HARDWARE (COMPILERS, OS, RUNTIME) AND C, C++, TO CUDA

ERNEST YEUNG ERNESTYALUMNI@GMAIL.COM

Contents

Date: 1 Nov 2017.
Key words and phrases. C, C++, CUDA, CUDA C/C++, Compilers, OS, runtime, classes, Object Oriented Programming, Segmentation Fault, memory, memory addresses.

Abstract. I review what, how, and why Segmentation Fault is and occurs in the specific context of C and hopefully to CUDA, virtual memory addresses in C++, and to CUDA.

1. Introduction; why I'm writing this

I didn't realize the importance of having a profound understanding of C/C++ and its relation to hardware, in particular memory (addresses), until I learned more specifically about the work done at NVIDIA from its news and job postings. Coming from a theoretical and mathematical physics background, I wanted to get myself up to speed as fast as possible, provide good textbook and online references, and directly relate these topics, which seem to be classic topics in computer science (at hardware level) and electrical engineering, to CUDA, to specifically GPU parallel programming with direct CUDA examples.

Note that there is a version of this LaTeX/pdf file in a "grande" format (not letter format but "landscape" format) that is exactly the same as the content here, but with different size dimensions.

I began looking up resources (books, texts, etc.) Recommend a text that explains the physical implementation of C (stack, heap, etc) [closed], and a Mehta recommended books including Hyde (2006) [4].

> "When programmers first began giving up assembly language in favor of using HLLs, they generally understood the low-level ramifications of the HLL statements they were using and could choose their HLL statements appropriately.Unfortunately, the generation of computer programmers that followed them did not have the benefit of mastering assembly language. As such, they were not in a position to wisely choose statements and data structures that HLLs could efficiently translate into machine code."

pp. 2, Ch. 1 of Hyde (2006) [4], where HLL stands for high-level language. I was part of that generation and knew of nothing of assembly.

Bryant and O'Hallaron (2015) [1]

2. How Computers Store Data

"How Computers Store Data" in *Data Structures: Introduction*, Neagole (2020) [22]

Persistent storage is slow.
CPU needs access to RAM and storage. CPU can access RAM a lot faster.
When a variable is declared, it'll hold value in memory RAM.
CPU grabs program in storage. To run fast, hold variable in RAM.
Think of RAM as massive storage area with numbered shelves - *addresses*. Each address holds 8 bits = 1 byte.
CPU is connected to a memory controller, and memory controller does actual reading of RAM memory, as well as writing to this memory.
When CPU asks what's at this address? memory controller actually has individual connections to each address; this is important, it means we can access 0th address and immediately access 7th address, or Nth address, thus RAM - random access memory - memory controller can jump to any address, but closer information is to CPU, less time to travel.
CPU cache- where CPU stores on tiny memory a copy of recent data; e.g. LRU cache.

Store variable `var a = 1` in "word" ("block") 32-bit, store next varaible in next block.

overflow (TODO: review overflow implementation).
Some data structures are organized in consecutive addresses, some apart.

3. 80x86 Assembly; 80x86 Architecture

cf. Ch. 3 of Hyde (2006) [4]

Main difference between complex instruction set computer (CISC) architectures such as 80x86 and reduced instruction set computer (RISC) architectures like PowerPC is the way they use memory. RISC provde relatively clumsy memory access, and applications avoid accessing memory. 80x86 access memory in many different ways and applications take advantage of these facilities.

3.1. Basic 80x86 Architecture.

Intel CPU generally classified as a *Von Neumann machine*, containing 3 main building blocks:

(1) CPU
(2) memory
(3) input/output (I/O) devices

These 3 components are connected together using the *system bus*. System bus consists of

- address bus
- data bus
- control bus

CPU communicates with memory and I/O devices by placing a numeric value on the address bus to select 1 of the memory locations or I/O device port locations, each of which has a unique binary numeric address.
Then the CPU, I/O, and memory devices pass data among themselves by placing data on the data bus.
Control bus contains signals that determine the direction of data transfer (to or from memory, and to or from an I/O device).

So

$$\textbf{Memory}$$
$$\text{Obj}(\textbf{Memory}) = \text{ memory locations}$$
$$\textbf{I/O}$$
$$\text{Obj}(\textbf{I/O}) = \text{ I/O device port locations}$$

and

$$\textbf{address bus}$$
$$\text{Obj}(\textbf{address bus}) = \text{ unique binary numeric addresses}$$

Likewise,

$$\textbf{data bus}$$
$$\text{Obj}(\textbf{data bus}) = \text{ data}$$

And so

$$\text{CPU} : \textbf{Memory} \times \textbf{address bus} \to \text{Hom}(\textbf{Memory}, \textbf{address bus})$$
$$\text{CPU} : (\text{memory location}, \text{unique binary numeric address}) \mapsto \&$$

in the language of category theory.

3.2. **Registers (for 80x86).** cf. 3.3 Basic 80x86 Architecture, from pp. 23 of Hyde (2006) [4].

Example: to add 2 variables, x, y and store $x + y$ into z, you must load 1 of the variables into a register, add the 2nd. operand to the register, and then store register's value into destination variable. "Registers are middlemen in almost every calculation." Hyde (2006) [4].

There are 4 categories of 80x86 CPU registers:

- general-purpose
- special-purpose application-accessible
- segment
- special-purpose kernel-mode

Segment registers not used very much in modern 32-bit operating systems (e.g. Windows, Linux; what about 64-bit?) and special-purpose kernel-mode registers are intended for writing operating systems, debuggers, and other system-level tools.

3.2.1. *80x86 General-Purpose Registers.* 80x86 CPUs provide several general-purpose registers, including 8 32-bit registers having the following names: (8 bits in 1 byte, 32-bit is 4 bytes)

$$EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP$$

where E is *extended*; 32-bit registers distinguished from 8 16-bit registers with following names

$$AX, BX, CX, DX, SI, DI, BP, SP$$

and finally

$$AL, AH, BL, BH, CL, CH, DL, DH$$

Hyde says that it's important to note about the general-purpose registers is they're not independent. 80x86 doesn't provide 24 separate registers, but overlaps the 32-bit registers with 16-bit registers, and overlaps 16-bit registers with 8-bit registers. e.g.

$$\&AL \neq \&AH, \text{ but}$$
$$\&AL, \&AH \in \&AX \text{and } \&AX \in \&EAX, \text{ so that}$$
$$\&AL, \&AH, \&AX \in \&EAX$$

4. Compilers; Compiler Operation and Code Generation

cf. Ch. 5 Compiler Operation and Code Generation, from pp. 62 and pp. 72- of Hyde (2006) [4]

5. gdp; good debugging processes (in C/C++/CUDA)

Learn C the hard way Lecture 4, Using Debugger (GDB)

6. Pointers in C; Pointers in C categorified (interpreted in Category Theory) and its relation to actual, physical, computer memory and (memory) addresses ((address) bus; pointers, structs, arrays in C

From Shaw (2015) [5], Exercise 15,

e.g. `ages[i]`, You're indexing into array `ages`, and you're using the number that's held in i to do it:

$$a : \mathbb{Z} \to \text{Type} \in \mathbf{Type} \quad \text{e.g.} \quad a : \mathbb{Z} \to \mathbb{R} \text{ or } \mathbb{Z}$$
$$a : i \mapsto a[i] \qquad\qquad a : i \mapsto a[i]$$

Index $i \in \mathbb{Z}$ is a location *inside* `ages` or a, which can also be called *address*. Thus, $a[i]$.

Indeed, from cppreference for Member access operators,

Built-in *subscript* operator provides access to an object pointed-to by pointer or array operand. And so `E1[E2]` is exactly identical to `*(E1+E2)`.

To C, e.g. `ages`, or a, is a location in computer's memory where, e.g., all these integers (of `ages`) start, i.e. where a starts.

$\mathbf{Memory}$, $\quad \text{Obj}(\mathbf{Memory}) \ni$ memory location. Also, to specify CPU,

$\mathbf{Memory}_{CPU}$, $\quad \text{Obj}(\mathbf{Memory}_{CPU}) \ni$ computer memory location

It's *also* an address, and C compiler will replace e.g. `ages` or array a, anywhere you type it, with address of very 1st integer (or 1st element) in, e.g. `ages`, or array a.

$$\mathbf{Arrays} \overset{\cong}{\longleftrightarrow} \mathbf{address}$$
$$\text{Obj}(\mathbf{Arrays}) \overset{\cong}{\longleftrightarrow} \text{Obj}(\mathbf{address})$$
$$a \overset{\cong}{\longleftrightarrow} \texttt{0x17}$$

"But here's the trick": e.g. "`ages` is an address inside the *entire computer*." (Shaw (2015) [5]).

It's not like i that's just an address inside `ages`. `ages` array name is actually an address in the computer.

"This leads to a certain realization: C thinks your whole computer is 1 massive array of bytes."

"What C does is layer on top of this massive array of bytes the concept of types and sizes of those types." (Shaw (2015) [5]).

Let

$\mathbf{Memory}_{CPU} :=$ 1 massive array of bytes

$\text{Obj}(\mathbf{Memory}_{CPU})$

$\mathbf{Type}$

$\text{Obj}(\mathbf{Type}) \ni \texttt{int}, \texttt{char}, \texttt{float}$

$\text{Obj}(\mathbf{Type}) \xrightarrow{\texttt{sizeof}} \mathbb{Z}^+$

$T \xmapsto{\texttt{sizeof}} \texttt{sizeof}(T)$

$\texttt{float} \xmapsto{\texttt{sizeof}} \texttt{sizeof}(\texttt{float})$

How C is doing the following with your arrays:

- *Create* a block of memory inside your computer:

$$\text{Obj}(\mathbf{Memory}_{CPU}) \supset \text{ Memory block}$$

  Let $\text{Obj}(\mathbf{Memory}_{CPU})$ be an ordered set. Clearly, then memory can be indexed. Let $b \in \mathbb{Z}^+$ be this index. Then $\text{Memory block}(0) = \text{Obj}(\mathbf{Memory}_{CPU})(b)$.
- *Pointing* the name `ages`, or a, to beginning of that (memory) block.
  Entertain, possibly, a category of pointers, **Pointers** $\equiv$ **ptrs**.

$$\mathbf{ptrs}$$

$$\text{Obj}(\mathbf{ptrs}) \ni a, \text{ e.g. } \texttt{ages}$$

$$a \mapsto \text{Memory block}(0)$$

$$\text{Obj}(\mathbf{ptrs}) \to \text{Obj}(\mathbf{Memory}_{CPU})$$

- *indexing* into the block, by taking the base address of `ages`, or a

$a \overset{\cong}{\mapsto}$ base address $0x17$

$\text{Obj}((T)\mathbf{array}) \overset{\cong}{\to} \text{Obj}(\mathbf{addresses})$

$a[i] \equiv a + i \overset{\cong}{\mapsto}$ base address + i * `sizeof`$(T) \overset{*}{\mapsto} a[i] \in T$ where T, e.g. $T = \mathbb{Z}, \mathbb{R}$

$\text{Obj}((T)\mathbf{array} \overset{\cong}{\to} \text{Obj}(\mathbf{addresses}) \to T$

"A pointer is simply an address pointing somewhere inside computer's memory with a type specifier." Shaw (2015) [5]

C knows where pointers are pointing, data type they point at, size of those types, and how to get the data for you.

6.0.1. *Practical Pointer Usage.*

- Ask OS for memory block (chunk of memory) and use pointer to work with it. This includes strings and `structs`.
- Pass by reference - pass large memory blocks (like large structs) to functions with a pointer, so you don't have to pass the entire thing to the function.
- Take the address of a function, for dynamic callback. (function of functions)

"You should go with arrays whenever you can, and then only use pointers as performance optimization if you absolutely have to." Shaw (2015) [5]

6.0.2. *Pointers are not Arrays.* No matter what, pointers and arrays are not the same thing, even though C lets you work with them in many of the same ways.

From Eli Bendersky's website, Are pointers and arrays equivalent in C?

He also emphasizes that

6.0.3. *Variable names in C are just labels.* "A variable in C is just a convenient, alphanumeric pseudonym of a memory location." (Bendersky, [6]). What a compiler does is, create label in some memory location, and then access this label instead of always hardcoding the memory value.

"Well, actually the address is not hard-coded in an absolute way because of loading and relocation issues, but for the sake of this discussion we don't have to get into these details." (Bendersky, [6]) (EY : 20171109 so it's on the address bus?)

Compiler assigns label *at compile time.* Thus, the great difference between arrays and pointers in C.

6.0.4. *Arrays passed to functions are converted to pointers.* cf. Bendersky, [6].

Arrays passed into functions are always converted into pointers. The argument declaration `char arr_place[]` in

```
void foo(char arr_arg[], char* ptr_arg)
{
char a = arr_arg[7];
char b = ptr_arg[7];
}
```

is just syntactic sugar to stand for `char* arr_place`.

From Kernighan and Ritchie (1988) [7], pp. 99 of Sec. 5.3, Ch. 5 Pointers and Arrays,

> When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address.

Why?

The C compiler has no choice here, since,
array name is a label the C compiler replaces *at compile time* with the address it represents (which for arrays is the address of the 1st element of the array).

But function isn't called at compile time; it's called *at run time*.
At run time, (where) something should be placed on the stack to be considered as an argument.

Compiler cannot treat array references inside a function as labels and replace them with addresses, because it has no idea what actual array will be passed in at run time.

EY : 20171109 It can't anticipate the exact arguments that'll it be given *at run-time*; at the very least, my guess is, it's given instructions.

Bendersky [6] concludes by saying the difference between arrays and pointers does affect you. One way is how arrays can't be manipulated the way pointers can. Pointer arithmetic isn't allowed for arrays and assignment to an array of a pointer isn't allowed. cf. van der Linden (1994) [8]. Ch. 4, 9, 10.

Bendersky [6] has this one difference example, "actually a common C gotcha":

"Suppose one file contains a global array:"

```
char my_Arr[256]
```

Programmer wants to use it in another file, *mistakingly* declares as

```
extern char* my_arr;
```

When he tries to access some element of the array using this pointer, he'll most likely get a segmentation fault or a fatal exception (nomenclature is OS-dependent).

To understand why, Bendersky [6] gave this hint: look at the assembly listing

```
char a = array_place[7];

0041137E  mov  al,byte ptr [_array_place+7 (417007h)]
00411383  mov  byte ptr [a],al

char b = ptr_place[7];

00411386  mov  eax,dword ptr [_ptr_place (417064h)]
0041138B  mov  cl,byte ptr [eax+7]
0041138E  mov  byte ptr [b],cl
```

or my own, generated from `gdb` on Fedora 25 Workstation Linux:

```
0x00000000004004b1 <main+11>:    movzbl 0x200b8f(%rip),%eax        # 0x601047 <array_place+7>
0x00000000004004b8 <main+18>:    mov    %al,-0x1(%rbp)

0x00000000004004bb <main+21>:    mov    0x200be6(%rip),%rax        # 0x6010a8 <ptr_place>
0x00000000004004c2 <main+28>:    movzbl 0x7(%rax),%eax
0x00000000004004c6 <main+32>:    mov    %al,-0x2(%rbp)
```

"How will the element be accessed via the pointer? What's going to happen if it's not actually a pointer but an array?" Bendersky [6]

EY : 20171106. Instruction-level, the pointer has to

- `mov 0x200be6(%rip),%rax` - 1st., copy value of the pointer (which holds an address), into `%rax` register.
- `movzbl 0x7(%rax),%eax` - off that address in register ‘
- `mov %al,-0x2(%rbp)` - `mov` contents `-0x2(%rbp)`into register `%al`

If it's not actually a pointer, but an array, the value is copied into the `%rax` register is an actual `char` (or `float`, some type). *Not* an address that the registers may have been expecting!

6.1. **Structs in C.** From Shaw (2015) [5], Exercise 16,

`struct` in C is a collection of other data types (variables) that are stored in 1 block of memory. You can access each variable independently by name.

- The `struct` you make, i.e.g `struct Person` is now a *compound data type*, meaning you can refer to `struct Person` using the same kinds of expressions you would for other (data) types.
- This lets you pass the whole `struct` to other functions
- You can access individual members of `struct` by their names using `x->y` if dealing with a ptr.

If you didn't have `struct`, you'd have to figure out the size, packing, and location of memory of the contents. In C, you'd let it handle the memory structure and structuring of these compound data types, `struct`s. (Shaw (2015) [5])

Part 1. C, Stack, Heap Memory Management in C

7. C, Stack and Heap Memory Management, Heap and Stack Memory Allocation

cf. Ex. 17 of Shaw (2015) [5]

Consider chunk of RAM called stack, another chunk of RAM called heap. Difference between heap and stack depends on where you get the storage.

Heap is all the remaining memory on computer. Access it with `malloc` to get more.

Each time you call `malloc`, the OS uses internal functions (EY : 20171110 address bus or overall system bus?) to register that piece of memory to you, then returns ptr to it.

When done, use `free` to return it to OS so OS can use it for other programs. Failing to do so will cause program to *leak* memory. (EY: 20171110, meaning this memory is unavailable to the OS?)

Stack, on a special region of memory, stores temporary variables, which each function creates as locals to that function. How stack works is that each argument to function is *pushed* onto stack and then used inside the function. Stack is really

a stack data structure, LIFO (last in, first out). This also happens with all local variables in `main`, such as `char action`, `int id`. The advantage of using stack is when function exits, *C compiler* pops these variables off of stack to clean up.

Shaw's mantra: If you didn't get it from `malloc`, or a function that got it from `malloc`, then it's on the stack.

3 primary problems with stacks and heaps:

- If you get a memory block from `malloc`, and have that ptr on the stack, then when function exits, ptr will get popped off and lost.
- If you put too much data on the stack (like large structs and arrays), then you can cause a *stack overflow* and program will abort. Use the heap with `malloc`.
- If you take a ptr, to something on stack, and then pass or return it from your function, then the function receiving it will *segmentation fault*, because actual data will get popped off and disappear. You'll be pointing at dead space.

cf. Ex. 17 of Shaw (2015) [5]

8. Data segments; Towards Segmentation Fault

cf. Ferres (2010) [9]

When program is loaded into memory, it's organized into 3 *segments* (3 areas of memory): let executable program generated by a compiler (e.g. `gcc`) be organized in memory over a range of addresses (EY : 20171111 assigned to physical RAM memory by address bus?), ordered, from low address to high address.

- *text* segment or code segment - where compiled code of program resides (from lowest address); code segment contains code executable or, i.e. code binary.

  As a memory region, text segment may be placed below heap, or stack, in order to prevent heaps and stack overflows from overwriting it.

  Usually, text segment is sharable so only a single copy needs to be in memory for frequently executed programs, such as text editors, C compiler, shells, etc. Also, text segment is often read-only, to prevent program from accidentally modifying its instructions. cf. Memory Layout of C Programs; GeeksforGeeks
- Data segment: data segment subdivided into 2 parts:
  - initialized data segment - all global, static, constant data stored in data segment. Ferres (2010) [9]. Data segment is a portion of virtual address of a program.
    Note that, data segment not read-only, since values of variables can be altered at run time.
    This segment can also be further classified into initialized read-only area and initialized read-write area.
    e.g. `char s[] = "hello world"` and `int debut = 1` *outside the main (i.e. global)* stored in initialized read-write area.
    `const char *string = "hello world"` in global C statement makes string literal "hello world" stored in initialized read-only area. Character pointer variable string in initialized read-write area. cf. Memory Layout of C Programs

  - uninitialized data stored in BSS. Data in this segment is initialized by kernel (OS?) to arithmetic 0 before program starts executing.
    Uninitialized data starts at end of data segment ("largest" address for data segment) and contains all global and static variables initialized to 0 or don't have explicit initialization in source code.
    e.g. `static int i;` in BSS segment.
    e.g. `int j;` global variable in BSS segment.
    cf. Memory Layout of C Programs
- Heap - "grows upward" (in (larger) address value, begins at end of BSS segment), allocated with `calloc, malloc`, "dynamic memory allocation".
  Heap area shared by all shared libraries and dynamically loaded modules in a process.
  Heap grows when memory allocator invokes `brk()` or `sbrk()` system call, mapping more pages of physical memory into process's virtual address space.
- Stack - store local variables, used for passing arguments to functions along with return address of the instruction which is to be executed after function call is over.
  When a new stack frame needs to be added (resulting from a *newly called function*), stack "grows downward." (Ferres (2010) [9])
  Stack grows automatically when accessed, up to size set by kernel (OS?) (which can be adjusted with setrlimit(`RLIMIT_STACK`,...)).

8.0.1. *Mathematical description of Program Memory (data segments), with **Memory**, **Addresses**.* Let **Address**, with Obj(**Address**) $\cong \mathbb{Z}^+$ be an ordered set.
Memory block $\subset$ Obj(**Memory**), s.t.

$$\text{Memory block} \xrightarrow{\cong} \{ \text{ low address , low address+}\texttt{sizeof}(T), \dots \text{ high address } \} \equiv \text{addresses}_{\text{Memory block}} \subset \mathbb{Z}^+$$

where $T \in$ Obj(**Types**) and $\cong$ assigned by address bus, or the virtual memory table, and $\text{addresses}_{\text{Memory block}} \subset$ Obj(**Addresses**).
Now,
text segment, (initialized) data segment, (uninitialized) data segment, heap, stack, command-line arguments and environmental variables $\subset \text{addresses}_{\text{Memory block}}$, that these so-called data segments are discrete subsets of the set of all addresses assigned for the memory block assigned for the program.
Now, $\forall\, i \in$ text segment , $\forall\, j \in$ (initialized) data segment , $i < j$ and $\forall\, j \in$ (initialized) data segment , $\forall\, k \in$ (uninitiaized) data segment , $j < k$, and so on. Let's describe this with the following notation:

$$\begin{aligned} &\text{text segment} < \text{(initialized) data segment} < \text{(uninitialized) data segment} < \\ &< \text{heap} < \text{stack} < \text{ command-line arguments and environmental variables} \end{aligned} \tag{1}$$

Consider stack of variable length $n_{\text{stack}} \in \mathbb{Z}^+$. Index the stack by $i_{\text{stack}} = 0, 1, \dots n_{\text{stack}} - 1$. "Top of the stack" is towards "decreasing" or "low" (memory) address, so that the relation between "top of stack" to beginning of the stack and high address to low address is *reversed*:

$$i_{\text{stack}} \mapsto \text{high addreess} - i_{\text{stack}}$$

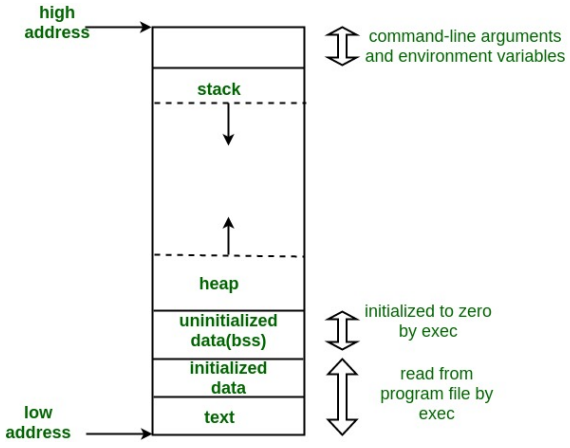


FIGURE 1. cf. https://www.geeksforgeeks.org/memory-layout-of-c-program/

Call stack is composed of stack frames (i.e. "activation records"), with each stack frame corresponding to a subroutine call that's not yet termined with a routine (at any time).

The *frame pointer* `FP` points to location where stack pointer was.

Stack pointer usually is a register that contains the "top of the stack", i.e. stack's "low address" currently, Understanding the stack, i.e.

$$\text{eval}(RSP) = \text{high address} - i_{\text{stack}} \tag{2}$$

8.0.2. *Mathematical description of strategy for stack buffer overflow exploitation.* Let $n_{\text{stack}} = n_{\text{stack}}(t)$. Index the stack with i_{stack} (from "bottom" of the stack to the "top" of the stack):

$$0 < i_{\text{stack}} < n_{\text{stack}} - 1$$

Recall that $i_{\text{stack}} \in \mathbb{Z}^+$ and

$$i_{\text{stack}} \mapsto \text{high address} - i_{\text{stack}} \equiv x = x(i_{\text{stack}}) \in \text{Addresses}_{\text{Memory block}} \subset \text{Obj}(\textbf{Address})$$

Let an array of length L (e.g. `char` array) `buf`, with `\&buf` = `\&buf(0)` $\in$ Obj(**Address**), be s.t. `&buf` $= x(n_{\text{stack}} - 1)$ (starts at "top of the stack and "lowest" address of stack at time t, s.t.

$$\texttt{\&buf}(j) = \texttt{\&buf}(0) + j\texttt{sizeof}(T)$$

with $T \in$ **Types**).

Suppose return address of a function (such as `main`), eval(`RIP`) be

$$\text{eval}(\texttt{RIP}) = \texttt{\&buf} + L \text{or at least } \text{eval}(\texttt{RIP}) \geq \texttt{\&buf} + L$$

If we write to `buf` more values than L, we can write over eval(`RIP`), making eval(`RIP`) a different value than before.

8.1. **Stack.** cf. https://www.geeksforgeeks.org/memory-layout-of-c-program/

Traditionally, stack area adjoined heap area and they grow in opposite direction. When stack ptr meets heap ptr, free memory exhausted.
(with modern large address spaces, virtual memory techniques, they may be placed almost anywhere, but they still typically grow opposite direction)

stack area contains program stack, UFO structure, typically located in higher parts of memory.
On standard x86, grows toward address 0.
"stack ptr" register tracks top of stack, adjusted to point to top each time a value is "pushed" onto stack.
set of values pushed for 1 function call is "stack frame", stack frame consists at minimum of return address.

Newly called function allocates room on stack for its automatic and temporary variables.
$\implies$ recursion: each time recursive function calls itself, new stack frame is used, so 1 set of variables (in recursion) doesn't interfere with variables from another instance of the function.

cf. Where are static variables in C/C++? Tutorialspoint

static variables remain in memory; lifetime is entire program, stored in data segment of memory (near heap); data segment is part of virtual address space of a program.

asked here https://www.codingame.com/work/cpp-interview-questions/, what is a static variable

cf. Ferres (2010) [9]

Stack and functions: When a function executes, it may add some of its state data to top of the stack (EY : 20171111, stack grows downward, so "top" is smallest address?); when function exits, stack is responsible for removing that data from stack.

In most modern computer systems, each thread has a reserved region of memory, stack. Thread's stack is used to store location of function calls in order to allow return statements to return to the correct location.

- OS allocates stack for each system-level thread when thread is created.
- Stack is attached to thread, so when thread exits, that stack is reclaimed, vs. heap typically allocated by application at runtime, and is reclaimed when application exits.
- When thread is created, stack size is set.
- Each byte in stack tends to be reused frequently, meaning it tends to be mapped to the processor's cache, making if very fast.
- Stored in computer RAM, *just like* the heap.
- Implemented with an actual stack data structure.
- stores local data, return addresses, used for parameter passing
- Stack overflow, when too much stack is used (mostly from infinite (or too much) recursion, and very large allocation)
- Data created on stack can be used without pointers.

Also note, for **physical location in memory**, because of Virtual Memory, makes your program think that you have access to certain addresses where physical

data is somewhere else (even on hard disc!). Addresses you get for stack are in increasing order as your call tree gets deeper.

memory management - What and where are the stack and heap? Stack Overflow, Tom Leys' answer

8.2. **Stack overflow.** If you use heap memory, and you overstep the bounds of your allocated block, you have a decent chance of triggering a segmentation fault (not 100

On stack, since variables created on stack are always contiguous with each other; writing out of bounds can change the value of another variable. e.g. buffer overflow.

8.3. **Heap.** Heap contains a linked list of used and free blocks. New allocations on the heap (by `new` or `malloc`) are satisfied by creating suitable blocks from free blocks.

This requires updating list of blocks on the heap. This meta information about the blocks on the heap is stored *on the heap* often in a small area in front of every block.

- Heap size set on application startup, but can grow as space is needed (allocator requests more memory from OS)
- heap, stored in computer RAM, like stack.

8.3.1. *Memory leaks.* Memory leaks occurs when computer program consumes memory, but memory isn't released back to operating system.

"Typically, a memory leak occurs because dynamically allocated memory becomes unreachable." (Ferres (2010) [9]).

Programs `./Cmemory/heapstack/Memleak.c` deliberately leaks memory by losing the pointer to allocated memory.

Note, generally, the OS delays real memory allocation until something is written into it, so program ends when virtual addresses run out of bounds (per process limits).

8.4. **Segmentation Fault.** cf. `https://en.wikipedia.org/wiki/Segmentation_fault` Failure condition (i.e. fault) that software attempted to access restricted area of memory (memory access violation).

"Segmentation" in this context refers to address space of a *program*.

8.5. **More Segmentation Faults.** The operating system (OS) is running the program (its instructions). Only from the hardware, with memory protection, with the OS be signaled to a memory access violation, such as writing to read-only memory or writing outside of allotted-to-the-program memory, i.e. data segments. On `x86_64` computers, this general protection fault is initiated by protection mechanisms from the hardware (processor). From there, OS can signal the fault to the (running) process, and stop it (abnormal termination) and sometimes core dump.

For virtual memory, the memory addresses are mapped by program called *virtual addresses* into *physical addresses* and the OS manages virtual addresses space, hardcare in the CPU called memory management unit (*MMU*) translates virtual addresses to physical addresses, and kernel manages memory hierarchy (eliminating possible overlays). In this case, it's the *hardware* that detects an attempt to refer to a non-existent segment, or location outside the bounds of a segment, or to refer to location not allowed by permissions for that segment (e.g. write on read-only memory).

8.5.1. *Dereferencing a ptr to a NULL ptr (in C) at OS, hardware level.* The problem, whether it's for dereferencing a pointer that is a null pointer, or uninitialized pointer, appears to (see the `./Cmemory/` subfolder) be at this instruction at the register level:

```
x000000000040056c <+38>:        movzbl (%rax),%eax

// or

0x00000000004004be <+24>:        movss  %xmm0,(%rax)
```

involving the register RAX, a temporary register and to return a value, upon assignment. And in either case, register RAX has trying to access virtual (memory) address `0x0` (to find this out in `gdb`, do `i r` or `info register`).

Modern OS's run user-level code in a mode, such as *protected mode*, that uses "paging" (using secondary memory source than main memory) to convert virtual addresses into physical addresses.

For each process (thread?), the OS keeps a *page table* dictating how addresses are mapped. Page table is stored in memory (and protected, so user-level code can't modify it). For every memory access, given (memory) address, CPU translates address according to the page table.

When address translation fails, as in the case that *not all addresses are valid*, and so if a memory access generates an invalid address, the processor (hardware!) raises a *page fault exception*. "This triggers a transition from *user mode* (aka *current privilege level (CPL) 3* on x86/x86-64) into *kernel mode* (aka CPL 0) to a specific location in the kernel's code, as defined by the *interrupt descriptor table* (IDT)." cf. [What happens in OS when we dereference a NULL pointer in C?](https://stackoverflow.com/questions/12645647/what-happens-in-os-when-we-dereference-a-null-pointer-in-c)

Kernel regains control and send signal (EY : 20171115 to the OS, I believe).

In modern OS's, page tables are usually set up to make the address 0 an invalid virtual address.

cf. [What happens in OS when we dereference a NULL pointer in C?](https://stackoverflow.com/questions/126 happens-in-os-when-we-dereference-a-null-pointer-in-c)

Part 2. C++

9. Free Store

GotW #9, Memory Management - Part I

cf. 11.2 Introduction of Ch. 11 Select Operations [10].

10. Copy vs. Move

cf. 17.1 Introduction of Ch. 17 Construction, Cleanup, Copy, and Move of Stroustrup [10].

Gottschling (2015) [11]

10.1. **Constructors and Invariants.** cf. 17.2.1 Constructors and Invariants, pp. 484, Ch. 17, Stroustrup [10].

class invariant - something that must hold whenever a member function is called (from outside the class). Typically, class *constructor*'s job is to initialize an object of its class and that initialization must establish the *class invariant*.

For example

```
Vector::Vector(int s)
{
if (s<0) throw Bad_size{s};
sz = s;
elem = new double[s];
}
```

Constructor must make true that an array size must be non-negative.

Why would you define an invariant?

- To focus the design effort for the class
- To clarify behavior of the class (e.g., under error conditions)
- To simplify definition of member functions
- To clarify class's management of resources
- To simplify documentation of the class

10.2. **Base and Member Destructors.** cf. 17.2.3 Base and Member Destructors, pp. 486, Stroustrup [10].

Ctors and dtors interact correctly with class hierarchies. A constructor builds a class object "from the bottom up":

(1) first, ctor invokes its base class constructors
(2) then, it invokes the member constructors, and
(3) finally, it executes its own body

A dtor "tears down" an object in reverse order:

(1) first, dtor executes its own body,
(2) then, it invokes its member dtors, and
(3) finally, it invokes its base class dtors

In particular, a `virtual` base is constructed before any base that might use it and destroyed after all such bases - this ordering ensures that a base or member isn't used before it has been initialized or used after it has been destroyed.

10.3. **virtual Destructors.** A dtor can be declared to be `virtual`, and usually should be for a class with a virtual function. The reason we need a `virtual` dtor is that an object usually manipulated through the interface provided by a base class is often also `delete`'d through that interface. It's because a derived class would have its own dtor, and without `virtual`, `delete` would fail to invoke appropriate derived class dtor.

```
class Shape
{
public:
virtual void draw() = 0;
virtual ~Shape();
};

class Circle
{
public:
void draw();
~Circle(); // overrides ~Shape()
};

void user(Shape* p)
{
p->draw(); // invoke appropriate draw()
```

```
delete p; // invoke appropriate dtor
}
```

Difference between *move* and *copy*: after a copy, 2 objects must have same value; whereas after a move, the source of the move isn't required to have its original value. So moves can be used when source object won't be used again.

Refs.: Sec. 3.2.1.2, Sec. 5.2, notion of moving a resource, Sec. 13.2-Sec.13.3, object lifetime and errors explored further in Stroustrup [10]

5 situations in which an object is copied or moved:

- as source of an *assignment*
- as object initializer
- as function argument
- as function return value
- as an exception

10.4. **Copy constructor.** cf. Copy constructors, cppreference.com

Copy constructor of class T is non-template constructor whose 1st parameter is `T&, const T&, volatile T&`, or `const volatile T&`.

```
class_name ( const class_name & )
class_name ( const class_name & ) = default;
class_name ( const class_name & ) = delete;
```

10.4.2. *Explanation.*

(1) Typical declaration of a copy constructor.
(2) Forcing copy constructor to be generated by the compiler.
(3) Avoiding implicit generation of copy constructor.

Copy constructor called whenever an object is **initialized** (by **direct-initialization** or **copy-initialization**) from another object of same type (unless **overload resolution** selects better match or call is **elided** (???)), which includes

- initialization `T a = b;` or `T a(b);`, where b is of type T;
- function argument passing: `f(a);`, where a is of type T and f is `void f(T t)`;
- function return: `return a;` inside function such as `T f()`, where a is of type T, which has no **move constructor**.

```
struct A
{
int n;
A(int n = 1) : n(n) { }
A(const A& a) : n(a.n) { } // user-defined copy ctor
};

struct B : A
{
// implicit default ctor B::B()
// implicit copy ctor B::B(const B&)
};

int main()
{
A a1(7);
A a2(a1); // calls the copy ctor
B b;
B b2 = b;
```

```
    A a3 = b; // conversion to A& and copy ctor
    }
```

i.e. cf. Copy Constructor in C++

Definition 1. ***Copy constructor*** *is a member function which initializes an object using another object of the same class.*

10.4.4. *When is copy constructor called?*

(1) When object of class returned by value
(2) When object of class is passed (to a function) by value as an **argument**.
(3) When object is constructed based on another object of same class (or overloaded)
(4) When compiler generates temporary object

However, it's not guaranteed copy constructor will be called in all cases, because C++ standard allows compiler to optimize the copy away in certain cases.

10.4.5. *When is used defined copy constructor needed? shallow copy, deep copy.* If we don't define our own copy constructor, C++ compiler creates default copy constructor which does member-wise copy between objects.

We need to define our own copy constructor only if an object has pointers or any run-time allocation of resource like file handle, network connection, etc.

10.4.6. *Default constructor does only shallow copy.*

10.4.7. *Deep copy is possible only with user-defined copy constructor.* We thus make sure pointers (or references) of copied object point to new memory locations.

```
MyClass t1, t2;
MyClass t3 = t1;          // ---------> (1)
t2 = t1;                             // ---------> (2)
```

Copy constructor called when new object created from an existing object, as copy of existing object, in (1). Assignment operator called when already initialized object is assigned a new value from another existing object, as assignment operator is called in (2).

10.4.9. *Why argument to a copy constructor should be const?* cf. Why copy constructor argument should be const in C++?, geeksforgeeks.org

(1) Use `const` in C++ whenever possible so objects aren't accidentally modified.
(2) e.g.

```
#include <iostream>

class Test
{
/* Class data members */
public:
Test(Test &t)   { /* Copy data members from t */ }
Test()                  { /* Initialize data members */ }
};
```

```
Test fun()
{
Test t;
return t;
};

int main()
{
Test t1;
Test t2 = fun();  error: invalid initialization of non-const reference of type Test& from an rvalue of
}
```

`fun()` returns by value, so compiler creates temporary object which is copied to t2 using copy constructor (because this temporary object is passed as argument to copy constructor since compiler generates temp. object). Compiler error is because **compiler-created temporary objects cannot be bound to non-const references**.

10.5. **Move Constructor.** For a class, to control what happens when we move, or move and assign object of this class type, use special member function *move constructor*, *move-assignment operator*, and define these operations. Move constructor and move-assignment operator take a (usually nonconst) rvalue reference, to its type. Typically, move constructor moves data from its parameter into the newly created object. After move, it must be safe to run the destructor on the given argument. cf. Ch. 13 of Lippman, Lajole, and Moo (2012) [13]

11. vtable; virtual table

I was given this answer to a question I posed to a 20 year C++ veteran and it was such an important answer (as I did not know a virtual table existed, at all before), that I will copy this, repeat this and explore this extensively:

"The keyword you're looking for is virtual table: " How are virtual functions and vtable implemented?, stackoverflow

Original question, from Brian R. Bondy:

11.1. **How are virtual functions and vtable implemented?** We all know what virtual functions are in C++, but how are they implemented at a deep level?

Can the vtable be modified or even directly accessed at runtime?

Does the vtable exist for all classes, or only those that have at least one virtual function?

Do abstract classes simply have a NULL for the function pointer of at least one entry?

Does having a single virtual function slow down the whole class? Or only the call to the function that is virtual? And does the speed get affected if the virtual function is actually overwritten or not, or does this have no effect so long as it is virtual.

Answer from *community wiki*:

11.1.1. *How are virtual functions implemented at a deep level?* From "Virtual Functions in C++"

Whenever a program has a virtual function declared, a v-table is constructed for the class. The v-table consists of addresses to the virtual functions for classes that contain one or more virtual functions. The object of the class containing the

virtual function contains a virtual pointer that points to the base address of the virtual table in memory.

Whenever there is a virtual function call, the v-table is used to resolve to the function address.

An object of the class that contains one or more virtual functions contains a virtual pointer called the vptr at the very beginning of the object in the memory. Hence the size of the object in this case increases by the size of the pointer. This vptr contains the base address of the virtual table in memory.

Note that virtual tables are class specific, i.e., there is only one virtual table for a class irrespective of the number of virtual functions it contains. This virtual table in turn contains the base addresses of one or more virtual functions of the class. At the time when a virtual function is called on an object, the vptr of that object provides the base address of the virtual table for that class in memory. This table is used to resolve the function call as it contains the addresses of all the virtual functions of that class. This is how dynamic binding is resolved during a virtual function call.

cf. "Virtual Functions in C++"

11.1.2. *What is a Virtual Function?* A virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes. It is one that is declared as virtual in the base class using the virtual keyword. The virtual nature is inherited in the subsequent derived classes and the virtual keyword need not be re-stated there. The whole function body can be replaced with a new set of implementation in the derived class.

11.1.3. *What is Binding?* Binding is associating an object or a class with its member. If we call a method `fn()` on an object `o` of a class `c`, we say that object `o` is binded with method `fn()`.

This happens at *compile time* and is known as *static* - or *compile-time* binding. Calls to virtual member functions are resolved during *run-time*. This mechanisms is known as *dynamic-binding.*

The most prominent reason why a virtual function will be used is to have a different functionality in the derived class. The difference between a non-virtual member function and a virtual member function is, the non-virtual member functions are resolved at compile time.

11.1.4. *How does a Virtual Function work?* When a program (code text?) has a virtual function declared, a **v-table** is *constructed* for the class.

The v-table consists of addresses to virtual functions for classes that contain 1 or more virtual functions.
The object of the class containing the virtual function *contains a virtual pointer* that points to the base address of the virtual table in memory. An object of the class that contains 1 or more virtual functions contains a virtual pointer called the **vptr** at the very beginning of the object in the memory. (Hence size of the object in this case increases by the size of the pointer; "memory/size overhead.")

This vptr is added as a hidden member of this object. As such, compiler must generate "hidden" code in the **constructors** of each class to initialize a new object's vptr to the address of its class's vtable.

Whenever there's a virtual function call, vtable is used to resolve to the function address. This vptr contains base address of the virtual table in memory.

Note that virtual tables are class specific, i.e. there's only 1 virtual table for a class, irrespective of number of virtual functions it contains, i.e.

vtable is same for all objects belonging to the same class, and typically is shared between them.

This virtual table in turn contains base addresses of 1 or more virtual functions of the class.

At the time when a virtual function is called on an object, the vptr of that object provides the base address of the virtual table for that class in memory. This table is used to resolve the function call as it contains the addresses of all the virtual functions of that class. This is how dynamic binding is resolved during a virtual function call, i.e.

class (inherited or base/parent) cannot, generally, be determined *statically* (i.e. **compile-time**), so compiler can't decide which function to call at that (compile) time. (Virtual function) call must be dispatched to the right function *dynamically* (i.e. **run-time**).

11.1.5. *Virtual Constructors and Destructors.* A constructor cannot be virtual because at the time when constructor is invoked, the vtable wouldn't be available in memory. Hence, we can't have a virtual constructor.

A virtual destructor is 1 that's declared as virtual in the base class, and is used to ensure that destructors are called in the proper order. Remember that destructors are called in reverse order of inheritance. If a base class pointer points to a derived class object, and we some time later use the delete operator to delete the object, then the derived class destructor is not called.

Finally, the article "Virtual Functions in C++" concludes, saying, "Virtual methods should be used judiciously as they are slow due to the overhead involved in searching the virtual table. They also increase the size of an object of a class by the size of a pointer. The size of a pointer depends on the size of an integer." I will have to check this with other references, because, first of all, how then would class inheritance be otherwise implemented?

cf. How are virtual functions and vtable implemented?, stackoverflow

11.1.6. *Can the vtable be modified or even directly accessed at runtime? No.* "Universally, I believe the answer is "no". You could do some memory mangling to find the vtable but you still wouldn't know what the function signature looks like to call it. Anything that you would want to achieve with this ability (that the language supports) should be possible without access to the vtable directly or modifying it at runtime. Also note, the C++ language spec does not specify that vtables are required - however that is how most compilers implement virtual functions."

11.1.7. *Do abstract classes simply have a NULL for the function pointer of at least one entry? Some do place NULL pointer in vtable, some place pointer to dummy method; in general, undefined behavior.* The answer is it is unspecified by the language spec so it depends on the implementation. Calling the pure virtual function results in undefined behavior if it is not defined (which it usually isn't) (ISO/IEC 14882:2003 10.4-2). In practice it does allocate a slot in the vtable for the function but does not assign an address to it. This leaves the vtable incomplete which

requires the derived classes to implement the function and complete the vtable. Some implementations do simply place a NULL pointer in the vtable entry; other implementations place a pointer to a dummy method that does something similar to an assertion.

Note that an abstract class can define an implementation for a pure virtual function, but that function can only be called with a qualified-id syntax (ie., fully specifying the class in the method name, similar to calling a base class method from a derived class). This is done to provide an easy to use default implementation, while still requiring that a derived class provide an override.

11.1.8. *Does having a single virtual function slow down the whole class or only the call to the function that is virtual? No, but space overhead is there.* "This is getting to the edge of my knowledge, so someone please help me out here if I'm wrong!

I believe that only the functions that are virtual in the class experience the time performance hit related to calling a virtual function vs. a non-virtual function. The space overhead for the class is there either way. Note that if there is a vtable, there is only 1 per class, not one per object."

11.1.9. *Does the speed get affected if the virtual function is actually overridden or not, or does this have no effect so long as it is virtual? No, but space overhead is there.* "I don't believe the execution time of a virtual function that is overridden decreases compared to calling the base virtual function. However, there is an additional space overhead for the class associated with defining another vtable for the derived class vs the base class."

11.2. **pImpl, shallow copy, deep copy.** cf. Item 22: "When using the Pimpl Idiom, define special member functions in the implementation file," pp. 147 of Meyers (2014) [14].

```
class Widget {                    // still in header "widget.h"
public:
Widget();
~Widget();                 // dtor is needed—see below
...

private:
struct Impl;     // declare implementation struct
Impl *pImpl;     // and pointer to it
};
```

Because `Widget` no longer mentions types `std::string, std::vector`, and `Gadget, Widget` clients no longer need to `\#include` headers for these types. That speeds compilation.

incomplete type is a type that has been declared, but not defined, e.g. `Widget::Impl`. There are very few things you can do with an incomplete type, but declaring a pointer to it is 1 of them.

`std::unique_ptr`s is advertised as supporting incomplete types. But, when `Widget w;`, `w`, is destroyed (e.g. goes out of scope), destructor is called and if in class definition using `std::unique_ptr`, we didn't declare destructor, compiler generates destructor, and so compiler inserts code to call destructor for `Widget`'s data member `m_Impl` (or `pImpl`).

`m_Impl` (or `pImpl`) is a `std::unique_ptr<Widget::Impl>`, i.e., a `std::unique_ptr` using default deleter. The default deleter is a function that uses `delete` on raw

pointer inside the `std::unique_ptr`. Prior to using `delete`, however, implementations typically have default deleter employ C++11's `static_assert` to ensure that raw pointer doesn't point to an incomplete type. When compiler generates code for the destruction of the `Widget w`, then, it generally encounters a `static_assert` that fails, and that's usually what leads to the error message.

To fix the problem, you need to make sure that at point where code to destroy `std::unique_ptr<Widget::Impl>` is generated, `Widget::Impl` is a complete type. The type becomes complete when its definition has been seen, and `Widget::Impl` is defined inside `widget.cpp`. For successful compilation, have compiler see body of `Widget`'s destructor (i.e. place where compiler will generate code to destroy the `std::unique_ptr` data member) only inside `widget.cpp` after `Widget::Impl` has been defined.

For compiler-generated move assignment operator, move assignment operator needs to destroy object pointed to by `m_Impl` (or `pImpl`) before reassigning it, but in the `Widget` header file, `m_Impl` (or `pImpl`) points to an incomplete type. Situation is different for move constructor. Problem there is that compilers typically generate code to destroy `pImpl` in the event that an exception arises inside the move constructor, and destroying `pImpl` requires `Impl` be complete.

Because problem is same as before, so is the fix - *move definition of move operations into the implementation file.*

For copying data members, support copy operations by writing these functions ourselves, because (1) compilers won't generate copy operations for classes with move-only types like `std::unique_ptr` and (2) even if they did, generated functions would copy only the `std::unique_ptr` (i.e. perform a *shallow copy*), and we want to copy what the pointer points to (i.e., perform a *deep copy*).

If we use `std::shared_ptr`, there'd be no need to declare destructor in `Widget`.

Difference stems from differing ways smart pointers support custom deleters. For `std::unique_ptr`, type of deleter is part of type of smart pointer, and this makes it possible for compilers to generate smaller runtime data structures and faster runtime code. A consequence of this greater efficiency is that pointed-to types must be complete when compiler-generated special functions (e.g. destructors or move operations) are used. For `std::shared_ptr`, type of deleter is not part of the type of smart pointer. This necessitates larger runtime data structures and somewhat slower code, but pointed-to types need not be complete when compiler-generated special functions are employed.

12. Class Hierarchies

12.1. **Inheritance.** Ctors of inherited classes are called in same order in which they're inherited. Dtors are called in reverse order of ctors.

12.1.1. *Diamond Problem.* cf. Geeks for Geeks, "Multiple Inheritance in C++".

Diamond problem occurs when 2 superclasses of a class have a common base class. The class gets 2 copies of all attributes of common base class, causing ambiguities.

13. Polymorphism

Polymorphism, TODO

cf. Vandevoorde, Josuttis, and Gregor (2017) [25], Ch. 22, pp. 517

Ch. 18 had described static polymorphism (templates), and dynamic polymorphism (inheritance, virtual functions). Tradeoffs: static polymorphism provides same performance as nonpolymorphic code, but set of types that can be used at run time is fixed at compile time. Dynamic polymorphism via inheritance allows a single version of the polymorphic function to work with types not known at time it's compiled, but it's less flexible because types must inherit from a common base class.

Ch. 22 - how to have smaller executable code size and (almost) entirely compiled nature of dynamic polymorphism, along with interface flexibility of static polymorphism that allows, for example, built-in types to work seemlessly.

Gutman (2016) [28]

13.1. **Operator Overloading.** cf. 18.2.4 "Passing Objects", pp. 532-533 of Stroustrup (2013) [10]

For arguments, we have 2 main choices for passing in objects to an operator:

- Pass-by-value
- Pass-by-reference

Typically, an operator returns a result. Returning a pointer or reference to a newly created object is usually a bad idea: using a pointer gives notational problems, and referring to an object on free store (whether by pointer or by reference) results in memory management problems. INstead, return objects by value. For large objects, define move operations to make such transfers of values (i.e. return by value) efficient.

If a function simply passes an object to another function, an rvalue reference argument should be used.

cf. 2.7.3 "Member of Free Function", Gottschling (2015) [11]

Writing all 3 implementations (binary operator adding two complex class instances, operator oveload accepting a double as second argument, etc.) provides better performance.

Adding 2 complex class values is more naturally expressed as a free function. To avoid ambiguity, we have to remove the member function with the complex argument.

Subtle difference: free functions are selected by overload resolution with potential implicit conversion on both arguments while methods are subject to overriding and name hiding when classes are derived.

Binary Operators: Prefer implementing binary operators as free functions.

14. Exception Handling, Error Handling, Exception Policy

Stroustrup (2013) [10]

15. Resources

Hundreds of easy examples. https://www.tenouk.com/cpluscodesnippet/cplusarraypointermemoryaddress.html

Part 3. Integers, numeric encodings, number representation, binary representation, hexidecimal representation

15.1. **Information storage.** cf. Ch. 2, Representing and Manipulating Information, pp. 31, Bryant and O'Hallaron (2015) [1]

A machine-level program views memory as a very large array of bytes, referred to as *virtual memory*. Every byte of memory is identified by a unique number, known as *address*, and set of all possible addresses known as *virtual address space*.

Actual implementation (Ch.9, Bryant and O'Hallaron (2015) [1]) uses combination of dynamic random access memory (DRAM), flash memory, disk storage, special hardware, and operation system software to provide program with what appears to be a monolithic byte array.

Bryant and O'Hallaron (2015) [1] covers in subsequent chapters how compiler and run-time system partitions this memory space into more manageable units to store the different *program objects*, i.e. program data, instructions, and control information.

C compiler maintains *type* information.

16. Introduction:

`int` $\neq \mathbb{Z}$, `float` $\neq \mathbb{R}$

cf. Systems 1, Introduction to Computer Systems, Introduction lecture at University of Texas, CS429; Don Fussell.

`int` $\neq \mathbb{Z}$, `float` $\neq \mathbb{R}$

e.g. $x^2 \geq 0$?

`float`: Yes!

`int`: $40000 \times 40000 \rightarrow 1600000000$

$50000 \times 50000 \rightarrow$??

Is $(x + y) + z = x + (y + z)$?

`unsigned` and `signed int`: Yes!

`float`'s

$$(1e20 + -1e20) + 3.14 \rightarrow 3.14$$

$$1e20 + (-1e20 + 3.14) \rightarrow ??$$

Can't assume "usual" properties due to finiteness of representations, so that integer operations satisfy "ring" properties - commutativity, associativity, distributivity

floating point operations satisfy "ordering" properties - monotonicity, values of signs

17. Bits and Bytes

cf. Lecture 2, Bits and Bytes, CS429.

Consider a (number) system with radix or base b ($b > 1$),
string of digits $d_1, \ldots, d_n$,

$$c = \sum_{i=1}^{n} d_i b^{n-i}, \qquad 0 \leq d_i < b \qquad \forall\, i = 1 \ldots n \tag{3}$$

with notation $c \equiv c_b$ to help denote the base or "radix", e.g. 15213_{10}, 00000000_2.

Byte $\equiv B$, 1 Byte $= 8$ bits. Note that $2^8 = 256$

e.g. Binary 00000000_2 to 11111111_2

Decimal: 0_{10} to 255_{10}

Hexadecimal: 00_{16} to FF_{16}

cf. Sec. 2.1.1. Hexadecimal Notation, pp. 37, Bryant and O'Hallaron (2015) [1]

A single byte B consists of 8 bits.
B ranges from, in binary notation, 00000000_2 to 11111111_2
B ranges from, in decimal integer, 0_{10} to 255_{10}
B ranges from, in hexadecimale, 00_{16} to FF_{16}.

In C, numeric constants starting with $0x$ or $0X$ are interpreted as hexadecimal. Characters 'A' through 'F' may be written in either upper- or lowercase. Even mixing upper- and lowercase (e.g. $0xFa1D37b$).

Digits can be converted by referring to Table 17. *One simple trick for doing the conversion in your head is to memorize the decimal equivalents of hex digits A, C, and F.*

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111
Hex digit 8	9	A	B	C	D	E	F	
Decimal value	8	9	10	11	12	13	14	15

Hex values B, D, and E can be translated to decimal by computing their values relative to the first 3.

e.g. Given number $0x173A4C$ - convert this to binary format by expanding each hexadecimal digit

Hexadecimal	1	7	3	A	4	C
Binary	0001	0111	0011	1010	0100	1100

This gives binary representation 000101110011101001001100

Conversely, given a binary number, you convert it to hexadecimal by first splitting into groups of 4 bits each.

However, if total number of bits isn't a multiple of 4, you should make the *leftmost* group be the one with fewer than 4 bits, effectively padding the number with leading zeros. e.g. 1111001010110110110011

Binary	11	1100	1010	1101	1011	0011
Hexadecimal	3	C	A	D	B	3

Problem Practice Problem 2.1.

A $0x39A7F8$ to binary, 001110011010011111111000
B binary 1100100101111011, to $0xC97B$
C $0xD5E4C$, 11010101111001001100
D binary 1001101110011110110101 to hexadecimal, $0x26E7B5$

17.1. **Machine Words.** Machine has "Word size".
Nominal size of integer-valued data, including addresses.

17.2. **Word-oriented Memory Organization.** Addresses specify Byte locations.
Addresses of successive words differ by 4 (32-bit) or 8 (64-bit).
32-bit words:
Addr = 0000, Addr = 0004
64-bit words:
Addr = 0000, Addr = 0008

17.3. **Byte-ordering: Big-Endian, Little-Endian.** $b = 8$

$$c = \sum_{i=0}^{n-1} d_i 8^{n-1-i} \tag{4}$$

Big-Endian: Least significant byte has highest address.
For address a,

$$a \mapsto d_0$$
$$a+1 \mapsto d_1$$
$$\vdots$$
$$a+n-1 \mapsto d_{n-1}$$

Little-Endian: Least significant byte has lowest address

$$a \mapsto d_{n-1}$$
$$a+1 \mapsto d_{n-2}$$
$$\vdots$$
$$a+n-1 \mapsto d_0$$

17.4. **Representing Strings.** C Strings
- represented by array of characters, each character encoded in ASCII format - string should by null-terminated meaning final character = 0

17.5. **Machine-Level Code Representation.** Encode program as sequence of Instructions.

Each simple operation - arithmetic operation, read or write memory, conditional branch. - instructions encoded as bytes, i.e. programs are byte sequences too!

17.6. **Boolean Algebra.** Boole's Algebraic representation of logic.
And:
$A\&B = 1$ when both $A = 1$ and $B = 1$
Or:
$A|B = 1$ when either $A = 1$ or $B = 1$
Not:
$\sim A = 1$ when $A = 0$,
$\sim A = 0$ when $A = 1$,

Exclusive-Or (Xor) $A\hat{}B = 1$ when either $A = 1$ or $B = 1$, but not both

$\hat{}$	0	1
0	0	1
1	1	0

17.6.1. *Application of Boolean Algebra.* Claude Shannon 1937 MIT Master's Thesis. Encode closed switch as 1, open switch as 0. Connection when $A\& \sim B| \sim A\&B = A \wedge B$

17.6.2. *Integer Algebra.* $\langle \mathbb{Z}, +, *, -, 0, 1 \rangle$ forms a ring

17.6.3. *Boolean Algebra.* $\langle \{0,1\}, |, \&, \sim, 0, 1\rangle$ forms a "Boolean algebra"

Or (|) is "sum" operation.
And (&) is "product" operation.
$\sim$ is "complement" operation.
0 is additive identity.
1 is multiplicative identity.

cf. https://en.wikipedia.org/wiki/Bitwise_operation

Bitwise OR may be used to set to 1 selected bits of register. e.g. fourth bit of 0010 may be set by performing bitwise OR with pattern with only 4th bit set.

Bitwise AND is equivalent to multiplying corresponding bits. Thus, if both bits in compared position are 1, bit in resulting binary representation is $(1 \times 1 = 1)$; otherwise result is 0 ($1 \times 0 = 0$ and $0 \times 0 = 0$).

Operation may be used to determine whether particular bit is set (1) or clear (0). e.g. given bit pattern 0011 (decimal 3), to determine whether 2nd bit is set, use bitwise AND with bit pattern containing 1 only in 2nd. bit. Because result 0010 is non-zero, we know 2nd. bit in original pattern is set. This is often called *bit masking* (by analogy, use of masking tape covers, or *masks*, portions that are not of interest)

Bitwise AND may be used to clear selected bits (or flags) of a register in which each bit represents an individual Boolean state.

- This technique is an efficient way to store a number of Boolean values using as little memory as possible.
- Also, easy to check parity (even or odd?) of binary number by checking value of lowest valued bit.

Bitwise XOR may be used to invert selected bits in a register (also called toggle or flip). Any bit may be toggled by XORing with 1. e.g. given bit pattern 0010, 2nd and 4th bits may be toggled by bitwise XOR with bit pattern containing 1 in 2nd. and 4th. positions.

Consider Boolean Ring:

$$\langle \{0,1\}, \wedge, \&, I, 0, 1\rangle \simeq \mathbb{Z}_2 \equiv \text{ integers } \mod 2$$

where I is identity operation $I(A) = A$,
and $A \wedge A = 0$ is the additive inverse (existence) property.

17.6.4. *De Morgan's Laws.*

$$A\&B = \sim (\sim A | \sim B)$$
$$A|B = \sim (\sim A\& \sim B)$$

17.6.5. *Exclusive-Or using Inclusive Or.*

$$A \wedge B = (\sim A\&B)|(A\& \sim B)$$
$$A \wedge B = (A|B)\& \sim (A\&B)$$

All properties of Boolean Algebra apply to bit vectors.

17.7. **Representing and Manipulating Sets; bit vectors as sets.** Width w bit vector represents subsets of $\{0, \ldots, w-1\}$, s.t. for $a \in \{0,1\}^w$, $a_j \in \{0,1\}$, $\forall\, j = 0, 1, \ldots w-1$.

If $a_j = 1, j \in A$; otherwise,

If $a_j = 0$, $j \notin A$

17.7.1. *Operations.*
$\& \to$ intersection
$| \to$ union
$\wedge \to$ symmetric difference
$\sim\!\!\to$ complement

Logic operations in C: &&, ||, !
Shift operations
cf. Systems 1, Integers lecture at University of Texas, CS429 is a **very good lecture**; both *mathematically rigorous* and full of useful, *clear* examples.

18. Unsigned integers

Consider $\mathbb{Z}^+_{2^w} \to \mathbb{Z}^+$, where $\mathbb{Z}^+_{2^w}$ represent "unsigned" integers.

$$\mathbb{Z}^+_{2^w} \to \mathbb{Z}^+ \tag{5}$$
$$(x_0, x_1, \dots, x_{w-1}) \mapsto \sum_{i=0}^{w-1} x_i \cdot 2^i$$

To recap using CS429's notation, for an unsigned integer x,
$B2U \equiv$ base-2-unsigned

$$B2U(x) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

where $B2U : "\mathbb{Z}^+" \to \mathbb{Z}^+_{2^w}$
So w is the total number of "bits" to represent x.
For hexidecimal (-based) numbers, observe this relationship:

$$2^w = 16^v = 2^{4v} \text{ so } w = 4v \tag{6}$$

So for a hexidecimal number, v hexidecimal numbers represent $w = 4v$ bits.
Observe that

$$\mathbb{Z}^+_{2^w} = \{0, \dots 2^w - 1\} \tag{7}$$

Also, **modular addition** $(\mathbb{Z}^+_{2^w}, +)$ forms an **abelian group**.

19. Two's complement

cf. Systems 1 Integers
Denote the notation for so-called "two's complement" numbers as $\mathbb{Z}_{2^w}$, which is a representation for integers in such a manner:

$$\mathbb{Z}_{2^w} \to \mathbb{Z} \tag{8}$$
$$(x_{w-1}, x_{w-2}, \dots x_0) \mapsto -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

with

$$\mathbb{Z}_{2^w} = \{-2^{w-1} \dots 2^{w-1} - 1\} \tag{9}$$

To recap using CS429's notation, for integer $x \in \mathbb{Z}$,
$B2T \equiv$ base-2-Two's complement

$$B2T(x) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

where $B2T : "\mathbb{Z}" \to \mathbb{Z}_{2^w} = \{-2^{w-1} \dots 2^{w-1} - 1\}$

Observe that

$$\begin{aligned} \max\left(\mathbb{Z}^+_{2^w}\right) &= 2 \max \mathbb{Z}_{2^w} + 1 \\ |\min \mathbb{Z}_{2^w}| &= \max \mathbb{Z}_{2^w} + 1 \end{aligned} \tag{10}$$

To recap and to clean up and unify the notation, recall the following:
radix (i.e. base) b
string of digits $d_1, \dots d_n$,

$$c = \sum_{i=1}^{n} d_i b^{n-i}, \quad 0 \le d_i < b \quad \forall\, i = 1 \dots n$$

cf. wikipedia, "Radix" i.e.

$$\{0, 1, \dots b-1\}^n \to \mathbb{Z}$$

$$(d_1, \dots d_n) \mapsto c = \sum_{i=1}^{n} d_i b^{n-i}$$

or (alternative notation),

$$\{01, , \dots b-1\}^w \to \mathbb{Z}$$

$$(d_{w-1}, d_{w-2}, \dots, d_0) \mapsto c = \sum_{i=0}^{w-1} d_i b^i$$

In Fussell's notation, Fussell (2011) [23],

$$\{0, 1, \dots b-1\}^w \to \mathbb{Z} \text{ or } \mathbb{Z}^+_{2^w} \to \mathbb{Z}^+$$

$$(x_{w-1}, x_{w-2}, \dots x_0) \mapsto c = \sum_{i=0}^{w-1} x_i 2^i,\ x_i < 2$$

$$B2U(X) = B2U(x_{w-1}, \dots x_0) = \sum_{i=0}^{w-1} x_i 2^i$$

I will also use this notation:

$$B2U(x_w, x_{w-1}, \dots x_1) = \sum_{i=1}^{w} x_i 2^{i-1}$$

For the *Two's Complement*,

$$B2T(X) = B2T(-x_{w-1}, x_{w-2}, \dots x_0) = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

$$\{0, 1, \dots b-1\}^w \to \mathbb{Z}$$

or with the notation I'll use,

$$B2T(-x_w, x_{w-1}, \dots x_1) = -x_w 2^{w-1} + \sum_{i=1}^{w-1} x_i 2^{i-1}$$

Note that for hexadecimal,

$$c = \sum_{i=1}^{w} x_i 2^{i-1} = \sum_{i=1}^{v} y_i (16)^{i-1} = \sum_{i=1}^{v} y_i 2^{4i-4}$$

v hexadecimal digits $\leftrightarrow 4v = w$ bits.

19.1. **Numeric Ranges.** For unsigned values,

$$\text{UMin} = 0$$
$$000\dots0$$
$$\text{UMax} = 2^w - 1$$

For why $\text{UMax} = 2^w - 1$, consider the geometric progression:

$$\sum_{k=1}^{n} ar^{k-1}$$
$$(1-r)\sum_{k=1}^{n} ar^{k-1} = a - ar^n \text{ or } \sum_{k=1}^{n} ar^{k-1} = \frac{a(1-r^n)}{1-r}$$
$$\implies \sum_{i=1}^{w} 2^{i-1} = \frac{1-2^w}{1-2} = 2^w - 1$$

$$\text{TMin} = -2^{w-1}$$
$$100\dots0$$
$$\text{TMax} = 2^{w-1} - 1011\dots1$$

$-1 \leftrightarrow 111\dots1$

19.2. **Negating with complement.**

$$\begin{gathered} \sim\colon \mathbb{Z}_{2^w} \to \mathbb{Z} \text{ s.t.} \\ \sim x + 1 = -x \quad \forall\, x \in \mathbb{Z}_{2^w} \\ \sim\colon \mathbb{Z}^+_{2^w} \to \mathbb{Z} \text{ s.t.} \\ \sim x = 2^w - 1 - x \end{gathered} \tag{11}$$

19.3. **Power of 2 Multiply with shift (left shift bitwise operation).** For both $\mathbb{Z}^+_{2^w}$, $\mathbb{Z}_{2^w}$,

$$u << k = u \cdot 2^k \qquad \forall\, u \in \mathbb{Z}^+_{2^w} \text{ or } \mathbb{Z}_{2^w} \text{and } k \in \mathbb{Z} \tag{12}$$

19.4. **Unsigned power-of-2 Divide, with shift (right shift bitwise operator).**

$$u >> k = \lfloor u/2^k \rfloor \qquad \forall\, u \in \mathbb{Z}^+_{2^w}, \quad k \in \mathbb{Z} \tag{13}$$

20. Endianness

Consider *address* $\in \mathbb{Z}_{2^3}^+$. Consider a *value* $\in \mathbb{Z}_{2^3}^+$ or $\mathbb{Z}_{2^3}$.
$\forall\, (x_0, x_1, \dots x_{w-1}) \in \mathbb{Z}_{2^3}^+$ or $\mathbb{Z}_{2^3}$
Given address a,

20.1. Big-endian.

Definition 2 (Big-endian)**.** *Most-significant byte value is at the lowest address, i.e.*

$$\begin{aligned} a &\mapsto x_{w-1} \\ a+1 &\mapsto x_{w-2} \\ a+2 &\mapsto x_{w-3} \\ &\vdots \\ a+w-1 &\mapsto x_0 \end{aligned} \tag{14}$$

20.2. Little-endian.

Definition 3 (Little-endian)**.** *Least-significant byte value is at the lowest address, i.e.*

$$\begin{aligned} a &\mapsto x_0 \\ a+1 &\mapsto x_1 \\ &\vdots \\ a+w-1 &\mapsto x_{w-1} \end{aligned} \tag{15}$$

For Little-endian $\to \mathbb{Z}$,
$(\ \textit{addresses}\ ) \in \mathbb{Z}_{2^3}^+ \mapsto \mathbb{Z}$,
$a \mapsto (x_0, x_1, \dots x_{w-1})$
For $\mathbb{Z} \to$ *Big-endian,*
$n \mapsto (x_{w-1}, x_{w-2}, \dots x_0)$.
Instead, to tackle the confusing problem of byte ordering, think about the integer as itself first, and then consider mapping multibyte binary values to memory.

$$\mathbb{Z} \to \left(\mathbb{Z}_{2^w} \longleftarrow \mathbb{Z}_{(2^3)^w} = \boldsymbol{addresses}\right) \tag{16}$$

$$\mathbb{Z} \ni (x_{w-1}, x_{w-2}, \dots x_1, x_0) \xmapsto{\textit{Little-Endian}} ((x_0, x_1, \dots x_{w-2}, x_{w-1}) \longleftarrow\!\shortmid\ (a, a+1, \dots a+w-2, a+w-1))$$

$$\mathbb{Z} \ni (x_{w-1}, x_{w-2}, \dots x_1, x_0) \xmapsto{\textit{Big-Endian}} ((x_{w-1}, x_{w-2}, \dots x_1, x_0) \longleftarrow\!\shortmid\ (a, a+1, \dots a+w-2, a+w-1))$$

Part 4. Floating Point Numbers

21. Floating point, IEEE Floating

IEEE Standard 754
Recall our previous notation for the sum representation of a number c:

$$c = \sum_{i=1}^{n} d_i b^{n-i}; \qquad 0 \le d_i < b, \quad \forall\, i = 1 \dots n \tag{17}$$

e.g.

$$c = 42_{10} = 4 \cdot 10^{2-1} + 2 \cdot 10^{2-2} = d_1 b^{n-1} + d_2 b^{n-2}$$

By writing the exponent to be $n - i$, then we can write the digit representing the largest value in the summation "starting from the left" as such:

$$c = (d_1, d_2, \dots d_{n-1}, d_n)$$

e.g. $(4, 2) = 42$.

However, the CS429h lectures uses the following notation for the `unsigned int`:

$$B2U(X) = \sum_{i=0}^{w-1} x_i 2^i \text{ or } (x_{w-1}, x_{w-2}, \dots x_1, x_0) \mapsto \sum_{i=0}^{w-1} x_i 2^i$$

e.g. $(x_3, x_2, x_1, x_0) = (1, 0, 1, 1) \mapsto x_0 2^0 + x_1 2^1 + x_2 2^2 + x_3 2^3$

21.1. **Fractional Binary Numbers.**

21.1.1. *Representation.* Bits to right of "binary point" represent fractional powers of 2.
Binary number is used to represent a rational number, which in turn is a floating-point *representation*.

$$\sum_{k=-j}^{i} b_k \cdot 2^k \equiv \sum_{k=-M}^{N} b_k 2^k \tag{18}$$

$$b \equiv (b_N, b_{N-1}, \dots b_1, b_0, b_{-1}, \dots b_{-M}) \mapsto \sum_{k=-M}^{n} b_k 2^k \equiv c \tag{19}$$

i.e.

$$\sum_{k=-M}^{N} b_k 2^k = b_N \cdot 2^N + b_{N-1} 2^{N-1} + \dots + b_1 \cdot 2 + b_0 \cdot 1 + b_{-1} \cdot \frac{1}{2} + \dots + b_{-M} \cdot 2^{-M}$$

e.g. $101.11_2 \mapsto 5\frac{3}{4}$

Divide by 2 by shifting right:

$$\frac{c}{2} = \sum_{k=-M}^{N} b_k 2^k / 2 = \sum_{k=-M-1}^{N-1} b_{k+1} 2^k \tag{20}$$

$$(b_N, b_{N-1}, \dots b_{-M}) \xrightarrow{\cdot \frac{1}{2}} (0, b_N, b_{N-1}, \dots b_{-M-1}, b_{-M})$$

Multiply by 2 by shifting left:

$$2 \cdot c = \sum_{k=-M}^{N} b_k \cdot 2^k \cdot 2 = \sum_{k=-M+1}^{N+1} b_{k-1} 2^k \tag{21}$$

$$(b_N, b_{N-1}, \dots b_{-M}) \xrightarrow{\cdot 2} (b_N, b_{N-1}, \dots b_{-M}, 0)$$

This implies that we first choose the "most significant bit", the digit b_k representing the largest value in the sum representation to start from the *left*.

Numbers of form $0.111111\dots_2$ just below 1.0 (remember those are bits, 0 or 1).

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^M} + \cdots \to 1.0$$

Use $1.0 - \epsilon$ notation.

21.1.2. *Limitation of representable numbers.* Can only exactly represent numbers of form $\frac{x}{2^k}$

The other numbers have repeating bit representations.

Value	Representation
$\frac{1}{3}$	$0.0101010101[01]\ldots_2$
$\frac{1}{5}$	$0.001100110011[0011]\ldots 2$
$\frac{1}{10}$	$0.0001100110011[0011]\ldots_2$

21.1.3. *Floating Point Representation. Numerical Form*:

$$-1^s\, M\, 2^E \tag{22}$$

where

sign bit s, $+1$ or -1
significand (mantissa, or coefficient) M, normally a fractional value in range $[1.0, 2.0)$
Exponent E weights value by power of 2.

From wikipedia,

$$\frac{s}{b^{p-1}} \cdot b^e \tag{23}$$

where s significand (ignoring an implied decimal point), p precision (number of digits in significand), base b.

e.g.

$$1.528535047 \times 10^5 = \frac{1528535047}{10^9} \times 10^5$$

Another example:

$$\left(\sum_{n=0}^{p-1} c_n 2^{-n}\right) \times 2^e =$$
$$(1 \times 2^{-0} + 1.2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + \cdots + 1 \cdot 2^{-23}) \times 2^1$$
$$\approx 1.5707964 \times 2 \approx 3.1415928$$

For the encoding of this numerical form,

Most significant bit (MSB) is sign bit.

exp field encodes E
frac field encodes M

s exp frac

Sizes for the encoding:

Single precision: 8 exp bits, 23 frac bits; 31 bits + 1 s bit = 32 bit
and so

2^{8-1}

$$\sum_{i=0}^{M-1} c_i 2^{-i} \xrightarrow{\max} 2^{M-1} = 2^{23-1}$$

Double precision: 11 exp bits, 52 frac bits; 63 bits + 1 s bit = 64 bits
$2^{\exp-1} = 2^{11-1} = 1024$. This is the max value of e in decimal.
$2^{M-1} = 2^{52-1}$

21.1.4. *"Normalized" Numeric Values.* Condition:

$$\exp \neq 000\ldots0 \text{and } \exp \neq 111\ldots1$$

Exponent coded as biased value:

$$E = \text{Exp} - \text{Bias} \tag{24}$$

Exp: unsigned value determined by exp
Bias: Bias value

Note that E is the actual value desired, e.g. $2^{13} = 2^E$, whereas Exp is the actual binary representation.

- Single precision: 127 (Exp: $1\ldots254$, E: $-126\ldots127$)
- Double precision: 1023 (Exp: $1\ldots2046$, E: $-1022\ldots1023$)

In general

$$\text{Bias} = 2^{e-1} - 1 \tag{25}$$

where e is number of exponent bits.

wikipedia says this: the exponent is "biased" in the engineering sense of the word - value stored is offset from actual value by the **exponent bias**. Biasing is done because exponents have to be signed value in order to be able to represent both tiny and huge values. It's not represented in two's complement, usual representation for signed values; it'd make comparison harder.

Exponent is stored as unsigned value which is suitable for comparison, and when interpreted, it's converted into an exponent within a *signed* range by subtracting the bias.

wikipedia uses this notation:

$$2^{k-1} - 1 = \text{ bias for floating point number}$$

where k is number of bits in exponent.

Number of normalized floating-point numbers in system (B, P, L, U) where

- base B
- precision of system to P numbers
- L smallest exponent representable in system
- U largest exponent used in system

$$2(B-1)(B^{P-1})(U-L+1)+1 \tag{26}$$

Smallest positive normalized floating-point number.

Underflow level = UFL = B^L

TODO: Understand all of this in https://en.wikipedia.org/wiki/Floating-point_arithmetic

Significand coded with implied leading 1:
Get extra leading bit for "free".
$xxx\ldots x$: bits of frac
Minimum when $000\ldots0$ ($M = 1.0$) Maximum when $111\ldots1$ ($M = 2.0 - \epsilon$)

21.1.5. *Denormalized Values and other Special Values.* Consider "denormalized" values, the special case with the following condition:

$$\text{exp} = 000\ldots0$$

The Value is this:
Exponent value $E = -\text{Bias} + 1$ (compare this to Eq. **??**).
Significand value $M = 0.xxx\ldots x_2$, i.e. $xxx...x$: bits of frac.
Consider these special cases:

- exp $= 000\ldots0$, frac $= 000\ldots0$, which represents value 0
  Note that there are distinct values for $+0$ and -0
- exp $= 000\ldots0$, frac $\neq 000\ldots0$
  These represent numbers that are very close to 0.0. They lose precision as they get smaller. There's "gradual underflow". TODO: understand what it means for "gradual underflow", cf. https://www.cs.utexas.edu/users/fussell/courses/cs429h/lectures/Lecture_4-429h.pdf, pp. 10

Consider special values with the condition that

$$\text{exp} = 111\ldots1$$

They include the following cases:

- exp $= 111\ldots1$, frac $= 000\ldots0$. This represents the value at infinity, ∞. Operation that overflows. There are positive infinity and negative infinity. e.g. $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
- exp $= 111\ldots1$, frac $\neq 000\ldots0$ This is Not-a-Number (NaN). It represents case when no numeric value can be determined. e.g. `sqrt(-1)`, $\infty - \infty$.

Part 5. Data Structures, Algorithms, Complexity

22. Containers, Relations, Abstract Data Types

cf. 2. Algorithmic Analysis, 2.1 Containers, Relations, and abstract data types (ADTs) Harder (2018) [17]

Definition 4 (Simple Containers)**.** ***Simple Containers*** *- containers that store individual objects.*

e.g. Temperature readings $\implies$ **Circular array**

Definition 5 (Associative Containers)**.** ***Associative Containers*** *- Containers that store keys but also store records associated with those keys.*

e.g. ID Number $\implies$ Server $\implies$ Student Academic Record.

23. Peak finding

cf. Demaine and Devadas (2011) [16], 1. Algorithmic Thinking, Peak Finding, MIT OCW MIT 6.006 Introduction to Algorithms, Fall 2011.

23.1. **1-dim. version of peak finding.** Given an array $a = a_i$, $i = 0, \dots N-1$, $a_i \in \mathbb{Z}$ (a_i are numbers). Position α is a peak if $a_\alpha \geq a_{\alpha-1}$ and $a_\alpha \geq a_{\alpha+1}$

In the case of the edges, look only to one side, e.g. a_{N-1} is a peak iff $a_{N-2} \leq a_{N-1}$.

Problem: Find a peak if it exists.

23.1.1. *Straightforward algorithm (peak finding, 1-dim.)* Start from left.

In case of $\geq$, then, homework, argue that $\exists$ peak, always. (true).
If case of $>$ can you make this argument? No.

You want to create algorithms that are general, so if problem definition changes on you, still have a starting point to attack 2nd. version of the problem.

If peak at $n/2$, look at $n/2$ elements.
Likewise, worst-case complexity $\theta(n)$. (you have to start from the left and go all the way to the right).
(Big O is upper bound)

Try break up array into smaller arrays, divide and conquer.
Divide & conquer: Given array a_i, $i = 1, 2, \dots n$
Look at $n/2$ position.
If $a[n/2] < a[n/2-1]$ then only look at left half $1 \dots n/2-1$ to look for a peak.
Else if $a[n/2] < a[n/2+1]$, then $\dots n/2+1 \dots n$ for a peak.
else $n/2$ position is a peak.
(Write an argument that this algorithm is correct).

If we define $T(n) \equiv$ work this algorithm does on input of size n, then for this divide and conquer algorithm,

$$T(n) = T(n/2) + \theta(1)$$

$\theta(1)$ work of 2 comparisons looking left and right

Base case: $T(1) = \theta(1)$

$$T(n) = \underbrace{\theta(1) + \dots + \theta(1)}_{\log_2 n \text{ times}} = \theta(\log_2 n)$$

23.2. **2-dim. version of peak finding.** For n rows, m columns, a_{ij} is a 2-dim. peak iff

$$a_{ij} \geq a_{ij-1}, a_{ij} \geq a_{ij+1}, a_{ij} \geq a_{i-1j}, a_{ij} \geq a_{i+1j}$$

(Devadas originally wrote a is a 2D peak iff)

$$a \geq b, a \geq d, a \geq c, a \geq e$$

23.2.1. *Greedy ascent algorithm.* (picks a direction, and follows that direction to find peak)

For any starting direction, starting pt. (e.g. middle), could touch many elements.

From worst case analysis, $O(nm)$ complexity.

23.2.2. *Recursive version, Attempt 2.* Pick middle column $j = m/2$.
Find global max on column j at (i, j).
Compare $(i, j-1), (i, j), (i, j+1)$.
Pick left columns if $(i, j-1) > (i, j)$. Similarly for right column.

If $(i, j) \geq (i, j-1)$, $(i, j+1) \Longrightarrow (i, j)$ is a 2D peak.
Solve the new problem with half the number of columns.
When you have a single oclumn, find the global max. $\Longleftarrow$ done.

$T(n, m) = T(n, m/2) + \theta(n) \Longleftarrow$ max: finding the global max. is for $\theta(n)$.
$T(n, 1) = \theta(n)$
$T(n, m) = \underbrace{\theta(n) + \cdots + \theta(n)}_{\log_2 m} = \theta(n \log_2 m)$

24. Zero-based numbering vs. One-based numbering

0-based numbering vs. 1-based numbering
Let N = total number of elements

$0, 1, \ldots N-1$ vs. $1, 2, \ldots N$
$N = 2M$ if N even, $N = 2M+1$, $M = 0, 1, \ldots$ if N odd.
$\frac{N}{2} = M$ if N even.
$\frac{N}{2}$ is the first element of the "right" half in 0-based numbering.
$\frac{N}{2}$ is the "last", most "right" element of the "left" half in 1-based numbering.

$\frac{N}{2} = M$ if N odd as well, in both cases.
$\frac{N}{2} = M$ is the middle element in 0-based numbering.
$\frac{N}{2} = M$ is the "left" adjacent element to the middle in 1-based counting.

25. Models of Computation, Document Distance

cf. 2. Models of Computation, Document Distance, MIT OCW, MIT 6.006 Introduction to Algorithms, Fall 2011.

25.1. **Model of computation.** Specifies
- what operations an algorithm is allowed
- cost (time, ...) of each operation.
EY: When you add up all the costs of all operations, you get the running time.

25.1.1. *Random Access Machine.* cf. 1. Random Access Machine.
EY: computer is to model of computation.
Random Access Memory (RAM) is to Random Access Machine (RAM).
RAM (memory) is essentially a giant array, access somewhere in constant time.
-Random Access Memory (RAM) modeled by big array.
-in $\theta(1)$ time, can

- load $O(1)$ words
- do $O(1)$ computations
- start $O(1)$ words

	word
0	
1	
2	
⋮	⋮

-$O(1)$ registers.

- **word** : w bits (e.g. $w = 32$ or 64 bits).

EY: If you can manipulate/compare words (e.g. peak finding), you can do comparisons in constant time $\theta(1)$.

w should be
log (size of memory)
in order to specify index of the array.

EY: $N \equiv$ size of memory.

$$\log_2 N = y; 2^y = N$$

If $y = 1$, $0, 1$ (total) indices to specify 2 addresses. $y = 2$, specify 4 addresses. So y bits, $y = \log_2 N$ to specify N addresses.

From 2.2 Analyzing algorithms, pp. 23 of Cormen, Leiserson, Rivest, and Stein (2009) [19], in the RAM model, instructions are executed one after another, with no concurrent (EY: parallel?) operations.

The RAM model should contain instructions commonly found in real computers: arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling), **data movement** (*load, store, copy*), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes constant amount of time $O(1)$.

Assume a limit on the size of each word w of data. For example, when working with inputs of size n, we typically assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. Require $c \geq 1$ so that each word can hold the value of n, enabling us to index individual input elements, and restrict c to be constant so that word size doesn't grow arbitrarily. (If word size could grow arbitrarily, we could store huge amounts of data in 1 word and operate on it all in constant time - clearly unrealistic!)

$$c \log n = y \qquad \text{e.g. } y = 64 \text{ bits}$$

$b^{y/c} = n$. n is size of the input. If n is the max. value of (unsigned) integer, will need y bits for base b to represent input uniquely on RAM model.

To define "size of input" carefully, note that the best notion for **input size** depends on problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, use the *number of items in the input* - for example array of size n for sorting. Sometimes, if input to algorithm is a graph, input size can be described by numbers of vertices and edges in the graph (2 or more numbers, instead of 1).

25.1.2. *Pointer Machine.* cf. 2. **Pointer Machine**.

EY: in RAM, there's no dynamic allocation.

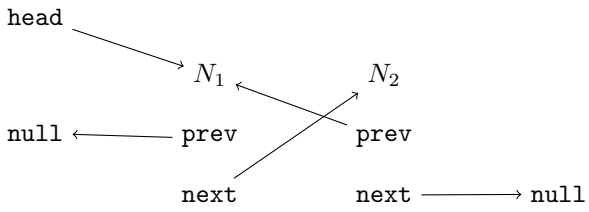
Pointer Machine analogous to OOP. (object oriented programming)

- dynamically allocated objects
- *object* has $O(1)$ *fields*
- *field* = *word* (e.g. `int`)
  or **pointer** to object or null (None)

e.g. Linked Lists:

val
prev
next

where
val ≡ value
prev ≡ pointer to some previous node
next ≡ pointer to some next node



You can implement this in the RAM model. A pointer becomes an index into this giant table. In C, do pointer arithmetic to get next element in RAM array.

In this model (pointer machine) you can only follow pointers (costs constant time).

Demaine: RAM model and pointer model are simple, load, store, follow pointer operations take $O(1)$ constant time.

25.1.3. *Python Model.* 1. "list" = array (list in Python is an array in RAM).

```
L[i] = L[j] + 5
```

(takes $O(1)$ time (in Python it's constant time, O(N) for access in a linked list)

2. Object with O(1) attributes

If object has 3 things or 10 things, constant time O(1) operations.

e.g. `x = x.next` is $\implies$ O(1) time.

For appending to a list *in Python*,

`L.append(x)`

$\implies$ *table doubling* (lecture 9) (Python uses an algorithm called table doubling to do appending to a list)

$\implies$ it's O(1) time

Demaine: There's a lot more

`L = L1 + L2` (concatenate 2 lists)

≡ `L = []`

```
for x in L1:          O(L1) time
L.append(x) // O(1) time for x in L2:       O(L2) time
L.append(x) // O(1) time
```

In total, O(—L1— + —L2—) time.

Demaine: don't think addition operation sign + means O(1); L1, L2 are large data structures, not words.

`x in L` operation in Python is O(N) worse case is testing every element.

`len(L)` O(1): Python stores length of list in beginning.

`L.sort()` $\implies O(|L| \log |L|)*$ time to compare 2 times. (Lecture 3; Python uses comparison sort)

Python dictionaries, **dict**.
`D[key]` = val $\implies$ O(1) (it's a hash table, lectures 8-10)
Also, alternative syntax: `key in D`

Note from Demaine: it's constant time with high probability w.h.p.

Demaine: *If you write any code today, use dictionaries to solve problems.*

long long integers in Python 3 version 2 (lecture 11)

How long does it take to add longs?
$x + y$ O(—x— + —y—) (EY: think of addition in arithmetic and digit by digit addition)

$x * y$ $O((|x| + |y|)^{\log_2 3})$ Demaine's notation $lg \equiv \log_2$. Now $\lg 3 \approx 1.6$.
heapq heap. Lecture 4.
Demaine: There are more data structures online on the online notes.

26. Complexity, Big-O

Also note, let N = number of elements in a collection.

$$\log_b N = n \leftrightarrow b^n = N$$

$N \log_b N = n \leftrightarrow b^n = N^N$ but rather, use Stirling's approximation:

$$N \ln N \cong \log_b N! = \sum_{i=1}^{N} \log_b i = n$$

$O(N \log_b N) = O(\log_b N!)$ via Stirling's approximation: $\ln N! = N \ln N - N + O(\ln N)$

cf. Ch. 3 Growth of Functions, 3.1 Asymptotic notation of Cormen, Leiserson, Rivest, and Stein (2009) [19].

26.1. Θ-notation - worst case.

Formally,
given function $g(n)$,
(27)

$$\Theta(g(n)) := \{f(n) | \exists\, c_1, c_2, n_0 > 0 \text{ s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall\, n \geq n_0\}$$

Formally,

$$f(n) \in \Theta(g(n)) \text{ but notation abuse } f(n) = \Theta(g(n))$$

Θ-notation asymptotically bounds function from above and below.

26.2. **O-notation - upper bound.** Use O-notation for **asymptotic upper bound**.
Given function $g(n)$,

(28)
$$O(g(n)) = \{f(n) | \exists \text{ positive constants } c, n_0 \text{ s.t. } 0 \leq f(n) \leq cg(n), \quad \forall n \geq n_0\}$$

Since for given $f(n) \in \Theta(g(n))$ (abuse notation $f(n) = \Theta(g(n))$),

$$\begin{gathered} \exists c_2, n_0 > 0 \text{ s.t. } f(n) \leq c_2 g(n) \quad \forall n \geq n_0, \\ f(n) \in O(g(n)) \\ \implies \Theta(g(n)) \supseteq O(g(n)) \end{gathered}$$

Ω-notation, lower bound.
Ω-notation for **asymptotic lower bound**.

(29)
$$\Omega(g(n)) = \{f(n) | \exists c, n_0 > 0 \text{ s.t. } 0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0\}$$

Clearly,

Theorem 1 (3.1 Cormen, Leiserson, Rivest, and Stein (2009) [19])**.** $\forall$ *functions* $f(n), g(n)$, $f(n) \in \Theta(g(n))$ *iff* $f(n) \in O(g(n))$, *and* $f(n) \in \Omega(g(n))$

cf. Demaine and Devadas (2011) [16], 1. Algorithmic Thinking, Peak Finding, MIT OCW MIT 6.006 Introduction to Algorithms, Fall 2011. Sept. 8, 2011. PS 1.
Problem PS1, 1-1. Asymptotic Practice
Group 1:

$$\begin{aligned} f_1(n) &= n^{0.999999} \log n \\ f_2(n) &= 10000000n \\ f_3(n) &= 1.000001^n \\ f_4(n) &= n^2 \end{aligned}$$

Solution 1-1, Group 2.
Now

$$\begin{aligned} f_1(n) &= 2^{2^{1000000}} = 2^{2^a} \in O(1) \\ f_2(n) &= 2^{100000n} = 2^{Bn} = \exp(Bn \ln 2) \\ f_3(n) &= \binom{n}{2} \in O\left(\left(\frac{n}{2}\right)^2\right) \end{aligned}$$

In general

$$\binom{n}{k} \in O\left(\left(\frac{n}{k}\right)^k\right)$$

$$f_4(n) = n\sqrt{n} = n^{3/2} \in O(n^{3/2})$$

$$\boxed{f_1(n) \in O(1), f_4(n) \in O(n^{3/2}), f_3(n) \in O(n^2), f_2(n) \in O(2^{Bn})}$$

For the binominal equation, we used stackexchange for this proof:
Group 3:

$$\begin{aligned} f_1(n) &= n^{\sqrt{n}} \\ f_2(n) &= 2^n \\ f_3(n) &= n^{10} \cdot 2^{n/2} \\ f_4(n) &= \sum_{i=1}^{n}(i+1) \end{aligned}$$

$$f_4(n) = \sum_{i=1}^{n}(i+1) = \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 = \frac{n(n+1)}{2} + n \in O(n^2)$$

$$f_1(n) = n^{\sqrt{n}} = n^{n^{1/2}}$$

$$\begin{aligned} y &= n^{n^{1/2}} & \ln y &= n^{1/2} \ln n \\ \lg y &= n^{1/2} \lg n & y &= \exp\left(n^{1/2} \ln n\right) \\ y &= 2^{n^{1/2} \lg n} \end{aligned}$$

$$f_2(n) = 2^n = \exp(n \ln 2)$$

$$\begin{aligned} y &= n^{10} \cdot 2^{n/2} & \ln y &= 10 \ln n + \frac{n}{2} \ln 2 \\ f_3(n) = n^{10} \cdot 2^{n/2} \lg y &= 10 \lg n + \frac{n}{2} & y &= \exp\left(10 \ln n + \frac{\ln 2}{2} n\right) \\ y &= 2^{10 \log n + n/2} \end{aligned}$$

So in order of increasing complexity,

$$\boxed{\begin{array}{l} f_4(n) \in O(n^2) \\ f_1(n) \in O(2^{n^{1/2} \lg n}) \subset O(2^n) \\ f_3(n) \in O(2^{n/2+10 \lg n}) \subset O(2^{n/2}) \\ f_2(n) \in O(2^n) \end{array}}$$

26.3. **Multi-Part Algorithms: Add vs. Multiply, e.g. $O(A+B)$ vs. $O(A*B)$.** If your algorithm is "do this (A), then, when completed, do that (B)" then add runtimes: $O(A + B)$.

If your algorithm is "do this for each time you do that", then multiply runtimes: $O(A * B)$.

26.4. **Amoritized Time (e.g. dynamically resizing array "ArrayList" or maybe `std::vector`).** Dynamically resize array x: if there are N elements, s.t. $N =$ max. capacity at N, $|x| = N \mapsto |x| = 2N$ and there are N copies to be made ($T = N$).

Let $X =$ number of elements to be inserted.

Suppose when $x = 1, 2, 4, 8, 16, \ldots 2^j, \ldots X$, $x = 2^j$ copies are made.

total number of copies:

$$\sum_{j=0}^{\lfloor \log_2 X \rfloor} 2^j = \sum_{j=0}^{\lfloor \log_2 X \rfloor} 2^{\lfloor \log_2 X \rfloor - j} = \sum_{j=0}^{\lfloor \log_2 X \rfloor} \frac{X}{2^j} = X\left(\frac{1 - 2^{-(\lfloor \log_2 X \rfloor + 1)}}{1 - 1/2}\right) = 2X\left(1 - 2/X\right)$$

$$\cong 2X \text{ if } X \text{ large}$$

Note that $X = 2^y$

$\log_2 X = y$

X insertions take $O(2X)$. "Amoritized" time for each insertion is $O(1)$.

26.5. $\log N$ **runtimes ($O(\log N)$).** e.g. binary search. find element (example) c in N-element *sorted* array. $x = x_i$, $i = 0, 1, \dots N-1 \mapsto i' = 1, 2 \dots N$.

First, compare c to midpoint of array.

If $c = x_{\lceil \frac{N}{2} \rceil}$, done.

So let $n_1 = N$.

if $c < x_{\lceil \frac{N}{2} \rceil}$, consider $x_1, x_2, \dots x_{\lfloor \frac{N}{2} \rfloor}$, if $c > x_{\lceil \frac{N}{2} \rceil}$, consider $x_{\lceil \frac{N}{2} \rceil + 1}, \dots x_N$

Let $n_2 = \lfloor \frac{N}{2} \rfloor$.

By induction, $n_j = \lfloor \frac{N}{2^{j-1}} \rfloor$

We stop when we either find the value c or we're down to just 1 element.

$\implies$ total runtime is the a matter of how many steps (dividing N by 2 each time).

$J =?$ s.t. $n_J = 1 = \lfloor \frac{N}{2^{J-1}} \rfloor \implies 0 = \log_2 N + (-J + 1)$ or

$$J = \log_2 N$$

When you see a problem where number of elements $n_0 = N, \dots n_j$ gets halved each time, it'll likely be $O(\log_2 N)$ runtime.

Finding an element in a **balanced binary search tree**: $O(\log N)$; with each comparison, we go either left or right.

26.6. **Recursive run times $O(2^N)$.**

$$\begin{array}{lcll} \text{Let } n_1 = N & & 2^1 & f(n_1 - 1) = f(N-1) \text{ calls} \\ \quad n_2 = N - 1 & & 4 = 2^2 & \quad f(n_2 - 1) \text{ calls} \\ \quad \dots & \mapsto & \dots & \\ \quad n_j = N - j + 1 & & 2^j & \quad f(n_j - 1) \text{ calls} \\ \quad \text{until } j = N & & 2^N & \end{array}$$

$$\sum_{j=1}^{N} 2^j = \sum_{j=0}^{N-1} 2^{N-j} = 2^N \sum_{j=0}^{N-1} 2^{-j} = 2^N \left(\frac{1 - 2^{-N}}{1 - 1/2} \right) = 2^{N+1}(1 - 2^{-N}) \cong 2^N \qquad (N \text{ large})$$

$$\implies 2^{N+1} - 1 \text{ nodes}$$

Try to remember this pattern. When you have a recursive function that makes multiple calls, runtime will often (not always) look like

$$\boxed{O(\text{branches}^{\text{depth}})} \tag{30}$$

branches = number of times each recursive call branches.

space complexity is $O(N)$, only $O(N)$ nodes exist at any time.

cf. VI Big O, pp. 46, McDowell, Example 1

```
for (int i=0; i < N; i++)
{ sum += array[i]; }
```

$O(N)$.

cf. VI Big O, pp. 46, McDowell, Example 2

```
for (int i = 0; i < N; i++)
{
for (int j = 0; j < N; j++)
{
std::cout << i << j;
}
}
```

$O(N * N) = O(N^2)$ Or see it as printing $O(N^2)$ total number of pairs.

cf. VI Big O, pp. 46, McDowell, Example 3

```
for (int i = 0; i < N; i++)
{
for (int j = i + 1; j < N; j++)
{
std::cout << i << j;
}
}
```

$$i = 1, 2 \dots N$$
$$j = i + 1, \dots N$$
$$\sum_{j=i+1}^{N} 1 = N - i$$

$$\sum_{i=1}^{N}(N - i) = N \cdot N - \sum_{i=1}^{N} i = N^2 - \frac{N(N+1)}{2} = N^2 - \frac{N^2}{2} - \frac{N}{2} = \frac{N^2 - N}{2} \cong N^2 \quad \text{(large)}$$

cf. VI Big O, pp. 46, McDowell, Example 4

```
for (int i = 0; i < NA; i++)
{
for (int j = i + 1; j < NB; j++)
{
if (a[i] < b[j])
{
std::cout << a[i] << b[j];
}
}
}
```

if statement within j's for loop is $O(1)$ time since it's just a sequence of constant time statements.

$\implies O(NA \cdot NB)$

cf. VI Big O, pp. 47, McDowell, Example 5

```
for (int i = 0; i < NA; i++)
{
for (int j = i + 1; j < NB; j++)
{
for (int k = 0; k < 1000000; k++)
{
//...
}
```

```
}
}
```

$\implies O(NA \cdot NB \cdot 1000000) = O(100000(NA)(NB)) \cong O(NA \cdot NB)$
100,000 units of work is still constant, so run time is $O(NA \cdot NB)$.
cf. VI Big O, pp. 48, McDowell, Example 6

```
void reverse(int array[], const int N)
{
for (int i = 0; i < N / 2; i++)
{
int other = N - i - 1;
int temp = array[i];
array[i] = array[other];
array[other] = temp;
}
}
```

$O(N)$ time. The fact that it only goes through half of the array (in terms of iterations) doesn't impact big O time.
cf. VI Big O, pp. 47, McDowell, Example 8
Let the longest string be of length L.
sort each string $\rightarrow L \log L$ (merge sort or best "worst" case for a sort)
N_a = number of strings in the array of strings.
$N_a L \log L$ = total number of sorts.

Sort the full array.
You should also take into account that you need to compare the strings.
Each string comparison takes $O(L)$ time (compare each string element at each position).
$O(N_a \log N_a)$ comparisons (sort the full array of strings)

$$\implies \boxed{O(N_a L \log L) + O(L N_a \log N_a) = O(N_a L (\log L N_a))}$$

26.7. **The master method for solving recurrences.** cf. 4.5 "The master method for solving recurrences", Cormen, Leiserson, Rivest, and Stein (2009) [19].
The master method proves a "cookbook" for solving recurrences of form

$$T(n) = aT(n/b) + f(n) \tag{31}$$

cf. Eqn. (4.20) of Cormen, Leiserson, Rivest, and Stein (2009) [19], where constants $a \geq 1, b > 1$, and $f(n)$ asymptotically positive function.
Consider memorizing 3 cases.
Recurrence Eqn. 31 describes running time of algorithm that divides problem size n into a subproblems, each of size n/b, where a, b positive constants.
The a subproblems are solved recursively, each in time $T(n/b)$. $f(n)$ encompasses cost of dividing problem and combining results of subproblems. e.g. recurrence from Strassen's algorithm has $a = 7, b = 2, f(n) = \Theta(n^2)$.
For technical correctness, n/b might not be an integer, so recurrence not well-defined.

Theorem 2 (Master Theorem, Thm. 4.1 of Cormen, Leiserson, Rivest, and Stein (2009) [19])**.** *Let constants $a \geq 1, b > 1$, let $f(n)$ be a function, let $T(n)$ be defined on nonnegative integers by recurrence:*

$$T(n) = aT(n/b) + f(n)$$

where n/b mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.
Then $T(n)$ has following asymptotic bounds:

1. *If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$*
2. *If $f(n) = \Theta(n^{\log_b a})$; then $T(n) = \Theta(n^{\log_b a} \log n)$*
3. *If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, if and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$*

In case 1, $n^{\log_b a}$ is larger, then $T(n) = \Theta(n^{\log_b a})$
In case 3, if $f(n)$ larger, then $T(n) = \Theta(f(n))$
In case 2, the 2 functions are the same size, multiply by logarithmic factor, solution is

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(f(n) \log n)$$

27. Data Structures

27.1. **Lists (arrays), Python Lists.** cf. https://wiki.python.org/moin/TimeComplexity, from 4. Quiz: Python Lists, Lesson 2: List-Based Collections, Udacity, Data Structures and Algorithms in Python

Operation	Average	Amoritized Worst Case
Copy	$O(N)$	$O(N)$
Append(1)	$O(1)$	$O(1)$
pop last	$O(1)$	$O(1)$
insert	$O(N)$	$O(N)$
Get item	$O(1)$	$O(1)$
Set item	$O(1)$	$O(1)$
Delete item	$O(N)$	$O(N)$
Get slice	$O(k)$	$O(k)$

Insert and delete is $O(N)$. Need to move every element with index greater than k, and reindex each element.

Get array length is $O(1)$. cf. https://stackoverflow.com/questions/21614298/what-is-the-runtime-of-array-length

27.2. **Linked Lists, $O(1)$ insertion.**

27.2.1. *Linked Lists vs. arrays.* main difference: Linked lists and arrays store difference information in each element.

Both cases: each element stores a value.
Both cases: stores 1 more information.

Array: index as a number.
Linked List: reference to next element, e.g. store addresses of next element, e.g.

$x0123$
e.g.

$x0122$	$x0122$
value : 8	value : 8
next : $x0123$	next : nullptr

Trick to remember: Given elements at addresses 1, 2, 3. If you delete the next reference for 1 (to 2), and replace it with a new object, you'll lose your reference to 3.
Assign your next reference to object at 2 before assigning next reference to 1.

Insertion takes constant time since you're just shifting around (constant, finite number of) pointers, instead of iterating every element of the list.

Doubly linked list: Also have pointers to previous element, e.g.

1	2	3
value:8	value:2	value:6
next:2	next:3	next:4
prev:0	prev:1	prev:2

cf. https://en.wikipedia.org/wiki/Linked_list
Disadvantages of Linked Lists:

- Use more memory than arrays because storage used by their pointers
- Nodes in linked list must be read in order from beginning: inherently sequential access
- nodes stored in continguously (possibly)
- difficult to reverse traverse, doubly linked list helps, but memory consumed in allocating space for back-ptr

Linked lists vs. dynamic arrays:
List data structures comparison

	Linked list	Array	Dynamic Array	Balanced Tree
indexing	$O(N)$	$O(1)$	$O(1)$	$O(\log N)$
insert/delete at beginning	$O(1)$		$O(N)$	$O(\log N)$
insert/delete at end	$O(1)$ when last element is known $O(N)$ when last element is unknown		$O(1)$ amoritized	$O(\log N)$
insert/delete in middle	search time $+O(1)$		$O(N)$	$O(\log N)$
wasted space (average)	$O(N)$	0	$O(N)$	$O(\log N)$

head of list is 1st. node.
tail of 1st. last node in 1st. (or rest of list)

cf. pp. 237-238 of Ch. 10 "Elementary Data Structures" of Cormen, Leiserson, Rivest, and Stein (2009) [19].

Given element x in the list,
if `x.next` = NIL, element x has no successor and is therefore the last element or **tail** of the list.

In a circular list, the prev pointer of the head of the list points to tail, and next pointer of the tail of the list points to the head.

27.2.2. *Inserting into a Linked List, $O(1)$ in front.* Given an element x whose key attribute has already been set, LIST-INSERT "splices" x onto front of linked list.

27.2.3. *Deleting from a Linked List, O(1) if given pointer, or O(N) if given key.* LIST-DELETE removes element x from linked list L.
It must be given a pointer to x, and it then "splices" x out of the list by updating pointers. O(1).

If we wish to delete an element with a given key, we must first call LIST-SEARCH to retrieve a pointer to the element. O(N) because we need to do a search.

27.2.4. *Sentinel.* Code for LIST-DELETE would be simpler if we can ignore boundary conditions at head and tail of the list:

LIST-DELETE'(L, x)
x.prev.next = x.next
x.next.prev = x.prev

instead of

if x is not head,
x.prev.next = x.next
else
L.head = x.next
if x is not tail
x.next.prev = x.prev

Sentinel is a dummy object that allows us to simplify boundary conditions.
Wherever we have NIL in the code, replace it by sentinel L.nil.
e.g. changes doubly linked list into a circular, doubly linked list with sentinel.
sentinel L.nil lies between head and tail.
attribute L.nil.next points to head of the list,
L.nil.prev points to tail.
Similarly, *next* attribute of tail and *prev* attribute of head point to L.nil

Since L.nil.next points to head, we can eliminate attribute L.head altogether.

Exercise 10.2-1, Cormen, Leiserson, Rivest, and Stein (2009) [19].
Can you implement the dynamic-set operation INSERT on a singly linked list in $O(1)$ time? How about DELETE?
cf. 3.05.`Linked_lists.pptx` of Harder (2018) [17], assign to the head a new node with the target value and pointing to the previous head, for its next attribute/field.
To delete, assign a temporary pointer to point to the node being deleted. Then assign head to its next attribute/field. Then delete the temporary pointer that pointed to the original head.

Exercise 10.2-2, Cormen, Leiserson, Rivest, and Stein (2009) [19].
Implement a stack using a singly linked list L. The operations PUSH and POP should still take $O(1)$ time.
PUSH is exactly push front for a linked list. POP is exactly pop front for a linked list. To repeat, push front is assign to head a new node with target value

and next attribute/field pointing to original head. pop front is first check if it's empty. If not, then assign a temporary pointer to the head node being deleted. Then assign to head its next. Then delete what the temporary pointer points to.

Exercise 10.2-3, Cormen, Leiserson, Rivest, and Stein (2009) [19].

Implement a queue by a singly linked list L. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time.

Keep a pointer to last element in linked list, as well as the head pointer.
To enqueue, insert element after last element of the list, and set it to new, last element (tail).
To dequeue, pop front, i.e. delete first element of list and return it.

Exercise 10.2-7, Cormen, Leiserson, Rivest, and Stein (2009) [19].

Given a $\Theta(n)$-time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

current = L.head.next
previous = L.head
while current $\neq$ nullptr,
next = current.next
current.next = previous (does the actual reversing)
previous = current
current = next
L.head = previous

cf. https://sites.math.rutgers.edu/~ajl213/CLRS/Ch10.pdf

Problem 10-1, Comparisons among lists, Cormen, Leiserson, Rivest, and Stein (2009) [19].

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, do
SEARCH(L, k)	N	N	N	
INSERT(L, x)	1	1	1	
DELETE(L, x)	N	N	1	
SUCCESSOR(L, x)	N	1	N	
PREDECCESSOR(L, x)	N	N	N	
MINIMUM(L)	N	1	N	
MAXIMUM(L)	N	N	N	

27.3. **Implementing pointers and objects for data structures.** cf. Sec. 10.3 "Implementing pointers and objects" in Cormen, Leiserson, Rivest, and Stein (2009) [19].

27.3.1. *A multiple-array representation of objects for linked lists.* 3 arrays, array *key* holds values of keys currently in the dynamic set.
pointers in arrays *next* and *prev* (notation: *previous*).
Given array index x (notation: i), key[i], next[i], previous[i] represents an object in the linked list.

27.3.2. *Allocating and freeing objects; example of doubly linked list represented by multiple arrays.* Suppose arrays in multiple-array representation have length m, and at some moment, dynamic set contains $n \leq m$ elements.
Then n objects represent elements currently in the dynamic set,
remaining $m - n$ objects are **free**.

Keep free objects in a singly linked list, called **free list**.

Free list uses only next array, which stores next pointers (i.e. index to next element).

head of free list held in global variable free.

free list acts like a stack.

Allocate-object():
if free == nil,
error "out of space"
else x = free
free = x.next
return x

free-object(x)
x.next = free
free = x

27.4. **Stack, $O(1)$ pop, $O(1)$ push.** "Stack Details", Udacity, Data Structures and Algorithms

top element $\mapsto$ Linked List head
push top element, pop top element. Last in, first out.
Last element you put in (new head), is the First out (when you pop).

27.5. **Queue, $O(1)$ insert, $O(1)$ delete, $O(N)$ search, $O(N)$ space complexities.** https://en.wikipedia.org/wiki/Queue_(abstract_data_type) Lesson 2: List-Based Collections, 11. Queues

Oldest element (tail) comes out first.

cf. pp. 121, Sec. 1.3 Bags, Queues, and Stacks, Sedgewick and Wayne (2011) [20]

First In, First Out.
first element you put in (tail) is the first out (when you pop).

Applications for Queues: cf. Harder (2018) [17], 3.03.Queues.pptx

- Queuing clients in a client-server model
- Breadth-first traversals of trees, i.e. *use queues for breadth-first traversals of trees*

27.5.1. *Queue as Circular Array.* cf. pp. 234, Ch. 10 Elementary Data Structures of Cormen, Leiserson, Rivest, and Stein (2009) [19]

`Q.head` attribute that indexes, or points to, its head.
`Q.tail` indexes next location at which newly arriving element will be inserted.
elements reside in `Q.head`, `Q.head` $+1, \ldots$ `Q.tail` $- 1$. When `Q.head` = `Q.tail`, queue is empty.

Figure 10.2 of Cormen, Leiserson, Rivest, and Stein (2009) [19] shows one way to implement a queue of **at most** $n-1$ **elements** using an **array** $Q[1 \dots n]$. This is an important point: see Stack overflow.

27.5.2. *Queue as Linked List.* head. "oldest element in the queue", "first"
tail. "newest element in queue", "last",
(I guess it grows from the tail)
add element to tail, enqueue.
Dequeue, remove head.
peek - peek at head.

Save references to head and tail.

Exercise 10.1-2, Cormen, Leiserson, Rivest, and Stein (2009) [19]. *Explain how to implement 2 stacks in 1 array $A[1 \dots n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in $O(1)$ time.*

The first stack starts at 1 and grows up towards n, while the second stack starts from n and grows down towards n. Stack overflow happens when an element is pushed when the 2 stack pointers are adjacent.

Exercise 10.1-6, Cormen, Leiserson, Rivest, and Stein (2009) [19]. *Show how to implement a queue using 2 stacks. Analyze the running time of the queue operations.*

For a queue q, recall a queue follows LIFO order. Given 2 stacks, s1, s2, recall stacks are FIFO.

The following are the solutions (implementation):
enqueue - enqueue pushes an element into s2. Check if s1 empty.
dequeue - dequeue pops elements from s1. Check if s1 empty.
If at any time, s1 is empty, then pop all elements in s2, into s1 so that the order is reversed.

Case 0. q empty, so s1 and s2 empty, enqueue element x1. Check if s1 empty then pops x1 into s1. Dequeue pops element x1.

Assume n1, n2 cases for s1, s2. Enqueue pushes x into s2. s1 not empty and still in LIFO order from assumption. Dequeue pops x1 from assumption. If n1 = 1, then n2 + 1 elements from s2 will be popped into s1 in reversed or LIFO order.

Average case time complexity for both is O(1). Worst case time complexity is O(N) because of the transfer from s2 to s1 to maintain LIFO order.

Exercise 10.1-7, Cormen, Leiserson, Rivest, and Stein (2009) [19]. *Show how to implement a stack using 2 queues. Analyze running time of stack operations.*

For a stack s, given 2 queues, choose 1 queue to be active q_a, and another to be not active q_{na}.

push: given element x, enqueue into q_a. O(1) time.

Because queues when popping or pushing "preserves order", the only way to get the last element in is to dequeue until the very last element.

pop: dequeue q_a until last element is reached (can check if empty afterwards). Then return that last element. Also, enqueue previous elements into q_{na} and label $q_{na} \mapsto q_a$ to be active.

27.5.3. *Deque. double-ended queue.* One can enqueue and dequeue from either end.

27.5.4. *Priority Queue.* . Assign each element in queue with a priority. Remove oldest and highest priority element first.

27.6. **STL Containers.** cf. Ch. 31, STL Containers, Bjarne Stroustrup (2013) [10]

pp. 886 Stroustrup (2013) [10] `std::vector<T, A>` contiguous allocated sequence of `T`s;

cf. pp. 888 Sec. 31.2.1 "Container Representation", [10]

`vector` element data structure is most likely an array:

`vector`: rep ← elements — free space

`list` : likely represented by sequence of links pointing to elements and number of elements; doubly-linked list of `T`; use when you need to insert and delete elements without moving existing elements.

`map` : likely implemented as (balanced) tree of nodes pointing to (key, value) pairs:

`unordered_map` likely implemented as hash table

`unordered_map` rep ← hash table ←, ←, ... (k, v), (k, v) , ...

`string` for short strings, characters are stored in the string handle itself, for longer strings elements are stored contiguously on free-store (like `vector` elements). Like `vector`, `string` can grow into "free space" allocated to avoid repeated reallocations.

`string`: rep ← characters – free space

Like a built-in array, `array` is simply sequence of elements, with no handle, `array`: elements

cf. pp. 894, Sec. 31.3, Operations Overview, Stroustrup (2013) [10]

Standard Container Operation Complexity

	`[]` Sec. 31.2.2	List Sec. 31.3.7	Front Sec. 31.4.2	Back Sec. 31.3.6	Iterators Sec. 33.1.
`vector`	const	$O(n)$+		const +	Ran
`list`		const	const	const	Bi
`forward_list`		const	const		For
`deque`	const	$O(N)$	const	const	Ran
`stack`				const	
`queue`			const	const	
`priority_queue`			$O(\log(n))$	$O(\log(n))$	
`map`	$O(\log(n))$	$O(\log(n))$+			Bi
`set`		$O(\log(n))$+			Bi
`unordered_map`	const+	const+			For
`unordered_set`		const+			For
`string`	const	$O(n)$+	$O(n)$+	const+	Ran
`array`	const				Ran

Ran - random-access iterator, "For" - "forward iterator", "Bi" - "bidirectional iterator"

27.7. **vector.**

27.7.1. *vector and growth.* Layout of `vector` : elem ← "front" of elements, space ← "front" of extra space (meets end of elements)

last ← end of extra space
alloc

Use of both size (number of elements), and capacity (number of available slots for elements without reallocation) makes growth through `push_back()` reasonably efficient: there's not an allocation operation each time we add an element, only every time we exceed capacity (Sec. 13.6)
adding half the size is common, standard doesn't specify by how much capacity is increased

27.7.2. *vector and Nesting.* `vector` (and similarly contiguously allocated data structures) has 3 major advantages:

- elements of `vector` compactly stored; no per-element memory overhead (contiguous on memory address)
  amount of memory consumed by `vec` of type `vector<X>` roughly `sizeof(vector<X>)+vec.size()*sizeof(X)`; `sizeof(vector<X>)` is about 12 bytes, insignificant for large vectors
- fast traversal, consecutive access, to get to next element, code doesn't have to indirect through a pointer
- simple and efficient random access, makes sort and binary search efficient

vs. doubly-linked list, `list`, incurs 4-words-per-element memory overhead (2 links plus free-store allocation header)

be careful don't unintentional compromise efficiency of access, e.g. 2-dim. matrix don't do `vector<vector<double>>`,
do `vector<double>` and compute locations from the indices

27.7.3. *vector vs. array.* `vector` resource handle, i.e. allows it to be resized and enable efficient move semantics,
disadvantage to arrays that don't rely on storing elements separately from handle; keeps sequence of elements on stack or in another object.

27.8. **Trees.** value, left pointer, right pointer,

levels - how many connections it takes to reach the root +1 child can only have 1 parent height - number of edges between it and farthest leaf
depth - number of edges to root.

height, depth are inverse

27.8.1. *Rooted Trees representation.* cf. Sec. 10.4 "Representing rooted trees", Cormen, Leiserson, Rivest, and Stein (2009) [19]

Assume each node contains a key attribute.

27.8.2. *Binary trees representation.* p, left, right store pointers to parent, left child, right child.
if x.p = NIL, then x is root
if x.left = NIL, node x has no left child.
if x.right = NIL, node x has no right child.

root of entire tree T is printed to by attribute T.root

27.8.3. *Rooted trees with unbounded branching.* If (class of) tree has number of children of each node is at most some constant k.

Replace the left, right attributes by child1, child2, ... childk

For arbitrary numbers of children, **left-child, right-sibling representation**
each node has parent pointer p
T.root points to root of tree T
each node has only 2 pointers
1. x.left-child points to left most child of node x
2. x.right-sibling points to sibling of x immediately to its right

If node x has no children, then x.left-child =NIL
If node x is rightmost child of its parent, then x.right-sibling = NIL

Exercise 10.4-2, Cormen, Leiserson, Rivest, and Stein (2009) [19].

Write an $O(n)$-time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree.

cf. https://sites.math.rutgers.edu/~ajl213/CLRS/Ch10.pdf

(this is for pre-order) print-tree(node)
if node == NIL, return
else
print node.key
print-tree(node.left)
print-tree(node.right)

27.8.4. *Preorder, postorder, inorder traversal, recursive.* To reiterate from wikipedia, "Tree Traversal",

preorder traversal

(1) visit current node
(2) recursively traverse left subtree
(3) recursively traverse right subtree

postorder traversal

(1) recursively traverse left subtree
(2) recursively traverse right subtree
(3) visit current node

inorder traversal

(1) recursively traverse left subtree
(2) visit current node
(3) recursively traverse right subtree

Exercise 10.4-3, Cormen, Leiserson, Rivest, and Stein (2009) [19].

Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

printWithStack(node)
let S by an empty stack.

push(S, node)
while S is not empty,
U = pop(S)
U $\neq$ NIL, then
print U.key
push(S, U.left)
push(S, U.right)

Keep in mind that this *does not* reproduce inorder recursive traversal. This instead exhibits interesting behavior of immediately printing a node. It's because recursion relies on a termination condition, which is typically that a node is a leaf.

27.8.5. *Postorder Traversal with stack.* Consider the following from geeksforgeeks, iterative postorder:

Create an empty stack.
2.1 while root is not NIL,
push root's right and then root to stack,
set root as root's left child.
2.2 pop item from stack and set as root
if popped item has right child, and right child is top of the stack, then remove right child from stack, push root back and set root as root's right child
else print root's data and set root as nullptr
2.3 repeat steps 2.1, 2.2 while stack isn't empty.

Consider also the following:

Create empty stack S, push(S, node)
currentptr=node, previousptr=NIL
while S is not empty,
currentptr = S.top()
if currentptr is a leaf or we've traversed right node of currentptr or left node of currentptr,
print currentptr.value
S.pop()
previousptr = currentptr
else
if currentptr.right is not NIL, S.push(currentptr.right)
if currentptr.left is not NIL, S.push(currentptr.left)

The predicate "we've traversed right node of currentptr or left node of currentptr" is to reproduce the recursive stack callback.

27.9. **Graphs.** edges can store data too.
Directed graph edges have a sense of direction.
A = set of ordered pairs of vertices.
undirected graph.
acyclic (no cycles)
DAG Directed graph with no cycles.

27.9.1. *Connectivity.* connected graph has only 1 connected component. $\exists$ path $\forall$ pair of vertices.
minimum number of elements (edges) to remove, to disconnect a component.

weakly connected directed graph if replacing all its directed edges with undirected edges produces a connected (undirected) graph.

27.9.2. *Graph Representation.* vertex object:
list of edges.
Edge Object
vertices.
Edge List: $= E$
Adjacency list $l = l(i)$ s.t. $\forall\, i \in V$, $l(i) \in$ **Set** s.t. $l(i) =$ set of all adjacent vertices.

Adjacent Matrix.
Let $V =$ set of all vertices.
$\forall\, v \in V$, label them: $v = v(i)$, $i \in \mathbb{N}$

$\forall\, v_i \in V$, so $\forall\, i = 0, 1, \ldots N-1$,
Consider $\forall\, w = w_j \in V$, so $\forall\, j = 0, 1, \ldots N-1$.

If v_i, w_j are adjacen, let $a(i,j) = 1$, otherwise $w(i,j) = 0$. (adjacent means $\exists$ edge s.t. $E = \{v_i, w_j\}$)
$\Longrightarrow a$ is an adjacency matrix.
Adjacency list.
$\forall\, v = v_i \in V$, $i = 0, 1, \ldots N-1$
$E = \{(v_o, v_t) | v_o, v_t \in V\}$
if $\exists\, e \in E$ s.t. $v_i = e(0) = v_o$, then $e(1) = v_t$ is adjacent to it.
Adjacency list useful for counting number of edges that a node has or number of adjacencies.

27.9.3. *Graph Traversal.* Depth-first search (DFS)
Breadth-first search (BFS)

DFS.
implementation: stack.
keep 2 structures: 1. Seen list, 2. stack.
If seen before, pop stack and go back.
$O(|E| + |V|)$ visit every edge twice $O(2|E| + |V|) = O(|E| + |V|)$.
$O(|V|)$ time to look up a vertex.
More on procedure:
begin with any node v_1. Put $v(v_1)$ into seen. Put v_1 on stack.
if $\exists\, \{e\} \subset E$ s.t. for $e = (v_o, v_t)$, $v_o = v_1$, then, put $v(v_t)$ into seen. Put v_t on stack.
Then consider v_t.
If $v(v_t) \in$ seen, consider another edge.
If you run out of edges with new node, pop stack.
Eulerian path $O(|E|)$

Hamiltonian path

27.10. **Tries.** Leaves of tree indicate the end of a word.

Each node contains a character of a word. Each branch (path) from child from root to leaf is a word in a collection of words.

class Nodes { array of all children or
lookup table to do you have child?
e.g. HashMap

is end of word

Used in word validation. Given this list of strings, word list validation. is CA prefix.

Keep state, build on a lookup by keeping state in trie or return by reference to the Node.

Data Structures: Tries, HackerRank

Insert and search costs O(key length) however, memory requirements of Trie is O(Alphabet size * key length * N) where N is number of keys in Trie.

28. Hash Tables

28.1. **Direct-address tables.** Direct addressing works well when universe U of keys is small.
Suppose dynamic set in which each element has key drawn from universe $U = \{0, 1 \dots m-1\}$ m not too large.
Assume no 2 elements have same key.
To represent dynamic set, use array, or **direct-address table**, $T[0, \dots m-1]$ in which each position, or **slot**, corresponds to key in universe U.
Slot k points to element in set with key k.
If set contains no element with key k, then T[k] = NIL

Direct-Address-Search(T, k)
return T[k]

Direct-Address-Insert(T, x)
T[x.key] = x

Direct-Address-Delete(T,x)
T[x.key] = NIL

O(1) time.

Exercise 11.1, Cormen, Leiserson, Rivest, and Stein (2009) [19].

Start at i = m-1. Consider T[i]. $\forall\, m-1, m-2, \dots 0$, until T[i] is an element O(m).

Exercise 11.1-2, Cormen, Leiserson, Rivest, and Stein (2009) [19].

*A **bit vector** is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in O(1) time.*

Let x be a bit vector of length m. Let dynamic set of n elements s.t. $n \leq m$ of distinct elements with key values from $0, 1, \ldots n-1$.

insert(k)
x[k] = 1

Delete(k)
x[k] = 0

search(k)
return x[k]

cf. 11.2 "Hash Tables" of Cormen, Leiserson, Rivest, and Stein (2009) [19]

Let K of keys stored in dictionary much smaller than universe U of all possible keys, we can reduce storage requirements to $\Theta(|K|)$, while search requires only O(1) time.

Given an element with key k,
direct addressing: k stored in slot k
hashing: element stored in slot H(k), i.e. **hash value** h(k) of key k

hash function $h : U \to \{0, 1, \ldots m-1\}$ s.t. T[h(k)] = k
$h : k \mapsto h(k)$

where size m of hash table typically $m < |U|$

collision: 2 keys may hash to same slot.

28.1.1. *Collision resolution by chaining for Hash Tables.* **chaining-** place all elements that hash to same slot into same linked list.
slot j contains pointer to head of list of all stored elements that hash to j
if there are no such elements, slot j contains NIL.

Dictionary operations:
Chained-Hash-Insert(T,x)
insert x at head of list T[j(x.key)]

Chained-Hash-Search(T, k)
search for element with key k in list T[h(k)]

Chained-Hash-Delete(T, x)
delete x from list T[h(x.key)]

For searching, worst-case running time is O(N).
Delete an element in O(1) time if list doubly linked. If lists only singly linked, then first have to find x.

28.1.2. *Analysis of hashing with chaining.* Given hash table T with m slots, stores n elements, define **load factor** $\alpha := \frac{n}{m}$, i.e. average number of elements stored in chain.

simple uniform hashing - assumption that any given element is equally likely to hash into any of m slots.

$\forall\, j = 0, 1, \ldots m-1$, let length of list T[j] $\equiv n_j$, so that

$$n = n_0 + n_1 + \cdots + n_{m-1}$$

expected value of $n_j \equiv E[n_j] = \alpha = \frac{n}{m}$

Theorem 3 (11.1, Cormen, Leiserson, Rivest, and Stein (2009) [19])**.** *In hash table in which collisions are resolved by chaining, unsuccessful search takes average case time* $\Theta(1+\alpha)$*, under assumption of simple uniform hash.*

Proof. By simple uniform hashing assumption, any key k not already stored is equally likely to hash to any m slots.

Expected time to search unsuccessfully for key k is expected time to search to end of list T[h(k)], which has expected length $E[n_{h(k)}] = \alpha$.

Thus expected number of elements examined is unsuccessful search is α.
$\implies$ take time required $\Theta(1+\alpha)$ □

Theorem 4 (11.2, Cormen, Leiserson, Rivest, and Stein (2009) [19])**.** *In a hash table in which collisions resolved by chaining, successful search takes average-case time* $\Theta(1+\alpha)$*, under simple uniform hashing assumption.*

Proof. Assume element being searched equally likely to be any of the n elements stored in table.
number of elements examined during successful sesarch for x is 1 more than number of elements that appear before x in x's list.
Let $x_i \equiv i$th element inserted into table, $i = 1, 2, \ldots n$
$k_i \equiv x_i$.key

$\forall$ keys k_i, k_j, define indicator random variable $X_{ij} = I\{h(k_i) = h(k_j)\}$
under simple uniform hashing assumption, $Pr\{h(k_i) = h(k_j)\} = \frac{1}{m}$, because probability to hash into any slot is $\frac{1}{m}$ and so probability to hash into this particular slot is also $\frac{1}{m}$ since it must be as equally likely as any other.

$E[X_{ij}] = \frac{1}{m}$ since Lemma 5.1 of Cormen, Leiserson, Rivest, and Stein (2009) [19] which is, recall, really simple: $E[X_{ij}] = 1 \cdot Pr\{h(k_i) = h(k_j)\} + 0 \cdot Pr\{h(k_i) = h(k_j)\}$.

Note that because new elements placed at front of list, elements before x in list inserted after x inserted.

Take average, over n elements in table, of 1 plus expected number of elements added to x's list after x was added to list.

Thus, expected number of elements examined in a successful search,

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{n}\right)=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)=$$
$$=1+\frac{n}{m}-\frac{1}{nm}\left(\frac{n(n+1)}{2}\right)=1+\frac{n-1}{2m}=1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Thus total time required for successful search is $\Theta\left(2+\frac{\alpha}{2}-\frac{\alpha}{2n}\right)=\Theta(1+\alpha)$ □

Exercise 11.2-1, Cormen, Leiserson, Rivest, and Stein (2009) [19].

cf. https://sites.math.rutgers.edu/~ajl213/CLRS/Ch11.pdf

Given n distinct keys, suppose keys are totally ordered $\{k_1,\dots k_n\}$. Given key k_i, consider $\forall\, l$ s.t. $l>k_i$ and $h(l)=h(k_i)$, $\quad Pr[h(l)=h(k)]=\frac{1}{m}$.

Probability that for some key there's a collision,

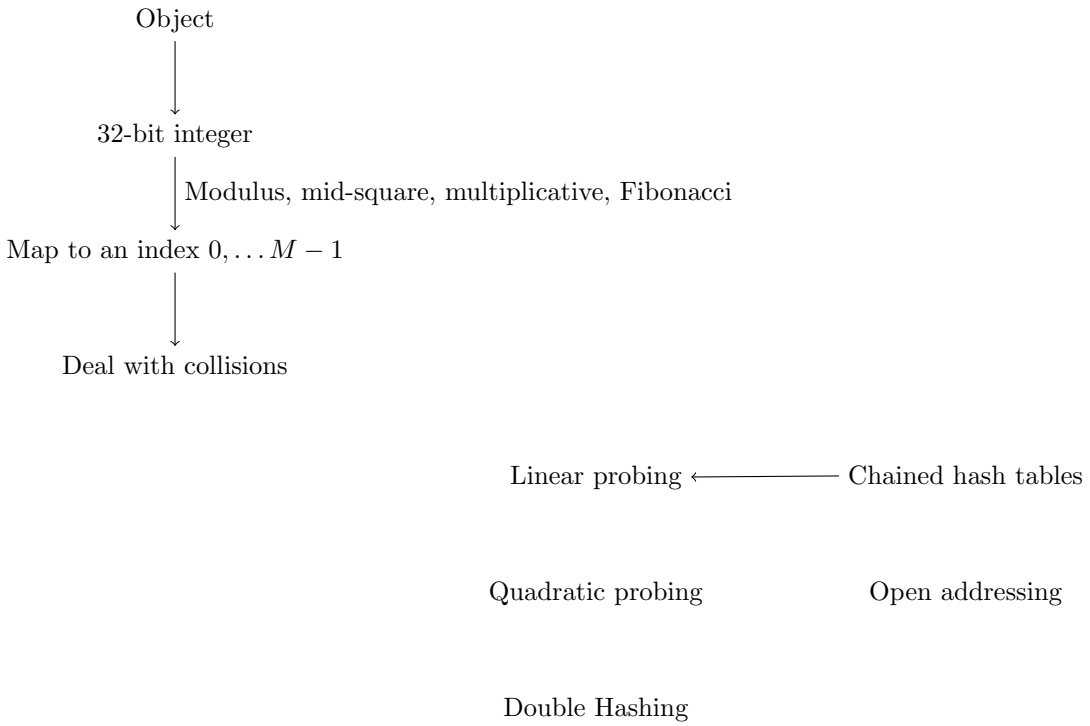
$$\sum_{i<j}^{n}Pr[h(k_j)=h(k_i)]=\sum_{j>i}\frac{1}{m}=\frac{n-i}{m}$$

Using simple uniform hashing assumption.

Expected number of collisions

$$\sum_{i=1}^{n}\frac{n-i}{m}=\frac{n^2-\frac{n(n+1)}{2}}{m}=\frac{n(n-1)}{m}$$

The hash process:



28.2. **Hash functions.** cf. 11.3 "Hash functions" of Cormen, Leiserson, Rivest, and Stein (2009) [19]

28.2.1. *What makes a good hash function?* A good hash function satisfies (approximately) assumption of simple uniform hashing: that each key equally likely to hash to any of m slots, independently of where any other key hashed to.
Unfortunately, probability distribution from which keys drawn unknown, and drawing itself might not be independent.

However, if we know keys are random real numbers k independently, uniformly distributed in range $0 \leq k < 1$, then

$$h(k) = \lfloor km \rfloor \tag{32}$$

satisfies condition of simple uniform hashing.

Create well-performing hash function:
e.g. consider compiler's symbol table, in which keys are character strings, representing identifiers in a program. Closely related symbols, e.g. pt and pts, often occur in same program.
- good hash function would minimize change such variants hash to the same slot.

A good approach derives hash value independent of patterns that might exist in data:
e.g. "division method"; key divided by prime number $\mapsto$ remainder; choose prime

number unrelated to problems in keys distribution.

example: want keys that are "close" to yield hash values far apart; e.g. universal hashing (11.3.3 of Cormen, Leiserson, Rivest, and Stein (2009) [19]).

28.2.2. *Interpreting keys as natural numbers for hash functions.* Most hash functions assume universe of keys is set $\mathbb{N} = \{0, 1, 2, \dots\}$ of natural numbers; thus if keys aren't natural numbers, find a way to interpret as natural numbers.
e.g. interpret a character string as integer expressed in suitable radix notation.
e.g. pt as p = 112, t = 116, radix-128 int $\mapsto (112 \cdot 128) + 116 = 14452$, i.e.

$$r(s) = \sum_{j=0}^{N-1} 128^j \cdot s[j]$$

28.2.3. *Division method.* map key k into m slots by taking remainder of k divided by m, i.e.

$$h(k) = k \mod m \tag{33}$$

m shouldn't be power of 2, since if $m = 2^p$, h(k) is just p lowest-order bits of k
Unless we know all low-order p bit patterns equally likely better off designing hash function to depend on all bits of key.
Exercise 11.3-3 (of Cormen, Leiserson, Rivest, and Stein (2009) [19]) shows choosing $m = 2^p - 1$ when k is character string interpreted in radix 2^p maybe poor choice, because permuting characters of k doesn't change hash value.

prime not too close to exact power of 2 often good choice for m
e.g. hash table, with collisions resolved by chaining, to hold n = 2000 character strings, where a character has 8 bits = 1 byte
We don't mind examining average of 3 elements in an unsuccessful search, so allocate hash table of size m = 701.
Choose m= 701 because it's a prime near 2000/3 but not near any power of 2

$$h(k) = k \mod 701 \tag{34}$$

28.2.4. *Multiplication method.*

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

where $0 < A < 1$, where $kA \mod 1$ means take the fractional part of kA, i.e. $kA - \lfloor kA \rfloor$

advantage of multiplication method: value of m is not critical.
typically choose m to be a power of 2 ($m = 2^p$ for some $p \in \mathbb{Z}$).

Suppose word size of machine is w bits, k fits into single word.

Restrict A to be fraction of form $s/2^w$, where s is an integer in range $0 < s < 2^w$.

So $k \times s = k \times A \cdot 2^w$,
where first multiply k by w-bit integer s = $A \cdot 2^w$.

Result is $2w$ bit value $r_1 2^w + r_0$ where r_1 is high-order word of product, r_0 is low-order word of product desired p-bit hash value consists of p most significant bits of r_0.

Although this method works with any value of constant A, optimal choice depends on characteristics of data being hashed. Knuth suggests

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887$$

e.g. $k = 123456$, $p = 14$ (14-bit has value), $m = 2^{14} = 16384$, $w = 32$ (32-bit word size).

Adapting Knuth's suggestion, choose $A = s/2^w$, and $A \approx (\sqrt{5} - 1)/2$, so $A = 2654435769/2^{32}$.
$k \cdot s = 327706022297664 = (76300 \cdot 2^{32}) + 17612864$, so $r_1 = 76300$, $r_0 = 17612864$.
14 most significant bits of r_0 yield $h(k) = 67$ since

$$17612864 \mapsto 0000000\,10000110011000\,000010\,00000$$

so 14 most significant bits of 32 are

$$00000\,001000011 = 67$$

28.2.5. *Universal hashing.*

29. Search

29.1. **Binary Search** $O(\log(N))$. https://classroom.udacity.com/courses/ud513/lessons/7123524086/concepts/71154040750923

$O(\log(N) + 1) = O(\log(N))$ Binary search efficiency.

Given array $a = a_i$, $i = 0, 1, \ldots N-1$, N elements in an array, s.t. $a(i) \leq a(j)$ if $i \leq j$, $i, j \in 0, 1 \ldots N-1$ (i.e. sorted array)
Given x to search for,
Consider m_j s.t.

For $j = 1$, given N, $m_1 := \begin{cases} \frac{N}{2} & \text{if } N \text{ odd} \\ \frac{N}{2} - 1 & \text{if } N \text{ even} \end{cases}$

$m_1 =$ midpoint to compare against.

if $x = a(m_1)$ done.
if $x < a(m_1)$, consider a_i s.t. $i = 0, \ldots m_1 - 1$
if $x > a(m_1)$, consider a_i s.t. $i = m_1 + 1 \ldots N - 1$

For j, given a_i, $i = l, \ldots r$ $l \leq r$, (l, r are included in the range),
Let $L_j := r - l + 1$

$$m_j := \begin{cases} \frac{L_j}{2} + l & \text{if } N \text{ odd} \\ \frac{L_j}{2} - 1 + l & \text{if } N \text{ even} \end{cases}$$

If $x = a(m_j)$ done,
if $x < a(m_j)$, consider a_i s.t. $i = 0, \ldots m_j - 1$
If $x > a(m_j)$, consider a_i s.t. $i = m_j + 1, \ldots N - 1$
Stop when $L_j = 0$.
cf. Pfenning (2016)

Given array A of N elements, $a_0, a_1, \dots a_{N-1}$, sorted, s.t. $a_0 \leq a_1 \leq \cdots \leq a_{N-1}$, target value T,
Pfenning (2016) defines variables l (`lo`), h (`hi`) s.t. subinterval in array to consider is from l to $h-1$. (i.e. subarray includes l and excludes h). $l \leq 0$, $h \leq N$ and $l < h$ (if $l \geq h$, contradiction since for $l = h$, l is included but h is excluded).

Given $l \geq 0$, $h \leq N$, $l < h$ (so subarray to consider is nonzero size).

Calculate midpoint, for $m \in \mathbb{Z}$, $m = l + \frac{h-l}{2}$. We don't use $n = \frac{h+l}{2}$ because of **possible overflow**.

$h - l =$ size of subarray to consider.
if $h - l$ even (i.e. $h - l = 2L_{h,l}$), m is the "left most" element of the "right" half.
if $h - l$ odd (i.e. $h - l = 2L_{h,l}$), m is the middle.

Suppose $\forall\, i = 0, 1, \dots l-1,\ a_i < T$, and $\forall\, i = h, h+1, \dots N-1,\ a_i > T$ i.e. $a_{l-1} < T$, and $a_h > T$.

Be careful, when accessing element of an array, *access must be in bounds*!

If $a_m < T$, let $l = m + 1$, if $a_m > T$, let $h = m$.

Exercise 2.3-5, Cormen, Leiserson, Rivest, and Stein (2009) [19].

Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The ***binary search*** *algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that worst-case running time of binary search is* $\Theta(\ln n)$

Iterative:

iterative-binary search(A, v, low, high)
while low $\leq$ high
mid = floor ((low + high) / 2) (gets highest value in left half; lies on "left side" of middle pt. if number of elements is even)
if v = A[mid]
return mid
else if v $\geq$ A[mid]
low = mid + 1
else high = mid - 1
return NIL

Recursive-binary search (A, v, low, high)
if low > high,
return NIL
mid = floor((low + high) / 2)
if v == A[mid]
return mid
else if v > A[mid]
return recursive-binary search(A, v, mid + 1, high)
else return recursive-binary search(A, v, low, mid - 1)

Time we do comparison of v with middle element, search range continues with range of elements halved.

Recurrence is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n/2) + O(1) & \text{if } n > 1 \end{cases}$$
$$\implies T(n) = \Theta(\log n)$$

29.1.1. *Binary Search for leftmost or rightmost occurrence.* Consider this question from leetcode, where we count the number of occurrences of a number in a sorted array.

To get the number of occurrences, it's easier to find the position of the left most appearance of the number and the position of the right most appearance of the number. Then we'll need a binary search that'll find such a position when the numbers are not unique in an array.

Consider finding the leftmost occurrence:
if (m= 0 or x > a (m - 1)) and a(m) = x,
(contrapositive (m ≠ 0 and x ≤ a(m-1)) or a(m) ≠ x
if a(m) < x
$\implies$ (m+1, h)
else
$\implies (l, m-1)$

Consider finding the rightmost occurrence:
if (m= n - 1 or x < a (m + 1)) and a(m) = x,
(contrapositive (m ≠ n - 1 and x ≥ a(m + 1)) or a(m) ≠ x
if a(m) > x
$\implies$ (l, m - 1)
else (a(m) < x or a(m+1) ≤ x and m ≠ n -1)
$\implies (m+1, h)$

29.2. **Breadth-first search (BFS).** https://en.wikipedia.org/wiki/Breadth-first_search

starts at tree root, or some arbitrary node, and explores all neighbor nodes at present depth prior to moving on to nodes at next depth level.

Worst-cast performance $O(|V| + |E|) = O(b^d)$, Worst-case space complexity $O(|V|) = O(b^d)$

https://www.quora.com/What-are-the-advantages-of-using-BFS-over-DFS-or-using-DFS-over-BFS

Pro:
1. Solution definitely found out by BFS, 2. never get trapped in blind alley, unwanted nodes, 3. if there are more than 1 solution, will find solution with shortest steps

Con:
1. Memory constraints: as it stores all nodes of present level to go for next level 2. if solution far away, consumes time

Application of BFS:
1. Find shortest Path, 2. Check graph with bipertiteness

29.3. **Depth-first search.** Pre-order - check off nodes as soon as you see it, before seeing children.

root check it off, pick first left child, ...

In-order.
check off node once left child is seen.
went left most to right, went through all nodes in order.

cf. Geeks for Geeks, Tree traversals (Inorder, Preorder and Postorder)
inorder traversal (left root right) is a depth first traversal. e.g.

$$\begin{array}{ccccc} & & 1 & & \\ & 2 & & 3 & \xrightarrow{\text{inorder traversal}} 4\,2\,5\,1\,3 \\ 4 & & 5 & & \end{array}$$

Algorithm for Inorder (tree)

(1) Traverse the left subtree, i.e., call Inorder(left-subtree)
(2) Visit the root.
(3) Traverse the right subtree, i.e., call Inorder(right-subtree)

29.3.1. *Uses of Inorder.* In case of binary search trees (BST), inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, variation of Inorder traversal where Inorder traversal is reversed can be used.

Post-order.
Check off leaf, don't check off parent, check right child.

Delete $O(N)$. insert.
When do we want to use these structures? pros and cons.

29.4. **Binary Search Tree (BST), $O(\log N)$ height of the tree, run-time complexity, worst case $O(N)$.** Given node x, functions l, r, v s.t. $l(x), r(x) \in$ **Nodes** $\cup \emptyset$, $v(x)$ is some ordered value, so $v(x) \in$ **OrderedSet**.

Binary Search tree has the condition that $\forall\, x$,

$$v(l(x)) \leq v(x) \leq v(r(x))$$

i.e., cf. LeetCode Binary Search Tree Overview, Introduction, value $\forall$ node must be greater than (or equal to) any values in its left subtree; but less than (or equal to) any value in its right subtree.

Like a normal binary tree, traverse binary search tree (BST) in preorder, inorder, postorder, level-order. Note, most frequently used is **inorder traversal** for BST.

cf. LeetCode, Binary Search Tree, Introduction to BST, Definition of the Binary Search Tree.

Given search value y,
if $v(x) = y$, done,
if $v(x) < y$, $x_1 = r(x)$
if $v(x) > y$, $x_1 = l(x)$

height of tree is $O(\log N)$, run-time complexity.
insert $O(\log(N))$
delete $O(\log(N))$

Unbalanced, distribution of nodes skewed, worse case $O(N)$ search, insert, delete
e.g. $5 \to 10 \to 15 \to 20$

Strength of BST is performing,
search, insertion, deletion in $O(h)$ time complexity (h = height of tree) even in *worst case*.

e.g. Design a class to find kth largest element in a stream.

cf. LeetCode, Introduction to Binary Search Tree - Conclusion.

If we use *array*, sort array in descending order and return kth element (that's one way).

But insert new element, must re-sort. Re-sorting required for $O(1)$ search (that's what we originally want).

Time complexity of insertion is $O(N)$ in average. Therefore time complexity will be $O(N^2)$ in total.

$\forall$ node, all values in right subtree are larger than value of node itself, while all values in left subtree are smaller than value of node.
Consider node x:

$$v(l(x)) < v(x) < v(r(x))$$

So while $v(l(r(x))) < v(r(x))$, but if we insert in accordingly, we could guarantee that $v(x) < v(l(r(x)))$ because $l(r(x))$ would not be the left node of x unless $v(x) < v(l(r(x)))$ was checked first.

If m nodes in the right subtree, node itself is $m+1$ largest for the current tree.

Consider counter $\forall$ node to indicate how many nodes there are in subtree rooted at this node.

29.5. **Heaps.** max-heap: parent must always have bigger value than its child.
min-heap: parent must always have smaller value than its children.
Unlike binarytree, heap can have any number of children.

Pro:
1. heap finds max, min (root) 2. heap data structure efficiently use graph algorithm, e.g. Dijkstra

Con:
1. takes more time to compare and execute.
heap = max. efficient implementation of priority queue.

Heap	find min	delete min	insert	decrease-value
Binary Heap	$O(1)$	$O(\log N)$	$O(\log N)$	$O(\log N)$

e.g. max binary heap.

"complete" all levels except last are completely full.
if not, continue adding values from left to right.

Search: $O(N)$ no guarantee child is $\lessgtr$. end up searching entire tree.

improved search: if $x >$ root, quit search (for max-heap)
In general, if v(node) = node value $< x$, no need to check children of node.

Search: worse cast: $O(N)$
Average case: $O(N/2) = O(N)$

29.5.1. *Heapify (Worst case $O(\log N)$).* stick new element in open spot of the tree. heapify - reorder tree based on heap property: $\forall$ given node C, if parent node P of C, $v(P) \geq v(C)$ (max heap)

Swap P and C when $v(C) > v(P)$
Extract; remove root. replace root with right most element. Then swap when necessary.

Heapify: worst case $O(\log N)$.
As many operations as height of the tree.

29.5.2. *Heap implementation (array).* max-binary heap: since we know how may children (2), $\forall$ parent,
Use some math to find next mode. If level l, nodes per level $l = 2^{l-1}$
Sorted array into tree.

Let sorted array be $a = a_i$ s.t.

$$a_0 \mapsto l = 1$$
$$a_1, a_2 \mapsto l = 2$$
$$a_3, a_4, a_5, a_6 \mapsto l = 3$$

$\left(\sum_{m=1}^{l-1} 2^{m-1}\right) - 1 = k$ index to start from.
Tree vs. Array
if tree uses nodes, need pointers: Arrays save space.

29.6. **Self-balancing tree.** Balanced minimize number of levels to use

29.7. **Red-Black Tree; search $O(\log N)$, space $O(N)$, insert $O(\log N)$, delete $O(\log N)$.** root black,
level 1 black
level 2 red
level 3 black

null leaf nodes must be colored black (i.e. all leaves are black)
Rule 4 (optional) root must be black.
Rule 5 every node; every path descending down must contain same number of black nodes. i.e. $\forall$ path from given node, to any of its descendant NIL nodes (leaf) must contain same number of black nodes.
red-black tree = self balancing binary search tree ($\forall$ node x, $v(l(x)) \leq v(x) \leq v(r(x))$)

29.7.1. *Red-Black Trees, insertion.* Insert Red Nodes Only.
runtime for insertion worst case $O(\log N)$. Binary search tree worse case $O(N)$ because BST could be unbalanced.

30. Recursion

cf. https://en.wikipedia.org/wiki/Recursion, Udacity
Recursion

- must call itself at some pt.
- base case

- alter input parameter

Theorem 5 (Recursion (from set theory))**.** *Given* $X \in \textbf{\textit{Set}}$, $a \in X$, $f : X \to X$
$\exists ! F : \mathbb{N} \to X$ *s.t.*
$F(0) = a$
$F(n+1) = f(F(n))$

e.g. $Fib(0) = 0$
$Fib(1) = 1$
$\forall n > 1,\ n \in \mathbb{Z},\ Fib(n) := Fib(n-1) + Fib(n-2) = f(Fib(n-1), Fib(n-2))$

cf. NCatlab

Let $\mathbb{N}$ be (parametrizable) natural number object in category with finite products with zero $0 : 1 \to \mathbb{N}$, and successor $s : \mathbb{N} \to \mathbb{N}$.
$\forall$ morphism $f_0 : Y \to Z$, (Y, Z are sets i.e. $Y, Z \in$ **Sets**), and $\forall$ morphism $h : \mathbb{N} \times Y \times Z \to Z$,
$\exists !$ morphism $f : \mathbb{N} \times Y \to Z$ s.t.

$$\begin{aligned} & f(0, -) = f_0 \text{ and} \\ & f(s(x), y) = h(s(x), y, f(x, y)) \end{aligned} \tag{35}$$

where $x : \mathbb{N} \times Y \to \mathbb{N}$ is the first projection and
$y : \mathbb{N} \times Y \to Y$ is the 2nd. projection.

From wikipedia on "Recursive" definition,
Let set A, $a_0 \in A$.
If $\rho : f \to A$ s.t.
f : nonempty section of positive integers $\to A$
the $\exists ! h : \mathbb{Z}^+ \to A$ s.t.

$$\begin{aligned} & h(1) = a_0 \\ & h(i) = \rho(h|_{\{1,2,\dots i-1\}}) \qquad \text{for } i > 1 \end{aligned} \tag{36}$$

cf. Neagole (2020) [22], "Algorithms:Recursion" - 151 Recursion Introduction,
- HackerRank, "Algorithms: Recursion", Gayle Laakmann McDowell
Bell, Grimson, Guttag (2016) [15]

30.1. **Time Complexity, Space Complexity.**

30.1.1. *Fibonacci,* $O(2^n)$ *time,* $O(n)$ *space (*n *stack frames).* For nth Fibonacci number, 2 branches as n decreases by 1 until base cases, or "leaves". n number of stack frames; once a base case is returned, pointer goes to previous stack frame, and then goes into next branch, and so there's at most n stack frames needed.

cf. Dynamic Programming - Learn to Solve Algorithmic Problems and Coding Challenges

30.2. **Binary Trees and Recursion.** cf. LeetCode, Solve Problems Recursively

30.2.1. *"Top-down" Solution.* "Top-down" means that in each recursive call, visit the node first to come up with some value, and pass these values to its children when calling the function recursively. So the "top-down" solution can be considered as a kind of *preorder* traversal.

For recursive function *topDown(root, params)*, it would work something like this:

(1) Return specific value for a null node
(2) Update answer if needed. (answer ¡- parameters)
(3) left answer = topDown(root left, left parameters) (left parameters ¡- root value, parameters)
(4) right answer = topDown(root right, right parameters) (right parameters ¡- root value, parameters)
(5) Return the answer if needed. (answer ¡- left answer, right answer)

30.2.2. *"Bottom-up" Solution.* "Bottom-up" - in each recursive call, first call the function recursively for all children nodes and then come up with answer according to the returned values and value of the current node itself. This process can be regarded as a kind of *postorder* traversal.

Typically, a "bottom-up" recursive function *bottomUp(root)* will be something like this:

(1) Return specific value for null node.
(2) left answer = bottomUp(root left) (call function recursively for left child)
(3) right answer = bottomUp(root right) (call function recursively for right child)
(4) Return answers (answer ¡- left answer, right answer, root value)

When you meet a tree problem, ask yourself 2 questions:

- Can you determine some parameters to help the node know its answer?
- Can you use these parameters and the value of the node itself to determine what should be the parameters passed to its children?

If the answer is yes to both, try to solve this problem using "top-down" recursive solution.

Or,

- If you know the answer of its children, can you calculate the answer of that node?

If answer is yes, solve problem recursively using a bottom-up approach.

31. Sorting

cf. Harder (2018) [17]
8.1 Sorting algorithms.

Definition 6 (Sorting)**.** ***sorting***

$$(a_0, a_1, \dots a_{n-1}) \mapsto (a'_0, a'_1, \dots a'_{n-1}) \text{ s.t. } a'_0 \leq a'_1 \leq \cdots \leq a'_{n-1} \tag{37}$$

Seldom will we sort isolated values.
Usually we sort a number of records containing a number of fields based on a *key*.

31.1. **Loop Invariant.** cf. pp. 18-20, Sec. 2.1 Insertion Sort of Cormen, Leiserson, Rivest, and Stein (2009) [19]

Definition 7 (Loop invariant)**.** *At the start of each iteration of a* ***for*** *loop (or other loop), this must hold true.*

A loop invariant must have 3 things:

- ***Initialization****: It is true prior to the first iteration of the loop.*
- ***Maintenance****: If it's true before an iteration of the loop, it remains true before the next iteration.*
- ***Termination****: When loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.*

e.g. $A[1..j-1]$ subarray constitutes the currently sorted hand, i.e. the elements *originally* in positions 1 through $j-1$, but now in sorted order. So a loop invariant is, at the start of each iteration of the for loop, subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$, but in sorted order.

e.g. **Maintenance**: move $A[1..j-1]$ to right until finding proper position for $A[j]$. Subarray $A[1..j]$ sorted. Increment j for next iteration of the **for** loop then preserves loop invariant.

Termination property causes loop to terminate; differs from mathematical induction in which we apply inductive step infinitely.

Typically, show Initialization and Maintenance of a Loop Invariant by induction. Use Termination to show the algorithm solves the problem and gives the final solution.

31.2. **Insertion Sort, Straight Insertion Sort. Best case** (Bester Fall): $O(N)$ comparisons, $O(1)$ swaps.
Worst case (Schlechter Fall): $O(N^2)$ comparisons and swaps (array sorted in reverse order)
Average: $O(N^2)$ comparisons and swaps.

Worst case *space complexity*: $O(N)$ total, $O(1)$ auxiliary.

cf. Dietzfelbinger (2019) [18]
Softieralgorithmen.

Input $x = (a_1, \dots a_n)$ von Objekten a_i mit Schlüssel-(key) und Datenteil ("data")
$\forall\, i$,
a_i. key, a_i. data

Gegeben Array $A[1, \dots n]$ oder lineare Liste (linear list).
Schongesehen: Sortieralgorithmus Straight Insertion Sort
Sortierverfahren heisst stabil.
4. wenn dieser nur $O(1)$, also von n unabhängig, ist: Algorithmus arbeitet "int situ" oder "in-place".
Straight Insertion Sort:

Schlechtester Fall: $\frac{1}{2}n(n-1) \approx \frac{n^2}{2}$ Vergleiche, Zeit $\Theta(n^2)$,
$\frac{1}{2}n(n-1) = \sum_{i=1}^{n-1} i$ i.e.

For $n-1$ iterations for input array of size n,
$k = 1, \dots n-1$,
for each sorted list of length k,
consider, in the *worst case* (Schlechtester Fall), k comparisons to find correct position to insert:

$$\implies \sum_{k=1}^{n-1} k = \frac{1}{2} n(n-1) \approx \Theta(n^2)$$

cf. 8.2 Harder (2011) [17].

Bester Fall: $n-1$ Vergleiche, Zeit $\Theta(n)$
Very few inversions, $d = O(n)$ or already sorted, so do n insertions. cf. 8.2.3 Harder (2018), "Insertion sort, Consequences of Our Analysis" [17].

Durchschnittlicher Fall, alle Eingabereihenfolgen gleich Wahrscheinlich: Zeit $\Theta(n^2)$

Average $O(d+n)$, slow if $d = \omega(n)$.

Stabil, in situ.

cf. Harder (2018) [17].

List with 1 element is sorted (a_0) (trivial proof).

If list of size $k-1$ sorted, insert kth new object, by starting from back, and swap adjacent values until find location for it.

In analyzing insertion sort, from pp. 27, 2.2 of Cormen, Leiserson, Rivest, and Stein (2009) [19], **worst-case running time** gives upper bound on running time for any input. Also, for some algorithms, worst case occurs fairly often. And the "average case" is often roughly as bad as the worst case.

One simplifying abstraction, **rate of growth** or **order of growth** just means to consider only the leading term of a formula for large values of n.

cf. pp. 18, Cormen, Leiserson, Rivest, and Stein (2009) [19]

insertion sort takes parameter array A[1... n]. Sorts inplace.

Insertion-Sort(A)
for j = 2 to A.length
key = A[j]
// insert A[j] into sorted sequence A[1 ... j - 1]
i = j - 1
while i ¿ 0 and A[i] ¿ key
A[i + 1] = A[i]
i = i - 1
A[i + 1] = key

31.3. **Bubble Sort,** $O(N^2)$ **in time,** $O(1)$ **in space.** cf. Efficiency of Bubble Sort, Lesson 3: Searching and Sorting, Data Structures and Algorithms, Python

Time complexity of bubble sort:
$\forall$ iteration, there were $N-1$ comparisons. Worst case is $N-1$ iterations to order all N elements.
$\implies O(N^2)$ worse case

$O(N^2)$ average case
$O(N)$ best case (it was already sorted!)

Space complexity $O(1)$ no extra arrays needed. Sort was "in-place."

From wikipedia, "Bubble Sort" (Optimizing Bubble Sort), cf. https://en.wikipedia.org/wiki/Bubble_sort, Observe that for an array of elements a indexed by $i = 1, 2 \dots N$,
$j = 1 \mapsto a(N)$ largest (otherwise it would not have been swapped; contradiction)
$j = 2 \mapsto a(N-1)$ 2nd. largest. Iterated on $2, \dots N-1$ $N-2$ total (don't include the first). $\vdots$
$j \mapsto a(N-j+1)$ jth largest, iterated on $2, \dots N-j+1$, $N-j$ total.

31.4. **Merge Sort, $O(N \log N)$ in time, $O(N)$ in space.** cf. Efficiency of Merge Sort, Lesson 3: Searching and Sorting, Data Structures and Algorithms, Python

Use approximation (approximate to the *worse case*!) to count the number of comparisons (operations) in a "pass" (iteration)
multiply each iteration with number of comparisons per iteration.

Number of iterations $\cong \log N$ since $2^J - 1 =$ array size.

N comparisons for $\log(N)$ steps $\implies O(N \log N)$

Space complexity. Auxilliary space $= O(N)$ at each step we need total arraying size N to copy into.

31.5. **Quick Sort.**

31.5.1. *First element pivot Implementation.* cf. https://stackoverflow.com/questions/22504837/how-to-implement-quick-sort-algorithm-in-c

Consider array $a = a_i$, $i = 0, 1, \dots N-1$; $|a| = N$ (array of length N).

Given $l, r \in 0, 1, \dots N-1$,
Let $p := a(l)$ (pivot value)
$i = i_i$; $i_1 = l$
Let $j = j_j$ s.t. $j_1 = l+1, \dots, j_{r-l-1} = r-1$ (range over $r-1-(l+1)+1 = r-l-1$)
if $a(j_j) \le p$
$i_{j+1} = i_j + 1$
$a(i_{j+1}) = a(i_j + 1) \leftrightarrow a(j_j)$.

$a(i_{r-l-1}) \leftrightarrow a(l)$
$\mapsto i_{r-l-1}$.

Let's work out a table for the first few steps:

iteration	i	j	swaps	condition	example
1	l	$l+1$			
	$l+1$	$l+1$	$a(l+1) \leftrightarrow a(l+1)$	$p \ge a(l+1)$	5 3 4
2	l	$l+2$			
	$l+1$	$l+2$			
	$l+1$	$l+2$	$a(l+1) \leftrightarrow a(l+2)$	$p \ge a(l+2)$	5 6 3
	$l+2$	$l+2$	$a(l+2) \leftrightarrow a(l+2)$	$p \ge a(l+2)$	5 3 4
3	$l+1$	$l+3$	$a(l+1) \leftrightarrow a(l+3)$	$p \ge a(l+3)$	5 8 6 3 $\mapsto$ 5 3 6 8

31.5.2. *Complexity of Quick Sort.* Worst case if *all* pivots are in place, e.g. $1, 2, 8, 13$
Cannot "split" (partition) by pivots since it's already sorted $\implies O(N^2)$.
Average case $O(N \log N)$ (N iterations (for each element) $\log N$ comparisons (via split))
$O(1)$ space complexity.

Best case; move pivot to middle and divide each partition by 2. Bad case; array already somewhat sorted, hard to move pivot to middle.

31.6. **Time Complexities of all Sorting Algorithms.** https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/

"Best" complexity (best of worst case) sort is $O(N \log N)$.

31.7. **Hashing, $O(1)$ lookup, or $O(n)$ bucket lookup.**

31.7.1. *Hash maps.* key $\xrightarrow{\text{Hash function}}$ Hash: Key
Value:

$\langle$Key, Value$\rangle$ $\xrightarrow{\text{Hash function on Key}}$ Hash: $\langle K, V \rangle$
Value:

Use keys are inputs to has function.
Store Key, Value pair in map $\langle K, V \rangle$

31.7.2. *Hash Functions.* Value $\xrightarrow{\text{Hash function}}$ Hash Value (often index in an array)
Fast $O(1)$ look up
e.g. ticket number
$8675309 \xrightarrow{?}$ some index.
Take last two digits of a big number
$01234956 \mapsto 56 \mapsto 56/100 = 5$
"last few digits are the most random"
collisions

e.g. $0123456 \mapsto 56\%10$
$6543216 \mapsto 16\%10$

To deal with collisions 1) change value(s used in hash function) in hash function or change hash function completely, or 2) change structure of array $\mapsto$ buckets.

Normal Array vs. Bucket
Value collection

(1) $X\%1000000$, $O(1)$
(2) Bucket, iterate through bucket $O(n)$, n allocated size of each bucket
    (a) Hash function inside

load factor - how "full" has table is.
Load factor = number of entries / number of buckets
e.g. 10 values in 1000 buckets, load factor = 0.01, majority of buckets in table will be empty.
waste memory with empty buckets, so rehash i.e. come up with new hash function

with less resulting buckets.

closer load factor to 1, better to rehash and add more buckets
any table with load value greater than 1, guaranteed to have collisions.
100 numbers N
a number is a multiple of 5: $x = 5n$
Number of values = 100 numbers = N
Number of buckets = 100 (0 to 99)
$\implies 100/100 = 1$ load factor.
$100/107 < 1$ (good)
125 is a multiple of 5.
$x = 5n$. $\frac{x}{125} = \frac{5n}{125} = \frac{n}{25} \mapsto$ loss of collisions.
$\frac{100}{87} > 1 \mapsto$ collisions.
$\frac{100}{1000} = 0.1$ but waste memory, ton of empty buckets.

31.7.3. *Hash Table.* Constant time lookup important to speed up.
String Keys.
ASCII values from letters.
30 or less words, Use ASCII.

e.g. "UD" $\mapsto U = 85$
$D = 68$
$31 = (32 : 326)$
Huge has values for 4-letter strings
31
$s(0)31^{n-1} + s(1)32^{n-2} + \cdots + s(n-1)$
cf. Dietzfelbinger (2019) [18]
WS19. Algorithmen und Datenstrukturen 1. 5. Kapitel. Martin Dietzfelbinger. Dezember 2019. **Hashverfahren** ()Hash procedures).
cf. 5.1 Grundbegriffe (First principle) [18]
Gegeben: Universum U (auch ohne Ordnung) und Wertebereich R. (Given collection U without order and range R),
Hashverfahren (oder Schlüsseltransformationsverfahren),

$$f : S \to R, \text{ mit } S \subseteq U \text{ endlich} \tag{38}$$

f is a mapping that goes from keys to the desired data.
Ziel: Zeit $O(1)$ pro Operation

Gegensatz Suchbäume: Zeit $O(\log(n))$ pro Operation.
Hashfunktion; to hash (engl): Kleinhacken, Kleinschneiden

Definition 8 (Hashfunction, Hashfunktion)**.**

$$\begin{gathered} h : U \to [m] \textit{ mit } U \equiv \textit{ Universum der Schlüssel (collection of keys?)} \\ [m] = \{0, \ldots, m-1\} \quad \textit{Indexbereich für Tabelle } T \\ S \subseteq U, |S| = n \end{gathered} \tag{39}$$

Grundansatz (basic, fundamental approach): Spoichrun von $f : S \to R$ in Tabelle $T[0, \ldots m-1]$

Aus Schlüssel, $x \in U$ wird ein
Index $h(x) \in [m]$

Idealvorstellung (ideal, concept, ideal conception):
Speichere Paar $(x, f(x))$ in Zelle (cell, booth, element) $T[h(x)] = T[j]$

Example, Beispeil: $U = \{A \dots Z, a \dots z\}^{3 \dots 20}$.
Words of length l s.t. $3 \leq l \leq 20$.

$$h : U \to [m]$$
$$h(c_1 c_2 c_3 \dots c_r) = \text{num}(c_3) \mod 13$$

Wobei (where)

$$\begin{aligned} \text{num}(A) &= \text{num}(a) = 0 \\ \text{num}(A) &= \text{num}(a) = 0 \\ \vdots & \quad \vdots \\ & \text{num}(A) = \text{num}(a) = 0 \end{aligned}$$

$xf(x)$(further desired target data)	num(c_3)$h(x)$	
Ja**n**uar31	130	$h(c_1 \dots c_r) = \text{num}(c_3) \mod 13$
Fe**b**ruar28	11	$j(x, f(x))$ mit $h(x) = j$
Ma**e**rz31	44	$0(\text{Januar}, 31), (\text{Juni}, 30)$
Ap**r**il30	174	$1(\text{Februar}, 28)$
⋮	⋮	⋮
Ju**n**i30	130	$4(\text{Maerz}, 31)$
⋮	⋮	

Kollisionen (Collision) bei $j = 0, 4, \dots$ Kollision $x, y \in S$, $x \neq y$, $h(x) = h(y)$.

Behälter, **bucket**

Definition 9 (Bucket)**.**

(40) $B_j := \{x \in S | h(x) = j\}$, $S \subseteq U$ *for hash function* $h : U \to [m]$

Fall: h "rein zufällig", $|S| = n$, $S = \{x_1 \dots x_n\}$
$h : U \to [m]$
$h : S \subset U \to [m]$; $h(x_1) \dots h(x_n)$ in $[m]^n$ sind rein zufällig".

Jeder der m^n Vectoren $(j_1 \dots J_n) \in [m]^n$ kommt mit derselben Wahrscheinlichkeit $1/m^n$
als $(h(x_1), \dots h(x_n))$ vor.

e.g. IP Addresses. Suppose we want to associate IP addresses and any corresponding domain names.

31.7.4. *Hashfunctions.* cf. LeetCode, Keys to Design a Hash Table
Hash function h examples:

Key Type	Key range, S,	Number of Buckets m	h examples
integer	$[0, 100000]$	1000	$y = x \mod 1000$
char	'a' ... 'z'	26	$y = x -' a'$
array of integers	size < 10, $\forall$ number $\in [0, 1]$	$1024 = 2^{10}$	$y = \sum_{i=0}^{9} x_i 2^i$
array of integers	size < 10, $\forall$ number $\in [0, 3]$	$1024 = 2^{10}$	$y = \sum_{i=0}^{4} x_i 4^i = \sum_{i=0}^{4} x_i 2^{2i}$

Remember,

$$\boxed{\begin{array}{l} f : S \to R,\ S \subseteq U \\ f : x \mapsto f(x) \\ h : U \to [m] \\ h : x \mapsto h(x) = j \end{array}} \tag{41}$$

Suppose $f = \mathrm{id} : S \to S$, $R = S$ (identity map)
$h : U \mapsto [m]$.

Consider $|S| = N$ large. Consider load balance N/m; we still need $m < N$ for a reasonable number of buckets.

32. References for Data Structures and Algorithms

https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/ https://www.tu-ilmenau.de/en/institute-of-theoretical-computer-science/lehre/lehre-ss-2017/aud/

Lecture Notes for Data Structures and Algorithms. Revised each year by John Bullinaria. School of Computer ScienceUniversity of BirminghamBirmingham, UK

https://www.tu-ilmenau.de/fileadmin/public/iti/Lehre/AuD/WS19Ing/AuD1-Kap-2-statisch2.pdf

https://www.tu-ilmenau.de/en/institute-of-theoretical-computer-science/lehre/lehre-ws-20192020/aud-1/

Part 6. Linux System Programming

33. File Descriptors

cf. https://medium.com/@copyconstruct/nonblocking-i-o-99948ad7c957

Let byte $b \in (\mathbb{Z}_2)^8 \equiv B \equiv o$ (when referring to exactly 8 bits).

The fundamental building block of I/O in Unix is a sequence of bytes $(b_n)_{n \in \mathbb{N}}$, $b_n \in B$.

Most programs work with an even simpler abstraction - a stream of bytes or an I/O stream.

File descriptors $\equiv fd$. fd references I/O streams. $fd \in \mathbb{N}$ (by definition and the Linux programmer's manual).

$$\mathbb{N} \to B^{\mathbb{N}}$$

$$fd \overset{\text{surj}}{\longmapsto} (b_n)_{n \in \mathbb{N}}$$

e.g. Pipes, files, FIFOs, POSIX IPC's (message queues, semaphores, shared memory), event queues are examples of I/O streams referenced by a fd, i.e. $fd \overset{\text{surj}}{\longmapsto} (b_n)_{n\in\mathbb{N}}$.

33.1. **Creation and Release of Descriptors.** fd's are either

- explicitly created by system calls like `open, pipe, socket`, etc. or
- inherited from parent process

fd's released when

- process exists
- calling `close` system call
- implicitly after an `exec` when fd marked as **close on exec**; i.e. fd automatically (and atomically) closed when `exec`-family function succeeds

34. Unix execution model

cf. https://stackoverflow.com/questions/4204915/please-explain-the-exec-function-and-its-fa
37558902

34.1. **Processes and Programs.** A process is something in which a program executes. $pid \in \mathbb{N} \overset{\text{surj}}{\longmapsto}$ process

These 2 operations that you can do is the entire UNIX execution model:

`fork` - creates new process containing duplicate of current program, including its state. There are differences between the processes so to be able to distinguish which is parent, which is child:

$$pid_P \overset{\text{surj}}{\longmapsto} pid_C > 0$$
$$\text{prog}_i \text{ s.t. } \text{prog}_i \in pid_P \mapsto \text{prog}_i \in pid_C$$

`exec` - replaces program in current process with brand new program:

$$\text{prog}_i \in pid_i \mapsto \text{prog}_{i'} \in pid_i$$
$$\text{s.t. } \text{prog}_i = \emptyset,\ \text{prog}_i \neq \text{prog}_{i'}$$

More details:

`fork()` makes a near duplicate of current process, identical in almost every way; 1 process calls `fork()` while 2 processes return from it:

$$pid_P \overset{\texttt{fork()}}{\longmapsto} pid_P, pid_C > 0$$

Examples:

- When `fork`, `exec` used in sequence: In shells, `find`:

$$pid_{\text{shell}} \overset{\texttt{fork()}}{\longmapsto} pid_{\text{shell}}, pid_C \qquad \text{prog}_j, pid_C \overset{\texttt{exec()}}{\longmapsto} \text{prog}_{j'} \in pid_C$$

Shell forks, then child loads `find` program into memory, setting up all command line arguments, standard I/O, etc.

- `fork()` only. Daemons simply listening on TCP port and fork copy of themselves to process specific request, while parent goes back to listening.

Notice that nominally, parent waits for child to exit.

close-on-exec flag, if set, will have fd automatically be closed during a successful **execve** (or `exec`) (cf. http://man7.org/linux/man-pages/man2/fcntl.2.html)

34.2. **File entry.** Every fd points to a data structure called *file entry* in the kernel.

$$fd \mapsto \text{ file entry}$$

i.e. $\forall\, fd, \exists$ file entry and pointer to that file entry.

open file description - an entry in system-wide table of open files.
open file description records file offset and file status flags.

$n_{\text{offset}} \equiv$ file offset, per fd, from the beginning of file entry.

`fork()` results in both parent and child processes using same fd and reference to same offset n_{offset} in file entry.

If process was last to reference file entry, kernel then deallocates that file entry.

Each file entry contains

- the type
- array of function pointers, that translates generic operations on fds to file-type specific implementations.

`open` call implementation greatly varies for different file types, even if higher level API exposed is the same.

e.g. sockets - Same `read, write` used for byte-stream type connections, but different system calls handle addressed messages like network datagrams.

For a process pid, consider fd table of fds. $\forall$ fd, $fd \mapsto$ open file description $\in$ open file table (system-wide).

$\forall$ open file description $\in$ open file table,
open file description $= n_{\text{offset}}$, status flags, and
open file description $\xmapsto{\text{inode ptr}}$ inode table.

A blocking system call is 1 that suspends or puts calling process on wait until an event occurs [1]

34.3. **Readiness of fds.** cf. https://medium.com/@copyconstruct/nonblocking-i-o-99948ad7c957

A fd is considered *ready* if process can perform I/O operation on fd without blocking (i.e. without process having to wait or be suspended). For fd to be considered "ready", it doesn't matter if **the operation actually transfer any data** - all that matters is that the I/O operation can be performed without blocking (i.e. the calling process won't wait or be suspended).

A fd changes into a *ready* state when an I/O *event* happens, such as the arrival of new input or completion of a socket connection or when space is available on previously full socket send buffer after TCP transmits queued data to the socket peer.

There are 2 ways to find out about the readiness status of a descriptor - edge triggered and level-triggered.

34.4. **Level Triggered.** cf. https://medium.com/@copyconstruct/nonblocking-i-o-99948ad7c957

To determine if fd is ready, the process tries to perform a non-blocking I/O operation. Also seen as "pull" or "poll" model. (User "pull" on the fd).

At time t_0, process pid tries I/O operation on non-blocking fd. If I/O operation blocks, system call returns error `err`.
At time $t_1 > t_0$, pid tries I/O on fd (again). Say call blocks again, and returns `err`.
At time $t_2 > t_1$, pid tries I/O on fd (again). Assume call blocks again, and returns

[1] https://stackoverflow.com/questions/19309136/what-is-meant-by-blocking-system-call, https://www.quora.com/What-is-the-difference-between-a-blocking-system-call-and-an-non-blocking-system-call

`err`.
At time $t_3 > t_2$, pid polls status of fd, and fd is ready. pid can then choose to actually perform entire I/O operation (e.g. read all data available on socket).
At time $t_4 > t_3$, assume pid polls status of fd, and fd isn't ready; call blocks (again), and I/O operation returns `err`.
At time $t_5 > t_4$, pid polls for status of fd, and fd is ready. pid can then choose to only perform a partial I/O operation (e.g., reading only half of all data available)
At time $t_6 > t_5$, pid polls status of fd, and fd is ready. This time, pid can choose to perform no I/O.

34.5. **Edge Triggered.** Process *pid* receives a notification only when fd is "ready" (usually when there's any new activity on fd since it was last monitored). This can be seen as the "push" model, in that a notification is pushed to the process about readiness of a fd.

process pid only notified that fd is ready for I/O, but not provided additional information like for instance how many bytes arrived on socket buffer.

Thus, pid only armed with incomplete data as it tries to perform any subsequent I/O operation e.g. at $t_2 > t_0$, pid gets notification about fd being ready.

byte stream $(b_n)_{n\in\mathbb{N}}$ available for I/O stored in buffer.

Assume $N_b = 1024$ bytes available for reading, when pid gets notification at $t = t_2$.

$$\implies (b_n)_{n<N_b} \subset (b_n)_{n\in\mathbb{N}}$$

Assume pid only reads $N_b' = 500$ bytes $< N_b$

That means for $t = t_3, t_4, \ldots t_{i_2}$, there are still $N_b - N_b' = 524$ bytes available in $(b_n)_{n<N_b}$ that pid can read without blocking.

But since pid can only perform I/O once it gets next notification, these $N_b - N_b'$ bytes remain sitting in buffer for that duration.

Assume pid gets next notification at $t_{i_2} = t_6$, when additional N_b bytes have arrived in buffer.

Then total amount of data available on $(b_n)_{n\in\mathbb{N}} = 2N_b - N_b' = 1548 = 524+1024$.

Assume pid reads in $N_b = 1024$ bytes.

This means that at end of 2nd. I/O operation, $N_b - N_b' = 524$ bytes still remain in $(b_n)_{n\in\mathbb{N}}$ that pid won't read before next notification arrives.

While it might be tempting to perform all I/O immediately once notification arrives, doing so has consequences.

- large I/O operation on single fd has potential to starve other fd's
- Furthermore, even with case of level-triggered notifications, an extremely large Write or send call has potential to block.

35. Concurrent Programming

cf. Ch. 12, Concurrent Programming, Bryant and O'Hallaron (2015) [1]

36. Multiplexing I/O on fds

In above, we only described how pid handles I/O on a single fd.

Often, pid wants to handle I/O on more than 1 fd.

e.g. program prog needs to log to `stdout`, `stderr`, while accept socket connections, and make outgoing RPC connections to other services.

There are several ways of multiplexing I/O on fds.

- nonblocking I/O (fd itself is marked as non-blocking; operations may finish partially)
- signal-drive I/O (pid owning fd is notified when I/O state of fd changes)
- polling I/O (e.g. `select`, `poll`, both provide level-triggered notifications about readiness of fd)

36.1. **Multiplexing I/O with Non-blocking I/O.** We could put all fds in non-blocking mode.

pid can try to perform I/O operations on fds to check if any I/O operations result in error.

kernel performs I/O operation on fd and returns err, or partial output, or result of I/O operation if it succeeds.

Cons:

- Frequent checks
- Infrequent checks. If such operations are conducted infrequently, then it might take pid unacceptably long time to respond to I/O event that's available.

When does this approach make sense?

Operations on output fds (e.g. writes) don't generally block.
- In such cases, it might help to try to perform I/O operation first, and revert back to polling when operation returns err.

It might also make sense to use non-blocking approach with edge-triggered notifications, where fds can be put in nonblocking mode, and once pid gets notified of I/O event, it can repeatedly try I/O operations until system calls would block with an `EAGAIN` or `EWOULDBLOCK`.

36.2. **Multiplexing I/O via signal driven I/O.** .

Kernel instructed to send the pid a signal when I/O can be performed on any of the fds.

pid will wait for signals to be delivered when an of the fds is ready for I/O operation.

Cons: Signals are expensive to catch, rendering signal drive I/O impractical for cases where large amount of I/O is performed.

Typically used for "exceptional conditions."

36.3. **Multiplexing I/O via polling I/O.** .

fds placed in non-blocking mode.

pid uses **level triggered mechanism** to ask kernel by means of system call (`select` or `poll`) which fds are capable of performing I/O.

36.3.1. *`poll`, and `select` vs. `poll`.* With `select`, we pass in 3 sets of fds we want to monitor for reads, writes, and exceptional cases.

With `poll`, we pass in a set of fds, **each marked with the events it specifically needs to track**.

36.3.2. *What happens in the kernel (with `select` and `poll`)?* Both `select` and `poll` are *stateless*, meaning,
every time a `select` or `poll` system call is made, the kernel checks *every fd*, in the input array passed as 1st. argument, for the occurrence of an event and return the result to the pid.

$\implies$ this means that the cost of `select`/`poll` is O(N), where N is the number of fds monitored.

The implementation of `select` and `poll` comprises 2 tiers:

- Specific top tier which decodes incoming request, as well as several device or socket specific bottom layers
- bottom layers comprise of *kernel poll functions* used both by `select`, `poll`

37. `epoll` Event poll

cf. "The method to epoll's madness" by Cindy Sridharan[24]

`epoll`, a Linux specific construct, allows for a pid to monitor multiple fds and get notifications when I/O is possible on them.

37.1. `epoll` **Syntax.** *epoll*, unlike *poll*, is not a system call. It's a kernel data structure that allows a pid to multiplex I/O on multiple fds.

This data structure can be created, modified, deleted by 3 system calls: `epoll_create`, `epoll_ctl`, `epoll_wait`.

38. Network Time Protocol (NTP), `ntpq`

https://www.eecis.udel.edu/~mills/ntp.html

39. Concurrency

cf. Concurrency, Goldman and Miller, 6.031 Spring 2020 [29].

There are 2 models for concurrent programming: *shared memory* and *message passing*.

39.1. **Shared memory concurrency model.** In the shared memory concurrency model, concurrent modules interact by reading and writing shared objects in memory. For example, if A and B are concurrent modules, A and B access the same shared memory space. In "shared" memory, A or B can have private objects (only 1 module can access it). But shared object is shared by both A and B (both modules have reference to it).

Examples of shared-memory model:

- A and B might be 2 processors or processor cores in same computer, sharing same physical memory.
- A and B might be 2 programs running on same computer, sharing a common filesystem with files they can read and write.
- A and B might be 2 threads in the same Java program, sharing same Java objects.

39.2. **Message passing concurrency model.** Concurrent modules interact by sending messages to each other through a communication channel. Modules send off messages, and incoming messages to each module are queued up for handling.

39.3. **Processes, threads, time-slicing.** Message-passing and shared-memory models are about how concurrent modules communicate. The concurrent modules *themselves* come in 2 different kinds: *processes* and *threads*.

39.3.1. *Process.* A process is an instance of a running program that's *isolated* from other processes on the same machine. Particularly, it has its own private section of the machine's memory.

Process abstraction is a *virtual computer*; it makes the program feel like it has the entire machine to itself - like a fresh computer has been created, with fresh memory, just to run that program.

- Processes normally share no memory between them, e.g. computers connected across a network.
  - Process can't access another process's memory or objects at all.
    - Sharing memory between processes is *possible* on most OS, but needs special effort.
- Process is automatically ready for message passing, because it's created with standard input and output streams.

Whenever you start any program on your computer, it starts a fresh process to contain the running program.

39.3.2. *Thread.* "A thread is a locus of control inside a running program." (EY: ???). Think of it as a place in the program that is being run, plus the stack of method calls that led to that place (so thread can go back up the stack when it reaches `return` statements).

Thread abstraction represents a *virtual processor*. Making a new thread simulates making a fresh processor inside virtual computer represented by the process. This new virtual processor runs the same program and shares the same memory as other threads in the process.

- Threads automatically ready for shared memory, because threads share all the memory in the process.
  - Takes special effort to get "thread-local" memory that's private to a single thread.
- Neccessary to set up message-passing explicitly, by creating and using queue data structures.

Whenever you run a Java program, program starts with 1 thread, which calls `main()` as its first step. This thread is referred to as the *main thread*.

39.3.3. *Time slicing.* When there are more threads than processors, concurrency is simulated by **time slicing**, meaning the processor switches between threads.

On most systems, time slicing happens unpredictably and nondeterministically, meaning that a thread may be paused or resumed at any time.

39.4. **Context Switching.** cf. wikipedia, "Context Switch"

context switch - process of storing state of a process or thread, so it can be restored and resume execution at a later point. This allows multiple processes to share a single CPU.

Context switch in varying contexts: in multitasking, refers to process of storing system state for 1 task, so task can be paused and another task resumed.

If interrupt, such as task needs disk storage access, context switch can occur to free up CPU time for other tasks.

Some OS require context switch to move between user mode and kernel mode.

39.4.1. *Costs of context switching. Costs*: time to save and load registers, memory maps, update tables and lists.

What's actually involved depends on architectures, OS, number of resources shared (threads that belong to same process share many resources compared to unrelated non-cooperating processes).
e.g. Linux kernel, context switching involves switching registers, stack pointer (its typical stack-pointer register), program counter, flushing translation lookaside buffer (TLB), and loading page table of next process to run (unless old process shares memory with new).
analogous context switching between user threads, notably green threads, and is very lightweight, saving and restoring minimal context.

39.4.2. *Switching cases, i.e. 3 potential triggers: Multitasking, Interrupt handling, User and kernel mode switching. Multitasking*: 1 process must be switched out of CPU so another process can run. This context switch can be triggered by the process making itself unrunnable, such as waiting for IO or synchronization operation to complete.
On pre-emptive multitasking system (use an interrupt to suspend currently executing process and invokes scheduler to determine which process should execute next so all processes get some amount of CPU time at any given time), scheduler may also switch out processes that are still runnable. To prevent other processes from being starved of CPU time, pre-emptive schedulers often configure timer interrupt to fire when process exceeds its time slice. This interrupt ensures scheduler will gain control to perform a context switch.

Interrupt handling: Interrupt driven - meaning, e.g. if CPU requests data from disk, it doesn't need to busy-wait until read is over; it can issues request (to IO device) and continue with some other task. When read is over, CPU can be interrupted (in this case by hardware, which sends interrupt request to PIC) and presented with read.

User and kernel mode switching: When system transitions between user mode and kernel mode, context switch isn't necessary; a *mode transition* isn't by itself a context switch. Depends on OS.

39.4.3. *Performance in Context Switching.* Context switching has cost in performance due to running task scheduler, TLB (translation lookaside buffer) flushes, and indirectly due to sharing CPU cache between multiple tasks.

Switching between threads of a single process can be faster than between 2 separate processes because threads share the same virtual memory maps, so a TLB flush isn't necessary.

Time to switch between 2 separate processes is **process switching latency**. Time to switch between 2 threads of same process is **thread switching latency**.

Switching between 2 processes in single address space OS can be faster than switching between 2 processes in OS with private per-process address spaces.

40. IPC: Interprocess Communications

40.1. **Queues and Message Passing.** cf. Reading 23: Queues and Message-Passing, Goldman and Miller, 6.031 Spring 2020 [29].

Recall the 2 models for concurrent programming: *shared memory* and *message passing*:

- *shared memory*, concurrent modules interact by reading and writing shared mutable objects in memory. Creating multiple threads inside a single process is a primary example of shared-memory concurrency.
- *message passing*, concurrent modules interact by sending immutable messages to 1 another over a communication channel; that communication channel might connect different computers over a network, e.g. web browsing, instant messaging, etc.

Advantages of message passing model over shared memory model:

- boils down to greater safety from bugs,
- Message passing shares only immutable objects (the messages) between modules, whereas shared memory *requires* sharing mutable objects, which can be a source of bugs.

Consider as an example in the following of implementing message passing within a single process, as opposed to between processes over the network. In the example in the following, use **blocking queues** (an existing threadsafe type) to implement message passing between threads within a process.

40.1.1. *Message passing with threads.* Previous message passing between processes: clients and servers communicating over network sockets.

We can also use message passing between threads within the same process, and this design is often preferable to shared memory design with locks.

Use a synchronized queue for message passing between threads.
Queue serves same function as buffered network communication channel in client/server message passing.

In ordinary queue (Java's `Queue`),

- Java's `add(e)` adds element `e` to end of queue
- Java's `remove()` removes and returns element at head of the queue, or throws an exception if queue is empty.

A blocking queue (Java's `BlockingQueue`) extends this interface, additionally supports operations that
wait for queue to become non-empty when retrieving an element, and
wait for space to become available in queue when storing an element.

Analogous to client/server pattern for message passing over a network is the **producer-consumer design pattern** for message passing between threads.

- Producer threads and consumer threads share a synchronized queue
- Producers put data or requests onto the queue, and consumers remove and process them
  - 1 or more producers and 1 or more consumers might all be adding and removing items from the same queue; this queue must be safe for concurrency

Java provides 2 implementations of a blocking queue:

- `ArrayBlockingQueue` is a fixed-size queue that uses an array representation. `put`ting a new item on queue will block if queue is full.

- `LinkedBlockingQueue` is a growable queue using linked-list representation. If no max. capacity is specified, queue will never fill up, so `put` will never block.

Unlike streams of bytes sent and received by sockets, these synchronized queues (like normal collections classes in Java) can hold objects of an arbitrary type. Instead of designing a wire protocol, we must choose or design a type for messages in the queue. **It must be an immutable type**.

40.1.2. *Bank account example.*

40.2. **Message Queues.** cf. Ch. 51, *Introduction to POSIX IPC*, Kerrisk (2010) [31].

Message queues can be used to pass messages between processes.
As with System V message queues, message boundaries are preserved, so readers and writers communicate in units of messages (as opposed to undelimited byte stream provided by a pipe).
POSIX message queues permit each message to be assigned a priority.

cf. **IPC object names**, Sec. 51.1 API Overview, pp. 1058, Kerrisk (2010) [31].

To access a POSIX IPC object, we must identify it and only portable means that SUSv3 specifies is using a *name consisting of an initial slash*, e.g. `/myobject`.

On Linux, names for POSIX message queue objects (and shared memory) are limited to `NAME_MAX` (255) characters.

Rules for creating IPC object names on some systems are different. e.g. on Tru64 5.1, IPC object names created as names within standard file system, so if caller doesn't have permission to create a file in that directory, then IPC *open* call fails.

cf. pp. 1063, Ch 52, *POSIX Message Queues*, Kerrisk (201) [31].

- POSIX message queues are reference counted - a queue that's marked for deletion is removed only after its closed by all processes that are currently using it.
- POSIX messages have associated priority and messages are always strictly queued (and thus received) in priority order. Each System V message has integer type and messages can be selected in a variety of ways using `msgrcv()`
- POSIX message queues provide feature that allows process to be asynchronously notified when message is available on a queue.

40.3. **TCP, UDP, Unix Network Programming.** cf. Stevens, Fenner, Rudolf (2004) [32]

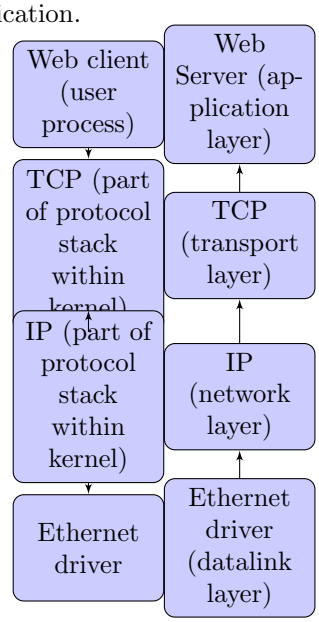
When writing programs that communicate across a computer network, one must first invent a **protocol**, an *agreement* on how those programs will communicate.

For example, a Web server is typically thought of as a long-running program (or **daemon**) that sends network messages only in response to requests coming in from the network.

Other side of the protocol is a Web client, e.g. browser, which always *initiates* communication with the server.

Deciding that client always initiates requests tends to simplify protocol - basic client/server model.

Some more complex network applications require *asynchronous callback* communication.



40.3.1. *Simple TCP/IP "Daytime" (get the time) client, server.* cf. pp. 6, Ch. 1, Introduction, Sec. 1.2 "A Simple Daytime Client", Stevens, Fenner, Rudolf (2004) [32]

In "Read and display server's reply", pp. 9, Sec. 1.2 of Stevens, Fenner, Rudolf (2004) [32], we must be careful when using TCP because it is a *byte-stream* protocol with no record boundaries.

Server's reply is normally a 26-byte string.

With byte-stream protocol, these 26-bytes can be returned in numerous ways:
single TCP segment containing all 26 bytes of data,
in 26 TCP segments each containing 1 byte,
any combination that totals to 26 bytes.

Normally, a single segment containing all 26 bytes returned, but with larger data sizes, can't assume server's reply will be returned by single `read`.
Therefore, when reading from TCP socket, we *always* need to code the `read` in a loop and terminate the loop when `read` either returns 0 (i.e. the other end closed connection) or value less than 0 (an error).

In this example and in HTTP 1.0, end of record denoted by server closing connection.
SMTP (Simple Mail Transfer Protocol), marks end of record with 2-byte sequence of an ASCII carriage return (`\r`) followed by ASCII linefeed (`\n`).
RPC (Sun Remote Procedure Call) and Domain Name System (DNS) place binary count containing record length in front of each record that's sent when using TCP.

The important concept here is that TCP itself provides no record markers.

cf. pp. 13, Sec. 1.5 "A Simple Daytime Server", Stevens, Fenner, Rudolf (2004) [32]

40.3.2. *Bind server's well-known port to socket.* Server's well-known port (13 for daytime service) is bound to socket by filling in an Internet socket address structure

and calling `bind`.
Specify IP address as `INADDR_ANY`, which allows server to accept client connection on any interface, in case server host has multiple interfaces.

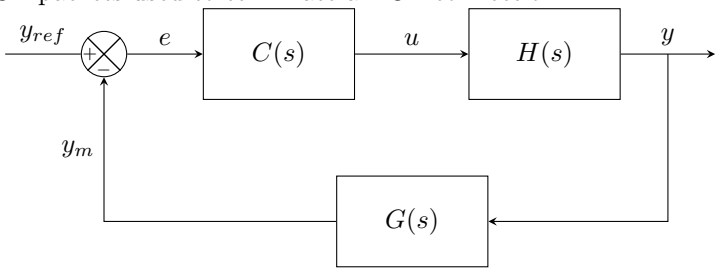
40.3.3. *Convert socket to listening socket.* By calling `listen`, socket is converted into a listening socket, on which incoming connections from clients will be accepted by the kernel. These 3 steps, `socket, bind, listen` are the normal steps for any TCP server to prepare the *listening descriptor*.

Specify maximum number of client connections kernel will queue for this listening descriptor.

40.3.4. *Accept client connection, send reply.* Normally, server process is put to sleep in call to `accept`, waiting for client connection to arrive and be accepted. A TCP connection uses what's called a **3-way handshake** to establish connection. When this handshake completes, `accept` returns, and return value from function is a new descriptor (`connfd`) that's called the *connected descriptor*. This new descriptor is used for communication with the new client.

A new descriptor is returned by accept for each client that connects to our server.

40.3.5. *Terminate connection.* Server closes its connection with client by calling `close`. This initiates the normal TCP connection termination sequence: a FIN is sent in each direction and each FIN is acknowledged by the other end. Sec. 2.6 of Stevens, Fenner, Rudolf (2004) [32] for more about TCP's 3-way handshake and 4 TCP packets used to terminate a TCP connection.



41. References and Resources for Linux System Programming

http://igm.univ-mlv.fr/~yahya/progsys/linux.pdf
http://www.cs.fsu.edu/~baker/opsys/examples/forkexec/fork_wait.c
http://cs241.cs.illinois.edu/coursebook/

CS556 - Distributed Systems. Tutorial by Eleftherios Kosmas. https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf

https://www.cs.rutgers.edu/~pxk/417/notes/content/02r-sockets-programming-slides.pdf https://www.cs.cmu.edu/~srini/15-441/S10/lectures/r01-sockets.pdf

https://db.in.tum.de/teaching/ss19/c++praktikum/

https://www.cs.cmu.edu/afs/cs/academic/class/15213-f99/www/class26/

Part 7. C++ Template Metaprogramming

From this question on stack overflow, comes this tutorial.

cf. Vandevoorde, Josuttis, and Gregor (2017) [25].

Part 8. Functional Programming and Category Theory

Best discussion of the relation to monads in category theory to monads in "computer science": https://ncatlab.org/nlab/show/monad+%28in+computer+science%29

Best article on practical implementation (real daily programming, real programming experiences) of functional programming without pulling punches on Category Theory:

https://nalaginrut.com/archives/2019/10/31/8%20essential%20patterns%20you%20should%20know%20about%20functional%20programming%20in%20c%2B%2B14

More links:

https://nbviewer.jupyter.org/github/ivanmurashko/articles/blob/master/cattheory/cattheory.pdf https://wiki.ifs.hsr.ch/SemProgAnTr/files/mof_categories_final.pdf

42. Actors, Concurrent Systems

https://sourceforge.net/projects/sobjectizer/ https://github.com/Stiffstream/sobjectizer#helloworld-example

Čukić (2018) [26]

Ch. 12, pp. 250, Čukić (2018) [26]

Consider that multiple (e.g. person) objects shouldn't have any shared data - real people *share* data by *talking* to each other; they don't have shared variables everyone can access and change.

actors - in actor model, actors are completely isolated entities that share nothing but can send messages to 1 another.

- actor receives messages and processes them 1 by 1, message → actor
- reaction to message can be change the actor itself or send message to another actor, actor → another actor in system

design actors as follows:

- Actors can receive only messages of a signle type, and send messages of a single (not necessarily same) type. If you need to support multiple different types for input or output messages, use `std::variant` or `std::any`, as in Ch. 9, Čukić (2018) [**?**]
- Leave choice of whom to send message to external controller so can compose actors in a functional way. External controller will schedule which sources an actor should listen to
- leave it up to external controller to decide which messages should be processed asynchronously and which shouldn't

https://cs.lmu.edu/~ray/notes/messagepassing/ Message passing notes

Part 9. Technical Interviews

"Interview Introduction", Data Structures and Algorithms in Python, Udacity

- Clarifying the Question
- Generating Inputs and Outputs
- Generating Test Cases
- Brainstorming

- Runtime Analysis
- Coding
- Debugging

43. Puzzles, Brain-teasers

http://www.mytechinterviews.com/four-people-on-a-rickety-bridge
20170525 Cyril Zeller, Nvidia had asked this.
Bridge crossing problem. 4 people on 1 side of a bridge. For each A, B, C, D
A takes 1 minutes
B takes 2 minutes
C takes 5 minutes
D 10 minutes.

At night, only 1 flash light.
No more than 2 people can walk together. Minimize time to cross.
To reduce the amount of time find a way for C, D to go together; if they cross together then we need another to come back to get the others. So have a "fast person" like A or B already on the other side.

A and B go cross. 2
A or B comes back. 1
C and D go across. 10
B or A comes back. 2
A and B go across (done).

Total $= 2 + 1 + 10 + 2 + 2 = 17$.

44. Suggested questions to ask the interviewer

What is the job function? Team? (e.g. what is the function of the team), last is SpaceX itself, What role does this team play in SpaceX?

45. How to do Code Review

cf. 6.005, Software Construction, MIT OCW, Reading 4: Code Review
Core review has 2 purposes:

- **Improving the code**:
  - Finding bugs,
  - anticipating possible bugs,
  - check clarity of the code,
  - check consistency with project's style standards
- **Improving the programmer**: It's an important way programmers learn and teach each other, about new language features, changes in design of the project or its coding standards, and new techniques

Further Code review guidelines for the MIT OCW 6.005 class:

45.1. Benefits of Code Review.

- *Reviewing helps find bugs*, complimentary to other techniques (static checking, testing, assertions, reasoning)

- uncovers code that's confusing, poorly documented, unsafe, or otherwise not ready for maintenance or future change
- spreads knowledge, allowing developers to learn from each other by explicit feedback and example

45.2. **What to look for. Bugs or potential bugs**

- Repetitive code (remember DRY, Don't Repeat Yourself)
- Disagreement between code and specification
- Off-by-one erros
- Global variables, and other too-large variable scopes
- optimistic, undefensive programming
- magic numberes
- . . . (other)

Unclear, messy code

- Bad variable or method names,
- Inconsistent indentation
- Convoluted control flow (if and while statements) that could be simplified
- Packing too much into one line of code, or too much into 1 method
- Failing to comment obscure code.
- Having too many trivial comments that are simply redundant with the code
- Variables used for more than 1 purpose.
- . . . (other)

Misusing (or failing to use) essential design concepts

- Incomplete or incorrect specification for a method or class,
- representation exposure for a data abstraction,
- Immutable datatypes that expose themselves to change,
- invariants that aren't really invariant, or aren't even stated

Positive comments are good; don't be afraid to make comments about things you really like, e.g.:

- **Unusually elegant code**
- **Creative solutions**
- **Great design**

45.2.1. *Respect in Code Review.* **Be polite**.
Be constructive. Don't just criticize, but be helpful: point the way toward solutions.

45.2.2. *Further FAQ.* **I don't feel like I know anything. How can I possibly review other people's code?** You know more than you think you do. You can read for clarity, comment on places where the code is confusing, look for problems we read about, know about, talked about, etc.

What if I can't find anything wrong? You can write a specific positive comment about something good.

45.2.3. *On Style, Style Standards.* cf. Reading 4: Code Review, Software in 6.005

It's *very* important to follow the conventions of the project. For instance, if you're the programmer who reformats every module you touch to match your personal style, your teammates will hate you, and rightly so. Be a team player.

45.2.4. *Smelly Example 1.* Programmers often describe bad code as having a "bad smell" that needs to be removed.

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
if (month == 2) {
dayOfMonth += 31;
} else if (month == 3) {
dayOfMonth += 59;
} else if (month == 4) {
dayOfMonth += 90;
} else if (month == 5) {
dayOfMonth += 31 + 28 + 31 + 30;
} else if (month == 6) {
dayOfMonth += 31 + 28 + 31 + 30 + 31;
} else if (month == 7) {
dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
} else if (month == 8) {
dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
} else if (month == 9) {
dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
} else if (month == 10) {
dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
} else if (month == 11) {
dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
} else if (month == 12) {
dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
}
return dayOfMonth;
}
```

45.2.5. *Don't Repeat Yourself.* Duplicated code is a risk to safety. If you have identical or very similar code in 2 places, then the fundamental risk is there's a bug in both copies, and some maintainer fixes the bug in 1 place, but not the other.

EY: If you're doing the same computation, with small change, consider defining it in another place.

45.2.6. *Comments Where Needed.* Good comments should make code easier to understand, safer from bugs (because important assumptions have been documented), and ready for change.

One kind of crucial comment is a *specification*, which appears above a method or above a class and documents the behavior of the method or class.

Specifications document assumptions.

Another crucial comment is one that specifies the provenance or source of a piece of code that was copied or adapted from elsewhere.

For example,

```
// read a web page into a string
// see http://stackoverflow.com/questions/4328711/read-url-to-string-in-few-lines-of-java-cod
String mitHomepage = new Scanner(new URL("http://www.mit.edu").openStream(), "UTF-8").useDeli
```

One reason for documenting sources is to avoid copyright violations. Also another is that code can fall out of date; a Stack Overflow answer from which code came from could have evolved significantly since the years it was first answered.

Some comments are bad and unnecessary. For instance, direct transliterations of code into English do nothing to improve understanding., e.g.

```
while (n != 1) { // test whether n is 1   (don't write comments like this!)
++i; // increment i
l.add(n); // add n to l
}
```

But obscure code should get a comment:

```
sendMessage("as you wish"); // this basically says "I love you"
```

45.2.7. *Fail Fast. Failing fast* means code should reveal its bugs as early as possible.

For instance, static checking fails faster than dynamic checking, and dynamic checking fails faster than producing a wrong answer that may corrupt subsequent computation.

In the `dayOfYear` example, there should be static checking of input arguments, because if you pass arguments in wrong order, it'll quietly return the wrong answer.

45.2.8. *Avoid Magic Numbers.*

45.2.9. *Don't Use Global Variables.* Avoid global variables.

In Java, `static` means there's a single instance of the variable.

In general, change global variables into parameters and return values, or put them inside objects that you're calling methods on.

45.3. **Summary for Code Review.**

- Don't Repeat Yourself (DRY)
- Comments where needed
- Fail fast
- Avoid magic numbers
- One purpose for each variable
- Use good names
- No global variables
- Use whitespace for readability

Part 10. **Embedded Systems**

Lee and Seshia (2016) [27].

cf. Ch. 3 Discrete Dynamics, Lee and Seshia (2016) [27].

showed how state machines can be used to model discrete dynamics.

cf. Sec. 3.1., Lee and Seshia (2016) [27].

pp. 43, Ex. 3.1., Lee and Seshia (2016) [27]. e.g. Consider system that counts number of cars that enter and leave parking garage in order to keep track of how many cars are in garage at any time.

Ignore for now how to design sensors that detect entry or departure of cars.

Assume `ArrivedDetector` actor produces an event when car arrives, and `DepartureDetector` actor produces an event when car departs.

`Counter` actor keeps running count, starting from initial value i.

Each time count changes, it produces an output event that updates display.

Each entry or departure modeled as **discrete event**. Discrete event **occurs at** an instant of time rather than over time.

EY: So there are *3* actors:

(1) Arrival Detector → produces event when car arrives
(2) Departure Detector → produces event when car departs
(3) Counter actor (keeps running count, each time count changes (so it's a FSM)) -¿ produces output event that updates a display

Signal u into `up` input port (from `ArrivalDetector`, upon arrival) is function of form

$$u : \mathbb{R} \to \{\text{absent}, \text{present}\}, \text{ i.e.}$$

$\forall$ time $t \in \mathbb{R}$, $u(t)$ either absent, i.e. there's no event at that time, or present, meaning there is a

pure signal - signal or function of form $u : \mathbb{R} \to \{$ absent, present$\}$, carries no value, but instead provides all its information by either being present or absent at any given time.

Counter: when **event** is present at up input port, increment count and produce output of new count value.
When event present at down input, decrements count and produces output new count value.
At other times, produces no output (count output is absent)

Hence,

$$c : \mathbb{R} \to \{ \text{ absent } \} \bigcup \mathbb{Z}$$

Counter signal is not pure.

Input to counter is pair of discrete signals that at certain times have an event, at other times have no event. Output is also discrete signals.

signals $\xrightarrow{\text{counter}}$ output signal

EY: separate FSM logic from this signal or message passing function.

Definition 10 (Signal and Discrete Signal)**.** *Signal is a function of the form* $e : \mathbb{R} \to \{absent\} \bigcup X$, $X \in Obj\mathbf{Set}$ *s.t.*
$\exists$ *1-to-1* $f : T \to \mathbb{N}$, *s.t.* f *order preserving, i.e.* $\forall t_1, t_2 \in T$, *if* $t_1 \leq t_2$, *then* $f(t_1) \leq f(t_2)$, *where* $T = T_{present} = \{t \in \mathbb{R} | e(t) \neq absent\}$.
e *is* ***discrete*** *if* $\exists$ *1-to-1* $f : T \to \mathbb{N}$ *such that* f *order preserving.*

cf. pp. 52 Lee and Seshia (2016) [27]

For example, in Figure 3.5, for the thermostat with hysteresis (so that it avoids **chattering**, heater would turn on and off rapidly when temperature is close to setpoint temperature), the finite state machine (FSM) could be **event triggered**, like the garage counter, in which case it'll react whenever a *temperature* input is provided.

Alternatively, it could be **time triggered** - it reacts at regular time intervals.

The definition of FSM doesn't change in these two cases: it's up to the environment in which an FSM operates when it should react.

cf. pp. 52, Sec. 3.3.2 "When a Reaction Occurs", Lee and Seshia (2016) [27]

Recall the notion of "reaction": from pp. 48, Sec. 3.2 "The Notion of State", formally define state to be encoding of everything about past that has an effect on system's reaction to current or future inputs. From Sec. 3.3 "Finite-State Machines",

Definition 11 (State Machine, Sec. 3.3 Finite State Machines, Lee and Seshia (2016)[27], pp. 48)**.** ***state machine*** *is a model of a system with discrete dynamics that at each* reaction *maps valuations of inputs to valuations of outputs, where map may depend on its current state.* ***finite-state machine*** *(FSM) is a state machine where the set* States *of possible states is finite.*

However, *nothing in the definition of a state machine constrains* ***when*** *it reacts.* The environment determines when machine reacts. Chapters 5, 6 of Lee and Seshia (2016) [27] describe a variety of mechanisms and give precise meaning to terms like **event triggered** and **time triggered**.

cf. Sec. 3.4 Extended State Machines, Lee and Seshia (2016) [27]

Pushes problem of large number of states into a variable, and there's only 1 state.

e.g. garage counter of Fig. 3.4, in Fig. 3.8

pp. 62, "Extended state machines can provide a convenient way to keep track of the passage of time." but merely pushes tracking time to input variable. e.g. Ex. 3.9, traffic light, Fig. 3.10.

Lee and Seshia mixes extended state machine state with discrete state and variables; but variables act as input.

Sec. 3.5 Nondeterminism, Fig. 3.11, of Lee and Seshia (2016)[27] nondeterministic model of pedestrians (as FSM)

state machine nondeterministic if more than 1 initial state (sufficient condition) or
if $\forall$ state, $\exists!$ 2 distinct transitions with guards that can evaluate to true in same reaction

Sec. 3.6 Behaviors and Traces, Lee and Seshia (2016)[27],

Consider port p of state machine with type V_p
Consider sequence of reactions $\in (V_p \bigcup \{ \text{ absent })^{\infty}$, represented

$$s_p : \mathbb{N} \to V_p \bigcup \{ \text{ absent } \}$$

This is signal received on that port (if it's an input), or produced on that port (if it's an output).
Behavior may be represented as sequence of valuations called **observable trace**.
Let x_i represent valuation of input ports,
y_i represent valuation of output ports at reaction i
Then observable trace is

$$((x_0, y_0), (x_1, y_1), \dots)$$

Observable trace is really just another representation of behavior.

Ch. 3. Discrete Dynamics, Exercise 8:

Exercise 8.

(a) $x : \mathbb{R} \to \{a, p\}$, $a \equiv$ absent, $p \equiv$ present

$$x(t) = \begin{cases} p & \text{if } t \in \mathbb{Z}^+ \\ a & \text{otherwise, } \forall\, t \in \mathbb{R} \end{cases}$$

Recall $T := \{t \in \mathbb{R} | x(t) \neq a\}$. Let $t_1, t_2 \in T$ s.t. $t_1 \leq t_2$. Then since $t_1, t_2 \in T$, $t_1, t_2 \in \mathbb{Z}^+$ (by def. of given x). Let $f : T \to \mathbb{N}$ by identity $t \mapsto t$. Then $f(t_1) = t_1 \leq t_2 = f(t_2)$

(b) Consider pure signal $y : \mathbb{R} \to \{p, a\}$ given by

$$y(t) = \begin{cases} p & \text{if } t = 1 - 1/n \,\forall\, n \in \mathbb{Z}^+ \\ a & \text{otherwise} \end{cases} \qquad \forall\, t \in \mathbb{R}$$

Let $T := \{t | y(t) \neq a\}$, let $t_1, t_2 \in T$ s.t. $t_1 \leq t_2$. Consider $t_1 = 1 - \frac{1}{n_1}$, $t_2 = 1 - \frac{1}{n_2}$.

Now

$$1 - \frac{1}{n_1} \leq 1 - \frac{1}{n_2} \text{ or } \frac{-1}{n_1} \leq -\frac{1}{n_2} \text{ or } \frac{1}{n_1} \geq \frac{1}{n_2} \text{ or } n_2 \geq n_1$$

Let $f(t) = \frac{1}{1-t}$ if $t_1 \leq t_2$. $f(t_1) = \frac{1}{1-\left(1-\frac{1}{n_1}\right)} = n_1 \leq n_2 = \frac{1}{1-t_2} = f(t_2)$

(1) Signal w is a merge of x and y, i.e. if $w(t) = p$ if either x or $y = p$, $w = a$ otherwise

Part 11. Design

cf. Daniel Jackson. 6.170 Software Studio. Spring 2013. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.

46. System Design

46.1. **Client-Server.** Let (notation)
client ≡ cli,
server ≡ ser,

cli actions : { request data, speak to ser}
ser actions: { listen, send data }

cli action: make DNS query. Obtain IP address.

e.g. cli ≡ browser.
browser makes DNS query. gets back IP address.
IP address ≡ unique identifier for machine on network and internet.

cli http request → ser

ser listen to requests on ports ∼ 16000 ports.
Need to specify port.
For http, port 80, https port 443 (decided upon long time ago).

Try the following:

```
dig algoexpert.io
dig google.com

nc -l 8081 # l for listen
nc 127.0.0.1 # 127.0.0.1 for local machine
```

46.2. **Relational Databases.** Structure imposed on data.
2 (major) structures.
Relational Databases (Db's) vs. Non-relational Db's
Tables or relations
columns represent attributes
rows referred to often as records.

Non-relational Db's.

46.2.1. *SQL query (time complexity).* $O(1)$ - hash index on id, or cached result on id from previous query.
$O(n)$ - Do full scan and look at each row;
$O(\log(n))$ sort id and do binary search.

cf. https://www.geeksforgeeks.org/sql-query-complexity/

46.2.2. *Databases definition.* Databases (db) uses disk or memory for 2 actions: record data, query data.
In general, they're themselves servers that are long lived and interact with rest of application through network calls, with protocols on top of TCP or even HTTP.

Some db's keep records in memory, users of such db's are aware records maybe lost forever if machine or process dies.

46.3. **Example: Payments, payments processing system.** Payment design.
Users authorize payments, system interfaces with external payment processes like Visa to copmlete payment.
Queue based architecture.
Payment systems demand high reliability. Not making payments users requested or charging users multiple times - consequences too high.

To achieve this reliability, system puts "payment task" onto a queue that's processed asynchronously.

(1) user u. action, payment request. front-end service accepts u's payment request.
Set up record in database that says u's request, says status request received, payment pending.
record will be used to communicate with user what states of payment is.
front-end service $\equiv s_{fr}$. s_{fr} has action accepts, accepts : u
(2) s_{fr} has action put message on payment queue.
message has information: reference to newly created database record, details about payment (which credit card to use, amount, etc.)
If either step or previous one failed, user immediately notified.
(3) asynchronous processor reads from queue, process individual payments.
primary objective if the processing is to make the payment using external payment service (like Visa) and update u's payment database record based on result.

(4) Once payment goes through or fails in nonrecoverable way (e.g. external service rejects payment due to insufficient funds), asynchronous processor notifies user.

cf. https://www.linkedin.com/pulse/system-design-practice-designing-payment-avik-das/

References

[1] Randal E. Bryant, David R. O'Hallaron. **Computer Systems: A Programmer's Perspective** (3rd Edition). ISBN-13 : 978-0134092669 ISBN-10 : 013409266X Pearson; 3rd Edition (March 12, 2015) https://csapp.cs.cmu.edu/

[2] Bertrand Meyer. **Object-Oriented Software Construction** (Book/CD-ROM) (2nd Edition) 2nd Edition. Prentice Hall; 2 edition (April 13, 1997). ISBN-13: 978-0136291558

[3] Randall Hyde. **Write Great Code: Volume 1: Understanding the Machine** October 25, 2004. No Starch Press; 1st edition (October 25, 2004). ISBN-13: 978-1593270032

[4] Randall Hyde. **Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level**. 1st Edition. No Starch Press; 1 edition (March 18, 2006). ISBN-13: 978-1593270650

[5] Zed A. Shaw. **Learn C the Hard Way: Practical Exercises on the Computational Subjects You Keep Avoiding (Like C)** (Zed Shaw's Hard Way Series) 1st Edition. Addison-Wesley Professional; 1 edition (September 14, 2015) ISBN-13: 978-0321884923.

[6] Eli Bendersky. Are pointers and arrays equivalent in C?

[7] Brian W. Kernighan, Dennis M. Ritchie. **C Programming Language**, 2nd Ed. 1988.

[8] Peter van der Linden. **Expert C Programming: Deep C Secrets** 1st Edition. Prentice Hall; 1st edition (June 24, 1994) ISBN-13: 978-0131774292

[9] Leo Ferres. "Memory management in C: The heap and the stack". Memory management in C: The heap and the stack

[10] Bjarne Stroustrup. **The C++ Programming Language**, 4th Edition. Addison-Wesley Professional; 4 edition (May 19, 2013). ISBN-13: 978-0321563842

[11] Peter Gottschling. **Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers (C++ In-Depth Series)**. Addison-Wesley Professional; 1st edition (December 17, 2015). ISBN-10 : 0134383583 ISBN-13 : 978-0134383583

[12] Carlo Ghezzi, Mehdi Jazayeri. **Programming Language Concepts** 3rd Edition. Wiley; 3 edition (June 23, 1997). ISBN-13: 978-0471104261 https://vowi.fsinf.at/images/7/72/TU_Wien-Programmiersprachen_VL_(Puntigam)_-_E-Book_SS08.pdf

[13] Stanley B. Lippman, Josée Lajoie, Barbara E. Moo. **C++ Primer** (5th Edition). Addison-Wesley Professional; 5 edition (August 16, 2012) ISBN-13: 978-0321714114

[14] Scott Meyers. **Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14**. 1st Edition. O'Reilly Media; 1 edition (December 5, 2014) ISBN-13: 978-1491903995

[15] Ana Bell, Eric Grimson, and John Guttag. *6.0001 Introduction to Computer Science and Programming in Python*. Fall 2016. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.

[16] Erik Demaine, and Srini Devadas. 6.006 Introduction to Algorithms. Fall 2011. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.

[17] Douglas Wilhelm Harder, M.Math. LEL. ECE 250 *Algorithms and Data Structures*. Waterloo Engineering. Department of Electrical and Computer Engineering. University of Waterloo. Waterloo, Ontario, Canada. https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/#introduction-and-reveiew

[18] Martin Dietzfelbinger. Algorithmen und Datenstrukturen 1 - WS 19/20. FG KtuEA, TU Ilmenau. Oktober 2019. Technische Universität Ilmenau. Fakultät für Informatik und Automatisierung. Algorithmen und Datenstrukturen 1, WS 19/20

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. **Introduction to Algorithms**, *Third Edition*. MIT Press; Sept. 1, 2009. ISBN-13 : 978-0262033848
Solutions to Introduction to Algorithms Third Edition

[20] Robert Sedgewick and Kevin Wayne. **Algorithms**. 4th Edition. Addison-Wesley Professional; 4th edition (April 3, 2011). ISBN-10 : 032157351X ISBN-13 : 978-0321573513

[21] Dr. Clifford A. Shaffer. **Data Structures and Algorithm Analysis in C++**, Third Edition (Dover Books on Computer Science) Third Edition. Dover Publications; (September 14, 2011). ISBN-10 : 048648582X ISBN-13 : 978-0486485829

[22] Andrei Neagole. Master the Coding Interview: Data Structures + Algorithms. https://www.udemy.com/course/master-the-coding-interview-data-structures-algorithms/

[23] Prof. Donald S. Fussell (Instructor). CS429H (378H) - Systems I (Honors) (Computer Organization, Architecture, and Programming). Spring 2011. University of Texas. https://www.cs.utexas.edu/users/fussell/courses/cs429h/

[24] Cindy Sridharan. "The method to epoll's madness." *Medium*. Oct. 29, 2017. https://medium.com/@copyconstruct/the-method-to-epolls-madness-d9d2d6378642

[25] David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor. **C++ Templates: The Complete Guide** (2nd Edition). Addison-Wesley Professional; 2 edition (September 18, 2017). ISBN-13: 978-0321714121

[26] Ivan Čukić. **Functional Programming in C++: How to improve your C++ programs using functional techniques**. 1st Edition. 2018

[27] Edward Ashford Lee, Sanjit Arunkumar Seshia. **Introduction to Embedded Systems: A Cyber-Physical Systems Approach** (The MIT Press) Second Edition. The MIT Press; Second edition (December 30, 2016). ISBN-10: 0262533812. ISBN-13: 978-0262533812

[28] John V. Guttag. **Introduction to Computation and Programming Using Python (With Application to Understanding Data)**. 2nd Edition. MIT Press. ISBN-13 : 978-0262529624

[29] **6.005: Software Construction**, **6.031: Software Construction**. 6.031: Software Construction, Spring 2020: Max Goldman, Rob Miller.

[30] Daniel Jackson. *6.170 Software Studio*. Spring 2013. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.[30]

[31] Michael Kerrisk. **The Linux Programming Interface: A Linux and UNIX System Programming Handbook**. No Starch Press; 1st edition (October 28, 2010). ISBN-10 : 1593272200 ISBN-13 : 978-1593272203

[32] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff. **UNIX Network Programming: The Sockets Networking API**. Volume 1. Third Edition. 2004. ISBN: 0-13-141155-1