COMPUTATIONAL EFFICIENCY

LING 1340/2340: Data Science for Linguists

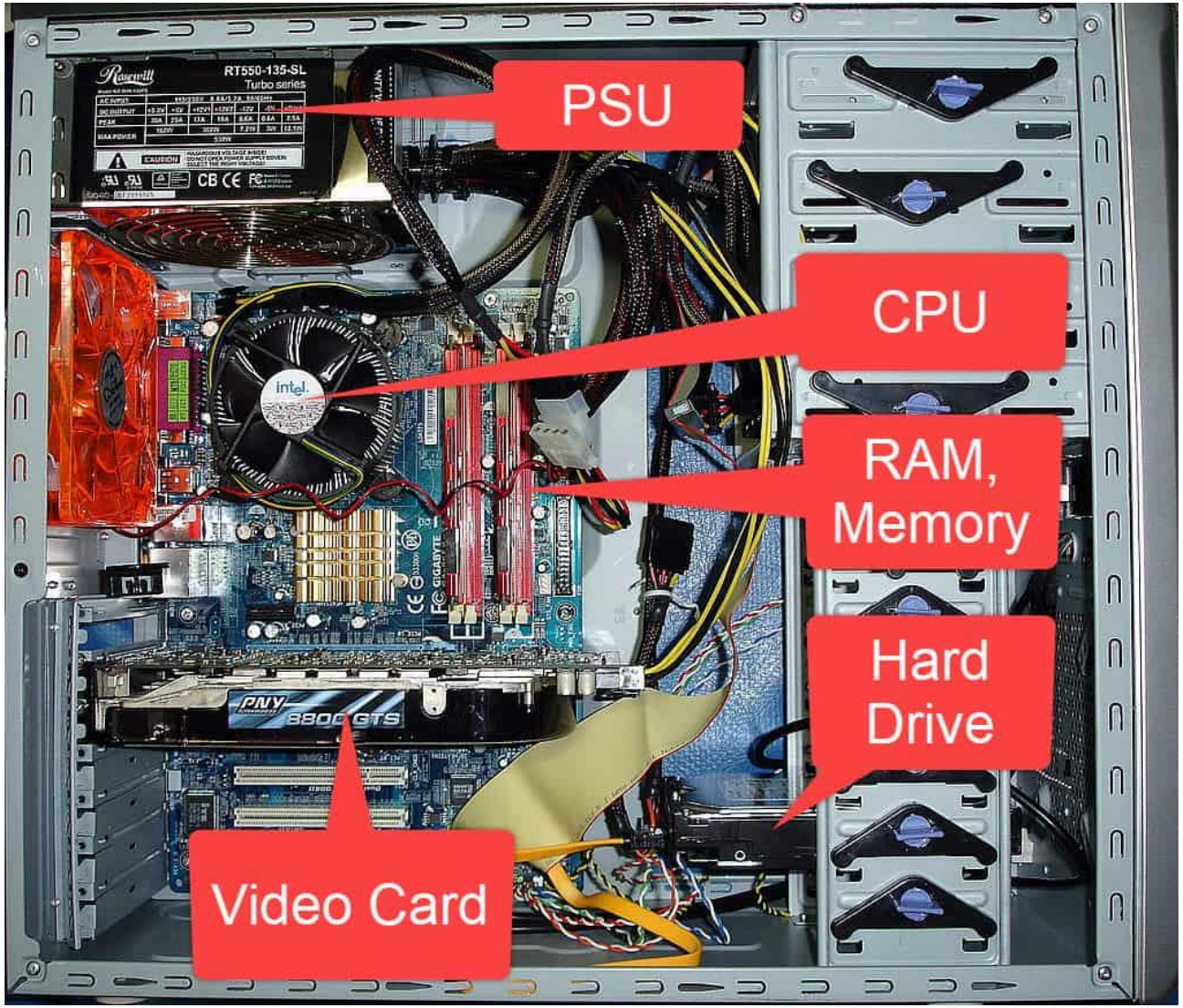
Joey Livorno

Objectives

- Computer Anatomy
  - Memory
  - Processing
- Computational Efficiency
  - Algorithmic complexity
  - Big O notation
- Solutions
  - Chunking data
  - Choosing the right data type

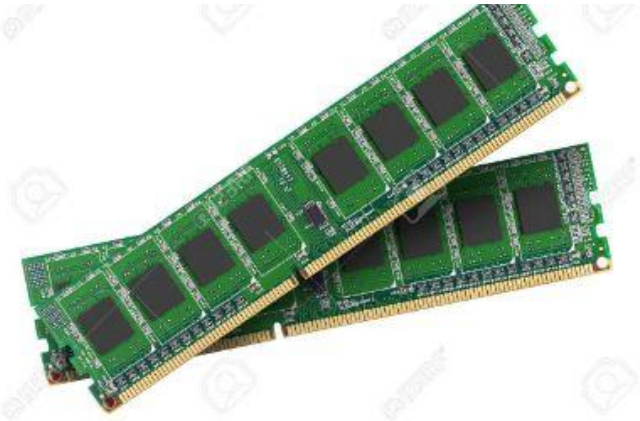
How Do Computers Work?

- Main Components
  - CPU
  - Video Card
  - RAM (Random Access Memory)
  - Hard Drive
  - Motherboard
  - Power Supply



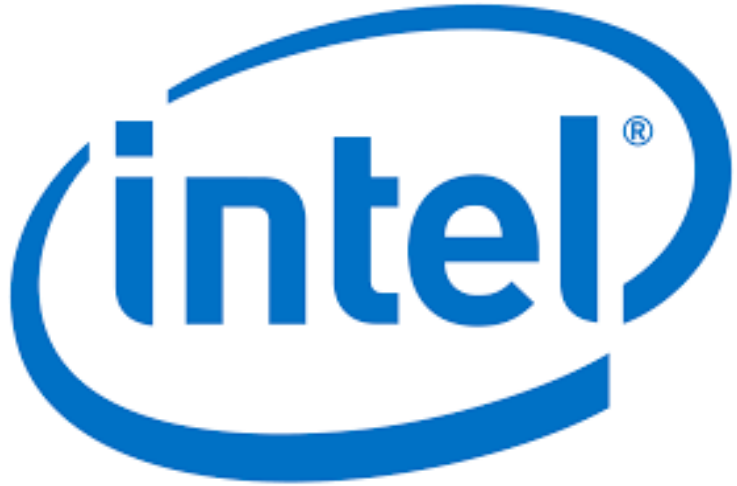
Random Access Memory (RAM)

- ”Random Access”
  - Can be read or changed in any order
  - Makes this type of storage insanely fast
  - Used for our “working” data
- Other storage mediums
  - HDD, SSD, Magnetic Tape, etc.
  - Not as fast
  - Used for large, permanent storage



Central Processing Unit (CPU)

- Circuit that executes instructions of programs
  - Arithmetic
  - Logic
  - Controlling
  - I/O
- Contain Cores
  - Individual Processors within the CPU
  - Each core can compute independently
  - Allows the CPU to process multiple tasks in parallel



Computational Efficiency: Space vs. Time

Space Complexity

- How much memory are we using?
- Do not create duplicate data objects
- Avoid creating data objects that do not need to be stored in their entirety
- Garbage Collection
  - Variable name acts as pointer
  - Data that does not have a pointer will be removed from memory automatically
  - If space is a concern, don't rely on this

Time Complexity

- How much processing are we doing?
- Avoid repeating steps that require heavy processing
- Use efficient algorithms
- Use data types optimal for a specific task

The goal is to optimize both. They often exist as a trade-off relationship, but by managing our resources we can achieve balance.

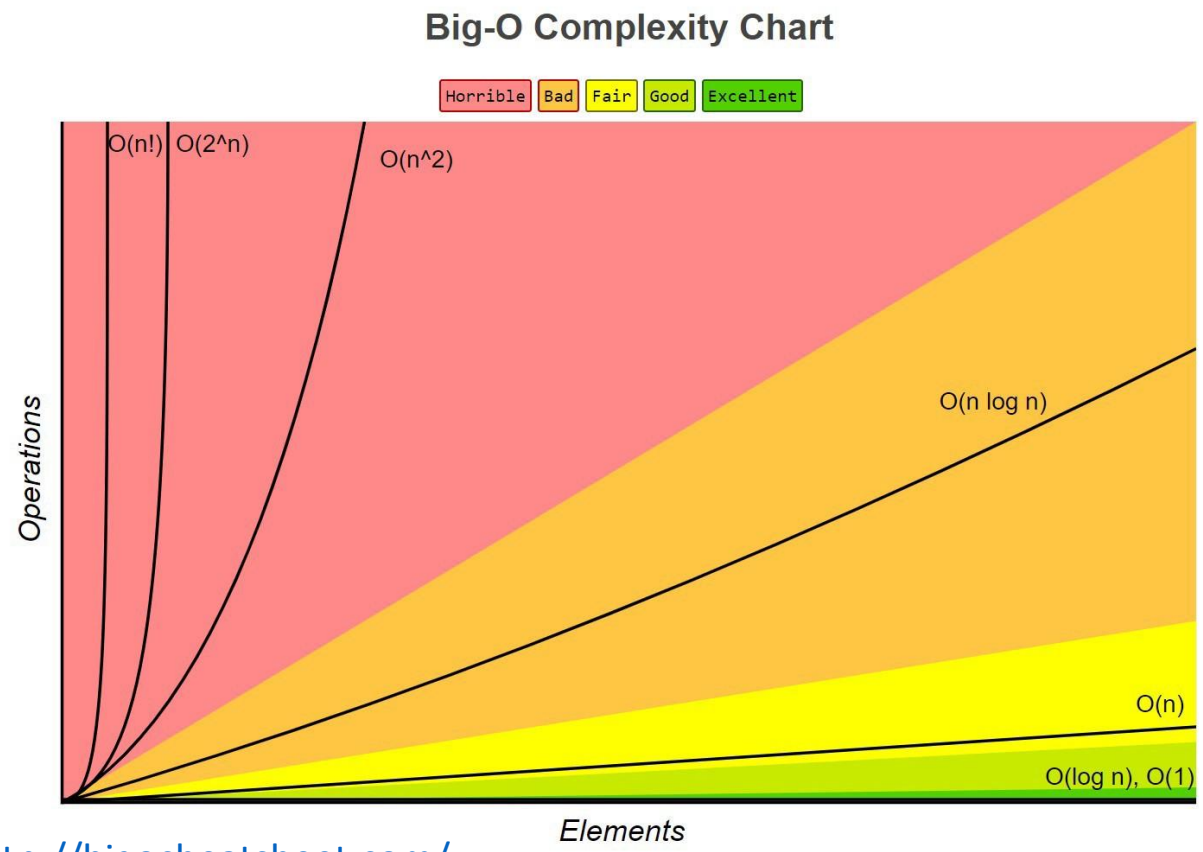
Algorithmic Complexity and the Big O

- https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
- Can be used to describe the worst-case performance of an algorithm
- We have an **unsorted list of *n* items**. Imagine *n* to be any value.

1. Is the first element in the list an even number?
   - Can be implemented in O(1), or constant time
   - Does not change as the size of n increases
2. Does the list contain 42?
   - Can be implemented in O(n), or linear time
   - Directly proportional to the size of n
3. Does the list contain duplicate values?
   - Can be implemented in $O(n^2)$, or quadratic time
   - Directly proportional to the square of the size of n
   - Can be optimized, but at the expense of space complexity

Algorithmic Complexity and the Big O

4. Sort the list
   - Can be implemented in O(n log n), or loglinear time
   - See: https://brilliant.org/wiki/sorting-algorithms/



http://bigocheatsheet.com/

Opening and Closing Files

```
f = open('review.json')
lines = f.readlines()
for l in lines:
    if 'horrible' in l:
        print(l)
f.close()
```

```
f = open('review.json')
for l in f:
    if 'horrible' in l:
        print(l)
f.close()
```

```
lines = open('review.json').readlines()
for l in lines :
    if 'horrible' in l:
        print(l)
```

```
with open('review.json') as f:
    for l in f:
        if 'horrible' in l:
            print(l)
```

Which methods are memory-efficient?

Answer: the ones on the right are more memory-efficient.

Left two boxes create `lines` object which holds the entirety of the .json file in memory!

Processing Big Files

- How much memory did we use for process_reviews.py?

joeylivorno — gil15@login0:~ — ssh gil15@h2p.crc.pitt.edu — 80×24

```
 GNU nano 2.3.1          File: process_reviews.py                Modified

import pandas as pd
import sys
from collections import Counter

filename = sys.argv[1]

# 5 GBs

df = pd.read_json(filename, lines=True, encoding='utf-8')
print(df.head(5))

# ~4 GBs

wtoks = ' '.join(df['text']).split()
wfreq = Counter(wtoks)
print(wfreq.most_common(20))



^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

This script is NOT memory efficient.

`df` holds the entire json file's content as DF, in memory.

`wtoks` is another big data object created from the entire text.

Processing 4-million Yelp reviews took 6.5 minutes and 33.5 GB of memory.

Handling Files in Chunks

- We can store the data in chunks so that we only have access to what we need at a given point in time
- https://realpython.com/introduction-to-python-generators/

Reading in the files as a generator takes up zero space.

Then we simply iterate through the smaller dfs.

naraehan@login0:~/todo11-12

GNU nano 2.3.1 File: process_reviews_eff.py

```
import pandas as pd
import sys
from collections import Counter

filename = sys.argv[1]

df_chunks = pd.read_json(filename, chunksize=10000, lines=True, encoding='utf-8')

wfreq = Counter()

for chunk in df_chunks:
    for text in chunk['text']:
        wfreq.update(text.split())

print(wfreq.most_common(20))
```

^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pc
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spe

Read_json() argument says that each chunk will be 10k lines.

This is memory efficient! Processing the same 4million Yelp reviews took only **12.35GB** of RAM.

Pandas vs. Large Data Tips

- “Why and How to Use Pandas with Large (but not big) Data”
  - https://towardsdatascience.com/why-and-how-to-use-pandas-with-large-data-9594dda2ea4c
- Use generators to access files in chunks
- Filter out unimportant columns to make your DataFrame smaller
- Use optimal data types
  - Float64 is bigger than float32, but we might only need float32

Data Types and Optimization

1. List Comprehend through alice words list against enable words list.

```
[5]: %time notfound = [w for w in awords if w not in enable]
```

2. Same, but filter against enable words as a *set*.

```
[7]: %time notfound = [w for w in awords if w not in enable_set]
```

3. Compute set difference.

```
[9]: %time notfound = awords_set.difference(enable_set)
```

Data Types and Optimization

1. List Comprehend through alice words list against enable words list.

```
[5]: %time notfound = [w for w in awords if w not in enable]

CPU times: user 39.7 s, sys: 16.2 ms, total: 39.7 s
Wall time: 39.8 s
```

Ew

2. Same, but filter against enable words as a *set*.

```
[7]: %time notfound = [w for w in awords if w not in enable_set]

CPU times: user 20.3 ms, sys: 4 ms, total: 24.3 ms
Wall time: 24.8 ms
```

A bit faster

List objects are not optimized for membership operations, but sets are!

3. Compute set difference.

```
[9]: %time notfound = awords_set.difference(enable_set)

CPU times: user 332 µs, sys: 2 µs, total: 334 µs
Wall time: 339 µs
```

Best option

Algorithmic Efficiency: Summary

- Solutions can be implemented with varying degrees of efficiency
- Certain problems have inherent limits to efficiency
  - Big O represents the upper bound to a problem as it relates to processing time and the input size
  - Example: Sorting algorithms can only be as fast as O(n log n) in the worst case
    - There are some sorting algorithms that are faster, but they only work with special input types

- Take Away
  - Use the most efficient algorithm possible
    - ”efficient” can be in terms of time or space complexity
    - Weigh your options and choose what’s best for your situation
  - Understand the relationship between the growth function and the input size
    - O(n) is rather scalable, but $O(n^2)$ is too inefficient for large inputs
  - More efficient algorithms = better memory usage and faster runtimes
  - Efficiency is not always the be-all and end-all
    - With smaller data sets, maybe readability is more important