	parentheses (function call operator)	left to right
`[]`	array subscript	
`.`	member selection via object	
`->`	member selection via pointer	
`++`	unary postincrement	
`--`	unary postdecrement	
`++`	unary preincrement	right to left
`--`	unary predecrement	
`+`	unary plus	
`-`	unary minus	
`!`	unary logical negation	
`~`	unary bitwise complement	
`( type )`	C-style unary cast	
`*`	dereference	
`&`	address	
`sizeof`	determine size in bytes	
`*`	multiplication	left to right
`/`	division	
`%`	modulus	
`+`	addition	left to right
`-`	subtraction	

`==`	equals	left to right
`!=`	does not equal	
`===`	strict equals (no type conversions allowed)	
`!==`	strict does not equal (no type conversions allowed)	
`&`	bitwise AND	left to right
`^`	bitwise XOR	left to right
`|`	bitwise OR	left to right
`&&`	logical AND	left to right
`||`	logical OR	left to right
`?:`	conditional	right to left
`=`	assignment	right to left
`+=`	addition assignment	
`-=`	subtraction assignment	
`*=`	multiplication assignment	
`/=`	division assignment	
`%=`	modulus assignment	
`&=`	bitwise AND assignment	
`^=`	bitwise exclusive OR assignment	
`|=`	bitwise inclusive OR assignment	
`<<=`	bitwise left shift assignment	
`>>=`	bitwise right shift with sign extension assignment	
`>>>=`	bitwise right shift with zero extension assignment	

Strings

There is no dedicated string data type in C. Instead, strings are represented in C by arrays of characters.

These arrays are expected to be null terminated (if this is not the case, bad things will happen in general).

Example of string declaration and initialization:

```
char myFirstString[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

…or equivalently (and much more conveniently):

```
char mySecondString[] = "hello";
```

'h'	'e'	'l'	'l'	'o'	'\0'
0	1	2	3	4	5

Note that the sample char array `myString` from above has length 6, i.e. it has one element more than the word «hello» has characters because of the null terminator. Further remarks:

– Only double quotes are allowed when initializing a string like in the second example above.

– Note that strings are thus *not* immutable in C (in contrast to Python).

Arrays

First of all, there is a difference between the usage of an array in pseudocode and the usage of an array in C. When it comes to pseudocode, we use position to describe the elements in an array, and the **position starts at 1**. While in C programming, we use indices to access the elements in an array, and **the index starts at 0**. It is important to learn this difference, and you need to be careful about this difference in your practise and your exam.

We take an array of integer as examples. The following also works for arrays of character, double numebr etc.

An array of integers in Pseudocode

In tasks of exercises and exams, you might see the description like "an array of Integer `A[]` with n integers; an array of Integer `A[1..n]` ".

scenario 1: an array of integer `A[]` with n integers

The array `A[]` is a general way to describe an array, and this array contains n integers. However, we don't know what these n integer are.

Strings

In C, there is no string data type. In practice, a string is an array of characters (char).

Terminating Character

A string is an array of characters with a terminating character `'\0'`.

Useful Functions

Assume that we have a string `char str[]`. The following function `int len(char str[])` calculates the length of a string.

```
int len(char str[]) {
    int i;
    while(str[i] != '\0') {
        i++;
    }
    return i;
}
```

Example

Pointers

```
#include <stdio.h>

int main()
{
  int i = 1;
  int* p_i = &i; // pointer to int i
  //show address of i and the pointer to i
  printf("The address of i is %p \n", &i);
  printf("The address of p_i is %p \n", &p_i);

  // show vaue of i and the pointer to i
  printf("The value of i is %d \n", i);
  printf("The value of p_i is %p \n", p_i);

  //show sizes of i and the pointer to i in memory
  printf("The size of i is %zu bytes \n", sizeof(i));
  printf("The size of p_i is %zu bytes \n", sizeof(p_i));
}
```

Sample output.

```
The address of i is 0x7ffee52736ac (you should have a different one)
The address of p_i is 0x7ffee52736a0 (you should have a different one)
The value of i is 1
The value of p_i is 0x7ffee52736ac (you should have a different one)
The size of i is 4 bytes
The size of p_i is 8 bytes
```

Use the `%p` formatting specifier to print addresses in hexadecimal. You should see something like this: "0xbfe55918". The initial characters "0x" tell you that hexadecimal notation is being used; the remainder of the digits give the address itself. Use `%f` to print a floating value. Use the sizeof operator to determine the memory size allocated for each variable, then use `%zu` to print it.

C Reference Card (ANSI)

Program Structure/Functions

type fnc(*type*$_1$, ...);	function prototype
type name;	variable declaration
`int main(void) {`	main routine
declarations	local variable declarations
statements	
}	
type fnc(*arg*$_1$, ...) {	function definition
declarations	local variable declarations
statements	
`return` *value*;	
}	
`/* */`	comments
`int main(int argc, char *argv[])`	main with args
`exit(`*arg*`);`	terminate execution

C Preprocessor

include library file	`#include <`*filename*`>`
include user file	`#include "`*filename*`"`
replacement text	`#define` *name text*
replacement macro	`#define` *name*`(`*var*`)` *text*
Example. `#define max(A,B) ((A)>(B) ? (A) : (B))`	
undefine	`#undef` *name*
quoted string in replace	`#`
Example. `#define msg(A) printf("%s = %d", #A, (A))`	
concatenate args and rescan	`##`
conditional execution	`#if, #else, #elif, #endif`
is *name* defined, not defined?	`#ifdef, #ifndef`
name defined?	`defined(`*name*`)`
line continuation char	`\`

Data Types/Declarations

character (1 byte)	`char`
integer	`int`
real number (single, double precision)	`float, double`
short (16 bit integer)	`short`
long (32 bit integer)	`long`
double long (64 bit integer)	`long long`
positive or negative	`signed`
non-negative modulo 2^m	`unsigned`
pointer to `int`, `float`,...	`int*, float*`,...
enumeration constant	`enum` *tag* {*name*$_1$`=`*value*$_1$,...};
constant (read-only) value	*type* `const` *name*;
declare external variable	`extern`
internal to source file	`static`
local persistent between calls	`static`
no value	`void`
structure	`struct` *tag* {...};
create new name for data type	`typedef` *type name*;
size of an object (type is `size_t`)	`sizeof` *object*
size of a data type (type is `size_t`)	`sizeof(`*type*`)`

Initialization

initialize variable	*type name*`=`*value*;
initialize array	*type name*`[]={`*value*$_1$,...};
initialize char string	`char` *name*`[]="`*string*`";`

Constants

suffix: long, unsigned, float	`65536L, -1U, 3.0F`
exponential form	`4.2e1`
prefix: octal, hexadecimal	`0, 0x` or `0X`
Example. `031` is 25, `0x31` is 49 decimal	
character constant (char, octal, hex)	`'a', '\`*ooo*`', '\x`*hh*`'`
newline, cr, tab, backspace	`\n, \r, \t, \b`
special characters	`\\, \?, \', \"`
string constant (ends with `'\0'`)	`"abc...de"`

Pointers, Arrays & Structures

declare pointer to *type*	*type* `*`*name*;
declare function returning pointer to *type*	*type* `*f();`
declare pointer to function returning *type*	*type* `(*pf)();`
generic pointer type	`void *`
null pointer constant	`NULL`
object pointed to by *pointer*	`*`*pointer*
address of object *name*	`&`*name*
array	*name*`[`*dim*`]`
multi-dim array	*name*`[`*dim*$_1$`][`*dim*$_2$`]`...

Structures

`struct` *tag* {	structure template
declarations	declaration of members
};	

create structure	`struct` *tag name*
member of structure from template	*name*`.`*member*
member of pointed-to structure	*pointer* `->` *member*
Example. `(*p).x` and `p->x` are the same	
single object, multiple possible types	`union`
bit field with *b* bits	`unsigned` *member*`:` *b*;

Operators (grouped by precedence)

struct member operator	*name*`.`*member*
struct member through pointer	*pointer*`->`*member*
increment, decrement	`++, --`
plus, minus, logical not, bitwise not	`+, -, !, ~`
indirection via pointer, address of object	`*`*pointer*`, &`*name*
cast expression to type	`(`*type*`)` *expr*
size of an object	`sizeof`
multiply, divide, modulus (remainder)	`*, /, %`
add, subtract	`+, -`
left, right shift [bit ops]	`<<, >>`
relational comparisons	`>, >=, <, <=`
equality comparisons	`==, !=`
and [bit op]	`&`
exclusive or [bit op]	`^`
or (inclusive) [bit op]	`|`
logical and	`&&`
logical or	`||`
conditional expression	*expr*$_1$ `?` *expr*$_2$ `:` *expr*$_3$
assignment operators	`+=, -=, *=, ...`
expression evaluation separator	`,`

Unary operators, conditional expression and assignment operators group right to left; all others group left to right.

Flow of Control

statement terminator	`;`
block delimiters	`{ }`
exit from `switch`, `while`, `do`, `for`	`break;`
next iteration of `while`, `do`, `for`	`continue;`
go to	`goto` *label*;
label	*label*`:` `statement`
return value from function	`return` *expr*

Flow Constructions

`if` statement	`if (`*expr*$_1$`)` *statement*$_1$ `else if (`*expr*$_2$`)` *statement*$_2$ `else` *statement*$_3$
`while` statement	`while (`*expr*`)` *statement*
`for` statement	`for (`*expr*$_1$`;` *expr*$_2$`;` *expr*$_3$`)` *statement*
`do` statement	`do` *statement* `while(`*expr*`);`

`switch` statement	`switch (`*expr*`) {`
	`case` *const*$_1$`:` *statement*$_1$ `break;`
	`case` *const*$_2$`:` *statement*$_2$ `break;`
	`default:` *statement*
	}

ANSI Standard Libraries

`<assert.h>`	`<ctype.h>`	`<errno.h>`	`<float.h>`	`<limits.h>`
`<locale.h>`	`<math.h>`	`<setjmp.h>`	`<signal.h>`	`<stdarg.h>`
`<stddef.h>`	`<stdio.h>`	`<stdlib.h>`	`<string.h>`	`<time.h>`

Character Class Tests `<ctype.h>`

alphanumeric?	`isalnum(c)`
alphabetic?	`isalpha(c)`
control character?	`iscntrl(c)`
decimal digit?	`isdigit(c)`
printing character (not incl space)?	`isgraph(c)`
lower case letter?	`islower(c)`
printing character (incl space)?	`isprint(c)`
printing char except space, letter, digit?	`ispunct(c)`
space, formfeed, newline, cr, tab, vtab?	`isspace(c)`
upper case letter?	`isupper(c)`
hexadecimal digit?	`isxdigit(c)`
convert to lower case	`tolower(c)`
convert to upper case	`toupper(c)`

String Operations `<string.h>`

`s` is a string; `cs`, `ct` are constant strings

length of `s`	`strlen(s)`
copy `ct` to `s`	`strcpy(s,ct)`
concatenate `ct` after `s`	`strcat(s,ct)`
compare `cs` to `ct`	`strcmp(cs,ct)`
only first `n` chars	`strncmp(cs,ct,n)`
pointer to first `c` in `cs`	`strchr(cs,c)`
pointer to last `c` in `cs`	`strrchr(cs,c)`
copy `n` chars from `ct` to `s`	`memcpy(s,ct,n)`
copy `n` chars from `ct` to `s` (may overlap)	`memmove(s,ct,n)`
compare `n` chars of `cs` with `ct`	`memcmp(cs,ct,n)`
pointer to first `c` in first `n` chars of `cs`	`memchr(cs,c,n)`
put `c` into first `n` chars of `s`	`memset(s,c,n)`

C Reference Card (ANSI)

Input/Output <stdio.h>

Standard I/O

standard input stream	stdin
standard output stream	stdout
standard error stream	stderr
end of file (type is int)	EOF
get a character	getchar()
print a character	putchar(*chr*)
print formatted data	printf("*format*",arg_1,...)
print to string s	sprintf(s,"*format*",arg_1,...)
read formatted data	scanf("*format*",&$name_1$,...)
read from string s	sscanf(s,"*format*",&$name_1$,...)
print string s	puts(s)

File I/O

declare file pointer	FILE *fp;
pointer to named file	fopen("*name*","*mode*")

modes: r (read), w (write), a (append), b (binary)

get a character	getc(*fp*)
write a character	putc(*chr*,*fp*)
write to file	fprintf(*fp*,"*format*",arg_1,...)
read from file	fscanf(*fp*,"*format*",arg_1,...)
read and store n elts to *ptr	fread(*ptr,eltsize,n,*fp*)
write n elts from *ptr to file	fwrite(*ptr,eltsize,n,*fp*)
close file	fclose(*fp*)
non-zero if error	ferror(*fp*)
non-zero if already reached EOF	feof(*fp*)
read line to string s (< max chars)	fgets(s,max,*fp*)
write string s	fputs(s,*fp*)

Codes for Formatted I/O: "%-+ 0*w.pmc*"

-	left justify
+	print with sign
space	print space if no sign
0	pad with leading zeros
w	min field width
p	precision
m	conversion character: h short, l long, L long double
c	conversion character:

d,i	integer	u	unsigned
c	single char	s	char string
f	double (printf)	e,E	exponential
f	float (scanf)	lf	double (scanf)
o	octal	x,X	hexadecimal
p	pointer	n	number of chars written
g,G	same as f or e,E depending on exponent		

Variable Argument Lists <stdarg.h>

declaration of pointer to arguments	va_list *ap*;
initialization of argument pointer	va_start(*ap*,*lastarg*);

lastarg is last named parameter of the function

access next unnamed arg, update pointer	va_arg(*ap*,*type*)
call before exiting function	va_end(*ap*);

Standard Utility Functions <stdlib.h>

absolute value of int n	abs(n)
absolute value of long n	labs(n)
quotient and remainder of ints n,d	div(n,d)

returns structure with div_t.quot and div_t.rem

quotient and remainder of longs n,d	ldiv(n,d)

returns structure with ldiv_t.quot and ldiv_t.rem

pseudo-random integer [0,RAND_MAX]	rand()
set random seed to n	srand(n)
terminate program execution	exit(status)
pass string s to system for execution	system(s)

Conversions

convert string s to double	atof(s)
convert string s to integer	atoi(s)
convert string s to long	atol(s)
convert prefix of s to double	strtod(s,&endp)
convert prefix of s (base b) to long	strtol(s,&endp,b)
same, but unsigned long	strtoul(s,&endp,b)

Storage Allocation

allocate storage	malloc(size), calloc(nobj,size)
change size of storage	newptr = realloc(ptr,size);
deallocate storage	free(ptr);

Array Functions

search array for key	bsearch(key,array,n,size,cmpf)
sort array ascending order	qsort(array,n,size,cmpf)

Time and Date Functions <time.h>

processor time used by program	clock()

Example. clock()/CLOCKS_PER_SEC is time in seconds

current calendar time	time()
$time_2$-$time_1$ in seconds (double)	difftime($time_2$,$time_1$)
arithmetic types representing times	clock_t,time_t
structure type for calendar time comps	struct tm

tm_sec	seconds after minute
tm_min	minutes after hour
tm_hour	hours since midnight
tm_mday	day of month
tm_mon	months since January
tm_year	years since 1900
tm_wday	days since Sunday
tm_yday	days since January 1
tm_isdst	Daylight Savings Time flag

convert local time to calendar time	mktime(tp)
convert time in tp to string	asctime(tp)
convert calendar time in tp to local time	ctime(tp)
convert calendar time to GMT	gmtime(tp)
convert calendar time to local time	localtime(tp)
format date and time info	strftime(s,smax,"*format*",tp)

tp is a pointer to a structure of type tm

Mathematical Functions <math.h>

Arguments and returned values are double

trig functions	sin(x), cos(x), tan(x)
inverse trig functions	asin(x), acos(x), atan(x)
arctan(y/x)	atan2(y,x)
hyperbolic trig functions	sinh(x), cosh(x), tanh(x)
exponentials & logs	exp(x), log(x), log10(x)
exponentials & logs (2 power)	ldexp(x,n), frexp(x,&e)
division & remainder	modf(x,ip), fmod(x,y)
powers	pow(x,y), sqrt(x)
rounding	ceil(x), floor(x), fabs(x)

Integer Type Limits <limits.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system, followed by minimum required values (if significantly different).

CHAR_BIT	bits in char	(8)
CHAR_MAX	max value of char	(SCHAR_MAX or UCHAR_MAX)
CHAR_MIN	min value of char	(SCHAR_MIN or 0)
SCHAR_MAX	max signed char	(+127)
SCHAR_MIN	min signed char	(−128)
SHRT_MAX	max value of short	(+32,767)
SHRT_MIN	min value of short	(−32,768)
INT_MAX	max value of int	(+2,147,483,647) (+32,767)
INT_MIN	min value of int	(−2,147,483,648) (−32,767)
LONG_MAX	max value of long	(+2,147,483,647)
LONG_MIN	min value of long	(−2,147,483,648)
UCHAR_MAX	max unsigned char	(255)
USHRT_MAX	max unsigned short	(65,535)
UINT_MAX	max unsigned int	(4,294,967,295) (65,535)
ULONG_MAX	max unsigned long	(4,294,967,295)

Float Type Limits <float.h>

The numbers given in parentheses are typical values for the constants on a 32-bit Unix system.

FLT_RADIX	radix of exponent rep	(2)
FLT_ROUNDS	floating point rounding mode	
FLT_DIG	decimal digits of precision	(6)
FLT_EPSILON	smallest x so $1.0f + x \neq 1.0f$	(1.1E − 7)
FLT_MANT_DIG	number of digits in mantissa	
FLT_MAX	maximum float number	(3.4E38)
FLT_MAX_EXP	maximum exponent	
FLT_MIN	minimum float number	(1.2E − 38)
FLT_MIN_EXP	minimum exponent	
DBL_DIG	decimal digits of precision	(15)
DBL_EPSILON	smallest x so $1.0 + x \neq 1.0$	(2.2E − 16)
DBL_MANT_DIG	number of digits in mantissa	
DBL_MAX	max double number	(1.8E308)
DBL_MAX_EXP	maximum exponent	
DBL_MIN	min double number	(2.2E − 308)
DBL_MIN_EXP	minimum exponent	

Printf: Format String

Non-exhaustive assortment of some commonly used format strings for the `printf` function:

Specifier	Description	Suitable Types
%c	single ASCII character	`char`
%d	decimal number	`int`
%e	scientific notation	`float`, `double`
%f	floating point number	`float`
%lf	floating point number	`double`
%s	strings	`char` array

Tips and Tricks When Devising Algorithms

- In the beginning, think about a solution on a high level and write it down as a (pseudocode) blueprint which you can later use when you're actually implementing it in C.
- Make drawings!
- Use pens or your fingers to think about iterative algorithms operating over arrays.
- Make a table to analyze loops, keeping track of iteration variables and other relevant values.
- Don't forget to think about potential special cases. Test your code thorougly.
- Execute your code regularly, do not write lots of code and only then try out. Make baby steps in the beginning!

C and Sorting Algorithms

https://visualgo.net/en/sorting

a) «The bubble sort algorithm can be implemented using two nested while loops.» ✓ true

→ `for` loops and `while` loops are interchangeable

b) «The insertion sort algorithm can be implemented using two nested for loops.» ✓ true

→ `for` loops and `while` loops are interchangeable

c) «Given the same input, all three sorting algorithms always need the same number of comparisons.» ✗ false

→ all algorithms will need a different number of comparisons in general (cf. lecture slides)

d) «All three sorting algorithms only compare two adjacent elements in an array.» ✗ false

→ counter example: Both selection sort and insertion sort in general will compare elements at positions which are not next to eachother

For loops can always be replaced by while loops

A `while` loop usually feels more natural to use when there is no way to know how many iterations will be done and the number of iterations might be different depending on the input, for example when searching in an array of integers for the first occurrence of two identical values next to each other.
A `for` loop usually feels more naturally do use when the number of iterations is fixed and clear from the beginning, for example when the sum of the elements of an array of integers is to be calculated. (Many more modern programming languages have a special kind of for loop `for` this purpose: `foreach`).

Exercise 1, Task 1: Conversion of `for` Loops Into `while` Loops

A `for` loop can always be converted into a `while` loop and vice-versa.

Disregarding special cases (e.g. break, continue, return within the loop), the conversion from a `for` loop into a `while` loop can be done as follows:

- put the initialization before the header of the `while` loop,
- put the condition in the header of the `while` loop, and
- use the increment as the last statement within the body of the `while` loop.

```
int i;
for (i = 0; i < n; i = i + 1) {
    /* loop body */
}
```

↕

```
int i = 0;
while (i < n) {
    /* loop body */
    i = i + 1;
}
```

Exercise 1, Task 1: Bubble Sort: For – For vs. While – While

Bubble sort with two nested `for` loops:

```
1  void bubble_sort(int A[], int n) {
2     int i;
3     for (i = n - 1; i >= 1; i--) {
4        int j;
5        for (j = 1; j <= i; j++) {
6           if (A[j] < A[j - 1]) {
7              int temp = A[j];
8              A[j] = A[j - 1];
9              A[j - 1] = temp;
10          }
11       }
12    }
13 }
```

Bubble sort with two nested `while` loops:

```
1  void bubble_sort(int A[], int n) {
2     int i = n - 1;
3     while (i >= 1) {
4        int j = 1;
5        while (j >= 1) {
6           if (A[j] < A[j - 1]) {
7              int temp = A[j];
8              A[j] = A[j - 1];
9              A[j - 1] = temp;
10          }
11          j++;
12       }
13       i--;
14    }
15 }
```

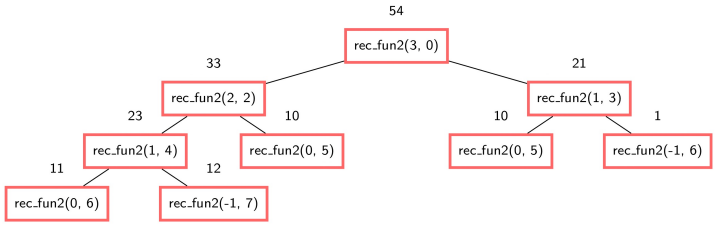
↳ Invariant of Bubble Sort: at the end of i iteration the right most i elements are sorted

Recursion

https://pythontutor.com/visualize.html#mode=edit

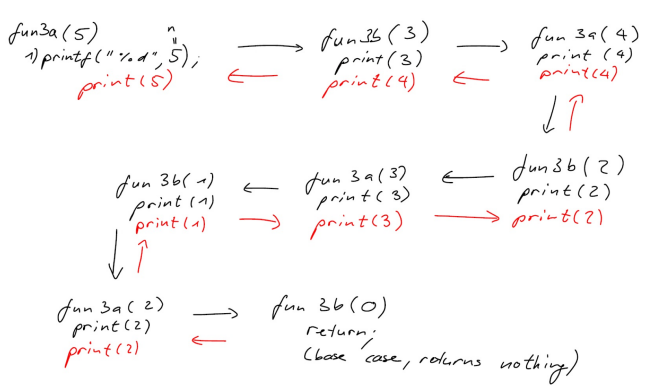
What will be the return value of the call `rec_fun2(3, 0)`?

```
int rec_fun2(int x, int y) {
  if (x <= 0) {
    y = y + 5;
    return y;
  }
  else {
    int t1 = rec_fun2(x - 1, y + 2);
    int t2 = rec_fun2(x - 2, y + 3);
    return t1 + t2;
  }
}
```



(c) Consider the following two C functions:
What will be the output on the console for the call `rec_fun3a(5)`?

```
void rec_fun3a(int n) {
    if (n == 0) {
        return;
    }
    printf("%d", n);
    rec_fun3b(n - 2);
    printf("%d", n);
}

void rec_fun3b(int n) {
    if (n == 0) {
        return;
    }
    printf("%d", n);
    rec_fun3a(n + 1);
    printf("%d", n);
}
```



A recursive function can have multiple base cases!

Implement a function in C which recursively calculates the sum of the elements of an integer array.

```
int sumrec(int arr[], int n, int i) {
    if (i == n - 1) {
        return arr[i];
    } else {
        return arr[i] + sumrec(arr, n, i + 1);
    }
}
```

Exercise Recursion: Average of an Integer Array

```
1 double average(double arr[], int n) {
2     if (n == 1) {
3         return arr[1];
4     } else {
5         return (average(arr, n-1) * (n-1) + arr[n-1]) / n;
6     }
7 }
```

base case

recursive call

Recursion: «Recursive Leap of Faith»

How can you transport 1000 elephants to the moon?

→ If all elephants already are on the moon, we're done. Else, ship one elephant to the moon first and then use the some approach on the remaining 999.

This kind of logic applied in recursive solutions oftentimes can seem a bit «magical». One just has a solution for a very small (sometimes even trivial) problem and then just calls the same approach on a reduced copy of the complete problem which then will «somehow» solve the problem.

This part of constructing a recursive solution is therefore sometimes called the «Recursive Leap of Faith».

Running time and Asymptotic Complexity

https://h5p.org/node/455075

Complexity of Search Algorithms

Linear Search -> $O(n)$

Binary Search -> $O(\log(n))$

Sorting

Insertion Sort, Bubble Sort -> $O(n^2)$

Quick Sort -> $O(n^2)$ but typically towards $O(n * \log(n))$

Merge Sort -> $O(n * \log(n))$

Theoretical Limit for Comparison Based Sorting is $O(n * \log(n))$. More cool things about sorting at sorting

---

O Big-Oh: upperbound

$f(n) = O(g(n))$

$f(n) \leq c * g(n)\ \forall n > n_0$

example $f(n) = 2n+3$ nearest 3 useful!

$2n+3 \leq 10 \cdot g(n)$ (c, $g(n)$) → $f(n) = O(n)$ → $f(n)$ is an upper bound of $g(n)$

↳ $f(n) = O(n)$

~~$f(n) = O(n^2)$~~

~~$f(n) = O(\log n)$ → would be a lower bound~~

$f(n)$ lower bound, average bound → upper bound

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

Ω Big omega (lower bound)

$f(n) \geq c * g(n)\ \forall n \geq n_0$

e.g. $f(n) = 2n+3$

$2n+3 \geq 1 \cdot n\ \forall n \geq 1$ (c, $g(n)$)

$f(n) = \Omega(n)$ nearest bound is useful!

$f(n) = \Omega(\log n)$

~~$f(n) = \Omega(n^2)$~~ → would be upper bound

Θ Big Theta: average case / tight bound

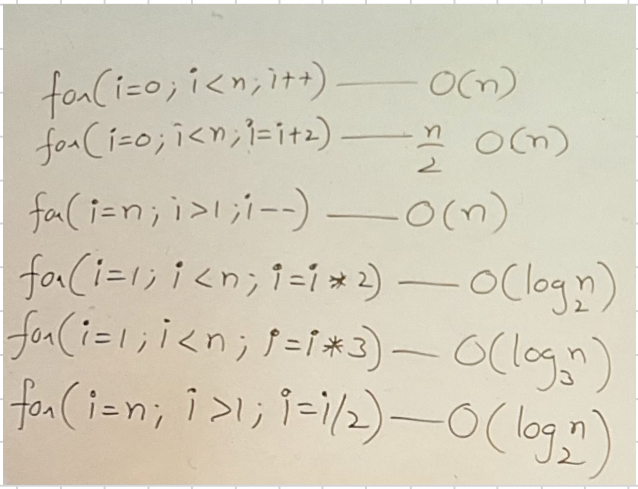
$f(n) = \Theta(g(n))$

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

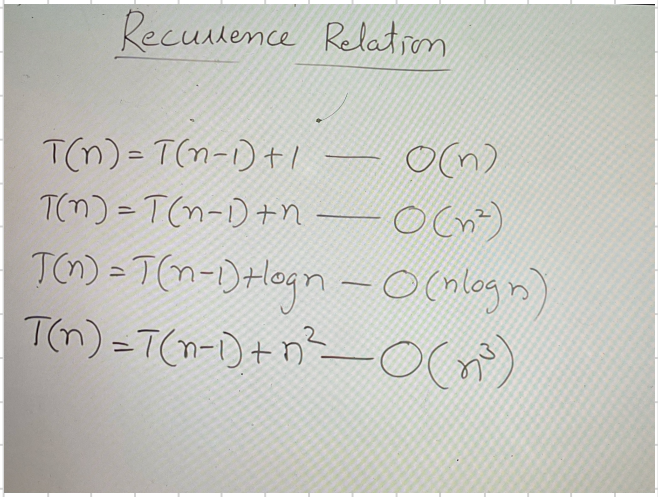
e.g. $f(n) = 2n+3$

$1 \cdot n \leq 2n+3 \leq 5 \cdot n$ (c_1, c_2, $g(n)$)

$f(n) = \Theta(n)$



$T(n) = 1 + \log n! = O(n \log n)$



Hierarchy of Asymptotic Complexities of Some Basic Functions

all depending on n as variable

growing slower ↑

Class	Name		Remarks	Suitability
$\mathbf{1}$	constant		$\sin(n), \cos(n) \in O(1)$	usually suitable for big problems
$\mathbf{\log_b(\log_b(n))}$			value of b does not matter	
$\mathbf{\log_b(n)}$	logarithmic		value of b does not matter	
$\mathbf{(\log_b(n))^k}$ where $k > 1$	polylogarithmisch		value of b does not matter	
$\mathbf{n^k}$ where $0 < k < ½$		polynomial	$= \sqrt[a]{n^b}$ where $k = b/a$	
$\mathbf{\sqrt[2]{n} = n^{0.5}}$			$= n^{½}$	
$\mathbf{n^k}$ where $½ < k < 1$			$= \sqrt[a]{n^b}$ where $k = b/a$	
$\mathbf{n}$	linear			
$\mathbf{n \cdot \log(n)}$ where $n > 1$	quasi-linear / log-linear		$\log(n!) \in \Theta(n \cdot \log(n))$	
$\mathbf{n^k}$ where $1 < k < 1.5$		polynomial		usually unsuitable for big problems
$\mathbf{n \cdot \sqrt[2]{n}}$			$= n^{1.5}$	
$\mathbf{n^k}$ where $1.5 < k < 2$				
$\mathbf{n^2}$	quadratic			
$\mathbf{n^k}$ where $2 < k < 3$				
$\mathbf{n^3}$	cubic			
$\mathbf{n^k}$ where $k > 3$				
$\mathbf{n^{\log_b(n)}}$	quasi-polynomial	superpolynomial	Value of b does not matter	
$\mathbf{k^n}$ where $k > 1$	exponential			
$\mathbf{n!}$	factorial		$n! \underset{n \to \infty}{\approx} \sqrt[2]{2 \cdot \pi \cdot n} \cdot (n/e)^n$	
$\mathbf{n^n}$			corresponds to $(n/k)^n$ where $k = 1$	

growing faster ↓

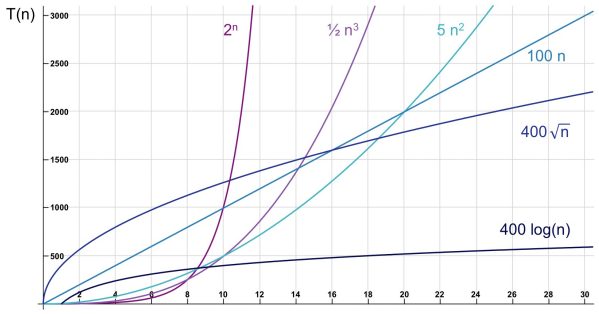
$O(2n!) > O(n^n)$

Algorithmic Complexity, Landau Notation

- Algorithmic «complexity» is actually not a very good name choice since it has nothing to do with how complicated an algorithm is. A very simple algorithm may have a high asymptotic complexity and oftentimes having a low running time complexity needs a very complicated algorithm. A better term may have been «algorithmic efficiency» / «performance» or something like this.
  (There are other notions of complexity, e.g. cyclomatic complexity / McCabe complexity – not part of this lecture.)

- It's a way to compare algorithms with respect to certain criteria (time, memory) while abstracting from details of the technical environment (CPU speed etc.).

Asymptotic Complexity Illustrated



Big O Notation: Types of Complexity Measures

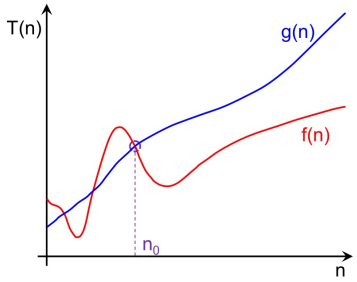
Symbol	*Symbol name*	*Meaning*	*Analogy*	*Definition*	*Example*
O	Big Omicron	Upper bound	$f \in O(g)$: " f **≤** g "	$\exists\, n_0 > 0,\ c > 0: \forall n \geq n_0,\ f(n) \leq c \cdot g(n)$	$4n + 2 \in O(n^2 + n - 7)$
Ω	Big Omega	Lower bound	$f \in \Omega(g)$: " f **≥** g "	$\exists\, n_0 > 0,\ c > 0: \forall n \geq n_0,\ f(n) \geq c \cdot g(n)$	$n^4 - 8 \in \Omega(5n^2 - 2n + 1)$
Θ	Theta	Tight bound	$f \in \Theta(g)$: " f **=** g "	$\exists\, n_0 > 0,\ c_1 > 0,\ c_2 > 0: \forall n \geq n_0,\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	$9n^2 - 3n \in \Theta(n^2 + 31)$
o	Little Omicron	Strict upper bound	$f \in o(g)$: " f **<** g "	$\exists\, n_0 > 0,\ \forall c > 0: \forall n \geq n_0,\ f(n) < c \cdot g(n)$	$6n^2 \in o(2n^3)$
ω	Little Omega	Strict lower bound	$f \in \omega(g)$: " f **>** g "	$\exists\, n_0 > 0,\ \forall c > 0: \forall n \geq n_0,\ f(n) > c \cdot g(n)$	$7n^3 \in \omega(4n)$

$f(n) = 42n^2 + 40n + 2 \in O(n^2)$	✓ true
$f(n) = 7^n \in O(5^n)$	✗ false
$f(n) = 5^n \in \Theta(7^n)$	✗ false
$f(n) = 5^n \in O(7^n)$	✓ true
$f(n) = n^{10} \in \Omega(10^n)$	✗ false
$f(n) = n^n \in \Omega(1)$	✓ true
$O(\log(n)) \subsetneq O(\sqrt{n})$	✓ true
$O(n) \subsetneq \Omega(n^2)$	✗ false

Asymptotic Complexity Definition: Potential Misconceptions

Remember the definition of the asymptotic upper bound:

$f(n) \in O(g(n))$ iff $\exists$ constants $n_0 > 0$, $c > 0$ such that $f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$



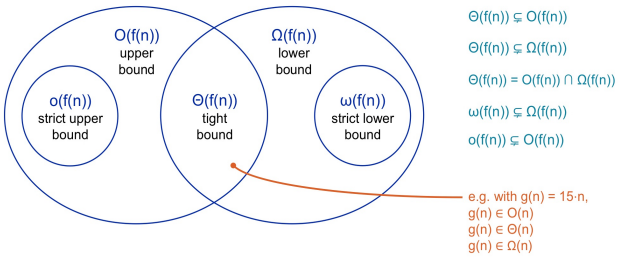
Which of the following statements are true, which are false?

a) «If $f(n) \in O(g(n))$ holds, then all values of g(n) have to be bigger than the values of f(n) beyond n_0.» *false* can also be equal

b) «If $f(n) \in O(g(n))$ holds, there cannot be any intersection of the two functions beyond n_0.» *false*

c) «The value of the constant c from the definition has to be chosen such that n_0 is at an intersection point of the two functions in question.» *false*

Complexity Measure Types from a Set Perspective



$\Theta(f(n)) \subsetneq O(f(n))$

$\Theta(f(n)) \subsetneq \Omega(f(n))$

$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

$\omega(f(n)) \subsetneq \Omega(f(n))$

$o(f(n)) \subsetneq O(f(n))$

e.g. with $g(n) = 15 \cdot n$,
$g(n) \in O(n)$
$g(n) \in \Theta(n)$
$g(n) \in \Omega(n)$

Algebraic Transformations: Exponentiation Rules

(1) $a^{-k} = 1 / a^k$

(2) $(a \cdot b)^k = a^k \cdot b^k$ and $(a \div b)^k = a^k \div b^k$

(3) $a^m \cdot a^n = a^{(m+n)}$ and $a^m \div a^n = a^{(m-n)} = a^m \cdot a^{-n}$

(4) $(a^m)^n = a^{(m \cdot n)}$

(5) $a^{1/k} = \sqrt[k]{a}$, e.g. $a^{3/4} = \sqrt[4]{a^3}$

(6) $a^0 := 1$ *regardless of the value of a*

- Beware: in general: $a^{(m^n)} \neq (a^m)^n = a^{(m \cdot n)}$
  a^{m^n} is by default to be read as $a^{(m^n)}$
- Beware: in general: $a^m + b^m \neq (a + b)^m$ $(a^2+b^2 \neq (a+b)^2)$
- Beware: in general: $(-a)^m \neq -a^m = -(a^m)$ $((-1)^2 \neq -(1^2))$

Algebraic Transformations: Logarithm Rules

(1) $\log_b(m \cdot n) = \log_b(m) + \log_b(n)$

(2) $\log_b(m \div n) = \log_b(m) - \log_b(n)$

(3) $\log_b(m^n) = n \cdot \log_b(m)$ *regardless of the value of b*

(4) $a = b^{\log_b(a)}$

(5) $\log_a(x) = \log_b(x) \div \log_b(a)$ used for base transformation

(6) $a^{\log_b(x)} = x^{\log_b(a)}$ $n^{\log c} = c^{\log n}$ *regardless of the value of b*

(7) $\log_b(1) := 0$ *regardless of the value of b*

Justification for (6): $a^{\log_b(x)} \overset{(4)}{=} (b^{\log_b(a)})^{\log_b(x)} = b^{\log_b(a) \cdot \log_b(x)} = b^{\log_b(x) \cdot \log_b(a)} = x^{\log_b(a)}$

Rules for and Some Notes on Asymptotic Complexity

Any addition of constants can be ignored:

$\Theta(f(n) + c) = \Theta(f(n))$

iff c = const

Example: n^2 is in $O(n^2)$ and $n^2 + 100000000000$ is in $O(n^2)$

Any multiplication with constants can be ignored:

$\Theta(c \cdot f(n)) = \Theta(f(n))$

iff c = const, c > 0

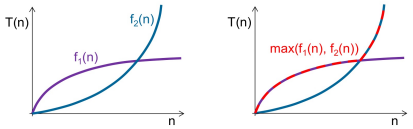
Example: 3n is in O(n) and 99999999n is in O(n)

Lower order terms get irrelevant asymptotically. In an sum of functions, the fastest growing function wins:

$O(f(n) \pm g(n)) = O(\max(f(n), g(n))$

The asymptotic complexity of the sum (or difference) of two functions is equal to the asymptotic complexity of the function summand which grows faster:

$\Theta(f(n) \pm g(n)) = \Theta(\max(f(n), g(n))$ (exception: $\Theta(f(n) - g(n))$ in case $\Theta(f(n) = \Theta(g(n))$)



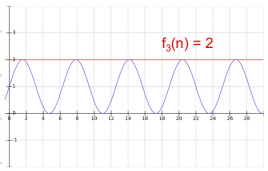
Another way to state this idea: $\Theta(f(n) \pm g(n)) = \Theta(g(n))$ iff $\Theta(g(n)) \text{ "}\geq\text{" } \Theta(f(n))$

The asymptotic complexity of the product of two functions is:

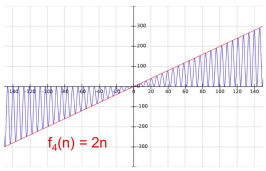
$$(f(n) \cdot g(n)) \in O(f(n) \cdot g(n))$$

Example revisited: What is the asymptotic complexity of the following functions?

$f_1(n) = 1 + \sin(n)$ → O(1)



$f_2(n) = n \cdot (1 + \sin(n))$ → O(n)



- The base of logarithms doesn't matter for asymptotic complexity (as long as it is constant and isn't 1):
  $\Theta(\log_a(n)) = \Theta(\log_b(n))$
  $= \Theta(\lg(n)) = \Theta(\ln(n)) = \Theta(\mathrm{ld}(n))$

  Beware: The base of a logarithm is only irrelevant in this kind of situation. It is very relevant for example when $f(n) = n^{\log_a(c)}$ is compared to $g(n) = n^{\log_b(c)}$.

- Potentially misleading notation:
  $\log^2(n) := \log(n) \cdot \log(n) = (\log(n))^2$
  and *not*, as one may probably expect, $\log(\log(n))$, i.e. $\log^2(n) \neq \log(\log(n))$

- Also note that $\log(n) \cdot \log(n)$ grows *faster* than than log(n) and $\log(\log(n))$ grows *slower* than log(n).

Find the asymptotic *tight* bounds (Θ notation) of the following functions and rank them by their order of growth (slowest first).

- $f_1(n) = n^{10} \cdot \sqrt[4]{n}$ $\in \Theta(n^{10.25})$
- $f_2(n) = \log(n^\pi \cdot \sqrt[2]{n})$ $\in \Theta(\log(n))$
- $f_3(n) = n^{15} \cdot \sqrt[6]{n} \cdot \log(n)$ $\in \Theta(n^{15\frac{1}{6}} \cdot \log(n))$
- $f_4(n) = \log_{\ln(5)}((\log(n))^{\lg(100)})$ $\in \Theta(\log(\log(n)))$
- $f_5(n) = (\sqrt{2})^{\log_2(n)} + \log(n)$ $\in \Theta(n^{0.5})$
- $f_6(n) = \log(n!) + n \cdot \log^5(n)$ $\in \Theta(n \cdot \log^5(n))$
- $f_7(n) = (\frac{2}{3})^n$ $\in \Theta((\frac{2}{3})^n)$

Ranking: (slowest growing) f_7, f_4, f_2, f_5, f_6, f_1, f_3 (fastest growing)

[2021 Final Exam] Assume $f_1(n) = O(1)$, $f_2(n) = O(N^2)$, and $f_3(n) = O(N \log N)$. From these complexities it follows that $f_1(n) + f_2(n) + f_3(n) = O(N \log N)$.

Answer: ☐ True ☒ False → $O(n^2)$

Rules for and Some Notes on Asymptotic Complexity: Polynomials

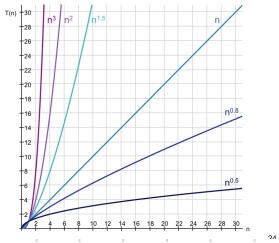
Polynomial family of functions, i.e. functions of type $f(n) = n^k$, $0 < k < \infty$, are all distinct from each other and have growing order of growth with growing exponent («polynomial family of functions»).

$\Theta(n^x) \neq \Theta(n^y)$ iff $x \neq y$

and

$n^x \in O(n^y)$ iff $y \geq x$

This also means, that the exponent of the root *does* matter:
$\Theta(\sqrt[2]{n}) \neq \Theta(\sqrt[3]{n})$



Any polynomial function, i.e. n^k, $0 < k < \infty$, regardless of exponent, grow faster than any logarithmic function.

Example: $\sqrt{n}$ grows faster than log(n) asymptotically

The basis of exponentiation *does* matter:

$\Theta(a^n) \neq \Theta(b^n)$ iff $a \neq b$

e.g. $3^n \notin \Theta(2^n)$ and $3^n \notin O(2^n)$

The logarithm of a factorial, i.e. log(n!), has the same asymptotic complexity as $n \cdot \log(n)$:

$$\log(n!) \in \Theta(n \cdot \log(n))$$

Constant additions to a variable from which the factorials is taken *do matter*:

$$\Theta((n + k)!) \notin \Theta(n!)$$

Justification: $(n + 1)! = n! \cdot (n + 1) = n \cdot n! + n! \in O(n \cdot n!) \neq O(n!)$

(in practice, though, it might be reasonable to just change the way how the input value n is defined…)

$f_1(n) = (2n + 3)!$	$\in \Theta((2n + 3)!)$
$f_2(n) = 2 \cdot \log(6^{\log(n^2)}) + \log(\pi \cdot n^2) + n^3$	$\in \Theta(n^3)$
$f_3(n) = 4^{\log_2(n)}$	$\in \Theta(n^2)$
$f_4(n) = 12\sqrt[2]{n} + 10^{223} + \log(5^n)$	$\in \Theta(n)$
$f_5(n) = 10^{\lg(20)}n^4 + 8^{229}n^3 + 20^{231}n^2 + 128n \cdot \log(n)$	$\in \Theta(n^4)$
$f_6(n) = \log(n^{2n+1})$	$\in \Theta(n \cdot \log(n))$
$f_7(n) = 101^{\sqrt{n}}$	$\in \Theta(101^{\sqrt{n}})$
$f_8(n) = \log^2(n) + 50\sqrt[2]{n} + \log(n)$	$\in \Theta(n^{0.5})$
$f_9(n) = n^n + 2^{2n} + 13^{124}$	$\in \Theta(n^n)$
$f_{10}(n) = 14400$	$\in \Theta(1)$

Ranking: (slowest growing) f_{10}, f_8, f_4, f_6, f_3, f_2, f_5, f_7, f_9, f_1 (fastest growing)

```
void functionB(int n)
{
    for (int i = 1; i < n; i++)
    {
        for (int j = n; j > n - i; j--)
        {
            printf(j);
        }
    }
}
```

Asymptotic complexity of nested loops multiplies itself (ac. outloop # ac. inner loop)

→ **O(n²)**

```
void functionC(int n)
{
    for (int i = n; i > 1; i = i / 2)
    {
        for (int j = 1; j <= n; j++)
        {
            printf("Hello World!");
        }
    }
}
```

The outer for loop has an asymptotic complexity of O(log(n)), the inner of O(n), in total there is a complexity of **O(n · log(n))**.

```
void functionE(int n)
{
    for (int i = n; i <= n; i++)
    {
        for (int j = n; j > 1; j = j / 2)
        {
            printf(j);
        }
    }
}
```

The outer for loop will be executed only exactly once (since the counter variable is initiated to n which is the upper boundary for the loop), the inner loop has an asymptotic complexity of O(log(n)), in total there is a complexity of **O(log(n))**.

```
int functionD(int n) {
    int z = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < j; k++) {
                z = z + 1;
            }
        }
    }
    return z;
}
```

→ **O(n³)**

Running Time Analysis: Loop With Arbitrary Fixed Step Size

In case of a (well-behaved) `for` loop with an arbitrary (but constant and integer) step size, the number of executions of the body can be calculated as follows:

up-counting: $\left\lfloor \frac{\text{end} - \text{start} + \text{step size}}{\text{step size}} \right\rfloor$ *down-counting:* $\left\lfloor \frac{\text{start} - \text{end} + \text{step size}}{\text{step size}} \right\rfloor$

Examples:

START	END	STEP	Number of times executed
`for i = 5 to n step 5 do` <loop body>			$\lfloor n / 5 \rfloor + 1$
			$\left\lfloor \frac{END - START + STEP}{STEP} \right\rfloor = \lfloor n / 5 \rfloor$

START	END	STEP	Number of times executed
`for (i = 57; i >= 23; i = i - 3) {`			13
`/* loop body */`			12

Running Time Analysis: Overview of Formulas for Calculation of Number of Executions

counting direction	*boundary inclusive?*	*step size*	*code example*	*no. of executions of the loop body*	*no. of executions of the loop header*
up	yes	1	`for(i = 1; i <= 99; i++)`	end − start + 1	end − start + 2
down	yes	1	`for(i = 99; i >= 1; i--)`	start − end + 1	start − end + 2
up	no	1	`for(i = 1; i < 99; i++)`	end − start	end − start + 1
down	no	1	`for(i = 99; i > 1; i--)`	start − end	start − end + 1
up	yes	arbitrary	`for(i = 1; i <= 99; i=i+3)`	$\left\lfloor \frac{\text{end} - \text{start} + \text{step}}{\text{step}} \right\rfloor$	$\left\lfloor \frac{\text{end} - \text{start} + \text{step}}{\text{step}} \right\rfloor + 1$
down	yes	arbitrary	`for(i = 99; i >= 1; i=i-7)`	$\left\lfloor \frac{\text{start} - \text{end} + \text{step}}{\text{step}} \right\rfloor$	$\left\lfloor \frac{\text{start} - \text{end} + \text{step}}{\text{step}} \right\rfloor + 1$

Algorithm: findKmax(A, k, n)	*Cost*	*Number of times executed*
1 **for** i = 1 **to** n − k + 1 **do**	c_1	$n - k + 2$
2 max = A[i]	c_2	$n - k + 1$
3 **for** j = 1 **to** k − 1 **do**	c_3	$k \cdot (n - k + 1)$
4 **if** A[i+ j] > max **then**	c_4	$(k - 1) \cdot (n - k + 1)$
5 max = A[i + j]	c_5	$\alpha \cdot (k - 1) \cdot (n - k + 1)$; $0 \le \alpha \le 1$
6 print(max)	c_6	$n - k + 1$

$T(n) = c_1 \cdot (n - k + 2) + c_2 \cdot (n - k + 1) + c_3 \cdot k \cdot (n - k + 1) + c_4 \cdot (k - 1) \cdot (n - k + 1) + c_5 \cdot \alpha \cdot (k - 1) \cdot (n - k + 1) + c_6 \cdot (n - k + 1) =$

$= c_1 + (c_1 + c_2 - c_4 + c_6 - c_5 \cdot \alpha + (c_3 + c_4 + c_5 \cdot \alpha) \cdot k) \cdot (n - k + 1)$ → in worst case $O(n^2)$

Running Time Analysis: Loops: Growth Rate and Step Size

The increment of a loop can have different growth rates, for example constant (e.g. i = i + 1), linear (e.g. i = 5 * i), quadratic (e.g. i = i * i) etc. Usually, the increment is a constant which is called the step size.

Loops with a constant increment (step size) can…

- be counting up (e.g. i = i + 1, i = i + 2, …) or counting down (e.g. i = i − 1, i = i − 2, …), and
- have a step size of 1 («counter») or a step size bigger than 1.

Example of a down-counting `for` loop with a step size of 42:

```
for (i = 1000; i > 0; i = i - 42) {
    /* loop body */
}
```

Universität Zürich[UZH]
Institut für Informatik

Number of Executions of the Body of Loops

	pseudo code fragment	*table visualization*	*«end-start+1» summation*	*pure summation*

Independent loops	`for i = 1 to n do`
	`  for j = 1 to n do`
	`  j = j + 1`
			$\sum_{i=1}^{n} \text{end} - \text{start} + 1 = \sum_{i=1}^{n} n - 1 + 1 = \sum_{i=1}^{n} n = n \cdot n = n^2$	$\sum_{i=1}^{n} \sum_{j=1}^{n} 1 = n^2$

Loops connected through initialization	`for i = 1 to n do`
	`  for j = i + 1 to n do`
	`  j = j + 1`
			$\sum_{i=1}^{n} \text{end} - \text{start} + 1 = \sum_{i=1}^{n} n - (i+1) + 1 = \sum_{i=1}^{n} n - i = n^2 - \frac{n \cdot (n+1)}{2}$	$\sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 = n^2 - \frac{n \cdot (n+1)}{2}$

Loops connected through stop condition	`for i = 1 to n do`
	`  for j = 1 to n − i do`
	`  j = j + 1`
			$\sum_{i=1}^{n} \text{end} - \text{start} + 1 = \sum_{i=1}^{n} (n - i) - 1 + 1 = \sum_{i=1}^{n} n - i = n^2 - \frac{n \cdot (n+1)}{2}$	$\sum_{i=1}^{n} \sum_{j=1}^{n-i} 1 = n^2 - \frac{n \cdot (n+1)}{2}$

The following example shows how to analyse the number executions of the body of the inner loop (lines 7 to 10) of the algorithm alg(A, n) using a table to track the evolution of the number of executions depending on the value of the outer loop variable.

Outer loop variable value i =	*Inner loop variable range* j = i+1…n−i	*Number of executions of the inner loop body*
1	2 … n−1	n − 2
2	3 … n−2	n − 4
3	4 … n−3	n − 6
⋮	⋮	⋮
k	k+1 … n−k	n − 2·k
⋮	⋮	⋮
⌊n/2⌋	⌊n/2⌋+1 … ⌈n/2⌉	0

Algorithm: alg(A,n)

```
1  for i = 1 to ⌊n/2⌋ do
2      min = i;
3      max = n − i + 1;
4      if A[min] > A[max] then
5          exchange A[min] and A[max];
6      for j = i + 1 to n − i do
7          if A[j] < A[min] then
8              min = j;
9          if A[j] > A[max] then
10             max = j;
11     exchange A[i] and A[min];
12     exchange A[n − i + 1] and A[max];
```

$$\sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{j=i+1}^{n-i} 1$$

Algorithm: alg(A,n)

```
1  for i = 1 to ⌊n/2⌋ do
2      min = i;
3      max = n − i + 1;
4      if A[min] > A[max] then
5          exchange A[min] and A[max];
6      for j = i + 1 to n − i do
7          if A[j] < A[min] then
8              min = j;
9          if A[j] > A[max] then
10             max = j;
11     exchange A[i] and A[min];
12     exchange A[n − i + 1] and A[max];
```

a) What does algorithm whatDoIDo(A, n, k) do?

This algorithm is an implementation of a descending selection sort with simultaneous summation, restricted to the first k elements of the input array: The algorithm will sort the first k elements of the array in descending order and at the same time will calculate and return the sum of the k biggest elements by adding up the current maximum.

For example, given the input array A = [12, 2, 10, 4, 8] and k = 3, the return value will be 12 + 10 + 8 = 30 and the state of array A after the algorithm will be A' = [12, 10, 8, 2, 4].

A =	12	4	10	2	8
	1	2	3	4	5

Algo: WHATIDO(A, n, k)

```
sum = 0;
for i = 1 to k do
    maxi = i;
    for j = i + 1 to n do
        if A[j] > A[maxi] then
            maxi = j;
    sum = sum + A[maxi];
    swp = A[i];
    A[i] = A[maxi];
    A[maxi] = swp;
return sum
```

Tips and Tricks for Finding Out What an Algorithm Does

- Having experience (i.e. having seen, thought about and understood many algorithms) and practicing definitely helps.
- Looking at the variable names may help (assuming they are not intentionally misleading).
- Find out what parts of the code are intended for (i.e. lines 8 to 10 are a swap) and wrap them into an abstraction (i.e. write «swap A[i] and A[max]» instead of lines 8 to 10).
- Make a drawing of the array on which the algorithm operates and think about how the loops go over it.
- Make a table and trace how the variables and their values change.
- Calculate a set of input and output values and compare them.
- …

increasing desperation

Summations: Polynomial Expressions

- Sum of the square of a sum:

$$\sum_{k=1}^{n} (a_k + b_k)^2 = (a_1 + b_1)^2 + (a_2 + b_2)^2 + \ldots + (a_n + b_n)^2 = \sum_{k=1}^{n} a_k^2 + 2 \cdot \sum_{k=1}^{n} a_k \cdot b_k + \sum_{k=1}^{n} b_k^2$$

Summations: Polynomial Expressions

- Sum of squares:

$$\sum_{k=1}^{n} k^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} \quad \in \Theta(n^3)$$

- Sum of cubes:

$$\sum_{k=1}^{n} k^3 = 1^3 + 2^3 + 3^3 + \ldots + n^3 = \left(\sum_{k=1}^{n} k \right)^2 = \left(\frac{n \cdot (n+1)}{2} \right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4} \quad \in \Theta(n^4)$$

- Sum of fourths:

$$\sum_{k=1}^{n} k^4 = 1^4 + 2^4 + 3^4 + \ldots + n^4 = \frac{n \cdot (n+1) \cdot (2n+1) \cdot (3n^2 + 3n - 1)}{30} = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} + \frac{n}{30} \quad \in \Theta(n^5)$$

Summations: Arithmetic Series

There is one particularly famous and important (and often used) case of the arithmetic series:

$$\sum_{k=0}^{n} k = \sum_{k=1}^{n} k = \frac{n \cdot (n+1)}{2}$$

(Gauß'sche Summenformel, $a_0 = 0$, $d = 1$)

Summations: Geometric Series

A summation of the form

$$\sum_{k=0}^{n} a^k = 1 + a^1 + a^2 + a^3 + \ldots + a^n$$

where a is some real number with $0 < a \neq 1$ and n is an integer with $n > 0$ is called a (finite) geometric series.

This finite geometric series sums up to $\frac{a^{n+1} - 1}{a - 1} = \frac{1 - a^{n+1}}{1 - a}$ (If a = 0, then the series sums just up to n + 1.)

The infinite geometric series $\sum_{k=1}^{\infty} a^k = 1 + a^1 + a^2 + a^3 + \ldots$

converges to 1 / (1 - a) iff $|a| < 1$, otherwise it diverges.

Invariants -> stays the same throughout a piece of code

Invariants are an useful tool for

- proofing the correctness of an algorithm and in particular of loops (loop invariants),
- better understanding and reasoning about algorithms in general.

An invariant is a logical expression, i.e. a statement which can be evaluated to true or false. This logical expression refers to some property of an algorithm or a part of an algorithm and has to be true in a certain range of the control flow for the algorithm to work as it should. Oftentimes, the invariant captures the main idea underlying an algorithm.

An invariant could look as follows for example:

- «the value of variable x has always to be positive», formally: $x > 0$
- «all numbers from 1 to 41 are present somewhere in array A of length n», formally: $\forall x: 0 < x < 42, \exists i \leq n: A[i] = x$

Invariant Finding Example

Find an invariant for the loop in the following C code fragment:

```
int c = 42;
int x = c;
int y = 0;
while (x > 0)
{
    x = x - 1;
    y = y + 1;
}
```

x decreases by 1 each time and y increases by 1 each time (-> sum will always stay the same)

c = x + y = 42 is an invariant of the while loop in this C code fragment.

Note that this is not the only invariant of the loop but there are many possibilities, e.g.

- $y \geq 0$
- $x \geq 0$
- $(x \% 2 = y \% 2) \wedge (x \cdot y \neq 1)$
- …

Obviously, in practice we will only be interested in an invariant which is useful for understanding the algorithm and proofing a certain property of it. What is a helpful invariant, depends on what the algorithm does.

Loop Invariants

For every loop, a loop invariant can be assigned.

To show that a loop invariant holds, three conditions have to be checked:

- initialization: the invariant has to be true (right) before the first iteration of the loop starts
- maintenance: if it is true before an iteration then it is true after that iteration; the invariant has to be true right at the start and right at the end of each iteration of the loop
- termination: the invariant has to be true (right) after the loop has ended; this is a useful property that helps to show that the algorithm is correct

This technique is similar to an inductive proof.

Exercise 3, Task 1c and d

Algo: whatDoesItDo(A, n, k)

```
result = −1000
for i = 1 to n do
    current = 0
    for j = i to n by k do
        current = current + A[j]
    if current > result then
        result = current
return result
```

$$\sum_{\ell=1}^{n} \left(\frac{n-\ell}{k} + 2\right) = \sum_{\ell=1}^{n} \frac{n-\ell}{k} + \sum_{\ell=1}^{n} 2 = \frac{1}{k}\left[\sum_{\ell=1}^{n} (n-\ell)\right] + 2n = \frac{1}{k}\left(\sum_{\ell=1}^{n} n - \sum_{\ell=1}^{n} \ell\right) + 2n = \frac{1}{k}\left(n^2 - \frac{n(n+1)}{2}\right) + 2n = \frac{n^2 - n}{2k} + 2n$$

Loop Invariants: Strategies

Two parts can be distinguished when working with loop invariants:

- finding the loop invariant,
- writing the loop invariant using formal language.

An unavoidable prerequisite for formulating a loop invariant is to understand what the loop actually does and should accomplish within the algorithm it is part of. Loop invariants are a formalization of one's intuition about how the loop works. For this part, it is difficult to give some general advise or recipes because algorithms use quite different ideas. Some algorithms use make use of similar ideas and it therefore helps if one has seen some examples of loop invariants.

If the loop invariant was found and formulated using natural language, it has to be written using formal language (predicate logic). This part is mainly a matter of training.

Precondition and Postcondition

-> promise of what is true at the begining of the function

Precondition and postcondition are logical expressions which can be evaluated to true or false.

- A precondition defines the range / set of arguments and environment situations which allow a meaningful execution a program, function or part of it. If the input arguments do not fulfill the precondition, the result of the respective calculations are undefined. The responsibility is outsourced to the person calling the respective program function. A precondition could be for example that an input parameter might not be equal to zero (because it is used in a division).
- A postcondition is a description of the output or state which is reached after successful execution of a program, function or part of it.

-> promise of what is true at the end of the function

Recurrences, Divide and Conquer, Master Method

Motivation: Another Running Time Analysis Example

What is the asymptotic upper bound for the following C code fragment?

```
int functionH(int a, int n)
{
        if (n == 1) {
                return a;
        }
        else {
                return a + functionH(a, n/2);
        }
}
```

always divides n by 2, so log(n)

→ O(log(n))

Follow-up question:

What is the result of `functionH(1, 100)`?

→ 7

Introduction Recurrences

The previous running time analysis example cannot be solved in the same way as this was done in earlier cases where we just determined the number of executions for each of the lines and multiplied this with the time needed (cost) for the respective line and add all together to get the total time needed.

Instead, here the time for execution of the function depends on the function itself: «The time needed to execute the function with parameter n is the time needed to execute the function for half the parameter size, n/2, plus some time for the non-recursive parts of the function (calculated as in the cases before)».

This kind of problem can be formulated as a recurrence relation:

T(n) = T(n/2) + 1.

We now need a method to rewrite this kind of equation into a non-recursive form where the T() function is only on one side of the equation. Recurrence relations can be thought of as mathematical models of the behaviour of recursive functions (in particular of divide and conquer algorithms).

- Recurrence (relation): description of a function which is recursive: f(n) = g(f(n)).

- The phrase «solving a recurrence» means to find a way to calculate a result for any input value of the function without the need of recursive expansion (also called «to find a closed solution / expression»). Thus the recursively defined function should only be on one side of the equation.

Solving recurrences is in a way similar to solving integrals: In both cases there is no single one-works-for-all procedure.

There are recurrences which cannot be solved.

(Example: most variants of the logistic map: T(n + 1) = r · T(n) · (1 − T(n))).

Methods for Solving Recurrences

- Repeated substitution and looking sharply (also called iteration method / iterating the recurrence)
- Recursion tree – basically the same as repeated substitution, just graphically visualized
- Good guessing, followed by formal proof (also called «substitution method»); the proof can also be done as a follow-up when using the previous two methods (note that the terms «substitution method» and «repeated substitution» refer to two different methods although the sound quite similar)

  (→ with induction)
- Master method: if it is a divide-and-conquer problem and you are only interested in order of growth (not an actual solution of the recurrence as defined before).
- (Characteristic equation: Certain types of recurrences – second-order linear homogeneous recurrence relations with constant coefficients – can be considered as a kind of polynomial and then a solution can be derived mechanically.)

Repeated Substitution

General application procedure:

1) Perform a sequence of substitute – expand – substitute – expand – … loops until you see a pattern.
2) Find a general (still recursive) formula for the situation after the k-th substitution.
3) Find out how many iterations have to be done in order to get to the base case.
4) Fill in the number of iterations for the base case into the general formula for k-th substitution.

- Tends to be confusing (at least for me).
- Can be done «in-line» or with a «helper row» (to probably avoid confusion).
- Quickly becomes unsuitable for more complexes cases, e.g. if multiple recursive calls are involved in the recurrence relation (an example recurrence where repeated substitution method would be a dead end street is for example: T(n) = T(n/3) + 2 · T(n/3²) + n). In those cases, another method needs to be used, for example the recursion tree method (which is basically just a clever graphical depiction of the same thing helping to sum up terms).

Repeated Substitution: Example

T(1) = 4; T(n) = 2 · T(n/2) + 4 · n

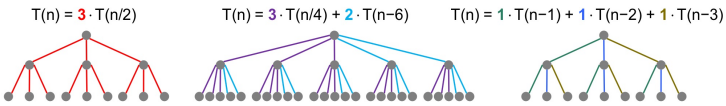
iteration h =	Expanded recurrence *Fill in the expanded term from right side as a substitution for the term that was expanded*	Term currently to be expanded *fill in the argument from left side everywhere you see n in the recurrence template; can be helpful to not simplifiy terms too early*
1	$T(n) = 2 \cdot T(n/2) + 4 \cdot n =$	$T(n/2) = 2 \cdot T((n/2)/2) + 4 \cdot (n/2)$
2	$= 2 \cdot [2 \cdot T((n/2)/2) + 4 \cdot (n/2)] + 4 \cdot n =$ $= 2^2 \cdot T(n/2^2) + 4 \cdot n + 4 \cdot n =$	$T(n/2^2) = 2 \cdot T((n/2^2)/2) + 4 \cdot (n/2^2)$
3	$= 2^2 \cdot [2 \cdot T((n/2^2)/2) + 4 \cdot (n/2^2)] + 8 \cdot n =$ $= 2^3 \cdot T(n/2^3) + 4 \cdot n + 4 \cdot n + 4 \cdot n = \dots$	$T(n) = 2^k \cdot T(n/2^k) + k \cdot (4 \cdot n)$

(3 · 4 · n)

Recursion Tree Method

→ term with largest complexity stays on top of tree!

- Basically the same idea as in the repeated substitution method, but graphically visualized as a tree.
- Number of recursive calls corresponds to number of branches per node.



- Values at nodes is work done outside recursive calls (at current recursive call)
- Sum of node values in the whole tree (down to the base case) is total work done.

Base case: $T(1) = 4$; $T(n) = \underbrace{2}_{\text{branches per node}} \cdot \underbrace{T(n/2)}_{\text{Recursive part}} + \underbrace{4 \cdot n}_{\text{Work outside recursive call = values at nodes}}$

level	number of nodes	recursive call	non-recursive value	recursion tree	sum per row
0	1	T(n)	4·n	4·n	4 · n
1	2	T(n/2)	4·n/2	4·n/2 4·n/2	4 · n
2	2^2	$T(n/2^2)$	$4 \cdot n/2^2$	$4 \cdot n/2^2$ $4 \cdot n/2^2$ $4 \cdot n/2^2$ $4 \cdot n/2^2$	4 · n
3	2^3	$T(n/2^3)$	$4 \cdot n/2^3$	$4 \cdot n/2^3$ … … … … $4 \cdot n/2^3$	4 · n
…	…	…	…		…
k	2^k	$T(n/2^k)$	$4 \cdot n/2^k$	$4 \cdot n/2^k$ … $4 \cdot n/2^k$	4 · n
…	…	…	…		…
x	2^x	T(1)	4	4 … 4	$\sum \dots$

Repeated Substitution: Example

- We found the general expression for the k-th iteration of substitution: $T(n) = 2^k \cdot T(n/2^k) + k \cdot (4 \cdot n)$.
- But when (at which k = x) will we reach the base case T(1)? (How often do we have to substitute / execute recursive calls?)
- This will be the case when $T(n/2^x) = T(1)$, i.e. when

  $n/2^x = 1$
- From this: $n = 2^k$, $x = \log_2(n)$
- Now insert the x where we reach the base case into the general expression from above and then substitute the value given for T(1):

$$T(n) = 2^{\log_2(n)} \cdot T(1) + \log_2(n) \cdot (4 \cdot n) =$$
$$= n \cdot 4 + \log_2(n) \cdot (4 \cdot n) =$$
$$4 \cdot n + 4 \cdot n \cdot \log_2(n) \in \Theta(n \cdot \log(n))$$

Recursion Tree Method: Example

- In order to get the total work done in the whole recursion tree, we have to sum up the row sums.
- Thus, we need to know the number of rows which is the same as the height / number of levels of the tree.
- Again, we have to find the number of iterations – or in this case: tree levels – needed to get to the base case. The respective considerations are analogous as for the repeated substitution method: The base case is reached when $T(n/2^x) = T(1)$ i.e. when $n/2^x = 1$ and from this we find: $n = 2^x$, $x = \log_2(n)$.
- Now, we sum up the row sums and find the result:

$$\sum_{k=0}^{x} 4 \cdot n = \sum_{k=0}^{\log_2(n)} 4 \cdot n = 4 \cdot n \cdot \sum_{k=0}^{\log_2(n)} 1 = 4 \cdot n \cdot (\log_2(n) + 1)$$

$$4 \cdot n + 4 \cdot n \cdot \log_2(n) \in \Theta(n \cdot \log(n))$$

Exercise 4, Task 3b

$T(n) = 3T(n-2) + n$, where $T(1) = 1$

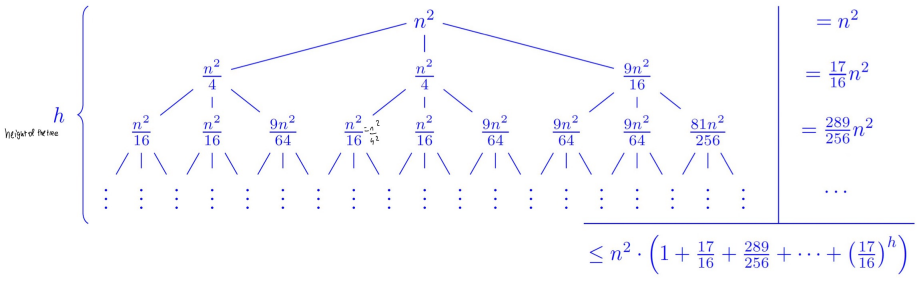
$$
\begin{aligned}
T(n) &= 3T(n-2) + n \\
&= 3(3T(n-2-2) + (n-2)) + n = 3^2T(n-2\cdot 2) + 3\cdot(n-2) + n = 9T(n-4) + 4n - 6 \\
&= 3(3(3T(n-2-2-2) + (n-4)) + (n-2)) + n \\
&= 3^3T(n-3\cdot 2) + 3^2\cdot(n-2\cdot 2) + 3\cdot(n-2) + n = 27T(n-6) + 13n - 42 \\
&= \dots \\
&= 3^kT(n-2k) + \sum_{i=1}^{k} 3^{i-1}\cdot(n-2\cdot(i-1)) \\
&< 3^kT(n-2k) + n\cdot\sum_{i=1}^{k} 3^{i-1}
\end{aligned}
$$

The base case will be reached when $T(n-2k_{max}) = T(1)$, i.e. for $k_{max} = \lceil \frac{n-1}{2} \rceil$
As a simplification, only even values of n will be considered which allows to reformulate the above result as $k_{max} = n/2$ and $T(0) = T(1) = 1$ will be assumed.
This yields for the base case:

$$T(n) < 3^{\frac{n}{2}}\cdot T(1) + \Theta(n)\cdot\Theta(3^{\frac{n}{2}})$$

$\Rightarrow \mathbf{T(n) \in O(n\sqrt[2]{3^n})}$

$$\text{T(n)} = \begin{cases} 1 & \text{if n = 1} \\ 2\text{T(n/2)} + \text{T(3n/4)} + \text{n}^2 & \text{if n > 1} \end{cases}$$



$$\leq n^2\cdot\left(1 + \tfrac{17}{16} + \tfrac{289}{256} + \dots + \left(\tfrac{17}{16}\right)^h\right)$$

Exercise 4, Task 3c

$$
\begin{aligned}
T(n) &= T(\sqrt[2]{n}) + \log(n) = T(n^{\frac{1}{2}}) + \log(n) \\
&= T\left(\sqrt[2]{\sqrt[2]{n}}\right) + \log(\sqrt[2]{n}) + \log(n) = T(n^{\frac{1}{2^2}}) + \log(n^{\frac{1}{2}}) + \log(n) = T(n^{\frac{1}{4}}) + \frac{1}{2}\log(n) + \log(n) \\
&= T\left(\sqrt[2]{\sqrt[2]{\sqrt[2]{n}}}\right) + \log(\sqrt[2]{\sqrt[2]{n}}) + \log(\sqrt[2]{n}) + \log(n) = \\
&= T(n^{\frac{1}{2^3}}) + \log(n^{\frac{1}{2^2}}) + \log(n^{\frac{1}{2}}) + \log(n) = T(n^{\frac{1}{8}}) + \frac{1}{4}\log(n) + \frac{1}{2}\log(n) + \log(n) \\
&= \dots \\
&= T(n^{\frac{1}{2^k}}) + \sum_{i=0}^{k-1}\frac{1}{2^i}\log(n) = T(n^{\frac{1}{2^k}}) + \log(n)\sum_{i=0}^{k-1}\left(\frac{1}{2}\right)^i
\end{aligned}
$$

The sum $\left(1 + \frac{17}{16} + \frac{289}{256} + \dots + \left(\frac{17}{16}\right)^h\right) = \sum_{i=0}^{h}\left(\frac{17}{16}\right)^i = \sum_{i=0}^{h} q^i$ is a finite geometric series. Since $|q| = \frac{17}{16} > 1$, the respective infinite geometric series $\sum_{i=0}^{\infty} q^i$ diverges and thus cannot be used to find an upper bound. The finite geometric series $\sum_{i=0}^{h} q^i$ is known to sum up to $\frac{q^{h+1}-1}{q-1}$. The recursion tree has its biggest height / longest branch on the left-hand side where it grows until $\frac{n^2}{4^h} = 1$ which implies $h = 2\log_4(n)$. To estimate the total work in the recursion tree's nodes, we will assume the tree has the height of the leftmost branch everywhere and this overestimation will yield an upper bound for the total work in the tree's nodes:

$$
\begin{aligned}
T(n) &< n^2\cdot\sum_{i=0}^{h}\left(\frac{17}{16}\right)^i = n^2\frac{\left(\frac{17}{16}\right)^{h+1}-1}{\frac{17}{16}-1} \\
&= n^2\frac{\left(\frac{17}{16}\right)^{2\log_4(n)+1}-1}{\frac{1}{16}} = 16n^2\cdot\left(n^{2\log_4(\frac{17}{16})} - 1\right) \\
&\approx 16n^{4.044} - 16n^2 \in O(n^5)
\end{aligned}
$$

Form / Recipe Master Method

(simplified version)

Given recurrence: $T(n) =$

1 Look and interpret

2 Perform pattern matching: $T(n) = a \cdot T(n/b) + f(n)$

$a =$ _____, $b =$ _____, $f(n) =$ _____

Pattern matching successful? ☐ yes ☐ no ⟶ *abort / try different method*

3 Check parameters

Parameter	*Criterion*	*Evaluation*
a	constant, $a \geq 1$	☐ ok ☐ not ok
b	constant, $b > 1$	☐ ok ☐ not ok
f(n)	asymptotically positive	☐ ok ☐ not ok

Master method applicable? ☐ yes ☐ no ⟶ *abort / try different method*

4 Determine case

Calculate $x = \log_b(a) =$ _____ (leaves in recursion tree: n^x)

Compare asymptotically $f(n) =$ _____ with $n^x =$ _____:

Comparison	*Description*	*Case*	*Result*	*Work*
f(n) grows polynomially slower than n^x.	$f(n) \text{ "}\preccurlyeq\text{" } \Theta(n^x)$ $\exists \varepsilon > 0: f(n) \in O(n^{x-\varepsilon})$	① ☐	$T(n) \in \Theta(n^x)$	mainly in the leaves
f(n) grows (about) as fast as n^x.	$f(n) \text{ "}\cong\text{" } \Theta(n^x)$ $\exists \varepsilon > 0: f(n) \in \Theta(n^{x-\varepsilon} \cdot \log(n))$	② ☐	$T(n) \in \Theta(n^x \cdot \log(n))$	distributed evenly
f(n) grows polynomially faster than n^x.	$f(n) \text{ "}\succcurlyeq\text{" } \Theta(n^x)$ $\exists \varepsilon > 0: f(n) \in \Omega(n^{x-\varepsilon})$	③ ☐	$T(n) \in \Theta(f(n))$	mainly in the root node

4a Check regularity for case ③

Find constants $c < 1$ and $n > n_0$ such that it holds for any n big enough that:

$$a \cdot f(n/b) \leq c \cdot f(n)$$

$$\_\_\_\_\_ \leq \_\_\_\_\_ \cdot \_\_\_\_\_$$

Regularity check successful? ☐ yes ☐ no ⟶ *abort / try different method*

5 Write down solution

$T(n) \in \Theta($_____$)$

Form / Recipe Master Method

(simplified version)

Given recurrence: $T(n) = 7 \cdot T(n/2) + n^2$

1 Look and interpret

The problem is split up in 7 subproblems each having half the size oft he original problem. The work which needs to be done outside of the recursive call depends quadratically on the size n of the problem.

2 Perform pattern matching: $T(n) = a \cdot T(n/b) + f(n)$

$a =$ 7, $b =$ 2, $f(n) =$ n^2

Pattern matching successful? ☒ yes ☐ no ⟶ *abort / try different method*

3 Check parameters

Parameter	*Criterion*	*Evaluation*
a	constant, $a \geq 1$	☒ ok ☐ not ok
b	constant, $b > 1$	☒ ok ☐ not ok
f(n)	asymptotically positive	☒ ok ☐ not ok

Master method applicable? ☒ yes ☐ no ⟶ *abort / try different method*

4 Determine case

Calculate $x = \log_b(a) =$ $\log_2(7) \approx 2.807$ (leaves in recursion tree: n^x)

Compare asymptotically $f(n) =$ n^2 with $n^x =$ $n^{\log_2(7)} \approx n^{2.807}$:

Comparison	*Description*	*Case*	*Result*	*Work*
f(n) grows polynomially slower than n^x.	$f(n) \text{ "}\preccurlyeq\text{" } \Theta(n^x)$ $\exists \varepsilon > 0$: $f(n) \in O(n^{x-\varepsilon})$	① ☒	$T(n) \in \Theta(n^x)$	mainly in the leaves
f(n) grows (about) as fast as n^x.	$f(n) \text{ "}\cong\text{" } \Theta(n^x)$ $\exists \varepsilon > 0$: $f(n) \in \Theta(n^{x-\varepsilon} \cdot \log(n))$	② ☐	$T(n) \in \Theta(n^x \cdot \log(n))$	distributed evenly
f(n) grows polynomially faster than n^x.	$f(n) \text{ "}\succcurlyeq\text{" } \Theta(n^x)$ $\exists \varepsilon > 0$: $f(n) \in \Omega(n^{x-\varepsilon})$	③ ☐	$T(n) \in \Theta(f(n))$	mainly in the root node

4a Check regularity for case ③

Find constants $c < 1$ and $n > n_0$ such that it holds for any n big enough that:

$a \cdot f(n/b) \leq c \cdot f(n)$

$\_\_\_\_\_ \leq \_\_\_\_\_ \cdot \_\_\_\_\_$

Regularity check successful? ☐ yes ☐ no ⟶ *abort / try different method*

5 Write down solution

$T(n) \in \Theta($ $n^{\log_2(7)}$ $)$

Form / Recipe Master Method

(simplified version)

Given recurrence: $T(n) = 32 \cdot T(n/4) + n^3$

1 Look and interpret

The problem is split up in 32 subproblems each having a quarter the size oft he original problem. The work which needs to be done outside of the recursive call depends cubically on the size n of the problem.

2 Perform pattern matching: $T(n) = a \cdot T(n/b) + f(n)$

$a =$ **32**, $b =$ **4**, $f(n) =$ **n^3**

Pattern matching successful? ☒ yes ☐ no → *abort / try different method*

3 Check parameters

Parameter	*Criterion*	*Evaluation*
a	constant, $a \geq 1$	☒ ok ☐ not ok
b	constant, $b > 1$	☒ ok ☐ not ok
f(n)	asymptotically positive	☒ ok ☐ not ok

Master method applicable? ☒ yes ☐ no → *abort / try different method*

4 Determine case

Calculate $x = \log_b(a) =$ **$\log_4(32) = 2.5$** (leaves in recursion tree: n^x)

Compare asymptotically $f(n) =$ **n^3** with $n^x =$ **$n^{\log_4(32)} = n^{2.5}$**:

Comparison	*Description*	*Case*	*Result*	*Work*
f(n) grows polynomially slower than n^x.	$f(n)$ "$\preccurlyeq$" $\Theta(n^x)$ $\exists \varepsilon > 0$: $f(n) \in O(n^{x-\varepsilon})$	① ☐	$T(n) \in \Theta(n^x)$	mainly in the leaves
f(n) grows (about) as fast as n^x.	$f(n)$ "$\cong$" $\Theta(n^x)$ $\exists \varepsilon > 0$: $f(n) \in \Theta(n^{x-\varepsilon} \cdot \log(n))$	② ☐	$T(n) \in \Theta(n^x \cdot \log(n))$	distributed evenly
f(n) grows polynomially faster than n^x.	$f(n)$ "$\succcurlyeq$" $\Theta(n^x)$ $\exists \varepsilon > 0$: $f(n) \in \Omega(n^{x-\varepsilon})$	③ ☒	$T(n) \in \Theta(f(n))$	mainly in the root node

4a Check regularity for case ③

Find constants $c < 1$ and $n > n_0$ such that it holds for any n big enough that:

$a \cdot f(n/b) \leq c \cdot f(n)$

$32 \cdot (n/4)^3 = 0.5 \cdot n^3$ $\leq$ **0.5** · **n^3**

→ The equation is fulfilled for $c = 0.5$ and any arbitrary n.

Regularity check successful? ☒ yes ☐ no → *abort / try different method*

5 Write down solution

$T(n) \in \Theta($**n^3**$)$

Master Method: General Remarks

- The term «master method» (or even «master theorem») is rather exaggerative and pretentious; actually it's merely a kind of cooking recipe – and can be applied as such.
- It is a method specific for solving divide-and-conquer recurrences – and only them.
- Will yield asymptotic bound solution only and not an exact running time analysis as in the case of repeated substitution and recursion tree method.

Master Method: Interpretation

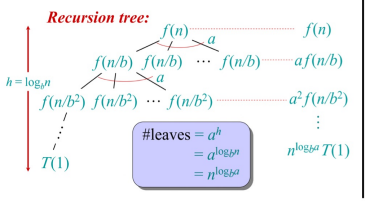
$$T(n) = a \cdot T(n/b) + f(n)$$

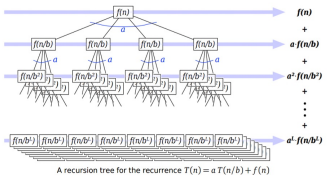
- n — size of problem
- a — number of subproblems
- b — factor by which the problem size is reduced in recursive calls
- f(n) — work done outside of recursive calls, work needed to split and merge

«The original problem of size n is split up into a new subproblems which are smaller by a factor of b. For splitting the original problem and merging of the solutions to the subproblems, work is needed which is described by f(n).»

Master Method: Recursion Tree

$$T(n) = a \cdot T(n/b) + f(n)$$





A recursion tree for the recurrence $T(n) = a\,T(n/b) + f(n)$

Master Method: Prerequisites

Prerequisites for the application of the master method:

- a, b constants
  method not suited for variable resizing of problem
- a ≥ 1
  otherwise the original problem would be deconstructed into less than one subproblem
- b > 1
  otherwise subproblems will be bigger than the original problem

f(n) asymptotically positive (i.e. for large enough n, f(n) is positive)
negative work doesn't make sense

Master Method: Admissibility Examples

Decide whether the master method can be applied to the following recurrences:

- $T_1(n) = 4n \cdot T_1(n/3) + n^{2n}$
  → not applicable: a = 4n is not a constant
- $T_2(n) = ½ \cdot T_2(n/3) + n/2$
  → not applicable: a < 1
- $T_3(n) = 3 \cdot T_3(n/2^{-1}) + \log(n)$
  → not applicable: b < 1
- $T_4(n) = 3 \cdot T_4(n - 3) + n$
  → not applicable: wrong format / parsing fails
- $T_5(n) = 4 \cdot T_5(n/4) - n^2$
  → not applicable: f(n) = $-n^2$ is not asymptotically positive
- $T_6(n) = 5 \cdot 2^n + 5 \cdot T_6(n/5) + n^2$
  → applicable: a = 5, b = 5, f(n) = $5 \cdot 2^n + n^2$
- $T_7(n) = T_7(n/2) + 42$
  → applicable: a = 1, b = 2, f(n) = 42

Master Method: Three Cases

First, calculate $x = \log_b(a)$. Then compare the non-recursive work f(n) to $n^x = n^{\log_b(a)}$.
(Note that n^x is the number of leaves in the corresponding recursion tree.)

- **Case ①**: f(n) grows polynomially slower than n^x.
  formally: $\exists \varepsilon > 0$: $f(n) \in O(n^{x-\varepsilon})$
  → **$T(n) \in \Theta(n^x)$**
- **Case ②**: f(n) grows (about) as fast as n^x.
  formally: $\exists \varepsilon > 0$: $f(n) \in \Theta(n^{x-\varepsilon} \cdot (\log(n))^k)$ for some $k \geq 0$
  → **$T(n) \in \Theta(n^x \cdot (\log(n))^{k+1})$** very often: k = 0
- **Case ③**: f(n) grows polynomially faster than n^x.
  formally: $\exists \varepsilon > 0$: $f(n) \in \Omega(n^{x-\varepsilon})$
  → **$T(n) \in \Theta(f(n))$**

Master Method: Three Cases

Remarks:

- Case ③ will require an additional regularity check.
- The three cases correspond to different distributions of work in the recursion tree:
  - Case ①: cost at the leaves dominating
  - Case ②: cost evenly distributed
  - Case ③: cost at the root dominating
- Note that it is not sufficient if f(n) just grows somehow slower or faster than n^x. It needs to be *polynomially* slower or faster.

Master Method: Cooking Recipe

1) Look, think, interpret
   To what kind of problem solving approach does the given recurrence relation correspond to (if any); is it a divide-and-conquer problem?
2) Pattern matching
   Try to find the a, b and f(n) parts in the given recurrence relation.
3) Check parameters
   Are the found parameters a, b, f(n) eligible for the application of the master method?
4) Determine the case
   Compare asymptotically n^x where $x = \log_b(a)$ with f(n).
   a) Perform additional regularity check if it is case ③
5) Write down the solution

Master Method: Technicalities

- The base case oftentimes is not described when dealing with the master method. (You could argue that all respective recurrences are actually incomplete.)
  → It is just assumed that T(0) will only affect the result by some constant which can be ignored asymptotically.
- Considerations on rounding of parameters and consequently floor and ceiling functions (Gauß-Klammern) are usually omitted / ignored.
  → It is just written T(n/b) instead of T(⌊n/b⌋) or T(⌈n/b⌉).

Substitution Method / Inductive Proof: Example

Task 1.4b from midterm 1 of FS 2016:

Consider the following recurrence: $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/5) + T(7n/10) + n & \text{if } n > 1 \end{cases}$

Prove the correctness of the estimate O(n) using induction (i.e. show that T(n) is in O(n)).

Note that we're given two things: a recurrence relation and an estimate / guess for the upper bound of T(n).

We will proceed in three steps:

1. Apply definition of asymptotic complexity and fill in the guess which is to be proofen
2. Assume, the guess was true (inductive step) and calculate the consequences of this
3. Find a constant c which fulfills the assumption from (2) for any n which is big enough

Given: $T(n) = 7 \cdot T(n/2) + n^2$ (e.g.: Strassen's algorithm for matrix multiplication)

- *Step 1: Look and think:*
  «The Problem of size n is split up into 7 subproblems, each of which has half the size of the original problem; the work for splitting and merging is quadratic with regard to the problem size.» → seems ok on first look
- *Step 2: Pattern matching:*
  a = 7, b = 2, $f(n) = n^2$ → successful
- *Step 3: Check parameters:*
  a ≥ 1, const ✓; b > 1, const ✓; f(n) asymptotically positive ✓ → successful
- *Step 4: Determine the case:*
  $x = \log_b(a) = \log_2(7) \approx 2.807$
  Compare: $f(n) = n^2$ to $g(n) = n^x \approx n^{2.807}$. ($n^x$ is the number of leaves in the recursion tree)
  Obviously, f(n) grows polynomially slower than n^x.
  $\exists \varepsilon > 0$: $f(n) \in O(n^{x-\varepsilon})$ → e.g. $\varepsilon = 0.8$
  ⇒ Case ①, running time dominated by the cost at the leaves
- *Step 5: Write down result:*
  $T(n) \in \Theta(n^x) = \Theta(n^{ld(7)}) \approx \Theta(n^{2.807})$

Substitution Method / Inductive Proof: Example

Consider the definition of big O notation:

$T(n) \in O(g(n))$ iff $\exists$ constants $n_0 > 0$, $c > 0$ such that $T(n) \leq c \cdot g(n) \quad \forall n \geq n_0$

Our goal is to proof that the recurrence $T(n)$ is in $O(g(n)) = O(n)$, i.e. that $T(n)$ is bounded from above by a linear function $g(n) = n$.

This is true if, for any sufficiently big n, we can find a constant c such that $T(n) \leq c \cdot g(n) = c \cdot n$.

↑ guess which is to proof by finding a suitable witness constant c

Let's assume that the assumption holds that T(n) can be bounded from above by g(n) = n.

If this is the case, we can always replace any occurrence of the term T(n) by $g(n) = c \cdot n$ and be sure that the result will be smaller or equal to what was written there before (by definition of big O).

$$T(n) \leq c \cdot g(n)$$

(This step constitutes the inductive step of a proof by induction.)

When this is applied to the given recurrence, this yields the following:

$$T(n) = T(n/5) + T(7n/10) + n \quad \leq \quad c \cdot g(n/5) + c \cdot g(7n/10) + n \quad = \quad c \cdot n/5 + c \cdot 7n/10 + n = \\ = c \cdot 9n/10 + n = \\ = n \cdot (0.9 \cdot c + 1)$$

↑ Replace T(n/k) with n/k
(Note that if T(n/k) is to be replaced, this has to be replaced by n/k.)

↑ Replace g(n) with the guess and include the constant c

From inductive step, we know want to find a witness constant c for which

$$(0.9 \cdot c + 1) \cdot n \quad \leq \quad c \cdot n$$

Let's just try with c = 10 and fill this into the inequality from above:

$$\begin{aligned} (0.9 \cdot 10 + 1) \cdot n &\leq 10 \cdot n \\ (9 + 1) \cdot n &\leq 10 \cdot n \\ 10 \cdot n &\leq 10 \cdot n \end{aligned}$$

...which is obviously true for any $n \geq 0$ (i.e. the threshold $n_0 > 0$ can be chosen arbitrarily here) and thus the required proof has succeded.

For any choice of c which is bigger than 10, the inequality holds even clearer, of course.

Therefore, we have found a withness c which shows that $g(n) = c \cdot n = 10 \cdot n$ bounds from above the recurrence T(n).

Substitution Method / Inductive Proof: Example

But how come, we've chosen c = 10 here? How can How can we systematically find such a constant c?

The constant c can be found systematically by getting rid of the variable n in the inequality and solving for the constant c. Consider the following sequence of reformattings:

$$\begin{aligned} (0.9 \cdot c + 1) \cdot n &\leq c \cdot n && | : n \\ (0.9 \cdot c + 1) &\leq c && | - 0.9 \cdot c \\ 1 &\leq c - 0.9 \cdot c && | \text{ subtraction} \\ 1 &\leq 0.1 \cdot c && | \cdot 10 \\ 10 &\leq c \end{aligned}$$

Divide and Conquer

Divide and Conquer Algorithms

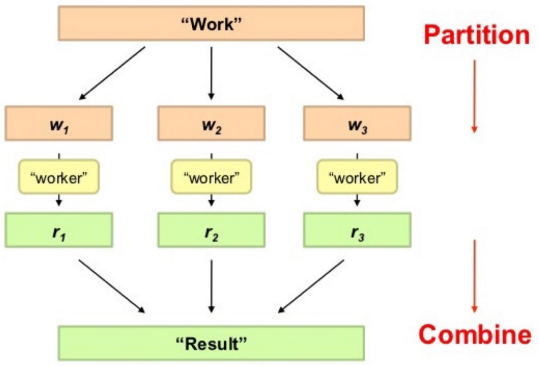
Divide and conquer algorithms consist of three (or four) parts:

(1) divide: split up the problem into subproblems (e.g. in the middle of an array)

(2) conquer: *recursively* apply the solution to the subproblems (until the base case is reached)

(3) combine: merge the solutions from smaller subproblems into the solution of a bigger subproblem

(2a) base case: abort recursion when a minimal problem is reached which can be solved directly

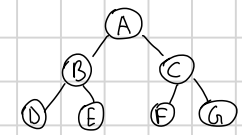


Template for Divide and Conquer Algorithms

Algorithm: *divcon*(problem, start, end)

if *is_small*(problem, start, end) **then**
    return *solution*(problem, start, end)
else
    middle = *divide*(problem, start, end)
    x = *divcon*(problem, start, middle)
    y = *divcon*(problem, middle + 1, end)
    result = *combine*(x, y)
    return result

HeapSort



complete binary tree: $2^n - 1$ nodes

Arr:

A	B	C	D	E	F	G

if a node is at index i

the left child is at index $2 \cdot i$

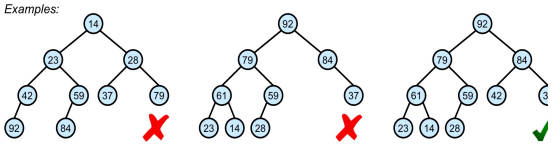
the right child is at index $2 \cdot i + 1$

the parent is at index $\lfloor i/2 \rfloor$

Heap Conditions

- A (binary) heap is a binary tree.
- All levels except the lowest are fully filled and the lowest level is occupied starting from left without gaps (= «nearly complete tree»).

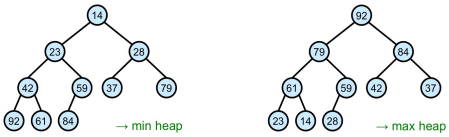
Examples:



→ not a binary heap → not a binary heap → valid binary max heap

Heap Conditions

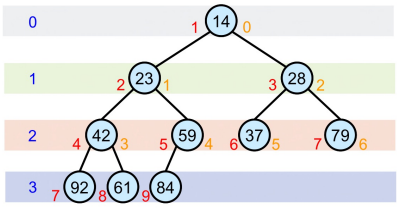
- The keys stored in the nodes of the heap are ordered:
  - in a max-heap: for all nodes, the key is bigger than or equal to all its children
    → largest element at root
  - in a min-heap: for all nodes, the father node is always smaller than or equal to his sons
    → smallest element at root



→ min heap → max heap

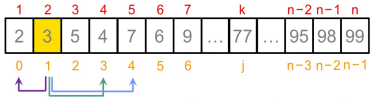
Buildheap: from 1 to $\lfloor n/2 \rfloor$ because it only works from $\lfloor n/2 \rfloor$ to n

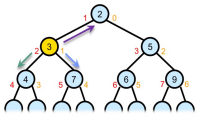
Representation of a Heap as an Array



Note: This kind of tree traversal is called level order.

Addressing Parent and Child Nodes





	Indexing starts at 1	*Indexing starts at 0*
Getting to parent of node with index k	$\lfloor k / 2 \rfloor$	$\lfloor (j - 1) / 2 \rfloor$
Getting to left child of node with index k	$2 \cdot k$	$2 \cdot j + 1$
Getting to right child of node with index k	$2 \cdot k + 1$	$2 \cdot j + 2$

Exercise 5, Task 1.4: Analysis of Heapify

How many times the function Heapify has been executed in HeapSort to completely sort an array?

- A. n/2
- B. (3n-2)/2
- C. n-1
- D. n

→ **B**

What is the worst case time complexity of HeapSort?

A. O(n)

B. O(2n)

C. O(n · log(n))

D. $O(n^2)$

Comparison of Heapifying Code for Min Heap vs. Max Heap

```
void min_heapify(int A[], int i, int n) {
    int min = i;
    int l = lchild(i);
    int r = rchild(i);

    if (l < n && A[l] < A[min]) {
        min = l;
    }
    if (r < n && A[r] < A[min]) {
        min = r;
    }

    if (min != i) {
        swap(A, i, min);
        heapify(A, min, n);
    }
}
```

```
void max_heapify(int A[], int i, int n) {
    int max = i;
    int l = lchild(i);
    int r = rchild(i);

    if (l < n && A[l] > A[max]) {
        max = l;
    }
    if (r < n && A[r] > A[max]) {
        max = r;
    }

    if (max != i) {
        swap(A, i, max);
        heapify(A, max, n);
    }
}
```

Asymptotic Running Times of Heap Operations

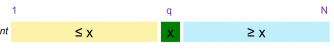
- Heapify: O(log(n))
- Build heap: O(n · log(n)) *(note: this is an upper bound; it can be shown that the tight bound is Θ(n))*
- Remove root and re-heapify: O(log(n))
- Insert and re-heapify: O(log(n))
- Heapsort: O(n · log(n))

QuickSort (kind of Divide and Conquer Algorithm but the combine step is missing)

Quicksort: Principle

1) **Pivot selection**: Choose an element from the input array which should be used for dividing it into two subarrays; this element is called the pivot element (denoted with x here). In the example below, the last element of the array is choosen as pivot.

2) **Divide**: partition the array into two subarrays such that elements in the left part are smaller or equal to the elements in the right part:

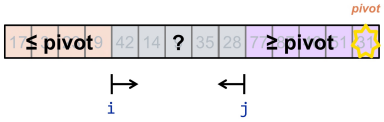


3) **Conquer:** recursively apply the steps 1 and 2 (pivot selection and partitioning) on the two subarrays.

Hoare Partitioning Algorithm

repeat i = i+1 **until** $A[i] \geq x$;
if $i < j$ **then** exchange A[i] ar

- The algorithm selects the rightmost element as the pivot x.
- It then grows two regions: one from left to right and one from right to left.



Partitioning Algorithms: Lomuto / Hoare

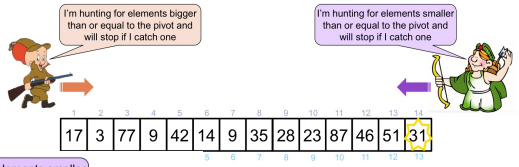
There are two main ways how the partitioning of the array in quicksort can be done:

- **Hoare's Algorithm**: -> a bit more efficient than Lomuto's
  - Grows two regions: A[p..i] from left to right and A[j..r] from right to left

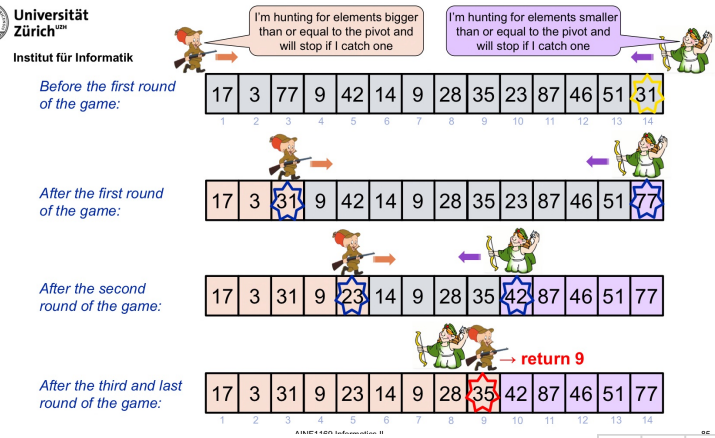
- **Lomuto's Algorithm**:
  - Works with a pivot; grows two regions from left: A[p..i] and A[i..j]; usually slower than Hoare's algorithm because it does three times more swaps on average

Hoare Partitioning: The Story of Two Hunters

Two hunters are marching along an array. One hunter starts at the left at a position before the first element and one hunter starts at the right at a position after the last element in the array. The left hunter will advance to the right (towards higher indices) and the right hunter will advance to the left (towards smaller indices). The hunters embark upon the following game: First, the right hunter advances to the right as long until he finds an element bigger than or equal to the pivot. When he found one, he will stop at once. Afterwards it's the turn of the left hunter who will do the same but looking for elements smaller than or equal to the pivot. When both hunters have stopped and they did not yet meet, they swap the elements they have found. The continue with this process until they meet which is when the game ends and they return the index of their meeting point.



Universität Zürich[UZH]
Institut für Informatik



Hoare Partitioning: The Story of Two Hunters

```
int partitionHoare(int A[], int leftArrayBoundary, int rightArrayBoundary) {
    int pivot = A[rightArrayBoundary];
    int i = leftArrayBoundary - 1;
    int j = rightArrayBoundary + 1;
    while (true) {
        do {
            i = i + 1;
        } while (A[i] < pivot);
        do {
            j = j - 1;
        } while (A[j] > pivot);

        if (i < j) {
            swap(A, i, j);
        }
        else {
            return i;
        }
    }
}
```

Selection of pivot (last element in array) and defining the starting position for the left hunter (i) and the right hunter (j).

The left hunter adavances to the right until he finds a value smaller than the pivot. (Note that this block is equivalent to: `while(A[++i] < pivot);`)

The right huntress adavances to the left until she finds a value bigger than the pivot. (Note that this block is equivalent to: `while(A[--j] > pivot);`)

If the two hunters have not yet met, the numbers they have caught will be swapped.

If the two hunters have met, their meeting position is returned and the algorithm ends.

Lomuto Partitioning Algorithm

The algorithm returns the position / index of the pivot in partitioned array.
It's complexity is Θ(n).

Algo: LomutoPartition(A,l,r)

```
x = A[r];
i = l-1;
for j = l to r-1 do
    if A[j]≤x then
        i = i+1;
        exchange A[i] and A[j];
exchange A[i+1] and A[r];
return i+1;
```

- Comprehension question: What happens when the input array is sorted ascendingly?
- Additional exercise: What is the loop invariant of Lomuto partitioning?

Consider the Lomuto algorithm as shown alongside and in the lecture. Why is the exchange statement on line 7 (marked in colour) necessary?

7 exchange A[i + 1] and A[r];

It is necessary only to handle the case when the pivot is the smallest element in the input array and could be omitted otherwise.

Quicksort: Comprehension Question

Quicksort has an asymptotic running time of O(n·log(n)) in the average case and thus is faster than algorithms like bubblesort, insertionsort or selectionsort.

This is achieved by the quicksort algorithm because it has to do considerably less comparisons of elements than the aforementioned algorithms (in the average case).

Question: How / where does quicksort save comparisons?

Quicksort needs less comparisons because of the **partitioning** it applies. All elements on the left (lower) side of a partitioning will never have to be compared with an element on the right (upper) side of the partitioning. This is not ensured in algorithms like bubblesort.

This also reveals that there is a very close relationship between binary search trees and quicksort. Actually, bottom line, they both apply exactly the same principle.

Exercise 5, Task 2.3: Comparing Heapsort and Quicksort

In which case, heap sort is asymptotically faster than quick sort?

- Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is n lg n, and the constant factors hidden in the n lg n notation are quite small. It also has the advantage of sorting in place.
- A good implementation of quicksort usually will beat heapsort in practice.

Comparison of Sorting Algorithms

Algorithm	*Asymptotic Time Complexity*			*Additional space required*	*Stable?*
	best case	*average case*	*worst case*		
Bubble sort	Θ(n)	Θ(n²)	Θ(n²)	(in place)	yes
Selection sort	Θ(n²)	Θ(n²)	Θ(n²)	(in place)	no
Insertion sort	Θ(n)	Θ(n²)	Θ(n²)	(in place)	yes
Merge sort	Θ(n·log(n))	Θ(n·log(n))	Θ(n·log(n))	Θ(n)	yes
Heapsort	Θ(n·log(n))	Θ(n·log(n))	Θ(n·log(n))	(in place)	no
Quicksort	Θ(n·log(n))	Θ(n·log(n))	Θ(n²)	Θ(log(n))	no

Stable? -> if yes, sorting also doesn't change order of elements with the same value

Pointers and Linked Lists

Structs: the declaration has to end with ;

- It is not allowed to initialize values within the declaration of a struct
- Initialization can be done with curly braces
- Access to the elements of the struct is achieved using the dot operator (e.g. Dario.name)

You can think of a pointer as a data type which contains a memory address (instead of a particular values). In this sense it points to another location in memory, where something of interest is stored. Accordingly, pointers are often visualized as arrows.

What kind of variable is (expected to be) actually stored at the location pointed to, is defined in the declaration of the pointer. (Therefore there will be a pointer type to any other data type, e.g. pointer to a integer, pointer to a float etc.)

pass by value → e.g. swap(int a, int b) changes value in function but not in main (→ doesn't swap)
↳ copy of variable is passed / not the actual variable
pass by reference → e.g. swap(int *a, int *b) changes value in function and in main (must swap!)
↳ actual parameter is passed

Pointers: Syntax

- Declare a pointer variable with name **p** which is pointing at a memory location where the bit pattern is to interpreted as of type **int**:

```
int *a;  or  int* a;
```

- Syntactically, `int * a;` and `int*a;` would also be valid, but are uncommon.
- All kinds of types can be used for pointers, in particular structs. (There is even the special case of `void*` which has special meaning.)
- Beware: `int* x, y, z;` is equivalent to `int *x; int y; int z;`.

Pointers: a Conceptual Perspective

Let's define the following:

- **address(x)** shall denote the memory address where the variable x is stored; takes a variable name as input.
- **content(p)** shall denote the content which is stored at the memory address p; takes a memory address as input.

Let's also use the symbol ← as the assignment operator. Thus when we write x ← 42, we mean that we assign value 42 to the variable x. In programming languages this operator is usually represented by a single equals sign (=) which is distinguished from the sign == which is used to make comparisons.

Pointers: Address-Of, Dereferencing

The functions address() and content() also exist in C language:

address(x)	`&x`	address-of operator
content(p)	`*p`	dereference operator

Remarks:

- Note that the same symbol which is used to access the content at a given memory address (i.e. *), is also used to define a pointer.
- Applying the dereference operator on a pointer p and thus accessing the content at the respective memory address is called dereferencing the pointer p.
- **Make sure that you are absolutely confident using those operators and really understand their meaning. Otherwise you will constantly run into trouble when using pointers!**

Pointers: Examples

What will be the output of the following code examples?

Example A:

```
int main(void) {
        int a = 0;
        int* b = &a;
        printf("%d", *b);
        return 0;
}
```

→ Segmentation fault: Program tries to access memory address 0 (which means: none, normally: NULL) at line 4.

Example B:

```
int main(void) {
        int a = 0;
        int* b = a;
        printf("%d", &b);
        return 0;
}
```

→ The address at which variable b is stored, interpreted as integer value, will be put to the console.

Pointers and Arrays

Array variables are actually pointers to the first element of the array.

This explains a lot of things that might have seemed a bit odd until now when with arrays, e.g. the syntax for passing an array to a function and the respective behaviour (pass by reference).

This also means, that for arrays `&(a[0])` is equivalent to `a` and `&(a[i])` is equivalent to `a+i`.

Therefore, we can also write

```
*(a + i)
```

printf("%p", arr)
printf("%p", &arr[0])
give the same result

instead of

```
a[i]
```

This leads to the concept of pointer arithmetics.

What problem is lurking in the following C code function which was intended to initialize a struct?

```
struct Point2D {
        int x;
        int y;
};

struct Point2D* initNewPoint() {
        struct Point2D p;
        p.x = 4;
        p.y = 2;
        return &p;
}
```

p = malloc(sizeof(Point2D))
Memory has not been allocated
At end of function memory is free again

→ The memory for the struct named p is allocated on the stack segment of memory. After the function `initNewPoint` has returned, anything may happen to this memory space since it is no longer regarded as needed afterwards.

Never try to use a variable which you have declared in a function outside of this function! If you need this behaviour, either allocate the respective memory dynamically (i.e. using `malloc`).

- Memory can be allocated dynamically using the `malloc()` function. The function `free()` is used the release the reserved memory for further usage.
- It is important to never forget to release dynamically allocated memory. This is of particular importance in case memory is dynamically allocated within a loop, since if it is not released, memory will shrink at every iteration (memory leak).

Syntax example:

```
int* pointer;
pointer = malloc(sizeof(int));
*pointer = 11;

printf("%d\n", *pointer);

free(pointer);
```

Use assign the address created by malloc to pointer

Reminder: Pre-Increment and Post-Increment (and Decrement)

If the value of an integer variable x needs to be incremented in C, this can be done by just reassigning using `x = x + 1` or `x += 1`. There are also two convenient operators for this purpose:

- **pre-increment operator**: **x++**
- **post-increment operator**: **++x**

These operators have (apart from being shorter) the advantage that they can be applied in places where an assignment like `x = x + 1` could not be used, for example when printing a value: `printf("%d", x++)`. In this case, it would be necessary to perform the increment on a separate line.

These two operators are not working in exactly the same manner. As their names indicate, the

- pre-increment operator will increment the value of the variable first, before it is used within the larger code context, and the
- post-increment operator will use the current, unchanged value of the variable first within the code and only then increment it.

Thus, you can think of pre-incrementation and post-incrementation as two-step processes.

An array is a constant pointer to its first location (array was allocated memory) when we pass an array to a function, we pass it as reference even if & is not there.

Array ≈ const int* arr (address of A cannot be changed) → a=b gives error

With pass by reference: we can change content of pointer but we can't change the pointer itself (address it points to)

C always passes by value

Reminder: Pre-Increment and Post-Increment (and Decrement)

pre-increment
++x
increment – use

```
x = ++i;   ≙   i = i + 1;
               x = i;
```

```
int x = 42;
printf("%d, ", ++x);
printf("%d", x);
```
} will print 43, 43

post-increment
x++
use – increment

```
x = i++;   ≙   x = i;
               i = i + 1;
```

```
int x = 42;
printf("%d, ", x++);
printf("%d", x);
```
} will print 42, 43

Exercise 6: Task 1a: Pointers

State if the following statements about pointers are true or false:

a) "Pointers in C language can only point to primitive data types (e.g., int, double), not non-primitive data types (e.g., struct, array)."
✘ false

b) "The expressions *ptr++ and ++*ptr produce the same results."
✘ false

c) "If we use a primitive data type variable as the argument to a function, and modify that variable inside the function, the original data will be changed."
✘ false

d) "If we use a pointer as the argument to a function, and change the pointer inside the function, the original data will be changed."
✓ true ✘ false → passed by reference
bcz can't change the pointer itself
depends on what "original data" refers to…

Exercise: Asymptotic Complexities of ADT Operations

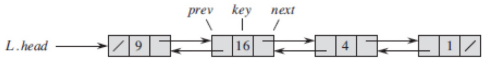
What is the smallest reachable worst-case asymptotic time complexity for each combination of operation and data structure? (Assume canonical properties of all involved data structures.)

Operation \ Data structure	*Ascendingly sorted singly-linked list*	*Ascendingly sorted doubly-linked list*	*Ascendingly sorted array*
finding the smallest element	O(1)	O(1)	O(1)
finding the largest element	O(n)	O(1) (using tail pointer)	O(1)
searching for a given element	O(n)	O(n)	O(log(n)) (using binary search)
deleting a found element (i.e. not including previous search)	O(1) (assuming a reference to the previous node is available; otherwise O(n))	O(1)	O(n)
finding the median	O(n)	O(n)	O(1)

Types of Linked Lists

There is a whole bunch of different flavours of linked lists:

- Singly linked list with tail pointer
- Doubly linked list
- Circular linked list (singly, doubly, with/without tail pointer)
- …



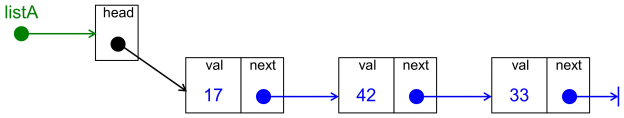
Blueprint for Iterating over a Linked List

In almost every problem, it will sooner or later be necessary to visit all nodes (or a subset of the nodes) in the linked list. This is usually done using a while loop which constantly checks whether a `NULL` pointer is encountered. The current pointer is advanced by reassigning it the next pointer of the current node.

```
while (current != NULL) {
        //do stuff
        current = current->next;
}
```

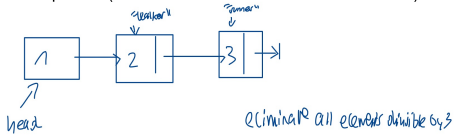
Remarks:

- The iteration can also be done using a for loop:
  `for (current = head; current != NULL; current = current->next) { … }`
- If there is a tail pointer, it is also possible to look for that instead.



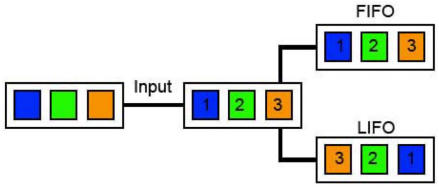
Toolkit for Building Algorithms Working on Linked Lists

- Iterating over a linked list
- Using a second pointer («walker and runner» / «rabbit and turtle»)



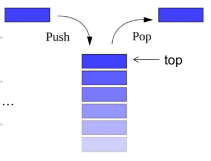
LIFO vs. FIFO

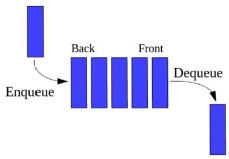
- *LIFO:* The output sequence is reverse of input sequence.
- *FIFO:* The output sequence is the same as the input sequence.



Stacks and Queues https://h5p.org/node/471454

Stack	Queue
	
LIFO	FIFO
– push(x) – pop()	– enqueue(x) – dequeue()





Stack: Implementation in C using Dynamic Array

- In the `main` function, dynamically allocate memory for a struct of type `stack` on the heap segment of memory. (Don't forget to free memory at the end of the `main` function.)
- Use a preprocessor directive to define the initial size of the stack:
  `#define INITIAL_STACK_SIZE 5`
- In the `initialize` function, dynamically allocate memory for an integer array of size `INITIAL_STACK_SIZE`. Assign the pointer to the location in dynamic memory to the field `elements`.

```
typedef struct stackADT {
    int *elements;
    int size;
    int count;
} stack;
```

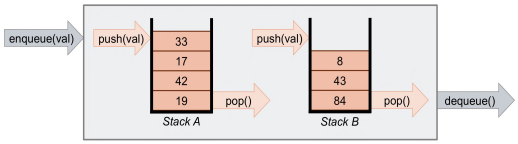
elements: (address of first element of an) integer array containing the contents of the stack
size: current size of the dynamical array `elements`
count: number of values currently stored in the stack

```
int main() {
    queue* myQueue = malloc(sizeof(queue));
    initialize(myQueue);

    for (int i = 1; i <= 6; i++) {
        enqueue(myQueue, i);
    }
    for (int i = 0; i < size(myQueue); i++) {
        printf("%d ", dequeue(myQueue));
    }
    printf("%d ", size(myQueue));

    free(myQueue);
    return 0;
}
```

Solution: 1 2 3 3

Beware: One might get lured into answering «1 2 3 4 5 6 0» which is **wrong**. Note that the upper bound of the second `for` loop (`size(myQueue)`) is not a constant but continuously changes throughout the iterations of the `for` loop because items are removed from the queue within the loop.

Task: Create a queue which is based on two stacks (i.e. wrap a queue interface around a pair of stack interfaces).



Consider a queue Q which currently contains n distinct integers as items. Assume you want to remove the all items from Q which are divisible by 3. The other elements in the queue shall be in the exact same order as before after the removal of these elements and you may not use another storage for the items other than Q and a single helper variable.
How many enqueue and dequeue operations are minimally required in the best case and worst case to achieve this task?

Best case: n operations (if all items are divisible by 3)
Worst case: $2n$ operations (if no item is divisible by 3)

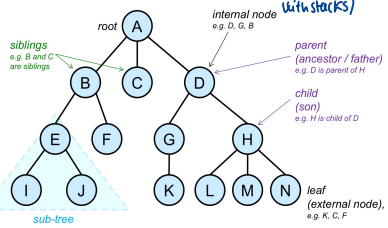
Algorithm: remove_divisible_three(*queue*, *n*)

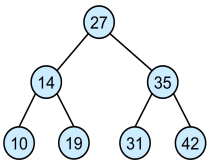
```
for i = 1 to n do
    temp = dequeue(queue)
    if temp mod 3 ≠ 0 then
        enqueue(queue, temp)
```

Trees and Traversals

Trees: Nomenclature

↳ Work with recursion (but can always implement iteratively with stacks)



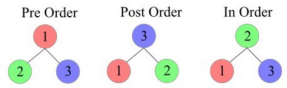


Inorder: 10, 14, 19, 27, 31, 35, 42
Preorder: 27, 14, 10, 19, 35, 31, 42
Postorder: 10, 19, 14, 31, 42, 35, 27
Levelorder: 27, 14, 35, 10, 19, 31, 42

Tree traversals are rules on how to visit each node of a binary tree exactly once. There are three famous kinds of depth-first traversals and one kind of breadth-first traversal (visiting neighbors first) which will be discussed in more detail later in the lecture.

Depth-first traversals:

- Inorder: left – root – right
- Preorder: root – left – right
- Postorder: left – right – root



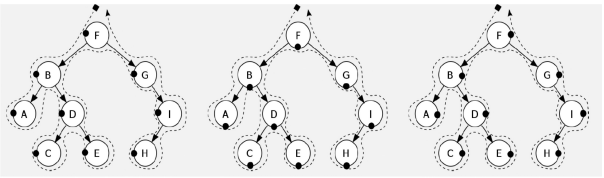
Breadth-first traversal:

- Levelorder: from left to right, from root level to leaf level

Note that the sequence of nodes stemming from a particular traversal method in general does *not* allow to unambiguously reconstruct the original tree.

- An exception is the preorder traversal which allows to reconstruct the original tree in an unambiguous and efficient manner; therefore, the preorder traversal can be used (and considered as) as representational structure for trees, e.g. in order to store a tree in a file.

preorder *inorder* *postorder*



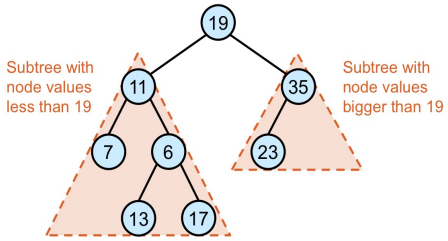
What are these tree traversal strategies useful for?

- Preorder: easy and efficient way to store binary search trees with the possibility to reconstruct them unambigously
- Inorder: retrieving the node values of a binary search tree in ascending order
- Postorder: important for compiler design (in assembly: operand operand opcode)
- Levelorder: used in heapsort algorithm

Binary Search Trees

A binary search tree (BST) is a binary tree where all descendent nodes to the left are smaller or equal than the ancestor node and all descendants to the right are bigger or equal than the ancestor node.

Example:



Remark: The above example assumes that there are no duplicates in the BST (which is a common and oftentimes applicable assumption in use cases where BST make sense as an ADT. If there can be duplicates, this comes with a bunch of problems / potential ambiguities that need to be dealt with.

Binary Search Trees: Degeneration

If keys are inserted in sorted order into a BST, the tree will degenerate to a list.

It is said: the tree is unbalanced.

We will see techniques later which will ensure that this does not happen (red-black trees).

Is they are balanced

(c) Is it true that the time complexity of search in a binary search tree is O(1) in the best case? Identify when this best case is achieved.

True, complexity is O(1) if the node happens to be at the root.

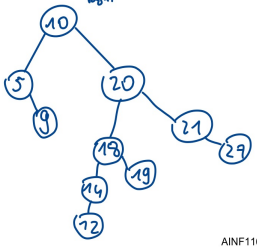
(d) Is it true that the time complexity of search in a binary search tree is O(log n) in the worst case where n is the number of nodes? Identify when this worst case is achieved.

False, complexity will be **O(n)** if the tree is degenerate.

Draw the binary search tree which will have a postorder traversal sequence as follows:

9, 5, 12, 14, 19, 18, 27, 21, 20, 10

Postorder: left-right-root
Inorder: 5 9 10 12 14 18 19 20 21 27 (left-root-right)
left right
Algorithm check: Inorder and post-order?

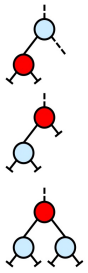


Check:
9 5 12 14 19 18 27 21 20 10 ✓
(By doing postorder on the constructed BST)

Binary Search Trees: Removing a Node

Three different cases are to be discerned:

- **A:** Node to be removed is a **leaf** (external node): just delete the leaf (the appropriate node of the parent is set to null and the node is deallocated)
- **B:** Node to be removed is **not a leaf** (internal node)
  - **B1:** Node to be removed has only **one child**: The node's parent can «adopt» the child; the node itself is then deallocated
  - **B2:** Node to be removed has **two children**: find maximum in left subtree and replace the root of the subtree with this value, than remove the maximum node of the subtree; since the maximum in the subtree can be a leaf node or an internal node of the subtree, therefore the removal has to be done using the techniques from cases A or B1 respectively (note that the maximum/minimum node cannot have two childs). Also note that this can also be done by taking the minimum in the right subtree.

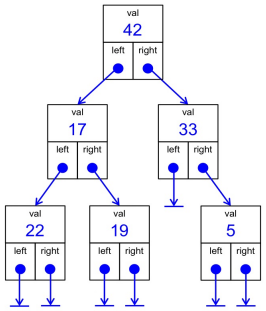


Binary Trees: Implementation

The nodes of a binary tree can be implemented in C using a struct as follows:

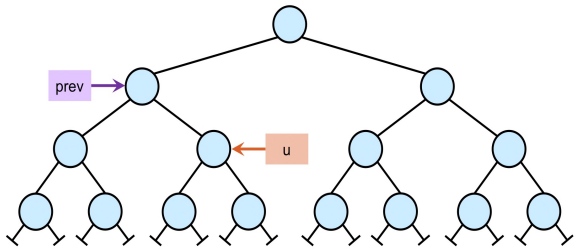
```
struct TreeNode {
        int val;
        struct TreeNode* left;
        struct TreeNode* right;
};
```

Optionally, a parent pointer may be added which allows to get the parent node of a node directly. This has advantages in some situations but can also have disadvantages in others (for example, this makes removal operations more difficult because there are more pointers which have to be changed correctly; also this needs more memory, obviously).



Implementation of deletion operation

Idea: Maintain two pointers **u** and **prev** which point to nodes of the tree such that prev always points to the parent of u (except in the very beginning, when u points to the root and prev to NULL). Advance both these pointers until **u** points to the node which shall be deleted.



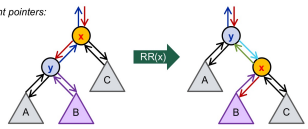
Red-Black Trees

Left rotations and right rotations will cancel out each other (they are inverse operations).

Right Rotation Implementation – With Parent Pointers

Pseudocode for right rotation with parent pointers:

```
rightRotate(T, x) {
        y = x->left;
        x->left = y->right;
        y->parent = x->parent;
        if (y->right != NIL) {
                y->right->parent = x;
        }
        if (x->parent == NIL) {
                T.root = y;
        } else {
                if (x == x->parent->right) {
                        x->parent->right = y;
                } else {
                        x->parent->left = y;
                }
        }
        y->right = x;
        x->parent = y;
}
```



Red-Black Trees: Insertion

- While insertions lead to a violation of the red property (because inserting of the new node as red results in two consecutive red nodes) and has to be restored, the depth property is violated by deletions (because deleting of a black node results in a change of the black height in the respective subtree).
- While for insertions, the color of the uncle is checked to decide which case has to be applied, for deletions the color of the brother / sibling is checked to decide which case has to be applied.

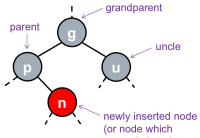
sertion works the same way as inserting into a binary search tree at first (i.e. smaller values go to the left, gger values go the right).

hen a node is inserted, it is always colored red initially.

xt, the RB conditions are checked. If the tree which arose is way violates the RB conditions, one or both of the llowing operations is applied to restore the RB conditions:

- node recolorings
- tree rotations

Note that the new node can have children in general, because it is possible that changes propagate through the tree. Also note that rotations do not change the black height of any node. (After a rotation, all subtrees are exactly the same relative to the root as they were before the rotation.)



Red-Black Trees: Deletions: Initial Preparatory Step

a) If the node which shall be deleted has no (non-nil) children or if it has only one (non-nil) child, i.e. if it has at least one nil child:

→ There is no preparatory work necessary and the process can be continued with step ②, i.e. restoring the RB conditions.

b) If the node which shall be deleted has two (non-nil) children, i.e. if it has no nil-children:

→ Copy: The value stored at the node which shall be deleted is overwritten with a copy of the largest value in its left subtree. Note that only the value is copied and the color and structure of the tree remain unchanged in this stage.

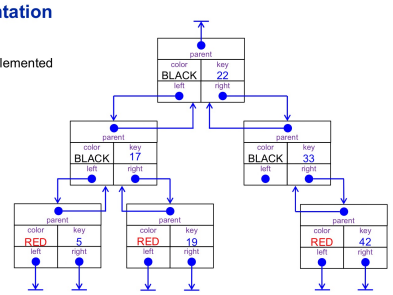
→ Reset the «flag of death»: The node from where this copy has been taken is considered to be the node which shall be deleted from now on and continue with step ②. This node is also called the replacement node.

Note that by applying the above technique, the node which is marked as the node to be deleted always will have either one non-nil child or no non-nil children. Also note that this initial preparatory step applies a similar approach as the removal of a node in a «normal» binary search tree with regard to the decision structure.

Red-Black Trees: Implementation

The nodes of a red-black tree can be implemented in C using a struct as follows:

```
struct rb_node {
        int key;
        int color;
        struct rb_node* left;
        struct rb_node* right;
        struct rb_node* parent;
};
```



Hash Tables

Hashing: Remark on the Modulo Function and Negative Numbers

In order to compute the modulus of a negative number, the following property of the modulo function can be used:

$$a \bmod m = (a + k \cdot m) \bmod m$$

Examples:

- $-3 \bmod 4 = (-3 + 4) \bmod 4 = 1 \bmod 4 = 1$
- $-17 \bmod 7 = (-17 + 3 \cdot 7) \bmod m = 4 \bmod 7 = 4$

This can also be calculated as follows:

$$-a \bmod m = (a \cdot (m - 1)) \bmod m$$

States of Slots, Searching and Deleting in an Hash Table

Each slot of a hash table is assigned one of three different states:

- **OCC**: the slot is occupied
- **EMP**: the slot is empty and always has been empty
- **DEL**: the slot is empty but was previously occupied (the content has been deleted)

These states have to be considered when inserting, searching and deleting in the hash table:

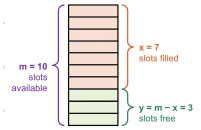
- Inserting: apply hash function; if resulting slot is occupied, apply next iteration of hash function; repeat until empty or deleted slot is reached; when slot found: insert and set state to occupied
- Searching: apply hash function; if resulting slot is occupied and value in the slot is the value we search for: return; if empty: return error; if occupied and not the key or if previously occupied, apply next iteration of hash function; repeat until key is found and has been returned or until empty slot is discovered which yields to returning «not found» message

Hashing: Load Factor (α)

For analysis of hashing and comparing different techniques for collision resolution, an important number is the load factor α. It is defined as the number x of items currently stored in the hash table divided by the number m of slots available in the hash table:

$$\alpha = \frac{x}{m}$$

Since at most element can be in a slot, i.e. $x \leq m$, it must be true that $\alpha \leq 1$ for a table using open addressing.



$\alpha = 7 / 10 = 0.7 = 70\%$

Hashing: Efficiency

Hashing is very efficient / fast.

	unsorted singly linked list, worst-case	sorted doubly linked list, worst-case	min-heap, worst-case	hash table, worst-case	hash table, average-case
SEARCH(L, k)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$
INSERT(L, x)	$O(1)$	$O(n)$	$O(lgn)$	$O(1)$ or $O(n)$	$O(1)$
DELETE(L, x)	$O(n)$	$O(1)$	$O(lgn)$	$O(1)$	$O(1)$