CS50 Notes
Tanishq Kumar

Learning to program tips: learn by project-based studies, not by mugging the syntax of a particular language. Make sure you have the principles down well, then do lots and lots of curious, creative projects to further your nuanced understanding. Do NOT focus on “learning a language”.

Website vs Web App: a website is an informational source, a set of static pages, like CareerFear. A web app is a dynamic, interactive site that stores and processes data, like Facebook or Quora.

Web applications are developed with HTML, CSS & JavaScript, use programming languages – PHP, Ruby, or Python apart from using frameworks – Rails, Django, and CakePHP.

During this coding journey, use Quora, Reddit (r/learnprogramming), StackOverflow, and your personal tutor, to go through any places where you’re deeply stuck, or want to know “what next”.

Week 0

Lecture 0

Introduces binary, ASCII, Unicode (UTF-8), RGB (3 bytes).

Abstraction: going from something very low level (bits) to higher level (human letters) to more meaningfully interpret the data.

Binary is contextual. A computer interprets a certain binary message in a certain way depending on what program is running; for example, a certain bit will represent colour in Paint, or a letter in a Messenger app, or a literal number in context of a calculator.

Introduction to Algorithms and what they are.

Backend code—that which runs on the servers, doing the actual processing.
Frontend code—that which runs on your device, doing the presentation.

Boolean expression: something with a true/false answer.

Threading is running two programs/routines in parallel.

Libraries include code for specific functionalities that other people have wrote and painstakingly revised the details of, so that we can seamlessly deploy it within our own program.

Week 1

Lecture 1

Introduction to C as a programming language, comparison to Scratch. C is called C because it comes after B, which is the programming language used by Bell Labs, the dominant computer company at the time, which almost monopolised the computer programming market.

C uses “while”, and can loop that forever by using “true”
printing something in C—> printf

the syntax for saying “repeat” is “for” in C—so to repeat something you just create a counter variable and add one each time you’ve executed it. make sure to define the data type beforehand (int), and

= is used to set one variable to a certain value, == is to actually check if two statements are equivalent

so you’d repeat hello world 50 times by doing:

```
	for (int i=0; i<50; i++)
	{
printf(“Hello World\n”);
	}
```

where \n is “end of line”

string *input* = get_string(“ask a question here and store the answer”)
printf(“hello, %s”, *input*); [or you could’ve put printf(“%s”, *input*); where the %s is simply saying I want you to put in a variable input here, with the variable given after this
if you wanted to have multiple uses of this, just use several %s, %s, and then at the end *input, input* (or use %i if *input* was an integer, %c for a character and so forth)

use CS50 sandbox as the IDE, and the code we write in as humans is called “source code”, as opposed to machine/assembly code (0s and 1s)

source code—> compiler—> machine code

the terminal is how you “talk to the computer”, telling it to, for example, run your program, so the first line after writing out your program would be “clang hello.c” which tells the computer to create a file that is hello.c but in machine/assembly code, and this code file is by convention called a.out (assembly output as a result of the previous command)
	the CS50 team abstracted this to a command called “make” followed by the .c name

C, as a language, doesn’t know what any specific function means—like printf, or the data type string. You, instead, have to declare libraries at the beginning which, in great detail, define all of this functionality.

an “argument” is something that you put inside the brackets of a function; it’s input if you will, for example, printf(“hello, %s”, *input*); where *input* is the argument of the function printf

int main(void) is how you start off a program in C

store float detail by saying %.7f to get 7dp
when you check large decimal expansions/stores in floats you find that they are imprecise, a little off, because RAM/other memory are finite and cannot store indefinitely, and so never want to use comparisons with variables that are defined as floats, because they are likely to be imprecise

to get as much detail as a double float, use the data type called “double”—8 bytes instead of 4

|| is OR

to create your own function without any inputs, like cough(), you write void cough(void), where the first void hints that the function doesn’t return any values (it prints something, but doesn’t return anything to store in the system, and, similarly, doesn’t take any input either, hence the argument being void, too. to abstract

```
void cough(void)
	{
	printf(“cough\n”);
	}
```

and to change this such that it takes an argument telling it how many times to cough, we can:

```
void cough(int n) [here we defined the argument n]
	{
	for(int i=0; i<n; i++)
		{
		printf(“cough\n”);
		}
	}
```

and now you can write cough(3) to get output “cough” three times. Remember that the initial value of the variable *does not matter*—it just tells you how many iterations will happen and isn’t relevant to the actual value of the variable itself.

Shorts 1

Conditionals

- if (), if-else, if-elseif-elseif… where in this list one the branches are obviously mutually exclusive
- if-if-if-else, where the branches are not mutually exclusive anymore, only the final if-else is mutually exclusive
- switch: this is where you can list cases of what to do and switch between them based on some input, so if x were an inputted integer, you have a list of cases that you can switch between based on what integer it was—but you have to make sure to write "break" after each one or every one of them will be executed
- cute short expression equivalent to if-else is ?: and is called a ternary operator
    - int x = (boolean expression) ?5:6; means that x will become 5 if the expression is true, and 6 if the expression is false.
- when you have one line inside a conditional, you can remove the curly braces without losing any functionality

Data types

- In most higher-level languages, you don't have to specify data types, but in c you do.
- integers take up 4 bytes of memory each (and hence highest value is 2^31 -1), since you split up half of them to go negative and the other first positive
- unsigned ints disallow negative numbers, and hence double the range of the positive values, other qualifiers (like const) exist in front of data types
- chars take up 1 byte each since only 255 exist in ASCII
- floats take up 4 bytes, and doubles 8 bytes, and hence have more precision
- printf always returns void—it prints something to the screen, but this is sort of a tie effect—it doesn't give a value back, or store the result in some variable. void can thus be a return type, or put into the argument area
- bool is a data type that comes with <cs50.h>
- you can define things as float sqrt2, sqrt3;
- you can declare and initialise (give a value to) a variable at the same time, like int x = 5;

Loops

- infinite loops like while(true)—while loops repeat until the bool is false
- ctrl+ c is how you kill your program
- do-while is guaranteed to run at least once since it checks whether the bool is true *after* evaluating the "do" steps once, commonly used for taking inputs and validating them
- for loops are traditionally used to repeat a fixed number of times
- for has this structure: for(start, bool—until false, increment)

Operators

- you can rewrite x = x*5 as x*=5, and a = a /90 as a/=90, and so forth for speed
- you can use x++ and x- -
- every non-zero value has a value of true when you check for an integer's boolean evaluation, but zero evaluates as false
- you use !x and !y to give the opposite value of a variable, note that this is different to testing for inequality via a != b

Problem Set 1

while loops check conditions first, and do…while loops check condition *after*

you have to be careful with variable declaration because variables have scope—they only exist between the curly braces in which they were declared, so if you're going to use something a lot, make sure you declare it up front

executing a command means typing something into terminal, -o is a command line argument that tells the compiler what you want the machine code file to be called. then you execute the machine code by using "./ filename"

for loops take the structure: for(initialization; condition; update)

When doing something like checking for change, you can use what are called “greedy” algorithms—series of steps that always take the biggest or most immediate solution (locally optimal) and iterate that way. Sometimes these algos give the globally optimal solution, and in other instances they don’t—with the problem of calculating fewest coins needed to give change, they happen to give the global optimal solution.

Week 2

Lecture 2—Arrays & Algorithms

a compiler really just does (a look under the hood to see what the computer is actually doing with that which we type into it):

- pre-processing

parsing all the libraries you ask it to import, such as #stdio.c, etc., so that it can import and be ready to utilise readymade libraries (so that it understands the function you’ll call like get_string etc. without you having to explicitly define it)

- compiling

conversion to assembly code, where each of the functions (printf, etc.) and actions are broken down into small palatable chunks that a processor can execute (move, add, subtract, etc.)

- assembling

conversion of everything written out in the program to binary that a processor can run through it’s logic gates to execute/process the program

- linking

converting any external functions/libraries to binary and weaving them into the machine code created in the previous step so now every single step of the program (whether you wrote it or called it from an external library) is in binary and is ready to the processed— run through it’s logic gates to execute/process the program

if you’re confused about debugging, use help50 before compiling using “make”, or, alternative, add printf statements throughout to figure out exactly what your program is doing at each point and help see wherein exactly the error lies, and use style50 to work out how to better improve code aesthetic
RAM is much faster than long-term (power-off) memory, and so anything currently being used is stuffed into RAM for fast-performance for task at hand.

You can think of an array as a chunk of memory, with back to back spaces in which things are stored—corresponding to a literal series of metal areas on a storage chip like RAM.

When you have the problem of designing a function after the code in which it’s used, create a prototype of that code at the top without explicitly designing the function in detail, and leave that on the bottom.

An array is just a list. You declare it in the following manner:

instead of creating score1, score2…, you just decare int scores[2] in the beginning, and then refer to each value by calling scores[1] etc. from then onwards. This scales better and offers cleaner solutions for large data sets. Indexing an array is going through it, starting from 0.

When you have repeats/copy paste of anything inside a program, that hints at bad design. For example, you may sidestep this problem by introducing global variables (variables introduced at the top of the program, outside any curly braces, that may or may not be called “const”—like the number of scores you want to input in a C program which you may want to change from time to time).

Strings are really just arrays, and can be treated as such.

Strlen(string) is a function that returns the length of a string, called from the library <string.h>

You can actually declare variables inside of loops, like saying for(“a, [declare variable];b; c”);

Under the hood, the computer is storing strings in an array, with one byte per character (ASCII, remember). The computer stores these arrays in memory next to each other, and tags the first character so that it knows where the array starts, but it adds a byte at the end (a null byte—/0) to mark when it ends. And so instead of

using an external library to call in strlen(s) as a crutch, we can actually define the function ourselves with this newfound knowledge:

```
void strlen(string s);
    {
        int i = 0;
        while(s[i] !=/0)
            {
            i++
            }
        printf("The length of this string is: %i\n", i);
    }
```

"Casting" is converting one data type to another, without changing the variable represented by the data type. This is a consequence of data being context dependent—a certain bit representation may mean colour or text in one context, and a number in another. And so, you can "explicitly cast" data by doing something like:

```
int a = (int) s[2]
```

which would convert the second character in the string s into an integer when you call a in the future, so it would show up the decimal representation of the character in ASCII. Alternatively, you can be implicit with this sort of thing, leaving out the (int).

You can write a program that converts input strings of all lower case to output string where all the characters are upper case, and this is done by treating the strings as arrays, and then casting each character into an integer that you add the difference between "a" and "A" to (this corresponds to the shift down the ASCII table to make everything capital), and then print the output. all of this is only done IF the original character was >=a && >=z, or, in other words, between those two ASCII values and hence lowercase.

There are of course library functions that have already been built to do this for us, so if you call in the library <ctype.h> and use functions called "toupper" and "islower", each of which take the argument (s[i]) and can be looped through the string via recursion.

In terminal, you can call a manual ("man") that tells you everything you need to know about a particular function. understanding what a function does well can be an incredibly powerful tool in streamlining your program.

if you ever find yourself copy and pasting functionality (action code, not just variable definitions etc.) then you should probably create a function that abstracts that away and makes it easier for you, so you can just call the function from then on out.

a function takes an input, and can give an output (do something—like print something or compute something) and then can return a value (store a value as a consequence of the input to be used in future—different from the output)

define an array by literally typing toys[5] to create toys1, toys2…and so forth.

turns out we can exert more power over the command line prompts—we can actually find a method to give words/input when we type out the program for execution (didn't it feel a bit silly to type ./mario or ./credit each time without anything else?) in the int main() {} section, normally we'd write void—that is, define the main function as one that doesn't take any inputs. but we can choose to change this, and have the main (overall, program) function take inputs (arguments).

by giving the argument main(int argc, string argv[]), you tell the compiler that you want to give more input alongside just executing the program by typing its name in. string argv[] just says be ready for input that is an array of vectors—like a series of strings. argc tells main how many strings are in argv. so writing:

```
#include whatever

int main(int argc, string argv[])
    {
    if (argc == 2) <that is, if there were 2 strings inputted alongside command line execution
command>
```

printf(“Hello, %s\n”, argv[1]); <that is, print Hello then the second—1 is 2nd item in an array that starts at 0—item that was inputted, in this case the name if you did ./program Tanishq would given Hello Tanishq>
}

so now we have an array of strings (argv[]) which are, in turn, an array of characters. Note that argv[2] is the second word in the inputted array.

by treating words as arrays, and using loops, you have access to not just the words themselves, by iterating through argv[], but to characters within those loops such as by doing argv[i][j], where j is a character counter. you have these command-line arguments because sometimes the program is designed as such that it needs input to be able to control how the program executes (eg. specifying the program you want to debug to a debugging program)

all functions return a value, and return is just the output of a function, that can then be assigned to a variable outside of the function definition. but main() is somewhat special in that it, by convention, returns a non-zero value if something went wrong (we say return 5 at the end for the broken case) and 0 if everything went well.

bubble sort was pairwise comparison—keep switching to correct until everything is right. selection sort was going through the whole group and selecting the smallest number and asking them to move to the end, then ignore them and continue, with the “end” now being defined with ignorance of those already sorted in mind.

fundamentally crucial to appreciate how once you sort through one variable (put 1 at the end etc.) you have 1 less step to go through from then on out, reducing workload and making the iteration process faster. we want to compare the steps required to go through these algorithms in terms of the amount of things we have to sort (n).

for selection sort, you had to compare the smallest value you had in mind with everyone else, and thus make (n-1) comparisons for a sample of n values (starting from 0), and repeat, ignoring the one you’ve sorted, thus making (n-2) comparisons, and so the steps taken is a linear function of n—a summation, in this manner. this consecutive sum, as we know, is [n(n-1)]/2. thus the efficiency of it is on the order of n^2 (highest degree—O is order of).

other common orders of magnitude include O(nlogn), O(n), O(logn), O(1). tearing through the phonebook linearly is on the order of n steps compared to a sample size of n, and so are O(n). going to the middle and then checking if you have to go forward or backward is order of O(logn), as when you make the sample size a trillion from half a trillion, you only have one additional step and so the steps vs sample size graph plateaus very quickly, resembling a logarithmic curve.

an algorithm that is constant time are written O(1), like “open the phone book”.

merge sort utilises extra space, and sorts each quarter of the array at a time, then merges them into halves and places the sorted quarters next to each other, and then the sorted halves next to each other, and so on and so forth until the whole thing is sorted. when thinking of anything that progressively halves things, thing logarithmic, just as the opposite is exponential, progressively doubling things. hence, since you’re halving the workload each time, it’s of complexity log n, but there are n things to sort at this complexity, and so the total complexity is nlogn. when thinking in context of 8 spots in an array, you have 1, 2, 4, 8—so 4 “bands” on screen and starting at band 1, you have to make 3 moves for 8 objects, i.e. log2(objects) * objects steps. That is, merge sort is of complexity O(nlogn)

Shorts 2

Algorithms Summary

sorting

- selection sort
- bubble sort
- merge sort
- insertion sort

search

- linear search

- binary search

worst case runtime is given by O(n) and best case runtime by omega(n)

Arrays

- think of it like a post-office. each post box corresponds to an element, all of which hold data of the same type like all mailboxes hold containers of the same type (small letters of packages), and each can be accessed by looking at the PO box number, the analog to the index number of an element in an array
- declaration takes form: type name[size], like double coolDecimals[10]—goes up to [9]
- you can do cool things like make: bool truthTable[10] and then manipulate them via things like— if (true == !truthTable[4])

```
{
printf("IT'S FALSE! \n");
}
```

- you can add values to an array by doing: truthTable[0] = false, truthTable[1] = true, and so on, or truthTable[3] = {true, false, false}
- you can think of doing truthTable[10][10] as making a two dimensional array, when really it' just making a long one-dimensional array with 100 element
- we can treat individual elements like truthTable[4] as variables, but not the entire array itself—so you can't assign one array to another, you have to loop through all of the elements, equating them one at a time
- most variables are passed by value in C, so when you're assigning one variable to another, you're really making a copy of the first and storing it in the second—this is not the case with arrays, where the second variable actually receive the *actual, original* array, and not just a duplicate of it—they are passed by "reference" while others are passed by "value" (intuition flipped). this is because copying large arrays would take too long, and so you "trust" the function not to alter the array before returning it

Linear Search

- just sift through everything one at a time, O(n), and omega(1)

Binary Search

- despite being much faster—O(logn) only works with sorted arrays, which could affect decision-making about efficiency
- first value at the middle between start and end of the array, and if the value you're looking for is greater than the middle value, set the next start point to the immediate right of that value, thereby eliminating half the problem. Thus you half it each time, and so it's time complexity is O(logn) for the worst-case, and omega(1) for the best case—when it's right in the middle.
- n in big-O notation is typically the number of bits to represent the input, and the output—O(n) is the number of "steps" needing to be taken as a function of that input size.

Bubble Sort

- Where in selection sort you go through all the numbers to find the smallest, then stick it on the left end, here, the biggest number dominated and out swaps every other number, and so this algorithm organises them from the right to the left.
- the pseudocode essentially creates a swap counter, counting the number of swaps you make each pass through the array, and repeats until the swap counter is 0 after the pass through the array has been made.
- remember that you discard elements as they bubble up to the top; that is, your array to arrange decreases in size as you arrange it.
- For the worst case scenario, it's in reverse order and each element has to be bubbled to its correct place, so you have to do n sweeps through the data, where n is the input size, and hence it's an O(n^2) algorithm since each sweep makes n potential swap comparisons.
- If the array is perfectly sorted, you only need to make n comparisons, and so it's omega(n)

Insertion Sort

- start by assuming the first one is sorted

- find the next unsorted one, and insert the sorted portion—or part of it—(which starts as just the first one) in the correct position relative to this first unsorted on, then you have two sorted instead of one, and you repeat

- this way, you only end up making one forward pass through the array, because you go backwards often to insert things in their correct spots

- has O() and omega(n)

Merge Sort

- leverages recursion, the act of taking a problem and repeatedly splitting it up into smaller versions of the same problem and iterating *on itself*

- you take an array, and divide it into halves until you have individual element as your sub array, then you "merge" these, which is when you make comparisons between the sorted portions (which may initially just be single elements) until it's all ordered. When you merge, you create a new sub-array/vertical line.

- if you have odd numbers and you want to split it down a half, choose that the left half is one element and the right half two—it doesn't matter as long as you're consistent

- when you're doing merges with two sub-arrays that have multiple elements, compare all of the first one to the first of the second one, and so forth—this is how merge sort makes comparisons.

- the worst case scenario is O(nlogn), as is the best case scenario, because it only makes comparisons by dividing everything down into individual elements, which requires the same amount of steps as before

- takes more memory than most other sorting algorithms since you have to store different sub arrays at the same time (in "parallel", if you will)

Selection Sort

- go through array and find smallest value, then replace the first element of the unsorted array with that smallest value, and now that portion becomes sorted, and you repeat the whole thing until the entire array is sorted

- O(n^2), and omega is same

Command Line

Unix is an OS, just like MacOS and Windows. You can control your computer and tell it what to do by navigating a GUI, but you get more precision and more power, in general when talking to your computer, by using the CLI. Command line prompts include:

- ls: list, see all files and folders in the current directory

- cd: change directory, so type cd and name of the directory, to see the current directory type "." and to go to the directory one level above that (the parent folder), type ".."

- ctrl+ L clears screen

- pwd tells me my present working directly

- to get all the way back up, just type cd

- mkdir is used to make a new directory

- cp, means copy, takes two arguments—source and destination, you can, for example, copy one file into another that you create by typing the destination

- rm means remove/delete (permanently), add -rf to forcibly do it without a prompt

- mv, from source to destination, to take it from one place to another—effectively the same as rename, so you can mv james1 tanishq1, which then effectively creates a file called tanishq1, copies james1 into it, and then deletes james1

Debugging

- help50 for compile time errors

- it's common to use printf as a diagnostic tool for where things are going wrong eprintf is a tool that can help with this—basically a more explicit print function that tells us very clearly where that particular print came from, so you know that line executed

- debug50 is a tool that can help us when the program compiles, but is not working as intended

- debug 50 then run the file—you can add break points by clicking on the left, and so the program will start and then break (stop) at a particular point you think is breaking down. then you can go to the debugging window on the right, and step into, over, or out of, particular functions to get a sense of what variables are what and what time, and what each line is doing, to figure out where your program is breaking down

Functions

- functions = methods = procedures = subroutines
- they're black boxes—so when we're using known ones we don't really care how they work (in other words how they are implemented), only that they work at all
- we use functions because they allow us to break the big problem into smaller problems, and it's easier to debug systematically the list of functions we have rather than the entire main() code, and they can be recycled and reused to streamline processes
- prototype a function before main(), even if you define it at the end
- to multiply two floating point numbers you might use:

```
double floatMultiplier(float a, float b) [this is a function definition]
    {
    double product =  a * b;
    return product;
    }
```

- double is double precision float, long is same as int, long long is twice the size.
- function definitions *do not have semicolons at the end, but declarations (top of program) do*
- a function that find whether a triangle is possible:

```
bool triangleTeller(float a, b, c); [declaration]
bool triangleTeller(float a, b, c); [definition]
    {
    if (a <=0 ll b <=0 ll c <=0)
        {
        return false;
        }
    else if (a+b>c && a + c > b && b + c > a)
        {
        return true;
        }
    else
        {
        return false;
        }
    }
```

Problem Set 2

- Unencrypted text is generally called *plaintext*. Encrypted text is generally called *ciphertext*. And the secret used is called a *key.* You can think that plaintext + key = ciphertext in a rotational encryption algo.
- c = (p + k) % 26 for a key of k, plaintext of p, cipher of p, in english alphabet, where each letter is assigned a number from 0 to 25
- return 0 is the same as an exit function—ends program
- the "atoi" function converts a strong to an integer, used in CLA, by ignoring the white space in front of the integer—casting would be okay if it were inside the actual program since you can eliminate the white space, but use atoi for CLAs
- convert a character to an integer by saying: char a = '7', int x = a - '0', so now int x = 7
- there are inbuilt isdigit() and isalpha() functions within the <stdin.h> library
- you can treat a character as an ASCII number
- if you want to print the elements of an array, do it iteratively via a loop
- be careful—"a" is not a char, but 'a' is

- often times, your code becomes hard to parse because you haven't done a full formal line by line sweep and so there are 1) redundancies 2) simple errors in plain sight, because your logic is usually correct on the whole
- you can iterate two separate things at the same time without using two loops—use one loop, and create another counter that you increment indefinitely within the first loop, and wrap that counter around using % to get a repeating pattern that you can then use as a loop in itself, embedded in a "parent" loop.
- things don't become any clearer by just sitting on them—you have to actively reflect on the problem at hand whenever you have free time, that's how it simmers in your mind—not just by being thrown on the back burner and moving on to other things.

- when you start, write out clear pseudocode and put those steps into comments in the program area, and then if you can't do one step, try doing an easier version of that step and then extending it—whatever it takes, just don't go look at other peoples'

Week 3—Memory

Lecture 3

- introduces CS50 IDE, where you can actually start implementing CLAs
- you see / at the end of names of folders, but not at the end files— the reason ./hello works is because "." is for current folder, and /hello is the subfile after compilations; and writing the name of the file runs it
- use check50 to check my own code, and now I can start using debug50 myself
- when using debug50, when it highlights a line as yellow, that means it has NOT yet executed that line of code, and instead paused there so you can inspect what everything is at
- strings are not actually a data type in C, that was just a construct of the cs50 library where they defined them to be an array of characters. the reason that any two strings will be != is because they are stored in different memory locations, and it is these locations that are compared when two strings are compared. in contract, other data types are fetched and compared directly (when comparing int 50 to int 49, you really are comparing the numerical values 50 and 49, as opposed to memory locations)
- you identify strings by the memory address they occupy, but also by their length so you can tell two strings apart
- remember that all strings terminate in \0 within memory
- string is the same as a char*, where the * indicates this isn't a char, but the address of that character in memory. this is also called a pointer, because it points to the memory location in which that character is stored—it stores an *address*. even more exactly, it is the address of the *first byte* of the string, because the computer can't pass back the whole string (it could be a book) every time—that would slow things down a lot
- strcmp is a function in <string.h> that compares two char*'s (strings) and outputs an int correspondingly
- lots of variables, when declared (if not assigned) will store garbage values, so don't be surprised
- when you pass arguments (variables) to a function, you're really just passing copies of those variables to the function, that are then being manipulated, instead of the specific original values (which are associated with a specific memory location). so, instead, we should pass *the address of those values*.
- when you specify * in the argument of a function, you ask not for an integer (or char, etc.), but the *address/pointer o that data value instead*. be careful, since * has this meaning in this context only. and alternatively, *a in a context outside the argument of a function means *go to this memory location and get the value*, and thus it outputs a value, as opposed to a memory location. this is a subtle but crucial distinction .
- when you just have a variables (outside the argument of a function), & means tell me the address of a variable and return that (the conceptual same as * *inside the argument of a function*), * means the opposite—get the value in that location and store that, not the pointer itself.
- segmentation fault output means that your program has run into a memory related problem, which is what happened in the "caesar" pset when you took in a null value as input, without making explicitly:
    - if(s == NULL); {return 1}, you can also write !s as a way of saying s is null or doesn't exist
    - OR if (s==0) when s is a pointer/memory location, pointer/location 0 doesn't exist so that is the same as saying when s doesn't exist/is NULL (not that the value stored in that location is 0, which can totally happen)
- when you assign strings to variables, think in terms of memory at a low level, or funky things will start to happen. in other words, saying string t = s, literally, allocates the pointer to the string (memory location 100, say) stores in s to memory location t. and so now you have both t and s pointing to the same memory location, and so any changes to one will affect the other, and they aren't independent like you wanted them to be. how to fix this? create a new memory location to store the same string in both, but now you have autonomy over manipulating one or the other. eg:
    - char *t (just creates a memory location to store a pointer) = malloc(strlen(s)+1) * sizeof(char))
    - this equation makes a new location t, containing a pointer, that points to another new location of size (one more than the string length to accommodate for \0) and multiplies by the memory each character will take (1 byte in this case, so redundant, but makes sense)—free is the opposite of malloc, and is necessary to do after you're done with that memory space so you don't get a memory leak which is basically when you run out after using all of it this way

- toupper (case sensitive) is a function that capitalises things, usually characters in an array that is a string

- we have a function strcopy that exists to do this very thing, and you implement it via strcopy(destination, source)

- scanf is the real “get_string”—the converse of printf(); if you will, used in the same way—after a printf() statement asking for input—just make sure instead of (“%i”, int x), you do &x to hint that you want to input it *into that memory location*

- a pointer has memory size 8 bytes by default

- arrays can be treated as pointers, since they’re both equal to strings

- you can also print the literal memory location of some input or stored value by using %p, which prints the location in hexadecimal

- we use hex for this sort of thing because base 16 is a power of 2 and leads to easy math and also saves a few digits when storing things because it’s more concise

- in the computer’s memory chip, you have the metaphorical “top”—storing the binary of your programs. below that, conceptually, you have the heap, which is the big open area which lots of functions store things in—so malloc(), which allocated memory to a specific location, allocated memory from the heap. then below that you, you have the stack, which is what functions use when they’re called. main is the first function called, and so the arguments/outputs associated with it are stored in the bottom of the stack. thus forth with successive, associated functions.

- when a function returns (finishes) the memory in the stack it was using to store its values (often copies of other variables that it’s manipulating—like swap() swapped a and b, which were copies of x and y from main(), but didn’t actually change x and y themselves), gets reclaimed, which is why you have to think in terms of granular memory space being re-allocated to other space that doesn’t get wiped upon returning.

- “Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, all of the variables pushed onto the stack by that function, are freed (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables.”

- stack overflow happens when you call too many functions to the extent that you overload the limited space in the stack of memory with the arguments/variables of those running functions, and it fills into the heap section, which is the large open space for dynamic allocation—i.e. filling extraneous things that haven’t yet found space to—controlled by malloc, and have to be freed, and stores things like global variables.

- you can create your own data types and programming structures (like cs50 did with “String”, giving you an unprecedented level of granular control over your programs) in a specific way. that is, for example say you ad to store a lot of information about particular students, you could do this by creating several different arrays (names, DOBs, address, etc.) with each number corresponding to a student, but there exists a quicker way.

- instead, say you declare an array called students, with “enrollment” number of students, but with *data type student*. you make your own custom data type, and pu tthem in your own custom header files, like “#include ‘struct.h’”, where struct.h is another file which defines a particular data type of your own. for example, the code defining data type “student” might be:

typedef struct (defining a data structure type being created)

```
{
        char *dorm;
        char *name;
}
```

student; (this is the name of the new data structure)

and then you have an array of students, that you can manipulate by saying things like:

(inside a loop:)

students[i].name = get_string(“Name: ”);

because the .name is a part of the students data structure, which the array “Students” has

- dereferencing a pointer means going to the particular address that it points to in memory

- fopen is a library function that takes in the name of a file and how you want to open it (filename, ‘w’ = write mode so it can be changed), and is called like: FILE *file(); because FILE is a data type built into C which is a pointer, bringing back to memory address of the file, which can then be modified by using the *, & utility.

Shorts 3

Recursion

- a recursive function is one that calls itself as part of its execution
- the factorial function is an example—n! = n * (n-1)!
- when you define a recursive function in this way, you need to have a base case (when n reaches 1 for the factorial function, for example), otherwise it'll iterate indefinitely; as well as a recursive case, which helps you go from starting point to the base case by chipping away at the problem each time.
- you can even have multiple base/starting cases (fibonacci sequence for example) and even multiple recursive steps (collatz)

—>implementing the collatz function collatz(n) that takes a starting value and tells you how many steps to get to the final value of 1 [ignoring declaration of prototype here, just defining it]

THIS IS AN ITERATIVE FUNCTION, NOT A RECURSIVE ONE

```
int collatz(int n)
        {
        int steps = 0;
        while (n != 1)

    {
        if (n % 2 == 0)
            n /= 2;
        else if (n % 2 == 1)
            n = 3n + 1
        steps++
    }


        }
```

THIS IS A RECURSIVE FUNCTION

```
int collatz(int n)
        {
            {
                if (n ==1)
                    return 0;
                else if (n % 2 == 0)
                    return 1 + collatz(n / 2);
                else if (n % 2 == 1)
                    return 1 + collatz(3n + 1)

            }
        }
```

Call Stacks

- This is where functions store all the things they need to run and work while they are being executed. These chunks of memory in the stack are called function frames/stack frames—several of these are piled on top of each other at a given time based on what the most recent function (active frame) being executed is.
- the most recently called function is always the one at the top of the stack, and when a new function is called it's "pushed" on top, and when it returns a value/finishes, it then is "popped" out—all its values are cleared to make new space in the stack
- now you can visualise how when a recursive function like "fact" is called, each iteration calls a new layer onto the call stack, like fact(5) then fact(4) until you reach the base case. Hence, when you have deep recursion or infinite recursion, you'll get a stack overflow error where you run out of memory to store the active frames in.

Pointers (confusing, but important)

- so far, we've only ever passed data between functions *by value*, that is, we've passed not variables to functions, but *copies* of variables to functions.

- this means that a change made to a variable in one function can impact the execution of another function that involves that very variable; and this interdependence between functions wasn't previously possible!

- recapping computer memory:

    - we use HD/SSD to store long-term things persistently. however manipulation of memory can only take place in RAM (short-term).

    - memory in this context = RAM, not hard disk.

    - int is 4 bytes, long long (huge integers beyond 2,147…) is 8 bytes, floats are 4 bytes, doubles are 8 bytes, a char is 1 byte, and when you assign a variable, you access an unused spot in RAM and create the appropriate amount of space, and then store the binary.

- the type of a pointer is the type of data i'll find after using the pointer to go to the desired address. so, for example, int* pointer = &x means that "pointer" is a box in memory storing the address of the variable x in memory, and it is storing *the address of an integer.* when you then access this by saying *pointer = 5, you say *go to the address you find in the storage space pointer* (and so it then goes to x and puts in 5).

- when you declare a pointer, make sure you always initialise it to "null" if you don't have anything to assign it to yet, like this—char *j = NULL; and this is so that if you dereference it, the program crashes, as opposed to going to the pointer associated with the "garbage" value inside the pointer because you haven't assigned it—that could then go and screw up something else, additionally.

- int arr[i] is *actually a pointer to the ith element of the array*. in this sense, arrays are actually pointers, which explains why they seemed "special" initially in that when they were modified in a function, they were *actually* modified, whereas normal variables were not (their copies were, instead).

- outside the argument of a function, &a gets the address of a, and *a gets the *contents* of a.

- * has *yet another niggly thing*; when you're declaring pointers you can't do the speedy int *x, y, z; as * is actually part of both the data type (address of an int, not int) and the pointer name (*x, not x) and so it's best to declare each pointer on a different line, or to do it like this: int* x, *y, *z;

- we said that string was the same thing as char*, now we know this is because arrays are pointers to specific memory locations, and strings are arrays, and therefore are just pointers to characters stored in memory next to each others, that is, char*

- all pointers take up 4 or 8 bytes in memory (depending on whether it's a 4 or 8 bit machine you're working with) since they're all just addresses it doesn't matter if it's a pointer to a float or to a character, you're still just pointing to a particular memory address

- static memory is called so because its size cannot change during the execution of a program, but with dynamic memory you can have it increase or decrease according to malloc() and free().

- the operating system is responsible for taking your malloc() input and finding locations in memory in which to put that information, coordinating it with everything else that is used or unused so that everything can be accessed at any time, in this sense it is the interface between the programmer and the hardware.

- in this sense, char word[length] and char *word = malloc(sizeof(char) * (length)) store the exact same amount of memory, but one in the stack (former) and one in the heap (latter). but with the latter, you keep word, the pointer to the array that is the word itself, in the stack where it occupies much less memory than the array itself which is kept in the heap (where it occupies the exact amount of space that you need for it, and is thus efficient)

- read pointer definitions right to left— in char * word = "a"; word is a pointer to a char. and so what would word, *word and &word convey? well, word would give a hex number—an address in memory where "a" is being stored. *word would give "a", the *value* being stored in word. &word here is somewhat meta—& operator gives the address of something, and so here the output would be the address of a pointer in the stack; that is, a pointer of a pointer.

- NULL is a sentinel value that is meaningless or empty. when a pointer is NULL, it points to 0th value in memory, which in history was a memory location off limits to the programmer, which therefore the computer interprets as meaning it is pointing to nothing.

- pointer vs array differences in C:

    - these two can mostly be treated as behaving the same, since an array is really just a pointer to the first element, at which point you keep reading bytes until you reach /0 to signal that the array has ended.

    - sizeof(array) obviously differs from sizeof(pointer)

    - &array gives a pointer to array[0] whereas &pointer gives the pointer to the pointer

    - defining char array[] = "abc" gives you a series of chunks of memory in the stack saying 'a', 'b', and so on, but defining char *pointer = "abc" gives you a pointer to the string "abc"

    - you can assign variables/values to a pointer to make it point elsewhere, like:

char *ptr1 = ptr2; // where x is an array or variable because memory addresses are just numerical manifestations of memory locations

but you can't do the same with arrays, that is you can't do this:

char arr[10];

arr = ptr2; // because although yes, an array is technically just a pointer to the beginning of the array, the computer still knows that you're storing an array of data so it doesn't really make sense to replace the array of data with one pointer .

- similarly, you can't do things like arr++, whereas you can increment a ptr++ because it's a just a hex number that can be increased by 1.

File Pointers

- we don't yet have a means of storing persistent information (storing things outside the program you run). but you can in fact do this via files, and to do this we will often use file* to point to the memory location in which a file is stored, and alter that—file pointers and pointers aren't synonymous but we don't need to know the subtleties.

- for example, you want to know how a user did in a game after it's over, you need a persistent file. in this section, we learn all about manipulating data within files for this purpose.

- FILE is a data structure (like arrays, hash tables) that stores files—it's an abstraction creates by C to help make our lives accessing files easier. And to edit what's inside the specific memory locations taken up by these files, we use file pointers, that is, FILE* x, where *x is the pointer.

- there are several functions that help us manipulate files and their contents.

- fopen(filename, operation—read/write/append); which returns FILE*, as well as opens the file. That is, it gives us a pointer to that file that can then be used in these ensuing functions. check that the pointer returned is not == NULL because then you'll get an error, as with normal pointers

- fclose(file pointer) which then goes to that address and closes it such that no more input/output operations can be performed on it

- fgetc(file pointer) which goes and gets the first (or next) character in a certain file—but the file must have been opened in read mode (powerful when looped—you can print it immediately after reading each character) and EOF refers to the last character in the file

- fputc(character to be written, file pointer) writing equivalent of the above—and this is how you can write commands that read from one file, and write to another (cp, in CLA, if you will)

- fread() is the macro version of fgetc(), and it returns lots of characters at one (also open in read mode, obv) and takes arguments (buffer—where you want the characters read to be stored—a pointer, size of memory each index must have, qty of memory units taken up, pointer to file), for example:

int arr[n];

fread(arr, sizeof(int), n, pointer);

- fwrite() is exact converse

- other functions include fgets(), fputs(), fprintf(), fseek(), ftell(), feof(), ferror()

Hexadecimal

- Used for memory addresses, which is why it's important

Dynamic Memory Allocation

- so far we've only seen static pointer usage, that is, pointing a pointer to another variable that already exists, and we know how much memory it'll take up. but what if we have no idea how big an input will be come input time, and so need to dynamically allocate memory after compilation? for this, we need access to new memory *while* the programming is running

- statically allocated memory is things that we usually give names to (functions, variables), and that's in the stack. opposite for the heap, which is where dynamic allocation occurs, which is why we said malloc() moves things there.

- stack is at the bottom and so has lower hex numbers and grows upwards, and vice versa.

- malloc() takes in bytes of memory neeed, and returns a pointer to that index where the memory is in the *heap*—since this is dynamically allocated.

- *always check for null after malloc()* to make sure we haven't run out of memory

statically allocate an integer:

int x;

dynamically allocate an integer;

int* px = malloc(sizeof(int));

another example of the contrast:
int x = get_int();

statically allocate to an array (on the stack):
float stack_array[x];
dynamically create an array (on the heap):
float* heap_array = malloc(x * sizeof(float)); [THIS POINTER HEAP_ARRAY POINTS TO A CHUNK OF MEMORY THAT DOES NOT HAVE A NAME—THIS IS ONE WAY WE KNOW IT HAS BEEN DYNAMICALLY ALLOCATED]

- normally all memory created for a function is cleared after it runs. with dynamic memory allocation, it doesn't automatically get cleared and so unless you actively free it, you can end up with a memory leak, slowing performance drastically. so with some browsers, which don't have good free() protocols built in, you leave them running for long enough and they cumulatively clog memory over time, slowing overall computer performance.
- use it via: free(pointerToMemory);

Problem Set 3

- a primitive data type in a language is a basic one, that's built into the language and is the building block for more complex data types that are then defined custom
- when you define a new structure/data type, in this case RGBtriple, and you give it properties like each RGBtriple has a rgbtRed value or each Student has a name value, then you can access those using dot notation, and change them individually in you C program.
- Misuse of malloc() when space in the heap of RAM (dynamically allocated) is not released is called a release because, from the computer's perspective, its pool of free memory to give to programs for dynamic allocation is leaking, decreasing, since one program is not releasing it after use.
- In an analogy of a tool workshop, the drawers and cabinet spaces that store regularly used tools in specific spots are the stack storing statically allocated memory. But if a friend brings in a whole bunch of assorted new tools and dumps them on the floor, that's dynamically allocated memory—there's not permanent, named place to store them, you just sort through them and call on them as time goes on.
- An operand is something that an operator acts on—like the 3 and 5 in the 3+5 operation.
- When you want to check if some hex value is, for example, 0xe_ where _ can be any hex digit, we can use what' called a bitwise & operator that looks like this:
- sprintf(info, "Tanishq, 18, CS/Math") assigns the value of the string on the right to the variable on the left
- fprintf() is like fwrite() but to write long strings to text into a file instead of bit manipulation on the granular level, and it also can be used to just print normally and also take the stderr argument in order to return the fact that there's been an error in input
- when you have problems with variable scoping in structuring a program, just declare the variable you're struggling with at the op globally and set it = NULL, then manipulate it as you did before in the same places
- when you're checking whether some value equals another, check whether you need to declare one of the variables as *unsigned* to get it to accommodate a specific range
- all of the movement of cursor functionality is already built into the f___() functions, you rarely have you manually fseek for something yourself

Week 4—Data Structures

Lecture 4

- valgrind is a CLA that looks for memory leaks or things that will lead to seg faults

- remember that malloc() returns a pointer to the memory you just allocated, and the pointer can be manipulated as an array to manipulate the memory chunks continuously after the first memory chunk —like int *x = malloc(10 * sizeof(int)); returns a pointer to the 40 bytes of memory, and so you can access the memory by using x[3] = 22; which then inputs 22 into the address pointed to for the 3rd of 10 chunks of memory. This shows how pointers point to an *array of memory*, with them storing inside

them both the starting point and how long the ensuing array is, which is why the x[3] trick works—not just the pointer to one memory address.

- usage: help50 valgrind ./nameOfProgram

- the text part of memory (above the heap and thus the stack) is what stores the machine code of the program being run in RAM

- stack is used for static allocation—calling local variables and the functions that employ them (this is where the call stack is because you call functions, and recursion heavy programs can thus cause stack overflow)

- when you call a function and want its output to return/influence something in another function, you have to remember the image of the call stack, where the logic in one function isn't affecting the memory stored in another function unless you employ a pointer to specifically go and make that change.

- *y means *go to the address stored in y (take a pointer, return a value)*, and &y means *give the address of y (take a value, return a pointer)* but this meaning is changed in the definition of a pointer like int *x which just says give me a variable x that stores an address in it.

- .h signifies a header files (bmp.h which defined HEADER data structure and string.h which defined the data type STRING) etc. and .c signifies a c file the within it, defines functions—a library, of sorts.

- we can thus define a structure—"typedef struct" and add the accessory topics within it we want to store data about (dorm, age, for students) and call it "student" at the bottom. you could, alternatively, make two arrays of char* that store dorm and age, and remember that the array index being the same represents the same student, but what if we had 20 pieces of information we wanted to store about each student? would we continue manipulating and working with 20 different arras at the same time? this is what the definition of data structures is for.

- This process of encapsulating all of this related information about a student (or any real world object in this way) is called, well, encapsulation.
- big O notation is a way to represent time complexity, which is a sub-field of asymptotic analysis, since you analyse the running time of programs as you increase the size of input to infinity (n—> infinite, which is why it's asymptotic)

- a huge limitation of storing data in arrays is that you can't change their size as you're using hem—if you start by creating an array of size 2, and then later during the running of the program want to add a 3rd value, you can't without creating a new array, copying the first 2 data values, then adding another one, then somehow freeing the first array (can't just use free() as it was statically allocated). this is extremely rigid, and runs in linear time—O(n)

- A data structure is a way of organising data such that some operations (search, sorting, etc.) can be performed on them with varying efficiency depending on the data structure. A data *type* is a collection of things that all share something in common—ints share the fact that they are all positive integers, chars share the fact that they are letters in the English language. We can also create data types, somewhat confusingly via using struct notation, like creating the student data type, which means all data of that type share the fact that they have a name and a dorm. We can create a node data type, which all share the fact that they have one integer and and one pointer (to another node—a somewhat recursive definition). A collection of these nodes (node being a data *type*) is called a linked list (a data *structure*, like an array, but not continuous, and calls only upon creating one more node at a time and therefore can be extended easily).

- the first node in a linked list doesn't contain a numerical value but instead jut a marker signifying that it is the first value in the linked list (and thus the most important one because you can find everyone else from this first value)

- another problem with arrays is that if yo over allocate space in anticipation, you just end up wasting it, but with linked lists you only use as much as you need. and for addition of a new member, you just

change two pointers and stick it in the beginning, and so now addition happens in constant time. now you start to see how this data structure, in many scenarios, runs in lower time, and is more efficient at using space, too, and this will continue to happen as we further introduce more and more sophisticated data structures.

- if you have two pointers instead of one at each node, you get a doubly linked list, and that ability to go back to the last node you can see how it might be useful for sorting if you go too far and how that you would make the time complexity of the same algorithm on different data structures (linked vs doubly linked list) different. you can also see, now as you develop nodes, how mathematical formulae from graph theory might be useful to implement computationally, and see the field of discrete mathematics and computer science being borne.

- you can create lists where you input numbers as a user, but you want to have the user input as many numbers as they want, but prays have a fixed pre-determined size. and so you can create this program 3 ways, two with arrays, and one with linked lists:
  - you can create a program where the user types in the number of numbers they intend to input, like they start the program by inputting 20, so an array of size 20 is created for their inputs. but this still doesn't accommodate those that don't know how many they want to input until they've inputted them all, so is a crude solution.
  - a more sophisticated approach, still with arrays, would use dynamic memory allocation, which allows for impromptu, on the spot, decisions to input the next number, and continue ad infinitum until ctrl+D is pressed to break out of it. this is done by:

```
int *numbers = NULL; // this creates a pointer storing NULL to start, remember that arrays are just pointers to certain bytes of memory where the lists of data start
int capacity, size = 0; // creates two measures of capacity and current size, both starting at 0

while (true)
{
    int number = get_int("Input a number: \n");
    if (number == INT_MAX) //CS50 version of ctrl+D uses a sentinel value
        break;
    if (size == capacity) // if we have no more space in our array and want more
        numbers = realloc(sizeof(int) * (size + 1)); // reallocates everything in the array that was too small into a different memory location, into a bigger array that can now fit the new element in
    numbers[size] = number;
    size++;
}
    free(numbers);
```

//LinkedList implementation

- the problem is that while the above approach works perfectly on the surface, it still runs in linear time and is inefficient by virtue of having to destroy and then create a new array. we can do better design by using linked lists, created for these purposes (example of how a data structure can fit context). these are the steps by which you'd implement a linked list, with important syntactical/structural features shown:

- start by making a pointer to a node so that when you create your first node, you can use the pointer to start operating on the linked list going forwards.

- forever prompt for numbers, constantly looking for the break operator

- if numerical input passes the test, dynamically allocate space for the newNumber that is being input using malloc();

- now that we've allocated space for the number, we want to actually put the newly inputted number into the space we've malloc'd().

- to do this, we say n->number = newNumber (for a node pointer n that points to the first node), which is shorthand for (*n).number, which means go to the node pointed to by n, and go to the number field, and put inside that the new number inputted.

- similarly, we say n->next = NULL; because we want this new node to be our now last node, which is characterised by NULL in its next pointer field (it can't point to anything if it's the last one in the list).

- but this only works for one inputted int. we now need some form of iteration to put it at the end of the linked list (i.e. to get it to have a relationship with the rest of the linked list), which is done by:

- use ptr check the (next pointer) of the next node to see if it's NULL, if not then set ptr to that pointer, which then checks the next pointer of the node *after that*, and continues until it reaches the node where the next pointer *is NULL* (the last pointer).

- then we change that NULL to point to the new node we just created for newNumber, and we're done, because we already initialised the nextPointer of the newNumber node to NULL—it is now successfully the last node on our linked list

- hash tables
  - we can combine some of these data structures to create novel ones. an example of this is if we were recording names of people who came to a class, we could put people in different "buckets" based on the first letter of their name. searching through that list would be 26 times faster—sure, since it's faster by the power of a constant, the search still runs in linear time, but now the "wall-clock time" (actual time taken to find a name) is that much faster, which is a real, meaningful improvement in performance. these combinations of arrays (the bucket titles each occupy one memory chunk of the array) and linked lists (the actual data being stored—that is, all the data in each bucket we've created).
  - a hash *function* is a mathematical/code operator that takes some input (in the name ordering hash table, that would be the name of a new student), and returns some output in relation to the hash table, (in the name ordering hash table, that corresponds to returning the memory location 0->25 that corresponds to that first letter).
  - an array is a type of data structure that support *random access*. this means you can jump to any point in the array irrespective of having accessed any other pointe in the array, purely by using the index number of that chunk in memory. this is as opposed to other data structures like a linked list, where you can't access node 15 without having been pointed to that index in memory by node 14, so you only support *sequential access* as far as linked lists are concerned.
  - this means that the role of the array in the hash table is to support random access to some property the data is put into buckets by, reducing search time by a large constant. in the name ordering hash table, putting peoples names into 26 different buckets means you can random access any letter of the alphabet, and so you then only have to search amongst all the "k" or "m" names for the name you're looking for, making your search effectively 26 times faster.

- trees
  - the type of tree you saw here was a "binary tree" because it had two children at each node, a left and right child, each left child being smaller than the parent and each right child being larger than the parent. this way, starting at the root (top, first value) it takes us a maximum of log(n) steps to get to any given value. this is because for however many values in the tree, the height of the tree is log(values), and to get to any value, the maximum effort needed is to go all the way through the tree, to the bottom.
  - to implement this data structure, you create a node data type like we did for the linked list, but this time with two links, one to the left child node and another to the right child node. it's different to a linked list in that the node has "children" and you can't go "back" a node since you can't go "up" the binary tree.

- tries
  - short for "retrieval" because its use permits searching for something in *constant* time, and it is a normal tree, but where each node is itself an *array*.
  - so, for a name, in something like a dictionary, you could say that the first node (array) represents the first letter, and so forth.
  - but this seems to demand a huge amount of memory—26 times the amount we seem to need. but, as is thematic, efficiency comes from re-use/overlap in use, so when you have names sharing letters, they can map through the same nodes, and you can have 5 arrays of the alphabet storing an *enormous* amount of names. thus, the storage of Bob is independent of Billy, and it takes the same amount to trace through bob no matter how many three letter names are in the list, which is why we say it runs in linear time).

Shorts 4

Defining custom data types

- you can rename existing data types to make them easier to use, just by using typedef <oldname—like unsigned char> <newname—like byte>

- you can create custom data types that bundles data sets that have shared features in the following way:
    - first you create your “structure” (a new, user-defined data type):

```
struct studentData
    {
        char* name;
        int tuitionPaid;
        int IDnumber;
        float age;
    };
```

    - and then you define this as a new data type that you give a name:

```
typedef struct studentData student; // and now you can just call “student” as a data type like int or char
```

- since the creation of structures and then their assignment to data types is so common, there has been syntax created to combine the two so that you can do the at the same time in this way:

```
typedef struct
    {
        char* name;
        int tuitionPaid;
        int IDnumber;
        float age;
    }
    student;
```

- where you can then go on to use student.age in the obvious way moving forward. if you, however, want to create a self-referential data structure, like a node in a Singly linked list, then you should make sure to refer to the self-referential bit in the beginning so the computer knows what you’re talking about, which is why you can’t create the pointer struct node *next because you haven’t created the idea of a node until the very end:

```
typedef struct nodeType
    {
        int number;
        struct nodeType *next;
    }
node;
```

Singly-Linked Lists

- there are some variations on things we’ve seen so far: trees and heaps are like tries, stacks and queues are quite similar to arrays/linked lists) etc.
- before these, we’ve only had one data structure—arrays—to store like values. structs allow us to store unlike values (different data types under a unified object, like student or node), but not like data values.
- the main limitations of this data structure include that it’s costly to insert things into the middle of an array, as well as that they’re inflexible—we can’t easily modify the size of an array once it’s been made to accommodate new data on the fly.
- an array is primitive (using stdio.h) in C, but we can fix these problems by creating a completely new data structure using pointers, structs, and arrays. this is how progress in CS is made.
- we need to be able to manipulate linked lists in a variety of ways; we want to be able to add and delete elements to the list, create lists from scratch, search through linked lists to find elements, etc.

- to create a singly-linked list, we want to:
    - create the node we're going to create by assigning it to a pointer, say, *new*, via malloc(node) (after definition of the data type at top of program)
    - check that the malloc() worked and that the pointer isn't pointing to NULL
    - initialise the new->number to the input you want to store, and new->next to NULL since this is now the last node
    - now you have created the first element in the list!
- to search through a linked list for the presence of a value:
    - we want to create a function that takes as arguments the pointer to the linked list, and the value we're looking to find within that list
    - create a duplicate of the pointer to the first node (since we don't want to accidentally change that pointer itself, as it's very important)
    - check the *pointer to see if it's the value we're looking for, if not then pointer = pointer->next (set the pointer equal to the pointer in the node pointed to by the current value of the pointer) and then check *that* value
    - iterate until you've gone through the whole list, else report failure
- insert a new node into the linked list:
    - here, we want to create a function that takes in as arguments the value we want to append to the linked list, and the pointer to the first node in the list
    - malloc() another node
    - you COULD insert it at the end, but the running time of doing that would be higher since you have to run your duplicate pointer through the whole linked list to then reconfigure the final node to point to this new node we're creating, but it would run a lot faster (constant as opposed to linear time) if you could just insert it at the beginning
    - so, now, to do so, we first set newNode.next = pointer->next, so that the new node has a pointer to the *current* first node (the soon-to-be second node) and *then set* the head pointer to point to the node we just created.
    - keep in mind that it is important to always want to connect the element into the old list before we move the head pointer to point to the new first element, as otherwise we risk orphaning the rest of the list
- to delete an entire list, we want to create a function like void destroy(headPointer)
    - this works recursively, whereby we use this algorithm: if you've reached a NULL pointer then stop and delete it, otherwise delete the rest of the list, and *then* free the current node. this deletes everything to the right of the current node before coming back to free it, and sets up multiple call stacks which repeat this until you reach the final node's .next, which is deleted, allowing all the others to thus be deleted.
    - if you did this in the wrong order (instead of end to beginning we did it beginning to end) then you would free the element that points to the next element, and so would have no way to access the new element and thus no way to delete it, leading to a memory leak.
- deleting a single member of a singly-linked list is tricky because you have no way of going *backwards* between elements on the singly-linked list, and so you risk orphaning everything after the point you're deleting if you try to do this (since you can't go back and connect the one after the one you want to delete the the one before it). enter *doubly-linked* lists, which make this easier.

Doubly-Linked Lists

- method for doing most things is the same as a singly-linked list, except to delete something in the middle:
    - say you have three elements linked to each other to start, A, B and C. To remove B, you must set A's forwardPointer to C's number, and C's backPointer to point to A's number, then free(B)
- linked-lists support very efficient insertion/deletion of elements (constant time), but in turn, give away the ability to random access elements (now in linear time instead of constant time)

Stacks

- has nothing to do with the stack (static) part of memory, and this can be implemented either an array or LL
- you can think of it as structurally similar to the call stack of functions (ironically which exists in the stack of memory) in that when data is added, it sits on top of the stack, and thus is the only one that can be removed (this last in, first out behaviour is what characterises it as a DS)

- the only two legal operations on a stack is pushing (adding to the top of the stack) and pop (removing it), and we store data not just about the array part of the stack, but also an integer that tells us which is the active (last in) part of the stack data structure right now

array based implementation:

- you'd define a push function that puts something onto the top of the stack, taking in the value to input as well as the *pointer* to the stack onto which you're going to push it, stored in the s.top part of the stack structure you define
- so you initialise s.top = 0; when you add something it is added to memory location 0, and then the s.top is incremented so you add successively to the stack
- you'd define a pop function that takes the pointer to the stack as a sole input, and decrements s.top by one, "removing" the current value at the former s.top location (leaves it there but makes it possible to overwrite it so it's effectively gone), but the pop function, when used, *returns* the number that we've popped

linked-list based implementation:

- essentially the same as a single-linked-list, with the rule that we can only remove from the most recently inputted element to the linked list.
- to add onto the linked list, we add at the beginning, and so via LIFO we must also remove from the beginning. thus, to pop, we move the head pointer to the linked list from the first to the second element, then free() the memory we created for the first element (since this was most recently added).
- stacks, as a data structure, we use if the context fuse demands that the most recently added element is the always one we need to modify/access/delete.

Queues

- FIFO analogue of stack, similarly implemented in two ways
- instead of popping and pushing, you queue and dequeue elements, where you add something at the top, and remove something at the bottom (in the "stack" way of visualising things, if you will)—it's "fairer" data structure in that the element that has been there the longest is the only one that can go.
- array implémentation
- when defining the structure, you include three things in each queue—the array that stores the values, the "front" (analog of stack.top—but here it's the oldest, not newest, thing in the array), and the size of the queue (this is the only unique feature different to stacks).
- array holds the data, the front holds the pointer to the value that is the next to be removed
- when you pop things in a stack, or dequeue things in a queue, you don't do it for no reason, but typically because you care about the number and want to assign it to something or use it in some way
- we have size variable because we always put the next variable at the memory location (front+size), which is always a constant, since if the size decreases by one then you've removed one element from the front/beginning, and so the front increases by one. this means that we can add elements onto the queue with no problem, even after taking one off the front.
- you may need to % by the array capacity as you approach the end of the allocated space in the array
- DLL implementation (SLL is also possible)
- you add the new node this time not to the beginning of the linked list, but to the end, and then remove from the beginning, or add to the beginning and remove from the end (though the former is easier to implement)—the enqueue function takes in head/tail as well as the value you want to append to the appropriate side of the queue
- you remove from the beginning (dequeue) by creating a duplicate pointer to the second element (element 1) and then free(head) and then set the pointer to the DLL to the place that the duplicate pointer is pointing. this function only takes in one argument—a pointer to the queue.

But when would you use stacks and queues?

To store node values in while traversing graphs. Particularly, a binary tree is a type of graph, which makes it easy to visualise depth-first and breadth-first searches, the two ways to most efficiently visit all the nodes on the binary tree/graph.

A *depth-first* algorithm goes deep from the get-go. It starts at the parent node, and then goes to child A, then child A *of child A*, going all the way to the left-most child at the very bottom. You traverse through the left sub-tree first, and then the right sub-tree, whereas in breadth-first search you traverse through levels of nodes —"generations", if you will. You can think of the difference as going vertically in and then across verses

horizontally across and then iterating vertically. For depth-first search, you would use a stack data structure to store the memory location of *d* when you're at *h,* for example (on the diagram above), and so you only need to access the memory location you were *just at before the current one*, hence LIFO. Opposite for breadth-first, which is FIFO, hence using queues.

But when would you need to traverse trees? Well, more specifically, when would you need to store data in trees? Trees are useful data structures because they allow you to store things hierarchically. So tags in HTML are stored hierarchically in relation to each other (this <h1> is inside that <body> etc.) when the computer processes then, and the file folder you're on: mac/user/desktop/leagueOfLegends, is also stored hierarchically. In this sense, you can see how hierarchical relationships are ubiquitous, and so tree traversal algorithms must be, too.

Hash Tables

- Hash tables combine the random access of an array with the dynamic resistibility of linked lists. They take advantages of both and disadvantages of none. This would be a data structure where insertion, deletion and lookup all tend towards constant time.
- Doesn't work to sort the data within it, since things will tend to linear time if you try to do that in this data structure.
- A hash table combines two things:
    - A hash function—mathematical operator that takes in a data value we want to input, and returns a nonnegative integer that is a *hash value*, which is the memory location in the array in which we will store that value.
    - The powerful thing about this is that we can later uses again that function with the data input we're looking for to get it to spit out the memory location it's stored in, and can thus take advantage of random access within the array. But arrays could already do this, this is just a fancier method; so, what's new?
- A hash function will do a few things: use *only* and *all* the data being inputted, be deterministic (so we get the same thing out every time for random access), uniformly distribute data within the array, and generate very different hash codes for very similar data (you don't want all the NSA codes next to each other in memory)
- Use hash functions from the internet, it's generally an art to design them.
- In the case of a collision (different data input returning the same hash value), we tend to iterate through the array until we find a nearby open spot in which to put the new data, so that lookup time in the future tends to O(1) in worst case because it's just a few spots away instead of *n* values away from the hash spot that was output by the function. This method of resolving collisions is called *linear probing.*
- When you start to fill up adjacent spots int he array, you start to get clustering, where if you land on that chunk (which is with increasing probability each time another spot fills up) then you have to move (asymptotically approaching) *n* spots to get to the next spot. Even if we change our probing technique (say, go forward and back when checking for empty spaces) we will inevitably run into this problem due to the finite size of the array.
- To resolve this, we need to somehow "un-cap" the size of this array operated by the hash function. This is where the linked list comes in. Ingeniously, how we solve it is by getting each element in the array to "hold" multiple pieces of data by making each memory chunk of the array a head pointer to a separate linked list, which can re-size dynamically.
- this is why insertion is now a constant time operation, and lookup is still n, but now it's approaching constant time as the size of the array in the hash table increases because when you have an array size 100 it's n/100, and as the number get's bigger, n approaches 1, thus tending towards linear time.
- in this way, while linked lists don't have random access, we can reduce that being a problem by jumping to the relevant "middle" of the hypothetical combined linked list by going to that "bucket" (the first letter in the name or however the data is assorted by the hash function)

Trie

- while hash tables have manipulation complexities that approach constant time, they still cannot consistently and predictably by inserted into, deleted, and searched into, in consistent linear time.
- data structures, in general, map key-value pairs. the key is how to find the data, and the value is the data itself/where the data can be found. for example, an array has keys as the index, and values stored *at* those indexes. hash tables, however, use the output of a hash function as the key to a certain value in an array of pointers to linked lists, and the value is the linked lists in which the data itself is stored.

- in tries, the key is guaranteed to be unique (from the outset—this is a condition) for example SSN number etc., and the value is simply to be though to fas a boolean because you look to trace that path through the "linked list" of arrays (where the arrays are nodes) and if you can, you return *true*, the value you're looking for exists in the trie.

- in the yearFounded—>uniName analogy, 1636 is the key that outputs the value *true* for Harvard

- you start at the root, going through nodes via branches (pointers) to reach leaves. in the yearFounded—>uniName analogy, this means there are 10 pointers that comprise each node, from 0->9. to define a trie node, you might define a structure trieNode that has two different properties—char* uni[20] (a string that stores uniName) and struct trieNode* node[10] (an array of 10 pointers to trieNodes—self referential like nodes on a linked list).

- to search for something, take the pointers dictated by the key, and if you ever reach a NULL it means that data you're looking for hasn't been stored in the trie yet.

- the downside to a trie as a data structure is that it occupies lots of memories (for each character you call in the whole goddamn alphabet/each number from 0-9) but, if it's any consolation, you start to quickly regain efficiency since many words/numbers take the same route through the node by virtue of sharing digits/letters

- these are commonly implemented to store dictionaries, as you can access/check for words in a dictionary in constant time, despite the enormous size of a dictionary as a data set (and we don't mind giving up memory for this since memory is cheap nowadays), and this is implemented in spell checkers, which underline a word if it isn't present in the dictionary.

Data structures overview

- each data structure can be distilled into one alternative of array, LL, hash table and tries. let's exam pros/cons and use cases.
- arrays
  - insertion and deletion are bad—you have to move everything to move one thing
  - lookup is good—constant time and random access
  - relatively easy to sort
  - inflexible in that it's a fixed size that can't dynamically change
  - relatively small memory occupancy which is good—pretty efficient
- linked lists
  - insertion and deletion are *good*—constant time (after searching for deletion)
  - lookup is bad—not random access, and linear time
  - relatively difficult to sort (unless you insert in sorted order, which compromises on the constant time that insertion into a linked list normally is)
  - relatively small memory occupancy which is good (but a large constant factor bigger than arrays)
- hash tables
  - insertion and deletion are fast—constant time
  - lookup still linear, but faster than linked list because you reduce the size of each linked list by a large constant factor
  - hard to sort elements in it
  - can occupy variable size—occupies more memory than a linked list of the same size (since you have *n* pointers to *n* linked lists) but not as much memory as a tie
- tries
  - insertion is tricky/complex because you have a lot of dynamic memory allocation (creating lots of nodes that are arrays on the fly) but fast, since it's constant time (takes in a 4-digit year or SSN number, all of which have a constant number of parts), but also gets easier as you use the trie more since you re-use the same nodes more and more, increasing efficiency
  - deletion is easy—just free some nodes
  - lookup is fast—constant time (8 steps in an 8-digit SSN being key, etc.) but not quite as fast as an array
  - sorting happens as you build it—when you're entering SSN numbers you go from 00000000 to 99999999 and so you sort of construct the trie in sorted order—but sorting as a concept doesn't really make as much sense since this is a structurally different data structure
  - rapidly occupies lots of memory, since each node you employ is an array, so occupies exponentially larger space with node size increasing

Problem Set 4

- to “hard-code” something is to explicitly include some special case in a program in a way that makes the program less general and elegant, and makes the code more convoluted and harder to maintain
- whenever you malloc() something, you have to make sure that malloc() didn’t fail (you didn’t run out of memory) by checking if that thing that you malloc()’d returned NULL, in which case you break

Week 5—Web Technologies; HTML/CSS/JavaScript
Lecture 5

- TCP/IP—these are names given to two of the most ubiquitous protocols that make the internet function; conventions and sets of rules for transmission of information that humans decided on to allow computers to exchange information effectively.
- when you mail someone a letter, you have their address written on the front, and yours on the corner in case it needs to be returned. analogously, computers/all devices have IP addresses, which uniquely identify them in the world of computers.
- IP addresses are 4 numbers, each from 0-255, and thus 4 bytes total. thus you can have 2^32 unique IP addresses. but now as the number of people that need IP’s exceeds 4 billion (2^32), we’ve started using a 128-bit system in place of this 32 bit system, which can house enormously more. this is IP version 6 as opposed to the old (but still ubiquitous) IP version 4. this new one is 8 sets of 16 bits (6 and 4 have nothing to do with the actual amounts inside, just coincidence)
- more than just the IP address though, computers need to add one more piece of information—port number—which tells the receiving computer what context that data is to be interpreted in. for example, you could send a virtual packet of information to a computer and it would receive it since you put the correct IP, but it needs to know whether you’ve send an email, a video chat request, or what—this is what the port number you add tells it.
- URL is the unique resource locator, and it’s the english equivalent of the IP address of the server that you’re inputting—so you’re just inputting the actual address of the website (server of FB, for example) that you want your computer to connect to
- when you enter the english URL, the device contacts the local DNS (domain name system) server (uni campus or apartment building or whatever) and asks it what that english means, and the DNS server returns to the device the numerical IP address. think of the DNS as the “phone-book of the internet”, so the metaphorical envelope you receive back after typing this URL will have your computer’s IP address on it, and the contents of it will be the HTML/CSS/JS code that comprise the web page you were trying to access, which you computer then processes to display to you.
- TLD (top level domain) is the last part of the URL that denotes a word or country (ae, uk, com, net etc.)
- the “www” refers to the specific server under “example.com" in which all its internet-based stuff is stored; a literal set of machines in the google headquarters store and process the data associated with www.google.com, and there can be hundreds of other servers like translate.google.com or mail.google.com and so forth. but “www” is no longer necessary since most DNS’s have been coded to recognise english inputs with or without it, returning the same IP output either way.
- when you input example.com/index.html it means the contents of the envelope you receive back from the web server will literally contain a text (hypertext markup language) file that your computer will then process.
- the “http” is (hypertext transfer protocol) is just another protocol. when you meet someone, they know to extend their hand for a handshake—this is a convention, a shared set of rules, that’s all a protocol is, even between computers—just like how clients at a restaurant just somehow know to request a meal, and waiters just know (have been pre-programmed/instructed to) just bring it to them. in this same way, the dynamic between computers is somewhat akin to the client-server relationship, where all computers have been programmed to obey certain rules in navigating that integration (HTTP defines this set of rules)
- when one computer extends the handshake, the other can respond with a number to clarify its position on the exchange, with 200 being OK—all good to go, 301 meaning error, 404 meaning the IP address you asked me for doesn’t exist, and so forth. you can learn more about these by looking at the “network” tab of the “developer tools” in chrome
- html is not a programming language—you can’t use it to get a computer to perform computation, logic or calculation, it’s just a markup language—it lays things out according to instructions.
- html is *interpreted*, instead of compiled, and it isn’t broken down into machine code—no binary, just literally translated line by line and put into action—all interpreted programs (incl. python, for example), are like this
- the <head> of a web page contains metadata/URL info etc., and the actual contents of the page only really begin in <body>
- you should start each document with <!DOCTYPE html> then <html lang = ‘en’>

- remember, port numbers on a packet tell us about the context of the package, and so when we run an html program on the online IDE, we need to have a different port number to 80, since the context of the package (a private server running some custom code) is different to that of the actual web page the IDE is built on (a public server running wide-scale code)
- the “alt” bit of an img src labels the image, for example for blind people getting an audio recitation from the software
- a link is a hyper-reference, and you need to anchor it into something (*something* needs to act as the link) hence the <a href>
- you can use tables to do things like make phone pads, start a <table>, then <tr> to open a row, which will contain lots of <td>s (one cell each)
- just inspect a web page you like and sit down and spend some time parsing how it was put together once you have a rudimentary knowledge of html/css/js and you can start learning how your favourite sites are made so you can copy their styles
- since each browser (chrome, safari, edge) has a slightly different protocol for how they interpret html, each will give your website (that has the same html code in each browser) a slightly different end product. validation checks exist for html in the same way as help50 and debug50, but different checks that you’ll be pointed to in the psets.
- when you type in a search result for google.com, you don’t see the URL as /index.html because it is a *dynamic page* and it’s not like they return the same one page regardless of input. and so to have this dynamism, they wrote a program that finds and outputs the search queries, and they run the program with the input of the search engine, thus giving us the URL we want, which is why you see google.com/search?q=cats where q can be thought of as the equivalent of the input, or the CLA in C. the ? reflects the fact that there is user input
- <form> gives you a space for user input, by having sub-tags like <input name = ‘q’ type=text> where the stuff after input is called an attribute (fittingly). this tag creates a space for user to type text, and then you can separately use another sub-tag to make a submit button as follows: <input type=“submit” value=“search”>, and this is the sort of thing you learn by going through the documentation or seeing it implemented on someone else’s site. the core functionality is important, not the jargon or vocabulary—that you can learn as you go along. you can then link this to Google’s search by adding an action attribute to the opening form tag as such: <form action “google.com/search" method=“get”>. all this does is mimic the URL format of google
- and thus, you’ve implemented the front-end of Google! you make a search button that has enough logic to display search queries that are output by the back-end search program (which is in turn written in python or c++ etc.)
- to input weird symbols like copyright, and you input a weird number like © for the copyright symbol
- any html page is broken up into, structurally, a header (not the same as the head), main, and footer, all inside body, and any tag can have a *style* attribute like <header style=“font-size: large**;** text-align: center”> and so forth. so you can see the structure inside each tag for employing CSS (the style bit, has keyword (font-size, for example), a colon : and a description (large, center, for example) and then a semicolon separating it from the next style tag (text-align, in this case)
- the reason it’s called *cascading style sheets* is because it has hierarchy built into it. for example if you were aligning the header, main and footer all to centre, then that’s a redundancy, just move the centre alignment to the body tag (the common parent tag) and then it applies to all through by itself, by cascading—this is better program design, as we were mindful of in C, also.
- in practice, even though we modify html tags by adding style attributes, we probably want to move all the CSS to a different file altogether for the sake of uniformity and clarity so someone who doesn’t know CSS can work on the html file and vice versa, just like in C files we would have header files that would get compiled with the file we were writing come execution time (this is an example of abstraction).
- the way we separate the data from the presentation of it is by abstracting things away to the type, by writing a <style> tag at the top, and defining “classes” in that store certain properties for easy retrieval/re-use later (a bit like creation of structures in C to be re-used), for example defining:

```
.big
{
        font-size: large;
}
```

and so whenever you then make text that you want to be big, you can just use <main class=“big”>TEXT</main> to make this simpler. moreover, you can actually imbibe normal HTML tags with properties if you’re going to use them like that:

```
body
{
```

```
        text-align: centre;
    }
```
which then aligns everything present in the body to the centre, and you can remove the use of “class” altogether, resulting in clean HTML with no CSS in it. taking it one step further *still*, you can even remove the CSS from the top of this file, and put it in a different file for another layer of abstraction to allow separation of data and its presentation. you do this by adding in the head:

```
<link href = “cssfile.css” ref=“stylesheet”>
</link>
```

// the “stylesheet” just makes clear the relationship between the .css file and the file in which its being called (the parent HTML file) is that the former is a stylesheet for the latter’s use, and this is similar to C’s #include mechanism.

- HTML is the structure/content, CSS is the aesthetic/style, *JavaScript* allows for *dynamism* and *intreactability*—the thing that makes it so that mail magically appears in your gmail inbox, the thing that makes Google maps move around so seamlessly.
- JS is again interpreted, like HTML/CSS and Python, but this time it *is* a programming language that allows for logic and computation. but it’s slightly easier in that it doesn’t have the pointers that we have in C, and doesn’t care so much about data types.
  - in C: int counter = 0, in JS: let counter = 0;
  - counter++, counter += 1, etc. are all the same in JS
  - for if statements, forever loops, etc., syntax is the same
  - for loops are the same (but defining i using let, not int)
- think of webpages as trees with different parts (headers, footers) being children of others, bound together by tree relationships. in C, we had to manually construct trees by allocating specific memory and keeping track of pointers, in web development, all of this is done “under the hood” (which is why it was so great to start by learning C)
- what the web browser (chrome, firebox, safari) that takes the metaphorical envelope containing HTML, and does the act of manually converting the line-by-line input into a data structure in memory that will store the data we’re manipulating using JS. so someone at each of the web browser companies had to write the code that builds these data structures using the HTML input in the same way as we implemented data structures in C. and after we’ve created a data structure in memory from the HTML input, we can now perform *logic* and *computation* on the data present within it (enter JS). so when you manipulate the data structure in gmail by a new email line popping up, it’s just the creation of a new node in the HTML data structure, and that has to be done by “someone”, and that “someone” is JS.
- this is how you can control the user’s browser *after* they have loaded your webpage, JS code gets processed on the client’s end, not the server’s (like C or python might, for more dirty back-end processing and computation of data the user inputs and sends to the server)
- the implementation of the Google search page wasn’t “interactive” by itself, we just statically made a program to create a URL of a certain structure, then pass it to Google, the place where the actual interactivity/computation would take place, and then Google would return the search results. now we want it all to happen with us when we’re introducing JS.
- you implement JS by using the <script> tag in the head of your HTML file (where <style> is kept as well)
- in JS, you forget about being explicit with data types, and so can define a function (that takes no input) as *function greet();* and instead of using printf(), you can just use *alert* to have a pop-up come up on screen.
- if you wanted to write a JS program to take in a name then output it on a web browser (definitionally interactive), you’d take an input name by having a <form> then under that, a sub-tag that is <input id=“name” type=“text”> and then, at the top (in the JS section), you’d assign some variable *name* to = *document.querySelector(‘#name’).value* where document is like a structure in C (which is why you see dot notation) and represents the tree storing whole HTML file, and querySelector is a primitive function that finds the node in the tree saying “name” (the # is to say you’re looking for something with *ID* name, not just a “name” tag, and the .value extension just takes the value inside that node (again, abstracting away the idea of pointers)
- you can add *autocomplete* and *autofocus* attributes to the input sub-tag of the form tag, where the latter automatically starts your cursor on any field open to the user without you having to click there, and you can set these attributes to “off”
- prime example of the difference between JS and C lies in how they concatenate. If you wanted to combine 2 strings and output them together, in C, you would have to make two char* array[] to store each, then make a second to store their combination, iterate through each of the first two to copy into the large array (being mindful of the null operator at the end of each array), then print the second array. in JS, you literally do: alert(string1 + string2); and you’re done. This is why even though C is more powerful in the sense that

you have more autonomy over every little thing, it's also so much more tedious and abstracting things away, as JS does, makes it a lot easier to work with and to focus on actually solving the real-life problem at hand instead of focus on technical implementation details

- JS is powerful because it enables the browser to respond to events (mouse movement, dragging, clicking etc.) in real-time because the code is processed client-side
- you'll see lots of *<tagName attribute = "value">*
- when you see document.querySelector(), immediately think of plucking a node out of the HTML tree structure and manipulating the data inside that. a program that changes background colour based on what button is clicked might look like:
- a program that changes font size upon change in selection in a menu might look like:

```
<!DOCTYPE HTML5>
	<html lang = "en">
		<head>
			<title> Dynamic Background </title>
		</head>
		<body>
			<button id="red">R</button>
			<button id="green">G</button>
			<button id="blue">B</button>

			<script>
				let body = document.querySelector("body");
				document.querySelector("#red").onclick = function()
					{
					body.style.backgroundColour ="red";
					};
				document.querySelector("#green").onclick = function()
					{
					body.style.backgroundColour ="green";
					};
				document.querySelector("#blue").onclick = function()
					{
					body.style.backgroundColour ="blue";
					};
			</script>
		</body>
	</html>
```

- note that when you're using . notation in JS, you change the style "background-colour" to backgroundColour—this is convention
- note the use of anonymous functions that are just action items but not explicitly named since they won't be used again
- now let's write a dynamic site that allows the user to change the font-size:

```
<!DOCTYPE HTML5>
	<html lang = "en">

			<head>
				<title> Dynamic Text Size </title>
			</head>

		<body>
		<p> TEXT TEXT TEXT </p>
			<select>
				<option value="xx-small">xx-small</option>
				<option value="small">small</option>
				<option value="medium">medium</option>
				<option value="large">large</option>
				<option value="xx-large">xx-large</option>
```

```
        </select>
    <script>
        document.querySelector("select").onchange = function()
            {
        document.querySelector("body").style.fontSize = this.value;
            }

    </script>
    </body>


</html>
```

- note that the dropdown selection box is known by the *select* tag
- note the use of .style.fontSize after the DQS being equated to *this*

Shorts 5

Internet Primer

- How do we get IP addresses? DHCP is an intermediate automated system between you and the internet that first assigns you an IP address based on available ones, but an actual human system administrator would have had to do this in the past

- no one DNS exists for the world, but micro ones instead exist for smaller locales, which are then aggregated by programs that demand larger sets of data (like google will need to be able to search through most that exist to present them as websites, and so they aggregate smaller DNS lists internally) and so the DNS system is decentralised across many servers

- The router in your house acts as a traffic cop that redirects all of your individual evics' request under the name of one unified IP address, thus sidestepping the limited number of IP addresses in IPv4 (thus it *routes*)

- A router is an example of an access pointer (a modem, switch, are others), instruments and devices that facilitate our connection to the internet

- at home we have small, local networks that are woven together, and these small networks are then interconnected via access points to create an *inter network*. Think of these small, local networks as isolated with only a single way in or out (via the access pointers) and draw the web picture in your mind).

- somewhere on each of these small networks exist the services/things we find on the "internet", but really, we're just connecting to another small, local network to use them (Google is its own small network which we connect to, and it, in turn, connects to Facebook to get some data, etc.). There is no one, unified thing that everyone connects to called the internet, the internet just refers to the rules and conventions ("protocols") by which these small networks share data on an equal footing

IP

- the image of small networks all connected to each other is misleading. for fast data transmission, data has to go through wires, and so we want to have wired connections between all nodes (networks) so that fast data transmission between them is possible, but not have all networks connected to each other (way too much wiring needed).

- so how we solve this is by connection node 1 to node 2 and to a router, which is connected to node 3 and node 4, and another router, and so forth so that there exists *some path* for data to get from node 1 to node X, but it doesn't need to be connected directly. each router stores information about IP addresses such that it can direct data to the correct node as needed, and the data reaches its destination node recursively—the router passes it one step closer, and then hands it off to another router.

- the internet protocol literally stands for how these networks are connected. network 1 will have IP address 1.X.X.X and so forth. and so the routers can read these first digits and send data along as needed (by a graph algorithm, obviously). so this is the set of rules that the routers and networks follow in moving data, and is the literal meaning of the inter-network protocol.

- on a small scale, this presence of routers can make things more inefficient, but it dramatically cuts costs on a large scale.

- data is sent in packets to speed up transfer because large chunks would bottleneck the network's data transmission if the sizes of messages being transmitted were different enough. this also means, in the case of failure, the packet being dropped (lost) just loses a tiny bit of data as opposed to the whole thing. IP

doesn't just dictate how information gets from A to B in a series of connected networks, but also how the data is broken into packets. and thus the routing table routers hold also has to hold information about the "cost" (time) of sending things via different routes in real-time, and the protocol for this is TCP (transmission control).

- in case of "traffic" in one path, the IP is responsible for re-routing data across another route to provide the optimal speed algorithmically, and this can only happen if multiple options to get from A to B exist, and so adding connections still makes networks more robust (with the same drawbacks as initially—cost and labour of wiring)

TCP

- responsible for getting a particular packet to the right place *within the receiving system*. to do this, it looks at what the packet is for (port—context), and thus you can see how TCP/IP are inseparable in networking.
- TCP *adds to the data*, adding metadata about how many packets the receiver should check they got, and in what order, guaranteeing perfect delivery (which IP cannot do since it doesn't do this) since the receiving machine may receive packets out of order (since they took different paths/times to get to end)
- common port numbers include 80 for HTTP, 443 for HTTPS, and 25 for SMTP (email), amongst others.
- TCP adds metadata to the packets of data and parses them (on the receiver's end, that is), and layered on top of that, IP adds another layer of metadata (what machine needs the data)—a layered doll, or an envelope, and there are various other subtle layers (like return address in case of failure) but that's the main gist.

HTTPS

- This is one of many "application layer" protocols (SMTP, FTP, etc.) that dictate the rules not for how things move from place to place, but for how to request data from the internet and return it in the first place. analogously, the protocol for meeting someone is to put out your hand, and shake theirs up and down, and then retreat your hand, and similarly, there is a series of rules for how computers may request information from each other. this does *not* have to do with moving data from place to place like TCP/IP
- takes the form of: GET / HTTP/1.1 // looking for you to return the homepage, talking in french (the analog of 1.1—a version of HTTP)
  Host: cats.com // from this address

and so the receiving computer parses the structure according to it's dictionary of what HTTP 1.1 means—first it'll have what it's looking for, then in the next line it'll have the address it wants it from, so that it can manipulate that information logically internally based on the structure they agreed the message would follow
- you can get various status report messages during the HTTP handshake, including ones for OK (200) or failure to find the page (404) etc.

HTML

- is an unordered list, with tags being <li> and same with <ol> which is numbered
- <input> is a sub-tag under <form> and a way for users to input data, with its attributes being *name* //unique identified and *type* // what sort of data it accepts, like text/password/checkbox or otherwise. input is a self-closing tag of the form <input/>
- is used to create a page division
- you make //comments in HTML by doing <!- - comment - - >

CSS

- aside from using the style="text-align" attribute within HTML tags, you can define all the css at the top like this:

```
body
{
        background-colour = "blue";
}
```

- the *body* in this case is the *selector* (from document.querySelector in JS, which is responsible for finding something in the HTML document) and the thing in between the curly braces for that selector is the *style sheet*. as such, the *selector* is modified by the *style sheet*

- other CSS properties to be used in style sheets to modify appearance of the site can include *border width style colour*, where style = dotted/dashed/ridged etc., *background-colour: hex*, *colour: hex* (this is text colour)

- *font-size: (small, 12pt, 80%)*, *font-family: Times New Roman*, *text-align*

- instead of just using a tag selector by starting with an existing HTML tag like body or h1, you can use an ID selector that only applies when you set the id="" attribute of the tag to that specified in the ID selector

- similarly, you can use a "class" selector that will only apply to tags given the specific class attribute, with an example being an attribute in a tag being *class* = "X", as this is a generalisation of the "id" tag since the ID tag acts as a unique identifier (you can't have many different tags operating under the same ID)

- to call a style sheet externally, use the <link href="stylesheet.css" rel="stylesheet"/> function.

- in css, comments are made using /* comment */

JS

- JS is actually derived from C, and is about as old as Python

- in the same way you can <link href="cssfile.cc" ref="stylesheets"> link in CSS from external files, you can also with JavaScript like this: <script src="jsfile.js">, which is particularly important to do with websites where one dynamic interaction is used again and again

- conditionals from C are the same, we define *local* variables using the *var* prefix in JS, and others using the *let* prefix

- you can mix data types when it comes to arrays in JS, done so by: *var array = [1, 3, 4.3. true];*

- JS can behave as an OOP language—an "object" is very similar to a "structure" in C, where our "fields" in C structure (like student.name) are analogous to *properties* in JS, where you can't access "name" (a field of the structure) without mentioned "student", since name only makes sense in context of student. objects have properties, which are analogous to these "fields" but also have *methods* which are the equivalents of functions, and these functions are meaningless outside of relation to their objects (just like name is meaningless outside of student context) and we define this method inside the creation of the object (you can't do this in C—imagine trying to define a function inside of a struct definition)

- object-oriented means that the object is central to everything; properties (fields), methods (functions), etc. and so if you wanted to call a function in OOP, you'd use object.function();

- a *key-value pair* is just some property and its value; that is, text-align (key): centre (value)

- normal variables (*var x = 0)* are also, in a sense, objects, just with one property instead of multiple that we define upon creation of the object

- iteration: if we wanted to iterate across all the key-value pairs of an object (since they're so ubiquitous) like changing the name, ID number, major, etc. of a student, how might we? in JS, iteration is used as follows:

```
for (var name in student)
        {
                // use student[name] in here to iterate across keys
        }
```

OR

```
for (var name of student)
        {
                // use name in here to iterate across values stored in those keys
        }
```

so, for example, we declare an array *var* wkArray = [Sunday, Monday,…]

to iterate across the *number/index* (keys in this case) of days, we use *in*

that is:

```
for (var day in wkArray)
        {
                // use wkArray[day] in here to iterate across keys
        }
```

giving output: 1,2—>7

to iterate across the *actual values* (keys in this case) of days, we use *of*
that is:

```
for (var day of wkArray)
    {
        // use day in here to iterate across keys
    }
```

- *console.log();* is the equivalent of printf();
- when concatenating if you want the + to be a binary operator for two integers, use the parseInt(string) to convert it to an integer (like atoi()). if we specified a data type for each variable, we wouldn't need this function to disambiguate, which is a trade-off we make.
- there are multiple primitive "methods" (functions) that can be used to manipulate arrays, which, in JS, are (surprise) objects (structure analogues), like array.shift() and array.pop() etc.
- another important method for use on arrays is *map(),* to be used to do some operation on each element in the set (array). say we wanted to double everything in an array, we could explicitly create a function double() and then iterate that through the array, OR we could do this:

```
arrayNums = arrayNums.map(function(num)        {
    return num * 2;
    }); // epitomises OOP, this is an anonymous function that takes in the function on num that is
```

defined within the curly braces that follow

- *event handlers* refer to some JS code that executes conditionally on some event taking place, like *onclick*, an attribute in HTML that executes some function if the tag it's nested in gets clicked, like say alertName, a JS function we define to execute in response to *onclick*:

```
function alertName(event)
    {
var trigger = event.srcElement; // this returned the element on the page that triggered the function
alert("You clicked on" + trigger.innerHTML); // this returns the name of the trigger we wrote
    }
```

more on OOP:

- first type of programming was functional/procedural, where you simply wrote a bunch of functions that outputted data, and then used those functions to accomplish the task at hand. this is what i've been taught in the C section of CS50 (C is a functional programming language)

- the problem with this approach is that if you have a complex program, and modify one function, you have to change all the others to accept different inputs

- in OOP, we combine related functions and variables into *objects* (remember how it's like a struct (stores variables called properties/fields) but it can store object-specific functions, too)

- you can think of a car as an object with properties like colour and license plate number, and methods like start(), stop() that only make sense in context of the object. this encapsulation of methods and properties is thusly termed.

- procedural implementations have variables and functions decoupled, and OOP the opposite. one of the telltale signs of procedural code is functions with lots of parameters (since in OOP you store the parameters (variables) as part of the object itself). the lack of parameters in functions makes it easier to maintain the functions when there's lots of them.

- another defining feature of OOP is abstraction—if you make changes to the inner variables and methods of an object, it doesn't affect the rest of your code, since on the outside you only ever deal with the object as an input/output, not what's inside it (and the object has not been changed)

- inheritance is when lots of different objects have structural features (properties or methods) in common and you so you can link all those objects to one generic object that has those features, thus eliminating any redundant code. polymorphism is an extension of this, where you can encode any specificity into the specific objects, while they all share access to the generic object, thus getting the combination you want with no wasted code. this is why OOP is just a more sophisticated, elegant way of structuring programming languages—they don't fundamentally change any of the logic, algorithms, or how we solve real-world problems.

Problem Set 5

- Bootstrap is a library of HTML, CSS, and JS code used to build responsive websites easily. Often times, creating appealing visual displays and fancy animations can be a chore to do by hand, requiring thousands of lines of HTML/CSS code. Bootstrap writes out templates for many different types of responsive sites that you can copy and use in the design of your own site, extending and filling in the gaps as required. All of this is free. It's like WordPress, but for web developers.
- It does things like define classes called "containers" and "jumbotron" so that you can just import the library file at the top, learn the functionality you can call by watching a tutorial, and start using all these pre-built CSS styles and JS scripts, thus accelerating the web design process.
- Bootstrap is a library mainly used to create "responsive" websites. Responsive, in this context, means that the sites are responsive to the medium on which they are opened—they will be optimised for mobile, laptop, iPad, and more, and the code is written so it opens in a style that works best for the device on which it is opened.
- The way this is done is as follows: it is impractical to write a new form of the website to be displayed on different screen sizes, resolutions, and shapes, and catering dynamism to different scripting abilities. Instead, we make all the code able to respond to changes in these parameters of the device it's being opened on by itself instead of hard-coding all possibilities.

Week 6—Python
Lecture 6

- In web development, the "DOM" is the tree—the storage of the HTML/CSS/JS as a linked list in memory—and thus you access things in the DOM using document.querySelector();
- Why Python? Why didn't humans decide on one language? the reasons one is made give it certain strengths, but those same defining features make each language unable to (or make it more difficult to) do certain things, which are then complemented by other programming languages.
- Python is used in the same way as C (command-line), and is multi purpose (from scientific computing to back-end web dev) but it can also be used to *generate other languages* and to factor out (abstract away) much of the commonalities in all our HTML/CSS files.
- if you rewrite *resize* in Python, it takes literally 20-30 lines of code, as you import some libraries, define some variables, and say outFile = infile(width *n, height *n) and that's that. Done. No dealing with padding, no thinking about memory chunks, no thinking about pointers or null characters or iterating over horizontally and vertically. In this sense, Python was built for exactly this, to execute these simple programs in a simple way.
- in Python, we define a variable by *counter = 0* (no more semicolons!) and x++ doesn't exist
- no more curly braces when dealing with conditionals in Python
- indentation and white-space is *important* in Python, even if it wasn't in HTML, CSS, JS, or C.
- we can use *for i in range 50:* as an iteration 50 times
- where the language gets particularly powerful is in the wide range of data types we have available, including *lists*, a data type that is arrays that can automatically resize (another way Python makes life easier!)
- we have a set data type (math) and a tuple (groups of related things etc. like pairs of coordinates), dict for lists of words, and more, hand-crafted for our use.
- instead of just #including a library, we now have to specify what functions we want to import from which library—so say *from cs50 import get_string, get_float*, etc.
- we don't compile Python programs because they're interpreted—we just write *python file.py* because the file *python* is the interpreter that does whatever is in the program itself
- when writing hello world in Python, we don't need to include any libraries from the outset, don't need to define a main() function that tis the actual program, and nothing else—literally just print("Hello, World!")
- you can add (f"{some operation or variable here}" to get Python to print something in the same way that we used %s in C
- in Python, division means division; if you divide 1 (an int) by 2 (an int) you get 0.5 returned, when in C you'd get 0 (an int). if you really want truncation like in C, you use the "//" operator
- in C you'd get integer overflow if you exceeded 2B (32 bits) but with Python, an integer can be orders of magnitude larger than that simply because they define it to include more memory than just 4 bits
- you can use the syntax: *z = x + y, print(f"{z}: .2f")* where the second f hints that it's a *floating point value* to the precision of two decimal points.
- Pythonic code doesn't make as much use of brackets () like after *if* statements, because they don't add much meaningful value. Also, comments are done by using #
- we can now use *or and* instead of ll and &&

- Python doesn't have the concept of a character, only strings. and if you want to compare two strings in Python, you don't have to create an elaborate functions like strcmp() like in C (because in C you needed to compare the value *at* certain memory locations so that you weren't just comparing duplicate copies made), comparison is just primitive to the language.
- to define a function in Python: *def function():*
  what the function does

no curly braces needed.

- the interpreter reads top to bottom, left to right, and so remember that it takes your code very literally in the order that you wrote it. and so if you define a function at the bottom of the program, you can't actually use it in the program because at that point you haven't defined it yet. and so, what often programmers do is actually define their program as the main() function, and then call it at the bottom of their code through this (cryptic code, just memorise):

```
if __name__ == "__main__":
    main()
```

- it matters where the functions are *called* as Python has *run* time errors, not *define* time errors, and so you can define functions as calling other functions before you introduce those other functions, as long as you don't execute anything before Python has had the chance to interpret all the definitions.
- Python is a "loosely-typed" language in that variables don't need to be defined with types, and so x could start as an int, and then be used as a string (*don't do this, though*)
- *break and return* both break out of loops they're embedded in
- function scope is slightly different in Python, in that, when inside a function, you can use a variable anywhere after you're declared it—even an indentation step "out" when you couldn't in C because it had strictly local scope
- you can disable the implicit nextline after pints by using another argument, *end=""*. this also shows how python makes use of named arguments; that is, you can pass in arguments in any order as long as you name them and then connect them to their values. in this case, you're simply replacing the default end="/n" with nothing so that you don't automatically jump to a new line.
- a string in python is primitive, it's powerful. it's like a primitive *structure* (an object), like the ones we'd define in C, containing the array of characters.
- strings come in with built-in functionality that allow you to operate on a given string/character using functions that python already knows by virtue of having defined and understood the concept of "strings", and so to capitalise something you just need to do string.upper(), and you can implement it in such a streamlined fashion:

```
    print(get_string("Name: ").upper())
```

- length of a string is simply called by the function *len*, which doesn't include the null character at the end of memory because Python abstracts that away for clarity and a better UX.
- We use Python 3, not 2
- we can import CLAs *from sys import argc*, and "python" at the beginning is ignored from this
- you can swap two things by just doing x, y = y, x (data type tuple being reallocated from one to another)
- python has lists, not arrays, and its "lists" are like resizable arrays; that is, they implemented linked lists under the hood, and we can use list[5] notation in the same way as we treated arrays in C even though these are linked lists, and we define them by doing list = []
- we have the .append() function with primitive functionality, because their functionality is embedded inside the python definition of a list—OOP
- you can check if something is in a list by literally doing *if a in b*, and it'll implement the for() loop that iterates through the array via linear/binary search underneath the hood—it reads very English-like.
- we can create the object *student*, just as we created the struct student in C, by doing:

```
    student = {"name": name, "dorm": dorm}
```

which now simply stores key-value pairs for any student where the keys are name and dorm, and have their corresponding values. you can also think of this as a hash table, where the array of pointers is "name" and "dorm" etc. pointing to the strings that correspond to each

- if you wanted to write a program that took in 3 students' names and dorms and printed the list:

```
from cs50 import get_string

students = []
for i in range(3):
    name = get_string("Name: ")
    dorm = get_string("Dorm: ")
```

student = {“name”: name, “dorm”: dorm}
students.append(student)

print() #just to give us a line space before we take in names and output them
for student in students:
    print(f“{student[“name”]} is in {student[“dorm”}”)

- in C, doing int(character) would return its ASCII equivalent, but in python, the int(character) is like the atoi(character) in C, in that it just makes sure we’re treating character as an integer, and not a string
- to resize you just call the Image library, and the Image.open() and Image.resize() functions are stored in that library
- the takeaway though is that even if you can write a python version of speller() in 5 minutes, the code takes twice as long to execute—it’s *slower*

Shorts 6

- we can use the for loop in the following way without issue: *for i in range(0, 100, 2)*

which will start at 0 and end at 99, but count in increments of 2 (note this argument being taken at the end of the bracket of the “range”)

- you can use a for loop to populate a Python list
- instead of noms = [] you can also say nums = list(), and you can do nums.insert() in the same way as .append()
- nums[len(nums):] = 5, 6, 7 will add the mini-list [5, 6, 7] to the end of the noms list (logically, this is because you’re inserting those elements in to the position that nums ends on)
- tuples are ordered and immutable
- you can manipulate tuples to output their contents—say you had a list of them—by doing

for arg1, arg2 in presidents #the name of the list
    print(“in {1}, {0} took office”.format(arg1, arg2))

OR you could just do this if you didn’t want arg2 to be printed before arg1:

for arg1, arg2 in presidents #the name of the list
    print(“in {}, {} took office”.format(arg1, arg2))

- we can also create *dictionaries* in Python, another type of storing data {} (aside from lists—sq brackets—and tuples—round brackets). a dictionary storing data on the pizza choices that many people have may look like this:

pizzas {“cheese”: 8, “pepperoni”: 10, “hawaii”: 3}
and then we can manipulate these by doing *pizzas[“cheese”] = 7*, and so forth or even create *pizzas[“bacon”] = 14,*

and these are internally implemented in memory using a hash table under the hood, and so can be thought of in that way, too.

- the *for* loop is very flexible in that it allows us to iterate through dictionaries, too, even though we’re stopped describing the indexes (keys) numerically. we can do this by: *for i in pizzas:* where *i* will represent cheese, then pepperoni, and so forth until it cycles through all of them.
- if you want to iterate over the values in a dictionary, you can do so by using the primitive .items() method like so: *for pie, price in pizzas.items(), print price* which would print 8, 10, 3, and would do so by internally converting the dictionary into a list which it rattles through
- exponentiation operator is x**n
- we define new types of objects (python is OOP, remember) using the *class* keyword
- in python you can’t just declare int x; you have to assign it some value *x = NULL;* and so similarly when you create an object (using a function called a *constructor)*, you must initialise a lot of the properties of the object while declaring the object.
- when you’re defining the methods of an object, the first parameter/argument is always *self* so that you can make sure python knows that the method is related to the object

- the constructor function is the first method you create in an object, and it's conventionally called __init__ for initialise. and then when you define variables, make sure you remember to do self.name or self.age = X. creating a student object might look something like:

```
class Student():
    def __init__(self, name, id):
        self.name = name
        self.id = id
    def changeID(self, id)
        self.id = id
    def print(self)
        print("{} - {}".format(self.name, self.id))
```

- the libraries/head files we call at the beginning/import are not called *modules*, and to run python files you just need to save your files as .py and write in the terminal python filename.py to get it to be interpreted. you can even write code to make the execution process of the python file a lot more like C, where you just ./fileName, but that requires addition of short but cryptic segment of code to tell the terminal how differently to process the file. Best to just pass it for interpretation.

Problem Set 6

- when you import some library (and only the library, not one specific function from that library) and are using its functions, you have to make sure to do cs50.isupper() etc. since you're operating in OOP in Python
- sys.exit() is the super return function, if you will, just cuts the program, and takes the argument that you want to return (like, return 1)
- the ord() function takes in a character and returns its unicode value (like in C how int('a') would represent 97, this returns the same value)
- the chr() function is the opposite — it takes in a numerical input and returns a character in unicode that corresponds to that input
- the .append() function is specifically for lists (arrays) and strings are not treated as such in Python, instead, you use concatenation functionality to add to an existing string
- in python, you use the open() function to open files and modify them, and it takes the file name, opening mode, and buffering status (0 always) as parameters
- a set is an *unordered* list with *no duplicates*, where you define an empty set by doing x = set(), and you can use a method "add()" that takes a parameter of an element that you want to add to the set
- LEARNING POINT—i tend to think in the same way I did for C when I write in python; that is, too low level and in terms of the weeds of implementation details. for example, i don't need to think in terms of defining array sizes in memories, and can instead just define an empty list at the top, and use the "add" (for set) or "append" (for lists) methods. also, to print stars in place of letters for a word, i can literally use this simple code (note how i literally multiply the character the amount of times i want to write it—how intuitive):

```
if (word in bannedSet):
    postCensorWordList.append("*" * len(word) + " ")
```

Week 7—Web Programming

Lecture 7

- Now we look at how we can use Python for web programming—we're now looking to use Python to *generate HTML*
- Today, we meet a framework to get around the process of re-typing the same lines of code in Python that are normally used to set up or get started with developing a web app (the parts in Python, at least)
- Bootstrap is a framework for CSS, Flask is a framework for Python that allows us to optimise for actually getting our projects done instead of re-implementing basic functionalities that have been written a million times over in the past by other people.
- So far in CS50, we've been working with *controller code,* logic and computation (back-end) in one or a few files, and now, we're also going to have *view code*, which controls aesthetic and design (in Python, still!), and for clarity, we separate concerns—we have "C" files (python files for logic and computation) and "V"

files (python files for aesthetics/UI), and next week we meet *model code*, or the “M” in “MVC” which represents use of SQL to manipulate databases and storage of input data.

- so far, in HTML, we know how to write a *file* that will just say “hello, world”, but now we’re learning to make a web app; that is, a *program* that will *generate* a web page saying “hello, world”
- To write a measly python program that generates that hello world html code, you need to write so much “formality” stuff where you just set up the python server and use self.wfile.write() 10 times to actually encode the HTML inside the python program that will write that code. this is repetitive, tedious, and not intellectually challenging. thus, what *Flask* does is import most of this “template” code out for us to get us started so we can start writing the exciting bits, which is why it’s called a “framework”
- something being dynamically added into a page just means that you’re not hard coding it in or writing out the code corresponding to that feature specifically/manually and it’s somehow responding to some user input to change accordingly (spontaneously = dynamically)
- if you didn’t use flask and just wrote a Python program to generate a “hello, world” HTML page, it would look something like:

```
from httpserver import baseHTTPeventhandler
    class HTTPServer_requestHandler
        def do_GET(self)
            self.sendresponse(200)
        self.sendheader(“Content-type”, “text/html”)
        self.wfile.write(b“<!DOCTYPE html>”)
        self.wfile.write(b“<html lang = ‘en’>”)
        self.wfile.write(b“<head><title> Hello, Title! </title></head>”)
```

and so forth… you can see how this involves a lot of esoteric, repetitive code. What flask does for us by acting as a “micro-framework” is abstract all this way. It handles all the nitty-gritty details of implemented python as a web language, like importing servers, choosing port numbers and sending GET and POST requests, as well as dynamically creating self.wfile.write files based on logic—making our lives easier.

- to implement flask, you’d write:

```
from flask import Flask, render_template, request #notice importing request from flask automatically
```

imports all of the necessary “httpserver” and “event handler” files/protocols without you having to be an expert in networking to understand them (abstraction 1)

```
app = Flask(__name__)

@app.route(“/“)
    def index():
        return“Hello, World”
```

the last few lines simply define one route through the web “app” (it isn’t dynamic just yet, so is strictly a site) by saying if the URL is “/“ then you proceed using this anonymous function, which, right now, just returns the string. a more sophisticated approach would separate controller code (python) from view code (HTML) by instead creating an HTML file called index.html that would be returned via *return render_template(“index.html”)* on the final line (abstraction 2)

- and within index.html, itself, we can use a further ability to abstract things supported by Flask by employing a language called Jinja (abstraction 3), which allows you to put variables within HTML that will dynamically return new HTML after inserting a certain layout file you store.
- within the python, you can get variables you define in jinja on other pages by defining them as such:

```
name = request.args.get(“name”) #notice this calls the request function we defined earlier, and args
```

simply states that we’re getting from a page that has method “GET” not “POST” and now you can manipulate the name variable to dynamically change what appears on index.html that employs {} Jinja BUT you have to somehow “connect” the *name* variable we just defined in python and the {{ name }} we called in the Jinja of index.html file. we do this by passing in another argument to the render_template(index.html) page, which is name (in the html file) = name (as we’ve defined it here in the python file). and to feed in the name we defined in the python file, we’d type name into the URL via a get request in the same way you type into q= in the google search URL. now you can see that even though we haven’t hard coded any one particular name, the web page can generate HTML to do “hello, Veronica” or “hello, Brian” based on the user input. this is an example of dynamic web programming.

- to take in a name in a text box: *<input name = “thistakesinname” placeholder=“name” type=“text”>*

- in a select menu <select><option> “optionName” you’d have to add the “value” attribute to the option tag so the browser knows what the innerHTML is and you can manipulate that logically in the future.

- more in detail about drop down menus: you can add a default value in the normal way, but add the attributes “Disabled” and “selected” to have it grey out and be the default value, respectively. and adding a submit button is as easy as creating another <input type=“submit” value=“whatUserSees”>
- if you want to define a route which happens once you “submit” a form (post to server) then you must add an argument to the @app.route() function for that route saying methods=[“POST”] lest it assume methods=[“GET”] by default
- with HTML alone, you can’t “factor out” or abstract away the template for any page, but with python we can now create and call a “layout.html” file that factors out all the repetitive <!doctype> <head> <body> bullshit, and when you have more complex design CSS features, those are also factored out (overall aesthetic, text/background colour and style and size on a page, etc.)
- layout might have all the parts of the page you want every page to have, but then when it come to the body (where the pages differ—in content), you’d use Jinja (abstraction 4 enabled by Flask) to do {% block body %} and then end it by {% endblock %} which would then allow you to just write in that code for each page (the interesting parts). note that while {{ variableName }} is different to {% chunkOfHTMLcode %}
- you call that layout page by starting your route page with {% extends layout.html %}, and in this case, {% block body %} since that’s what you defined as missing in your layout page
- now you can see that any web app we make from now on will have similar organisational structure in that it’ll have an “application.py” that contains most of the controller code that will do all the routing of the user through different routes of the web app based on user behaviour, layout.html to set the base front-end code, and the rest will correspond to different routes the user takes that the application.py can render_template() of
- you can even add some logic into Jinja, like if you have a page that extends the layout page (thus makes use of Jinja) and wants to iterate over some server-side stored list, you can do something like:

```
<ul>
 {% for s in students %}
        <li> s </li>
 {% endfor %}
<ul> #prints the list of tuples or strings stored in “students” instead of using a database
```

- if you restart the server (kill the program in terminal) then the data we stored in the web server (AWS or whatever) is erased, and you lose the information in the list
- you can send emails in a program after registering a form by importing the smtp library and using library functions you can learn if you read the documentation, like emailing an RA whenever someone signs up for an IM
- alternatively, you can save to CSV files, which are permanently stored in the hard drive so the data isn’t wiped every time you close the Flask program. you literally use open(filename.csv) to create a new file, and open it in append mode (“a”). you might store and append the input data of name, dorm of a student into a CSV (comma separated value, a spreadsheet file, if you will) like this:

```
fileName = open(students.csv, ‘a’)
csv.writer(fileName)
writer.writerow((request.form.get(“name”), request.form.get(“dorm”)))
file.close()
render_template(“fileName.csv”)
```

- whereas in the manipulation of CSV files, you could error check for multiple student entries by iterating through the CSV looking for the same as the current student input, but this task would take a few lines of code, and is made much easier in SQL, a language designed specifically for the implementation and manipulation of databases.

- we can validate data on the server (by checking with python *if not name or not form then render “failure”*) or we can validate it client-side using javascript (script tag linking to a JS file that says *if (! document.querySelector(“input”)) then alert* to see whether they actually input things into the fields or not. trade-offs to both as individuals, so probably use both together, but important to know the user can disable javascript in their browser if they want via browser tools.
- One important coding design decision is separation of concerns, and different people do things differently. React is a popular framework, analogous to Bootstrap and Flask, that will automatically create different files, all in the same folder, that separate HTML, CSS, JS, Python this way.

- jQuery is the name of a previously-really famous JS library (but still powerful, and used by Bootstrap for its JS components/intractability)

- to implement an autocomplete dynamic-esque function, you use JS as such:

```
/* import jQuery in script source before this */
let input = document.querySelector(“‘input”);
input.onkeyup = function( {
        $.get(‘/search?q=‘ + input.value, function(data) {
                document.querySelector(‘ul’).innerHTML = data
        }
}
```

the “data” is what is returned by the server upon the anonymous function called by the onkeyup, and the data is really just a long list of <li> word tags that are then stuck inside the <ul> tag by the second anonymous function we called

- there exist Flask functions like *jsonify()* that take the files you want to read and return a JS object, which is a more efficient storage structure in this case than redundantly adding every time, and so you can remove the need to return “search.html” every time (which would return the <li> {{ words }} code> and can instead just return jsonify(words)
- you could run the whole “autocomplete” program locally without any network traffic/back-end python at all, just JS by downloading the dictionary onto the liven’t computer with code like:

```
<!- - continuing code from an HTML page - ->
        <script src=“dictionary.js”></src>
                let input = document.querySelector(‘input’)
                input.onkeyup() = function() {
                        let html = “”;
                        if (input.value) {
                                for word of WORDS {
                                        if word.startsWith(input.value) {
                                                html += ‘<li>’ + word + ‘</li>’
                                        }
                                }
                        }
                        document.querySelector(‘ul’).innerHTML = html;
                }
```

- so now we have three versions of autocomplete implemented in three different ways, one just created a list and iterated over it, generating a new page every time we clicked submit, one used jQuery (that used AJAX — the $) to talk to the server to get new HTML code, and then we just used a CSV file to eliminate all the <li> being returned and just using *jsonify* to output a JS object, and finally we implemented it all in JavaScript with no Python/Flask involvment at all. Each approach has pros/cons, and the process of making informed, specific and contextual choices is part of what an early stage CTO would do, in addition to writing core code himself.

Week 7 Shorts

Flask

- web frameworks are another layer of abstraction when doing back-end web programming, since they are written in python themselves, and do a lot of the repetitive, tedious work you’d otherwise have to do yourself, like creating a web server manually, or rewriting similar lines of code, etc. this is an abstraction further to the fact that we’re using python, a very high level language that already abstracts a lot away from something like C. Django and Pyramid are other examples of frameworks, along with Flask.
- use of python fundamentally allows dynamism without having human intervention.
- always do *from flask import Flask* (flask is a module, Flask is a function (a class))
- you initialise an application by doing app = flask(__name__) where __name__ represents the name of the application this line of code is written in, and thus you create a flask application from *application.py*, the page that this code is usually in.

- a “decorator” is a route through a web app defined by @app.route(“ULRhere”) in the context of web programming
- to run our flask application:

    export FLASK_APP application.py
    export FLASK_DEBUG = 1 #this just ensures we’re in debug mode
    flask run

which returns a URL that becomes the home page of your application

- a client submits an HTTP request to the server (the code that reads in the form HTTP 1.1 GET/POST index.html) and is replied with the same + a status code (404 not found or 200 OK), and GET/POST are just the two most common methods for clients to interact with servers (these are the two things a browser may want to do with a server)
- instead of having your browser manually send a GET request, you can just modify the URL to pass in data in the same way, and define a route:

```
@app.route("/show/ <number>")
def show(number):
    return "you passed in {}".format(number)
```

which will dynamically show which number the user “posted”

- you can have on URL do two different things depending on whether the client computer communicated a GET or POST request to the web server, and use *if request.method == GET or POST* then execute accordingly
- request, as used above, is a flask function that allows humans to easily communicate with server communication messages, another abstraction feature that makes web development easier for us
- other useful flask functions to import are redirect() and session() where the latter checks whether the user is logged in, and render_template()

AJAX

- AJAX is used to embed dynamism into a *part* of the page without reloading the full page, like when sports sites update the score on a page or when gmail adds a new row dynamically. it is not a language, but a technique for writing javascript within an HTML context
- normally, when you click a button, you just write some logic to create some change in the page to manipulate existing data/code, with AJAX, you actually *send a request to the server to get more data in real-time* when the button is pressed, and then integrate that data into the page without reloading the whole thing
- to do so, we have to use a special, existing JS object called XMLHttpRequest, which allows us to make an asynchronous (not synchronous with the reloading of the page, that is, dynamic/real-time)
- after defining a new one of these objects (like var X = new XMLHttpRequest;) you need to instantly define its “onreadystatechange” behaviour, that is, define an anonymous function that tells it exactly what is going to change on the site or what the behaviour of this new variable with the XMLHttpRequest is going to be. an example of a readyState change is the browser sending a request, and receiving data packets from the server, where the readyState is defined numerically as something from 0 to 4 depending on whether it’s the browser or server sending a request etc. we want to get readyState to 4 (request series complete) and status of HTTP requests to 200
- you create a request via open() and send via send()
- you can write the pure JS version of this request, but it’s easier to just use features of jQuery (a JS library) to implement it easier

- an example of using AJAX to dynamically create a new div that outputs a picture of someone and some information about them without changing the whole page is, with comments:

<! - - you’d have to add some links to jQuery libraries to use the $, and to ajax.js (shown below this) for the below code to function as intended - ->

```
<body>
    <h2> Ajax test </h2>
    <div id="infodiv">
        information about a person that will appear dynamically will refresh this div here
    </div>
    <div>
```

<select name = "student" onchange = "studentInfo(this.value);"> #notice the use of "this" and the semicolon at the end calling the studentInfo() function that is defined in ajax.js as a logical operator within an html document, where "this" will be the value of the option that triggered the change mentioned in "onchange"

<option value = "" disabled placeholder> Select a student </option>
<option value = "Tanishq"> Tanishq </option>
<option value = "James"> James </option>
<option value = "Andy"> Andy </option>

</select>
* *
</body>

AJAX.JS file code:

// this function takes in the name value from the file above
function studentInfo(name):
if name == ""
return;
// create a new ajax object that is the request we send to the server
var ajax = new XMLHttpRequest;

// when the client-server transaction of the new data we want is complete, load the
// new div as we want, and note that ajax has a property called status since it's an
// object
ajax.onreadystatechange = function() {
if (readystate == 4 && ajax.status == 200) {
$('#infodiv').html(ajax.responseText) // replaces infodiv html with output, this is the jQuery method to describe doing so since responseText is output of ajax.send(), and the cryptic symbols at the beginning call the JS code in jQuery to say "replace"
}
};
// sends a GET request to open the file called Tanishq.html or whatever that will contain the snippet of html information I want to put in the div tag dynamically about me—note that you can send this GET request to any server! you can even download information from Yahoo! Finance in real time if you want
ajax.open(GET, name + '.html', true)
// sends the GET request, and waits to receive responseText
ajax.send()

Problem Set 7

Lines

- often times when you're confused about implemented some intricate logical function, there is already a native function that will do what you're looking for, so just scan the documentation looking for that and use it
- reading the python 3 documentation is crucial! there are so many functionalities inbuilt for things I want to do, like instead of setting up a for loop to iterate over two sets to find and store items that are in both, you can just do list(setA & setB) and *done!*
- the "sent_tokenize" function from the nltl.tokenize module implements the separating of input strings into a list of sentences for you. as such, you can see that most things you'd ever want to implement have already been implemented for you. it's when you're building something hyper-specific and fundamentally new that you need to start devising and tweaking these modules—which is key to learn how to do, and is why we study programming in high detail and learn ds/a.
- you can do a & b for sets a, b to return a list of things that are in both (same with other logical operators—this is why python is said to be executable pseudocode)
- the "in range" part of *for i in range()* takes in TWO arguments defining the range of executing the following code on the input in that range. however, the first argument has been coded to be optional, and by default set to 0, so it can be used for simple iteration, too.
- to again reiterate the importance of reading and exploring documentation (this is what actual software engineers do every single day—*and* they recommend different modules and technologies to each

other that they haven't heard of to help solve their friends' problems. In trying to implement *sub-strings* part of compare, I essentially re-wrote the implementation of a library function that already existed, and racked by brains trying to figure it out when the actual online solution uses a few lines by exploiting the str[i:j] functionality that strings can take in Python. This functionality, say for i=0 and j=2, will iterate 0 and 1 (but NOT two itself). I used two nested for loops to iterate over each character in each string in a given textual input to extract every combination of sub-strings within a given string input. The actual solution has one simple loop:

```
def substringList(name, n):
    substrings = []
    for i in range(len(name) - n + 1)
        substrings.append(name[i: i + n])
    return substrings
```

and this function returned the subtrings of length n in an inputted string, and you implement the same functionality for lines/sentences hereafter. again, reading the documentation for the string functionality would've made you aware of this instead of implementing it bottom-up as you would (and have done!) with C.

Week 8—SQL & Databases

Lecture 8

- today we learn about the "model" part of MVC, having done python for C and front-end for V, csv is not the most expressive type of database, since you can't insert/delete or manipulate anything, it's just a list of data—akin to a spreadsheet

- SQL is structured query language. an API is an application programming interface is just akin to a third party website that communicates some data to us

- a cookie is a little file stored in my computer after I log into a site/app that sends the login information every time I click a link on a web/mobile app so you don't have to log in after every new click and it can make the experience more seamless—by storing data in "cookies" do you build in the login functionality. this naive implementation involves storing the login information in a file on a computer where it can re-send that information everytime. obviously, storing your personal information on a file stored on your computers means anyone that logs onto your computer has access to it. so how might we get continual access to the login without actually storing the username and password? perhaps we can store something else to show the server we've actually logged in, without bringing out the user and pass again, much like when you re-enter a club after first time, you don't show them the ID all over again, but instead just the stamp on your hand. similarly, the cookies store an enormous number on your computer acting as the stamp to get back into the website, which is termed the cookie. on first login, you "get-cookie" to create that number and associate it with your specific log-in credentials on the *server* (where not just anyone can see it—but this is why hacking the main server compromises the integrity of everyone's data), and so you just send the cookie with the HTTP request in the future and the server immediately recognises that you're logged in. if you disable cookies, the UX deteriorates significantly because of then having to re-log-in several times.

- cookies are sensitive topics in terms of privacy because they can be used for tracking. if a website (say, facebook) collects cookies that keep on top of your login details and information, and walmart shows can ad on your Facebook feed, they have access to those cookies. if walmart also shows ads on google, and youtube, and other websites, then they have access to when you logged into any of those websites. for many big advertisers, this means they have complete understanding of your search history, thus crossing the privacy line.

- cookies are so powerful not just because they are the hand-stamp at the club for entry, but because they represent the link to your account on the server, which contains all the information about your buying history, shopping card, and a number of other things which you may potentially not want other people to know about

- when you're setting the cookie in the first HTTP interaction between the web server and the client, you use "get-cookie", and set the value using *session = value*, where "session" represents the shopping cart —your particular log-in session associated with your account and personality on the web server that is set equal to the large number.

- to build log-in functionality into a web application, you use the *session* feature on the flask micro-framework. this is a dictionary that stores the amount of each item in your "shopping cart" (the information

associated with each account in the web server's database), as well as a boolean value that remembers whether a user is logged-in or not (changed every time they log in or out).

- to store something, a csv file isn't great because to search and insert/delete, it's all in linear time, and we can do better than having to repeatedly open—iterate—close…this is when a database that people have built comes in.

- a database is pretty much like excel (an example of a *relational* database since it specifies relationships between the data) and just stores data, but for a much larger scale than excel was designed to handle (the industrial scale). SQL is the language you type into terminal to manipulate the database. this data is stored in the hard drive of your computer if it's your data, or the permanent memory of the corporation. this is compared to *data structures* which are stored in RAM, and are important to understand because RAM is where programs are run so understanding how to store data in RAM dictates the efficiency of how a program runs since it's quite small.

- a table is one part of a database, just as how one spreadsheet is part of an excel file.

- There are several versions of the SQL language, that can be thought of as dialects, each optimised for different use cases. We use SQLite in cs50, which is used in iOS and Android, too, but for more rigorous/demanding applications you'd want to use postgress, another dialect of SQL.

- in a SQL table, you have to specify the data type in advance for each type of column—in SQLite the data types include *blob, integer (smallint/integer/bigint each store different amounts of bits, middle use case is most common), numeric (boolean, date, time, etc.) , real (real or double precision, where the latter gives you double control over the decimal point), text (char(n), varchar(n), and text)* where *varchar(n)* gives you an upper bound of characters, but sacrifices random access (speed/time) on accessing the data since it can no longer do random access like it could with a char() column of data.

- data structures and data basas aren't completely unrelated, though, when databases are designed, they often have clever data structures baked in to optimise the computer's ability to access particular values in those data bases.

- to create a database, just input the command *sqlite3* and to create a table, use *CREATE TABLE 'tableName' ('field1' dataType, 'field2' dataType, 'field3' dataType);*

- and you write *.schema* to see the database/table you've created

- to insert, use *INSERT INTO 'tableName' ('field1') VALUES (value1);* # with "" if string

- use *SELECT * FROM 'tableName' WHERE field = value;* to display all contents

- to change a field use *UPDATE 'tableName' SET field = value WHERE id = value;*

- in the terminal window, you can "touch" a file, which just creates it but puts nothing inside of it, as well as creating one using "sqlite3 fileName.db"

- pHpLiteAdmin is just a GUI for our databases that appears whenever we double click a file in cs50 IDE. instead of memorising all the commands, you can just use the options displayed in the GUI

- when designing a database, you need a piece of data (a field) that will identify all pieces of data uniquely, and this is the primary key—best practise is to have one primary key

- with csv we would have to write code to manipulate and traverse the file (in slow, linear time), but with databases the searching/insertion and manipulation functionality is baked into it (at a higher optimisation) so we don't need to write code to manipulate it, we just write out what we want to do using SQL commands

- when taking in a data field like phone number, you have to make a design decision about what data type you're going to use, since many might be appropppriate, and each has its own weaknesses. for a phone number, you might use char(10) for US numbers, and then write python code to throw away all non-numeric characters (like parentheses or dashes) entered by the user before entering it into the databases, and this is how different languages can mesh together their various strengths.

- you mainly use the GUI to create, organise and set up the tables, but then SQL code to manipulate it on a granular level

- you can, by importing the SQL function from the cs50 library, write python code that interfaces directly with the database (essentially translating your python into SQL for database manipulation)

- if you created a database (via terminal SQL or GUI use) that stored, say, students in a class, and wanted to create a python application that outputted those students (and thus, say, build it out into a web application), you'd use the SQL and .execute() functions, where the second one executes SQL code within a python program. the code might look like:

```
db = SQL("SQLite:///students.db") #this uses db to create a "portal"/pointer to access our database and call it "db" from now on in this python application
rows = db.execute("SELECT * FROM students") #this SQL will return a list of rows
for row in rows:
    print(f"{row["name"]} is in the class")
```

- when you have lots of repetition anywhere, it should scream optimisation. for example, in a students database, if you have thousands of students in 4 houses (think hogwarts), then there will be hundreds of “slytherin” written your database. if you then assigned slytherin to 4, you’d save lots of byte by subbing in char(9)’s 9 bytes into a smallint’s 2 bytes (multiplied thousands of times). but then, in response, you’d have to make another table that related the smallints to the names of the houses—a key of sorts. and it would get tedious swapping from one to the other, so you might want to query to show students next to their houses, and you’d do so by using SQL’s *join* functionality:

SELECT * FROM ‘tableName1’ JOIN ‘tableName2’ ON table1.field1 = table2.field2 (where field1 = field2 such as houseID being 1-4 in both tables)

the process of optimisation of the database in the way described above is called “normalisation”

- if something is a primary ID in a particular table and is referenced in another table (like houseID 1-4 is the primary key in the 1 = gryffindor etc. table) then it’s a “foreign key” in the other table since it is a key but not the primary one in that table

- in databases, a common problem is a “race condition” where two people observe the status of a particular variable at the same time, and there is a period of time in which they could acton it (buy more milk, register a username, whatever), and the danger comes in when they both take actions at similar times, and the database isn’t built to understand that only one of those people should be allowed to take an action. for example, extracting $200 from an account that only has $100 by logging in on two separate computers and withdrawing at the same time.

- this problem arises because there’s no flag, no sharing of state, nothing saying when one goes out to buy more milk, or one inspects a username to see if its free, there is no notification system in place to notify others that the first person is inspecting the username/buying more milk, and so they are not allowed to do so. well, in databases, this locking is exactly how protection against race conditions are implemented, which is what you see when you’re booking things online, and they give you a certain amount of time to do it (like airline tickets), because they don’t want you to spend an hour booking only to find that the database has been updated and the seats are no longer available, and so they lock everyone else out while you book the tickets, and they hold those tickets only for you, so only you can modify the database during this time. maintaining database integrity in this way is known as maintaining “adamicity” of the database.

- another threat to databases is an SQL injection, where someone injects written SQL into your server when you take user input. this is an example of why you always distrust the user completely. an example of this is when you’re taking some input, and they type *input; DELETE * FROM ‘tableName’* they finish the input by doing input; and then start to inject their own line of SQL that will delete your table, and thus make you lose all the data on the server, causing your site to crash and potentially millions of users’ experiences to be negatively impacted. make sure, for this reason, not to simply blindly plug in user’s input into your code, and instead scrub/sanitise the user’s input always before using it to manipulate a database. there is a special SQL convention where you change the way the inputted data is embedded into your back-end code to sidestep this.

Shorts 8

- SQL’s primary purpose is to query a database and manipulate the data within it.
- Different dialects include MySQL and SQLite (which we use in cs50)
- most of these installations/versions come with a built in GUI you can use
- when you first create a table (best done through the GUI), you have to not just specify the names of thecolumns, but also the data type you want to store therein
- the *enum* data type stores an enumerated list of which you can only choose certain value (like red, green or blue if you choose to use a RGB definitions in your list)
- all of the 20+ data types have an “affinity”, or a type they are associated with—like smallint, bigint, all come under the umbrella of “char”.
- to insert: INSERT INTO tableName (column1, column2) VALUES (value1, value2)
- to select (return info): SELECT (column1, column2) FROM tableName *WHERE* (condition)
- we don’t need to store every relevant piece of information about an object in a single table—you can have multiple for different properties of it, and then use *join* to connect the two and extract information as if were one table (but not have to deal with the cumbersome nature of a table with 20 columns)
- the relationships between these different tables (like both share a specific field) characterise a *relational* database
- to use JOIN: SELECT table1.column1 table2.column4 FROM table1 JOIN table2 WHERE table1.column2 = table2.column2
- to update: UPDATE tableName SET column = value WHERE *predicate*

- DELETE FROM table1 WHERE *predicate*

Problem Set 8

pk_d011f4318e1f48e3b934d0bc7391c820

- you have to register with the host site to use their API—application programming interface (a connection to their server that passes us data that we need to make our site functional in some way), and then paste their API key into your IDE to make the connection between the two
- you manipulate a dict (unordered) by accessing a particular ["key"] to output that corresponding value. In a list, you also use [] notation to get at the various elements.
- when debugging, professional software engineers use an incisive, asymptotic approach. They carefully read the errors to see where in *their* application is going wrong, then experiment with printing values of different things to see where the root of the error could be. a useful tool for this is to print out various different variables to see what type they are by using the type() function.
- The key that helped me finish was sticking with it over time. The mistakes were fucking retarded—JS being above code instead of below, misusing JS syntax with if-else statements, and things like that… debugging by constantly checking prints() and systematically making changes got me to the end.

Week 10—Review, Moving Forwards

Lecture 9

- xcode is the standard IDE to use on the mac environment, even for non-app making things
- atom and VS code, too
- vim is a fast command line tool

- Git is a command-line language you use with GitHub, a tool that allows for versioning a code-repo and sharing with others so everyone can collaborate on it at the same time
- to host a personal site, use pages.github.com or netlify.com
- to host a web app, try heroku.com/platform, and for more heavyweight applications use what they actually use in industry—AWS or Microsoft Azure