More Recursion, CS Concepts, Symmetric Encryption

- in this lecture we will cover
  - Symmetric Encryption
  - Cases for recursion
  - Basic CS concepts and algorithms

Your future in CS

I used to include this on my slides, but since these slides have changed - going to just leave it up here for every notebook. I get a lot of questions about more programming courses, the concentrations, and minors in computer science. Here is a brief reminder.

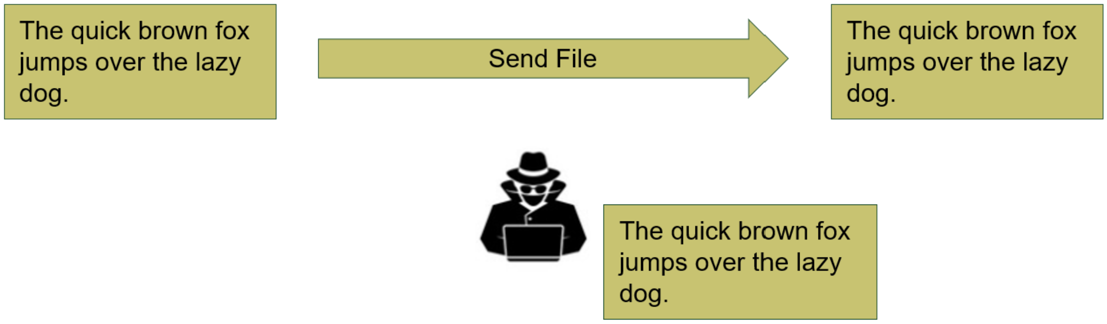
CS 164 – Next Course In Sequence, also consider CS 220 (math and stats especially)

- CO Jobs Report 2021 – 77% of *all* new jobs in Colorado require programming
- 60% of all STEM jobs requires *advanced* (200-300 level)
- 31% of all Bachelor of Arts degree titled jobs also required coding skills
- 2016 Report found on average jobs that require coding skills paid $22,000 more

- Concentrations in CS:

  - Computer science has a number of concentrations.
    - General concentration is the most flexible, and even allows students to double major or minor pretty easily.
    - Software Engineering
    - Computing Systems
    - Human Centered Computing
    - Networks and Security
    - Artificial Intelligence
    - Computer Science Education.
  - Minors:
    - Minor in Computer Science - choose your own adventure minor
    - Minor in Machine Learning - popular with stats/math, and engineering
    - Minor in Bioinformatics - Biology + Computer Science

Reminder from last lecture



- Thinking about what we know about the internet
- Computers send packets through other computers
- What prevents a computer from slightly modifying a packet?

What are packets?

Header	• Destination IP • Originating IP • Protocol (email/www/ftp/video/etc) • ...
Payload	Payload these are the file contents
Footer	Info needed for file reassembly

- Given the contents of a packet, and how the internet works, which parts can you encrypt?
- How can we do this so it seamless to the client?

Encryption Types

- Symmetric Encryption
  - We use this almost daily
  - HTTPS - or TLS
- Public Key Encryption (asymmetric)
  - We should use this more
  - Will cover this in a future lecture
    - We need to cover more CS concepts first

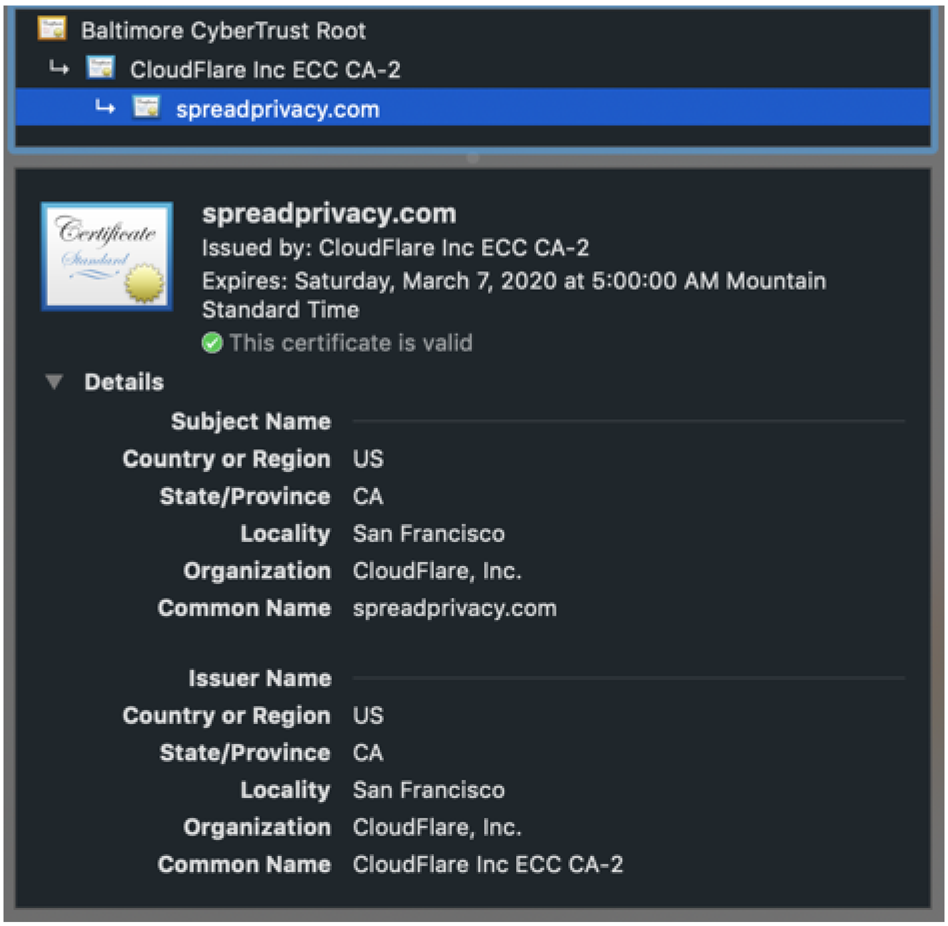
Transport Layer Security

- A way to encrypt the **payload** of packet
- Protocol
  - Used on top of the web protocols (www, IP, etc)
- Stages:
  - Handshake
    - Double check certificate

        - Figure out encryption key
    - Connection is now encrypted for a limited time
- We see it as: HTTPS
- Also used in Virtual Private Networks (VPNs)

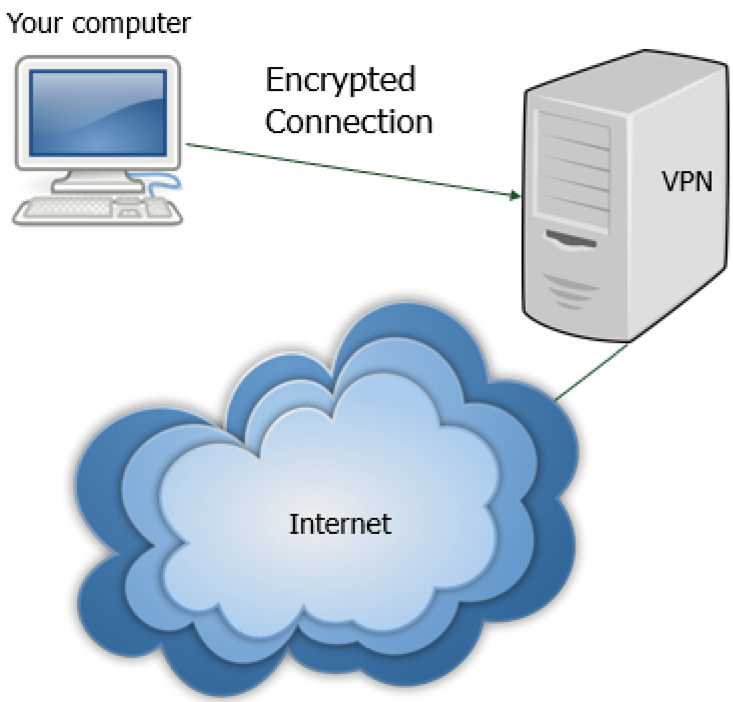
> **Action Item:**
>
> - At your tables, open up a webpage (google or csu for example)
> - Click the Lock next to the URL
> - Look for the 'certificate' to popup (slightly different in various browsers)

This is dependant on 3rd party **trusted** authorities, so that means the 'distributed' nature of the internet is a bit more limited once we need certificates for everything.



Side comment: it is possible to spoof certificates or root them. CS 356 and the CS 456 are great classes to learn how (and ways against). Open to minors and majors in CS.

Virtual Private Networks (VPN)

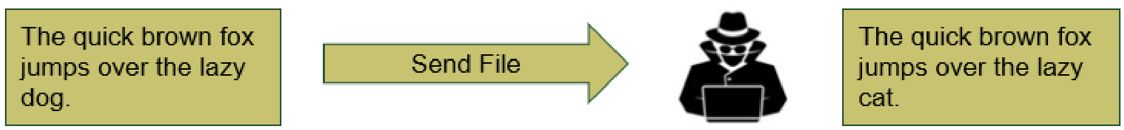


- Ways to encrypt
  - Payload
  - Mask your Origination IP
  - Add additional layer of encryption
- You **first** connect to a VPN server
  - Tunnel all your information through that server
    - Called a tunnel as it creates a mostly direct connection
    - The connection between you and the server is encrypted
    - Often requires two factor authentication, so no one else can 'be you'
  - Server then relays the info the websites you are browsing
    - Ideally through TSL also!
- PROs:
  - Your IP is masked
  - Your data to the server is encrypted, and then your data to the website is encrypted
  - You act like you are from their server
    - often granting access to websites you may not have had access to before
  - Those who manage VPNs are experts at securing their servers
- CONS:
  - VPN becomes a single point of failure for multiple parties
  - You must trust the VPN service
  - Some of them charge for it!
    - Free ones often have issues
- We all have a *free* one that is pretty well known to be secure
  - Colorado State University
  - Allows you to access to subscriptions that CSU gives to everyone
    - scholarly journals

        - access to on-campus resources
        - etc
    - Also, secures your connection, masks your origination, and adds a layer of security!
- CSU uses pulse secure
    - Migrating to global protect (early access now)
- Requires 2 factor authentication
- Easy to setup
    - Recommended you set it up as an application, so all traffic is encrypted, not the single browser session.

To think about over the week

- How do you know the file sent is the **correct** file?
- How do you make it so **only** your friend can open the file?



However, before we really talk about this, we need to explore some more CS concepts.

Repeating Recursion

- Recursion
    - Allows us to use a method to 'repeat' our actions
    - Causes us to focus more on the local aspect of our code
        - Divide-conquer-glue
- Why do we want to use it?
    - Let's look at a few examples

Linear Search

- we have done this in labs
- if we want to **find** an item in a list, we can iterate looking at every element!

In class activity

- Write a function that finds a value in a list (don't use the built in one)
- Return the **index** of the value found!
    - return -1 if it isn't found

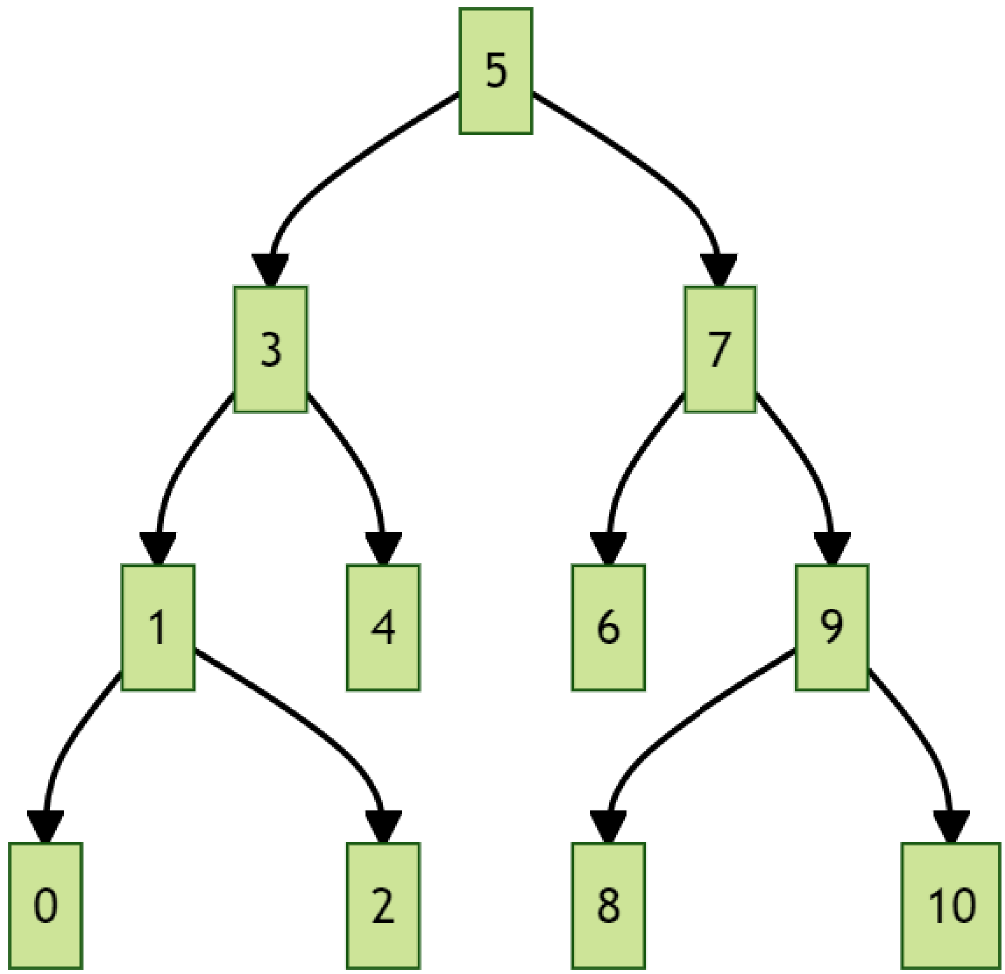
In [ ]:
```
def find_it(lst, value):
    index = 0
    while(index < len(lst)):
        if value == lst[index]: return index
        index += 1
    return -1


lst = ["Kaz", "Inej", "Jasper", "Nina", "Matthias", "Wylan"]
print(find_it(lst, "Kaz"))  # should return 0
print(find_it(lst, "Matthias")) # should return 4
print(find_it(lst, "Yarl"))  # should return -1
```

```
0
4
-1
```

- To find Wylan, it would take 6 iterations of the loop
- No matter how large the list, the worst time is the last element
    - so we say size is N, we can say a linear search takes N times worst case!

Binary Search

- However, what if our data was *sorted*?
- For example:
    - [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- Could we reduce the amount of time it takes to find an element?
- What if we 'redrew' the sorted list as follows?



- Looking at it this way:
  - We can check to see if the value we are looking for is
    - larger, equal to, or less than five
  - Then we can split our data and search in half.
    - Continue to look at the middle value
    - And move down
- This is a binary search
  - Oldest documented search method
  - Goes back to roman times and looking up addresses in an address book
- It is also naturally recursive in nature!

In [ ]:
```
def binary_search(sorted_list, value, start=0, end=None):
  if end is None:
    end = len(sorted_list)
  print(sorted_list[start:end]) ## just so we can see what is going on
  if (end - start) + 1 < 2:
    return -1
  ## get middle index!
  middle = (end+start) // 2  # notice! why?
```

```
  if value == sorted_list[middle]:
    index = middle
  elif value < sorted_list[middle]:
    index = binary_search(sorted_list, value, start=0, end=middle)
  else:
    index = binary_search(sorted_list, value, start=middle+1, end=end)
  return index
```

Activity

- Discuss the above code
- Can you draw out what it is doing
  - Use the code below to help with drawing.

In [ ]:
```
sorted = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  # i could have said sorted = range(1,11)

print("The index is", binary_search(sorted, 2))
print()
print("The index is", binary_search(sorted, 7))
print()
print("The index is", binary_search(sorted, 5))
print()
print("The index is", binary_search(sorted, 12))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5]
[1, 2]
The index is 1

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8]
[6, 7, 8]
The index is 6

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5]
[4, 5]
The index is 4

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[7, 8, 9, 10]
[10]
[]
The index is -1
```

More Recursion?

- You will cover recursion and cases for recursions in future classes
  - 164, 165, 220, 320, etc
  - It has a high representational power, but at a cost
- Good at:
  - building sequences
  - breakdown complex iteration to simple methods

- Divide-Conquer-Glue
  - Essential thinking for recursion

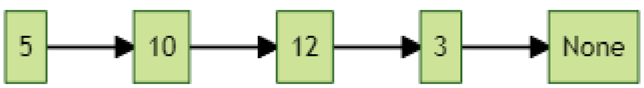
What about other structures?

- We have been talking about lists using indexes
- What if we don't know all the indexes?
- These *chain* style lists are called linked-lists

> Activity
>
> Stand up, form mostly equal groups with each TA. We will go over lists as these groups.

Linked list in code

- assume the following information [value, next element]
- Based on that, I can build a *list of lists*
- There are actual uses of this, lists is just one way to show it



In [ ]:
```
lstfinal = [3, None]
lst3 = [12, lstfinal]
lst2 = [10, lst3]
lst = [5, lst2]


print(lst)

lst3[0] = 13

print(lst)  #notice the middle list is modified
```

```
[5, [10, [12, [3, None]]]]
[5, [10, [13, [3, None]]]]
```

Finding an element?

- Finding an element can be a challenge
- Why because we don't know the 'list' depth
- Each list is stored in a separate spot in memory, and `lst` just references all those memory spots
- Recursion to the rescue

In [ ]:
```
def find_recursive(lst, val):
    print(lst)
    if(lst[0] == val): return lst ## found
    if(lst[1] == None): return None ## not found!
    return find_recursive(lst[1], val) # keep moving down the links
```

In [ ]:
```
print("start of node is:", find_recursive(lst, 10))
print()
print("start of node is:", find_recursive(lst, 12))
```

```
[5, [10, [13, [3, None]]]]
[10, [13, [3, None]]]
start of node is: [10, [13, [3, None]]]

[5, [10, [13, [3, None]]]]
[10, [13, [3, None]]]
[13, [3, None]]
[3, None]
start of node is: None
```

What if we want to add to the end?

In [ ]:
```
def find_end(lst):
    if(lst[1] is None): return lst
    return find_end(lst[1])

end = find_end(lst)
print(end)
```

```
[3, None]
```

Now let's add a value!

In [ ]:
```
def add_to_links(lst, value):
    end = find_end(lst)
    end[1] = [value, None]
```

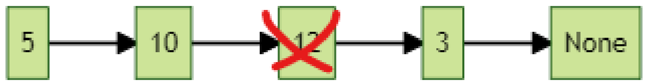
In [ ]:
```
nlst = [1, None]
add_to_links(nlst, 10)
print(nlst)
print()
add_to_links(nlst, 3.4)
print(nlst)
print()
add_to_links(nlst, "another type")
print(nlst)
print()
```

```
[1, [10, None]]

[1, [10, [3.4, None]]]

[1, [10, [3.4, ['another type', None]]]]
```

Thinking Further

- Advantages
    - More memory efficient
- Disadvantage
    - Takes longer to 'jump' to a location
- Also, the 'chain' is distinct!

- What happens when a node is removed incorrectly?
  - The chain breaks!
  - Which means if we compare chains, we can easily find differences



Overview

- The study of these algorithms is part of CS
  - CS 320, 420, etc
  - So much so, we have ways to define 'impossible problems'
    - and then we tackle them using ML or other algorithms.
- Speed matters!
  - Especially if you deal with internet of things devices
- Very open field