Lecture 14: Computational Efficiency, OnDemand on CRC

LING 1340/2340: Data Science for Linguists

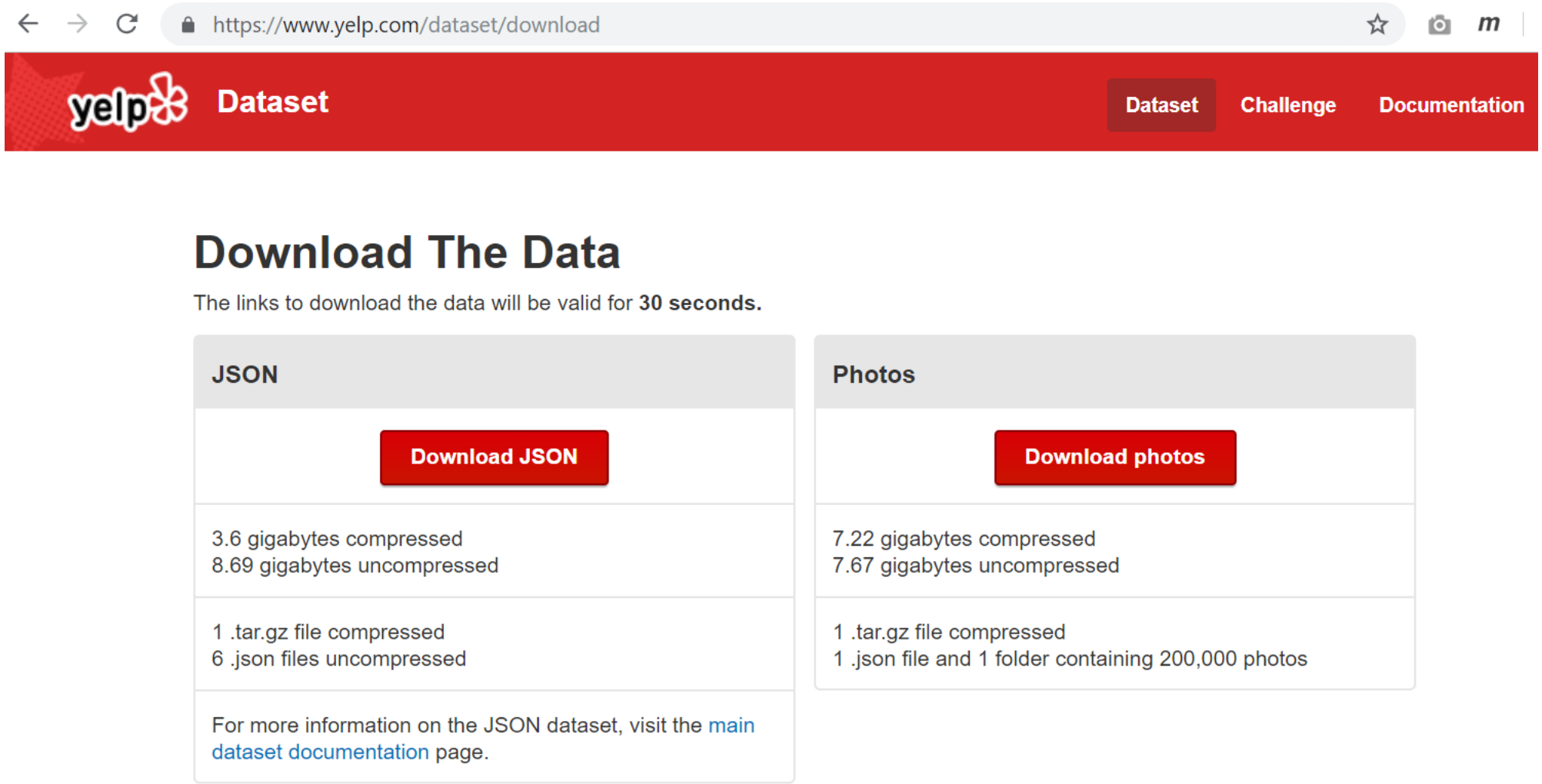
Na-Rae Han

Objectives

- Big data considerations
- Computational efficiency
  - Memory vs. processing time
  - Algorithmic complexity
  - Big O notation
- OnDemand platform & JNB at CRC (GUI!)
  - Clustering, topic modelling

The Yelp Dataset Challenge

- https://www.yelp.com/dataset/challenge



Working with big data files

```
naraehan@login0:/zfs2/ling1340-2019s/shared_data/yelp_dataset_13
[naraehan@login0 yelp_dataset_13]$ ls -lah
total 5.1G
drwxr-xr-x 2 naraehan ling1340-2019s   10 Mar 21 12:03 .
drwxr-xr-x 6 naraehan ling1340-2019s    6 Mar 21 12:05 ..
-rw-r--r-- 1 naraehan ling1340-2019s 132M Mar 21 12:03 business.json
-rw-r--r-- 1 naraehan ling1340-2019s 390M Mar 21 12:03 checkin.json
-rw-r--r-- 1 naraehan ling1340-2019s  99K Mar 21 12:03 Dataset_Challenge_Dataset_Agreement.pdf
-rw-r--r-- 1 naraehan ling1340-2019s  25M Mar 21 12:03 photo.json
-rw-r--r-- 1 naraehan ling1340-2019s 5.0G Mar 21 12:03 review.json
-rw-r--r-- 1 naraehan ling1340-2019s 234M Mar 21 12:03 tip.json
-rw-r--r-- 1 naraehan ling1340-2019s 2.4G Mar 21 12:03 user.json
-rw-r--r-- 1 naraehan ling1340-2019s 110K Mar 21 12:03 Yelp_Dataset_Challenge_Round_13.pdf
[naraehan@login0 yelp_dataset_13]$ wc -l review.json user.json
   6685900 review.json
   1637138 user.json
   8323038 total
```

- Each file is in JSON format, and they are huge:
  - review.json is 5.0GB with 6.7 million records
  - user.json is 2.4GB with 1.6 million records
  - ← Too big to open in most text editors (Notepad++ couldn't.)
  - ← How to explore them?

    In command line. head/tail, grep and regular expression-based searching.

Command line exploration

naraehan@login0:/zfs2/ling1340-2019s/shared_data/yelp_dataset_13

```
[naraehan@login0 yelp_dataset_13]$ wc -l review.json user.json
   6685900 review.json
   1637138 user.json
   8323038 total
[naraehan@login0 yelp_dataset_13]$ head -1 review.json
{"review_id":"Q1sbwvVQXV2734tPgoKj4Q","user_id":"hG7b0MtEbXx5QzbzE6C_VA","business_id":"ujmEBvifdJM6h6RLv4wQIg
","stars":1.0,"useful":6,"funny":1,"cool":0,"text":"Total bill for this horrible service? Over $8Gs. These cro
oks actually had the nerve to charge us $69 for 3 pills. I checked online the pills can be had for 19 cents EA
CH! Avoid Hospital ERs at all costs.","date":"2013-05-07 04:34:36"}
[naraehan@login0 yelp_dataset_13]$ grep -i 'scrumptious' review.json | wc -l
9223
[naraehan@login0 yelp_dataset_13]$ grep -i 'horrible' review.json | wc -l
149059
[naraehan@login0 yelp_dataset_13]$ grep -i 'horrible' review.json | cut -d, -f4 | head -3
"stars":1.0
"stars":1.0
"stars":1.0
[naraehan@login0 yelp_dataset_13]$ grep -i 'horrible' review.json | cut -d, -f4 | sort | uniq -c | sort -nr
 101450 "stars":1.0
  21380 "stars":2.0
  11463 "stars":3.0
   8436 "stars":5.0
   6330 "stars":4.0
[naraehan@login0 yelp_dataset_13]$
```

Opening + processing big files

- How much resource does it take to process review.json file (5.0GB)?

process_reviews.py - C:\Users\narae\Documents\Data_Science\dataset\process_reviews.py (3.5.3)

File Edit Format Run Options Window Help

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

filename = sys.argv[1]

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

print(df.head(5))

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

There's 5 GB

Another 4~ GB

Not as big

This code is NOT memory-efficient.

After exceeding the 8GB default memory allocation, the job gets killed.

Memory consideration

- How much space needed for bigrams? Trigrams?

process_reviews2.py - C:/Users/narae/Documents/Data_Science/dataset/process_reviews2.py (3.5.3)

File Edit Format Run Options Window Help

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

filename = sys.argv[1]

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

print(df.head(5))

wtoks = ' '.join(df['text']).split()
bigrams = nltk.bigrams(wtoks)
trigrams = nltk.trigrams(wtoks)

bifreq = Counter(bigrams)
print(bifreq.most_common(20))

trifreq = Counter(trigrams)
print(trifreq.most_common(20))
```

Good news! These are built as *generator* objects and take up almost zero space.

But these frequency counter objects will take up space.

```
>>> import nltk
>>> sent = 'Colorless green ideas sleep oh so very furiously'
>>> toks = sent.split()
>>> toks
['Colorless', 'green', 'ideas', 'sleep', 'oh', 'so', 'very', 'furiously']
>>> bigrams = nltk.bigrams(toks)
>>> bigrams
<generator object bigrams at 0x00000236371E2BF8>
>>> for b in bigrams:
        print(b)

('Colorless', 'green')
('green', 'ideas')
('ideas', 'sleep')
('sleep', 'oh')
('oh', 'so')
('so', 'very')
('very', 'furiously')
>>> bigrams
<generator object bigrams at 0x00000236371E2BF8>
>>> list(bigrams)
[]
>>> bigrams = nltk.bigrams(toks)
>>> list(bigrams)
[('Colorless', 'green'), ('green', 'ideas'), ('ideas', 'sleep'), ('sleep', 'oh')
, ('oh', 'so'), ('so', 'very'), ('very', 'furiously')]
>>>
```

Generator type objects take up little memory space; meant to be used in a loop-like environment.

Content has been exhausted

Casting as list.
If you store the returned list, it will take up memory space.

File opening & closing methods

```
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)
```

Python will close up this file handle.

Which methods are memory-efficient?

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

No need to close f later. Some folks swear by using `with`.

Handling files in chunks

```
f = open('review.json')
lines1 = f.readlines(1000000000)
lines2 = f.readlines(1000000000)
lines3 = f.readlines(1000000000)
lines4 = f.readlines(1000000000)
lines5 = f.readlines()
f.close()
```

Optional # of bytes to read. (When used like this without a loop, offers no memory advantage.)

```
dfs = pd.read_json('review.json', lines=True, chunksize=10000, encoding='utf8')

wfreq = Counter()

for df in dfs:
    wtoks = ' '.join(df['text']).split()
    temp = Counter(wtoks)
    wfreq.update(temp)

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

Generator object. Takes up zero space.

`chunksize` optional parameter in pandas' `read_json` method reads in 10,000 lines at a time…

then, iterate through each small df.

Memory-efficient! This code uses only **290MB** of memory!!

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

1. Read CSV file data in chunk size
2. Filter out unimportant columns in DF to save memory
3. Change `dtypes` for columns
   - float64 takes up more space than float32.

Vectorizing and training in chunks

```
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import HashingVectorizer
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

filename = 'review_10k.json'
length = 10000
chunk_size = 1000
chunks = length/chunk_size

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

clf = MultinomialNB()
vectorizer = HashingVectorizer(alternate_sign=False)

for i, df in enumerate(df_chunks):
    if i < 0.8 * chunks:
        clf.partial_fit(vectorizer.transform(df['text']), df['stars'], classes=[1,2,3,4,5])
    else:
        pred = clf.predict(vectorizer.transform(df['text']))
        print('batch {}, {} accuracy'.format(i, np.mean(pred == df['stars'])))
```

If vectorizer/ML model depends only on individual row of data, it can be implemented in chunks.

(Caveat: TF-IDF vectorizer and most ML models can't.)

Hashing vectorizer **skips the IDF part** of TF/IDF, can be implemented in chunks!

NB classifier can be trained in partial bits!

```
batch 8, 0.444 accuracy
batch 9, 0.439 accuracy
```

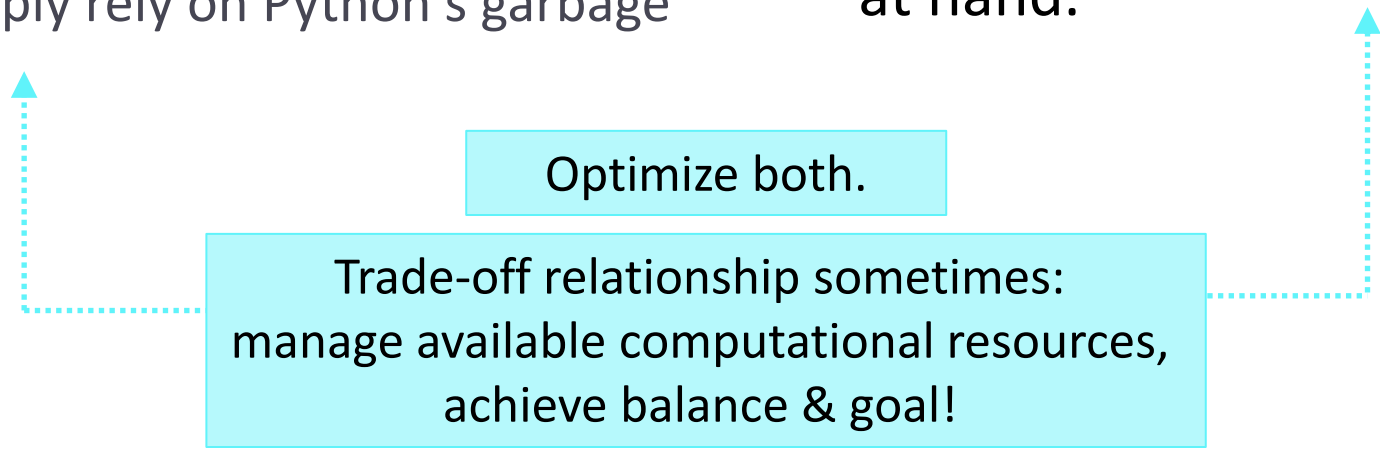
Computational efficiency: space vs. time

SPACE: **memory** footprint

- Do not create duplicate data objects.
- Avoid creating a data object that does not need to be stored in its entirety.
- In code, delete large data objects that will no longer be used
  - Do not simply rely on Python's garbage collection

TIME: **processor** runtime

- Avoid duplicating an expensive processing step: process once, store result as an object, then reuse.
- Use an efficient algorithm.
- Use the data type optimal for the task at hand.



Data types and optimization

[1]:
```
import nltk
nltk.data.path.append('/zfs2/ling1340-2019s/shared_data/nltk_data')
```

[2]:
```
from nltk.corpus import gutenberg
%pprint
gutenberg.fileids()
```

```
Pretty printing has been turned OFF
```

[2]:
```
['austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt', 'blake-poems.txt', 'bryan
t-stories.txt', 'burgess-busterbrown.txt', 'carroll-alice.txt', 'chesterton-ball.txt', 'chesterton-brown.tx
t', 'chesterton-thursday.txt', 'edgeworth-parents.txt', 'melville-moby_dick.txt', 'milton-paradise.txt', 's
hakespeare-caesar.txt', 'shakespeare-hamlet.txt', 'shakespeare-macbeth.txt', 'whitman-leaves.txt']
```

[3]:
```
awords = gutenberg.words('carroll-alice.txt')
print(awords[:100])
print(len(awords))
```

```
['[', 'Alice', "'", 's', 'Adventures', 'in', 'Wonderland', 'by', 'Lewis', 'Carroll', '1865', ']', 'CHAPTE
R', 'I', '.', 'Down', 'the', 'Rabbit', '-', 'Hole', 'Alice', 'was', 'beginning', 'to', 'get', 'very', 'tire
d', 'of', 'sitting', 'by', 'her', 'sister', 'on', 'the', 'bank', ',', 'and', 'of', 'having', 'nothing', 't
o', 'do', ':', 'once', 'or', 'twice', 'she', 'had', 'peeped', 'into', 'the', 'book', 'her', 'sister', 'wa
s', 'reading', ',', 'but', 'it', 'had', 'no', 'pictures', 'or', 'conversations', 'in', 'it', ',', "'", 'an
d', 'what', 'is', 'the', 'use', 'of', 'a', 'book', ",'", 'thought', 'Alice', "'", 'without', 'pictures', 'o
r', 'conversation', "?'", 'So', 'she', 'was', 'considering', 'in', 'her', 'own', 'mind', '(', 'as', 'well',
'as', 'she', 'could', ',']
34110
```

[4]:
```
enable = open('/zfs2/ling1340-2019s/shared_data/enable1.txt').read().split()
print(enable[:30])
print(len(enable))
```

```
['aa', 'aah', 'aahed', 'aahing', 'aahs', 'aal', 'aalii', 'aaliis', 'aals', 'aardvark', 'aardvarks', 'aardwo
lf', 'aardwolves', 'aargh', 'aarrgh', 'aarrghh', 'aas', 'aasvogel', 'aasvogels', 'ab', 'aba', 'abaca', 'aba
cas', 'abaci', 'aback', 'abacterial', 'abacus', 'abacuses', 'abaft', 'abaka']
172820
```

"Alice in Wonderland", 34K tokens

"Enable" word list, 173K total words

Task: find Alice words that are not found in enable list

- Try 1: list-comprehend through awords (list), filter against enable (list)

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

- Try 2: same, but filter against enable list as a SET

```
[6]: enable_set = set(enable)  # this one is a set data type
     print(len(enable_set))    # same size
```

```
172820
```

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

- Try 3: compute the SET difference

```
[8]: awords_set = set(awords)
     print(len(awords_set))    # now a set, smaller size
```

```
3016
```

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

- Try 1: list-comprehend through awords (list), filter against enable (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
```

- Try 2: same, but filter against enable list as a SET

```
[6]: enable_set = set(enable)  # this one is a set data type
     print(len(enable_set))    # same size

172820

[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
```

much faster

List as a data type is NOT optimized for membership operations…

but **set** is!

- Try 3: compute the SET difference

```
[8]: awords_set = set(awords)
     print(len(awords_set))    # now a set, smaller size

3016

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

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

blazing fast

Keep efficiency in mind, pick right combination of data structure and operation

Algorithmic complexity and the Big O

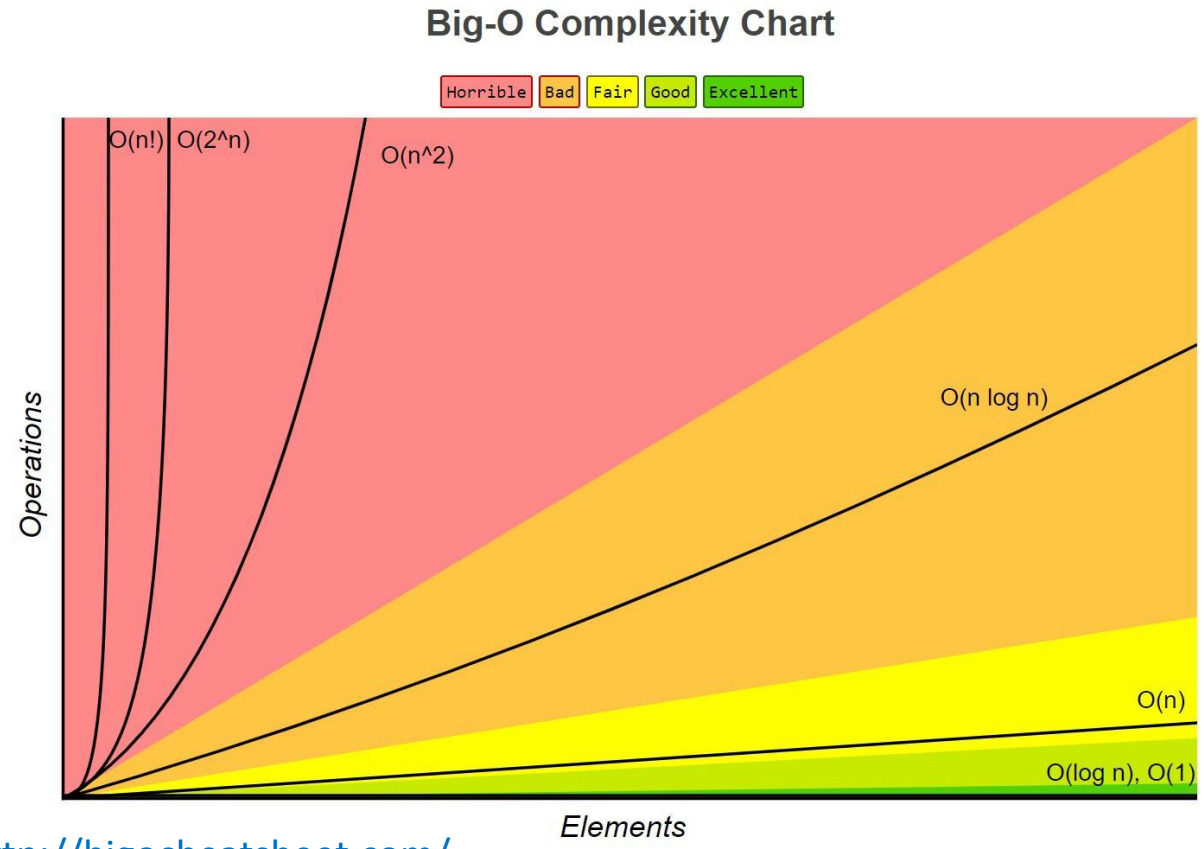
- https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
- We have a **list of *n* items**. Imagine *n* is 100, 1000 or even 1 million.

1. Is the first element an even number?
   - Can be implemented in O(1): an algorithm that executes in a **constant** time regardless of the size of the input dataset.
2. Does the list contain value 42?
   - Can be implemented in O(n): an algorithm whose performance will grow **linearly** in proportion to the size of the input data.
3. Does the list contain duplicate values?
   - Can be implemented in $O(n^2)$:an algorithm whose performance is directly proportional to the square of the size of the input data set (**quadratic**).

Algorithmic complexity and the Big O

- https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
- We have a **list of *n* items**. Imagine *n* is 100, 1000 or even 1 million.

4. Sort the list (ascending or descending)
   - Can be implemented in O(n log n): an algorithm that executes in **loglinear** time.
   - See: https://brilliant.org/wiki/sorting-algorithms/



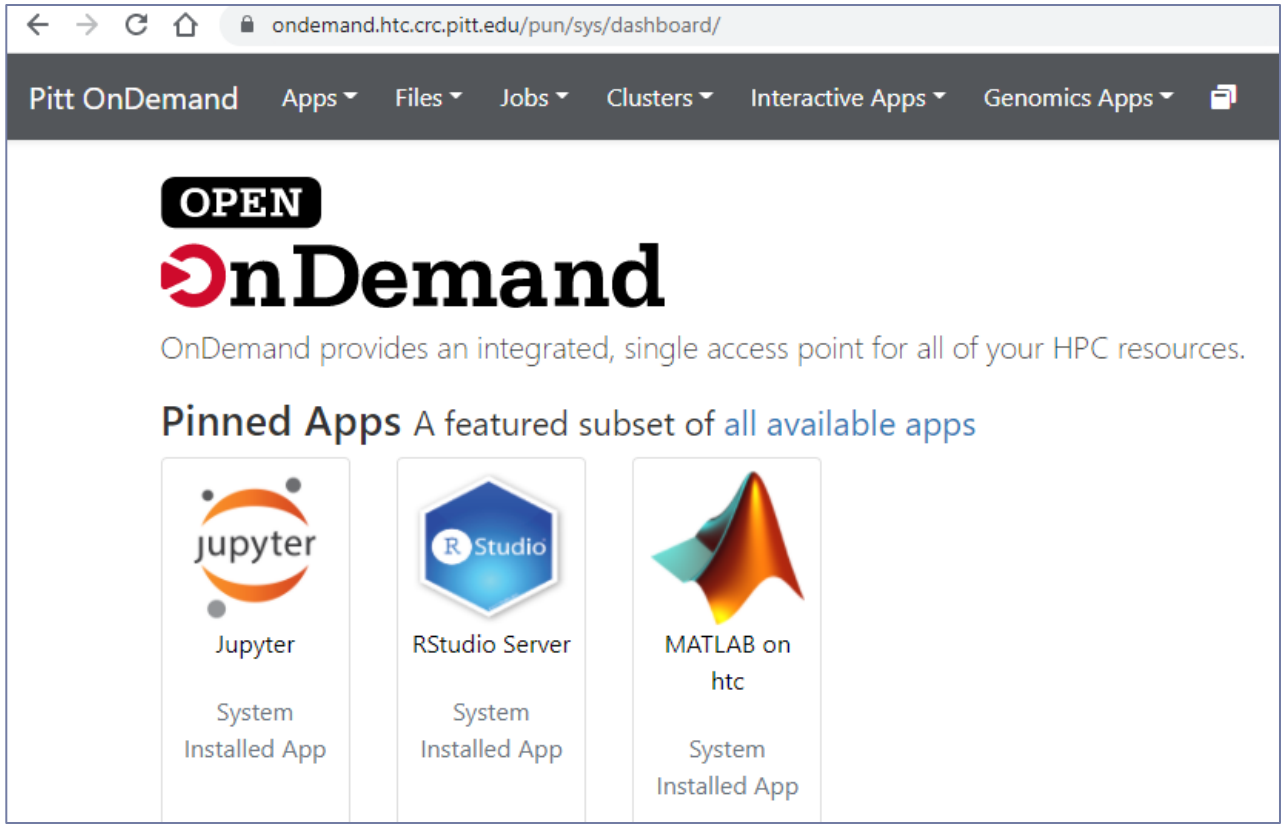
http://bigocheatsheet.com/

Algorithmic efficiency: summary

- A problem can be implemented with varying degrees of algorithmic efficiency.
- A problem comes with its own inherent algorithmic complexity limit.
  - **Big O notation** is a mathematical notation that encapsulates the relationship between the processing time and the input data size.
  - Example: the most efficient known sorting algorithm bottoms out at O(n log n).
- In a nutshell…
  - Compose the most efficient algorithm that you can.
  - Understand the relationship between the data size growth and the processing time growth. O(n) has fair scalability, $O(n^2)$ becomes intractable.
  - Efficiency of an algorithm can lead to dramatic runtime difference when dealing with big data.

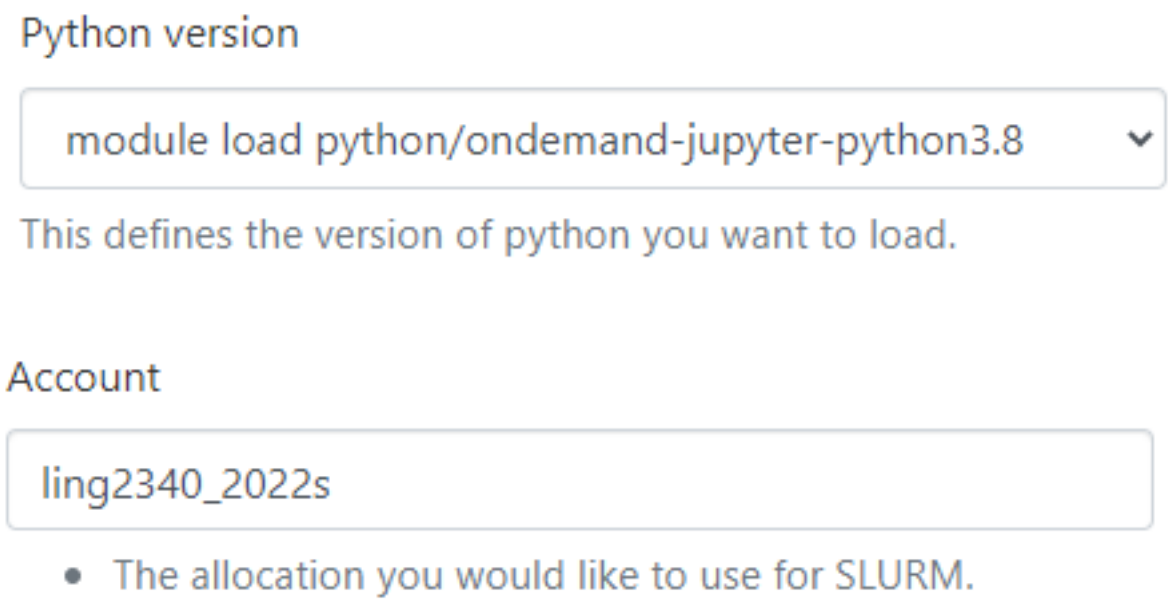
OnDemand on CRC!

- Browser-based gateway to CRC resources!
  - https://ondemand.htc.crc.pitt.edu/
- Jupyter Notebook (Lab) etc. are available!
- Help documentation is somewhat buried:
  - https://crc.pitt.edu/resources/htc-cluster/open-ondemand



Launching a session

- Python version: `module load python/ondemand-jupyter-python3.8`
- Account: `ling2340_2022s`



Jupyter (663715) 1 node | 1 core | Running

Host: >_htc-n16.crc.pitt.edu

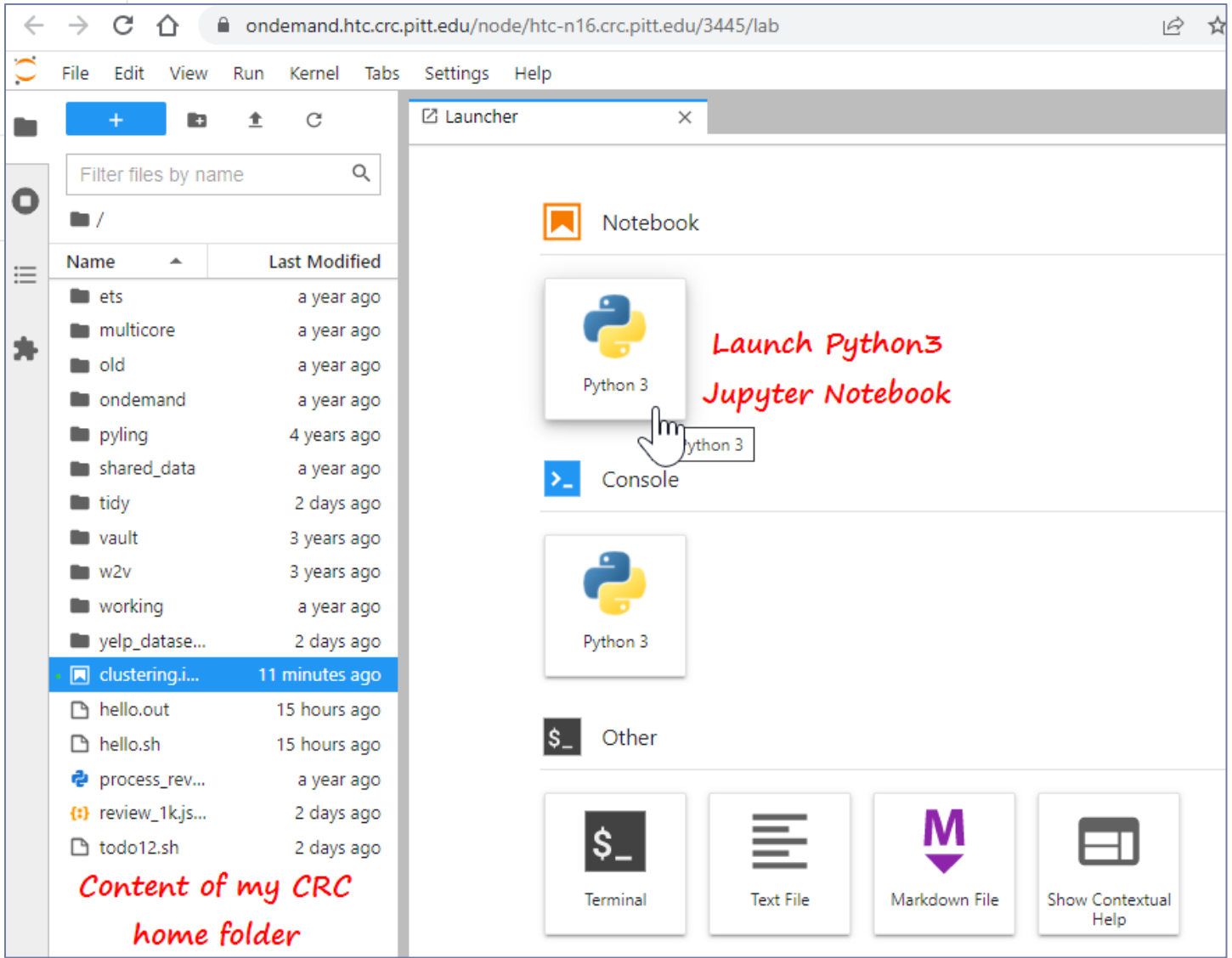
Delete

Created at: 2022-03-31 14:11:10 EDT

Time Remaining: 59 minutes

Session ID: 4f790652-695e-4043-9e38-27d4b9ace746

Connect to Jupyter



Wrap up

- Homework #4 out – don’t be too ambitious!
- Progress report #3, presentation up coming!