CMPT 120

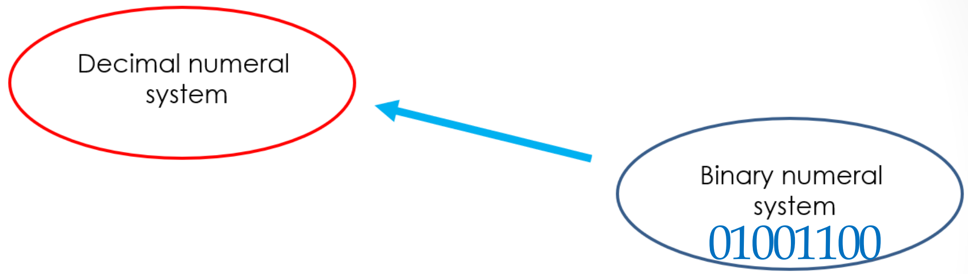
Lecture 32 – Internet and Big Data
Algorithm - Searching

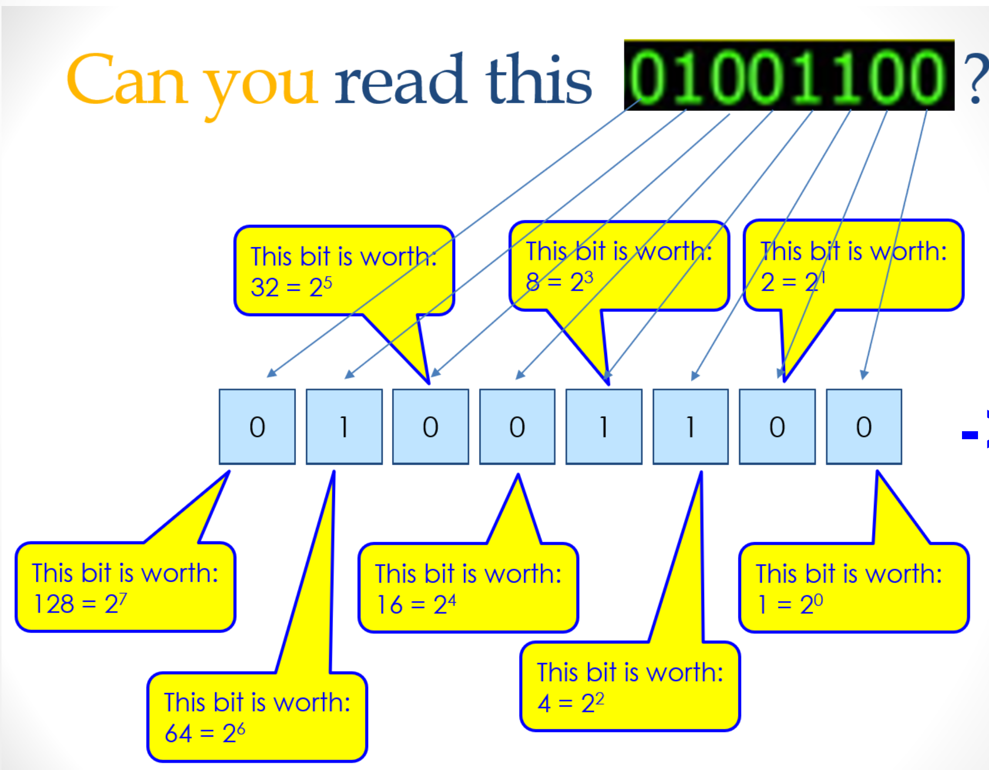
Last Lecture

- We continued looking **under the hood**, continued having a close look at the computer memory!
  - We learnt how to convert binary numbers into decimal numbers and decimal numbers into binary numbers
  - We learnt how to convert binary numbers into ASCII characters and given the ASCII table, we certainty could convert ASCII characters into binary numbers (and decimal numbers)
  - We learnt why **255** meant ***full on green*** *(or **red** or **blue**)*
  - We learnt what **Kb**, **Mb**, **Gb**, etc. (as well as **KB**, **MB**, **GB**, **TB**) stood for

Source: https://www.dreamstime.com/stock-photos-car-open-hood-hand-drawn-sketch-cartoon-illustration-image36401843

Review:





-> $1 \times 2^6 + 1 \times 2^3 + 1 \times 2^2$

$= 64 + 8 + 4$

$= 76$

Review:

Decimal numeral system → Binary numeral system

1 →	00000001
2 →	00000010
3 →	00000011
4 →	00000100
5 →	00000101
6 →	00000110
7 →	00000111
8 →	00001000

9 →	00001001
10 →	00001010
11	00001011
12	00001100
13	00001101
14	00001110
15	00001111
31	00011111
64	01000000
255	11111111
256	100000000

Why does **255** means ***full on green*** or ***red*** or ***blue?***

The value of each colour component of the RGB colour scheme is expressed using 8 bits (1 byte). We call this ***colour depth***. So, the bitmap representing our images has a colour depth of 24 bits (3 bytes): 1 byte to express the intensity of **R**, 1 byte to express the intensity of **G** and 1 byte to express the intensity of **B**. And the largest value we can expressed using 8 bits (1 byte) is 11111111_2, i.e., 255_{10} and the smallest value is 0.

Review:

Decimal numeral system
106

Binary numeral system

Remember:

128	64	32	16	8	4	2	1

106 – **128** = nop! 128 does not fit in 106

106 – **64** = 42

42 – **32** = 10

10 – **16** = nop! 16 does not fit in 10

10 – **8** = 2

2 – **4** = nop! 4 does not fit in 2

2 – **2** = 0

0 –**1**= nop! 1 does not fit in 0

0	1	1	0	1	0	1	0

i.e., 128 does not fit in 106

i.e., 64 does fit in 106

etc……………………………………………

Homework

Seagate Portable 2TB External Hard Drive Portable HDD – USB 3.0 for PC, Mac, PS4, & Xbox - 1-Year Rescue Service (STGX2000400), Black

Visit the Seagate Store

4.6 246,061 ratings

#1 Best Seller in External Hard Drives

$97.99

Binary decoder?

Step 1 - Problem Statement

Write a binary to decimal decode program that converts the binary number the user enters.

Step 2 - Design

Today's Menu

A LOT of data!

The Internet and Big Data

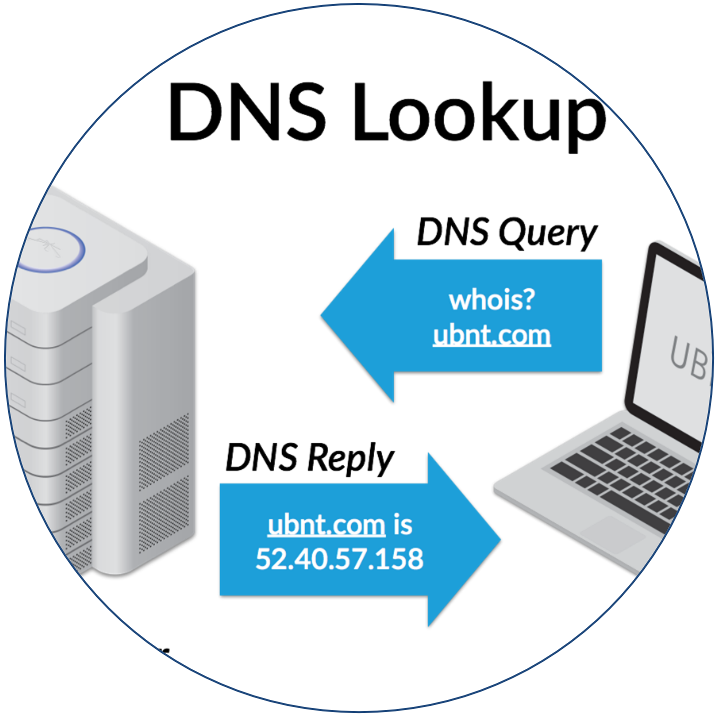
- Internet browsers like **Google** must be able to search through billions of web pages very efficiently
- In this unit, we shall learn various algorithms that allow us, and **Google**, to **search** and **sort** through ***lots data*** and we shall learn to compute how **fast** (time efficient) these algorithms are
- Today, we shall learn a few **searching** algorithms
  - **Linear** search
  - **Binary** search

Where is **search** used?

- Google
- Domain name system (DNS) servers
- Music databases
- Amazon customer databases

… everywhere!

Because there is a large amount of data out there, **searching** needs to be done especially fast!



DNS: Domain Name System
[Example of addresses for the sfu.ca domain]

Searching

First, the basic algorithm: **Linear Search**

Let's try!
Is ____ in this sequence?

Called "target"

42 8 12 34 2 67 33 26 89 54

Our reading calls it **The Sequential Search**!

What did we do to find the target?

Linear Search – Take 1

Step 1 – Problem Statement

Given a list of numbers (e.g. of shoe sizes) and a target number we are looking for in the given list, can we write a function that returns **True** if the target number is in the given list of numbers, otherwise, it returns **False**?

Requirements:

We cannot use a Python conditional statement such as `if x in list:` Can you see why?

Step 2 – Design

Step 3 – Implementation

Let's start coding!

Step 4 - Testing

```
testData1 =
testData2 =
testData3 = []

linearSearchBool(testData1,    ) ->
linearSearchBool(testData1,    ) ->
linearSearchBool(testData1,    ) ->
linearSearchBool(testData2,    ) ->
linearSearchBool(testData2,    ) ->
linearSearchBool(testData2,    ) ->
linearSearchBool(testData3,    ) ->
```

We are expecting:

Observations

Reminder: Generalisation guideline and testing

- When we design algorithms, programs and functions, we want to make sure they solve as many similar problems as possible, i.e., they work with as many different data configurations as possible

  -> **generalisation** guideline
- Therefore, we shall test our algorithms, programs and functions accordingly:
  - For example: Does our linear search algorithm work successfully with …
    - An empty list? A sorted list?
    - An list containing the same element?
    - A list containing the target number once?
    - A list containing the target number several times, etc…

Linear Search – Take 2

Step 1 – Problem Statement

Given a list of numbers (e.g. of shoe sizes) and a target number we are looking for in the given list, can we write a function that returns the **location** of the target number in the given list of numbers, e.g., its index, otherwise it returns **None**?

Requirements:

We cannot use a Python conditional statement such as **`if x in list:`**

Step 2 – Design

Step 3 – Implementation

Let's start coding!

Linear search algorithm

- Advantages
  - Simple to understand, implement and test
- Disadvantages
  - Slow (time inefficient) because it looks at every element
  - Wait a minute! Not always!
    - We saw that for some of our test cases linear search did not look at every element

That is true! We'll come back to this real soon!

Time inefficient linear search?

- **Linear search** could be quick if the target element we are looking for is the first element in our data:

42	8	12	34	2	67	33	26	89	54

- or it could be slow if the target element we are looking for is the last element in our data:

42	8	12	34	2	67	33	26	89	54

- Conclusion:

  Sometimes **linear search** is time efficient and sometimes, it isn't!

Various scenarios

- Best case scenario

```
ourList = [ 0, 6, 9, 2, 5, 3, 7, 1, 4 ]
target:
```

- Average case scenario

```
ourList = [ 0, 6, 9, 2, 5, 3, 7, 1, 4 ]
target:
```

- Worst case scenario

```
ourList = [ 0, 6, 9, 2, 5, 3, 7, 1, 4 ]
target:
```

Data organization vs. Searching

- Would the way we organize our data (e.g., data in order, not in order) affect the time it takes to find a target element in a sequence?
- What if we were to **sort** our data?
- Would this affect the time efficiency of **linear search**?

Various scenarios

- Best case scenario

```
ourList = [ 0, 1, 2, 3, 4, 5, 6, 7, 9 ]
target:
```

- Average case scenario

```
ourList = [ 0, 1, 2, 3, 4, 5, 6, 7, 9 ]
target:
```

- Worst case scenario

```
ourList = [ 0, 1, 2, 3, 4, 5, 6, 7, 9 ]
target:
```

Consider …

- Consider our vinyl collection. How could we organize our LPs to make searching through them faster?

- Although it will take some time to do the initial sorting (we will see in our next lectures), a sorted sequence of elements will make looking for an target element much quicker later, every single time.

Source: https://victrola.com/blogs/articles/8-classic-vinyl-records-you-should-own

How Binary Search works!

Searching for target element **17** and return **True** if found in this sequence:

-2	-1	8	14	17	23	29	37	74	75	81	87	95

Iteration #1:

1. Find middle element of sequence:

-2	-1	8	14	17	23	29	37	74	75	81	87	95

2. Compare middle element **29** with target element **17**
3. Because **17** < **29**, we can ignore the 2^{nd} half of array:

-2	-1	8	14	17	23	~~29~~	~~37~~	~~74~~	~~75~~	~~81~~	~~87~~	~~95~~

How Binary Search works!

Iteration #2:

1. Find middle element of the remaining sequence:

-2	-1	8	14	17	23	29	37	74	75	81	87	95

2. Compare middle element **8** with target element **17**
3. Because **17** > **8**, we can ignore half of array:

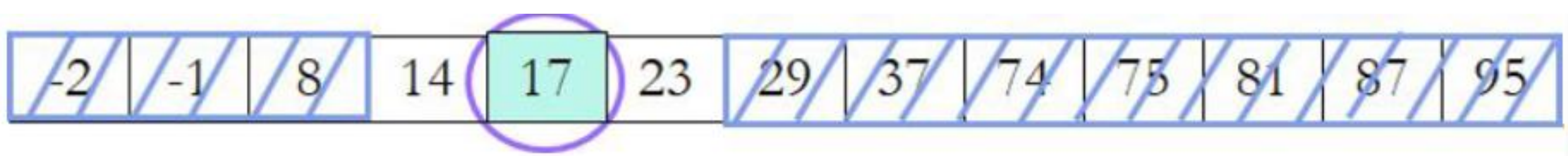
-2	-1	8	14	17	23	29	37	74	75	81	87	95

How Binary Search works!

Iteration #3:

1. Find middle element of the remaining sequence:

2. Compare middle element **17** with target element **17**
3. Because **17** == **17**, return **True**!



Next Lectures

- We shall finish exploring **Binary** search
- We shall investigate which of the two search algorithms: **Linear** or **Binary** search, is the most time efficient, i.e., fastest?

- We shall look into **sorting**:
  - **Insertion sort**
  - **Merge sort**
- And figure out which one is the most time efficient, i.e., fastest!