Review:
Data Structures

PyObject

```c
typedef struct _object {
  Py_ssize_t ob_refcnt;
  struct _typeobject *ob_type;
} PyObject;
```

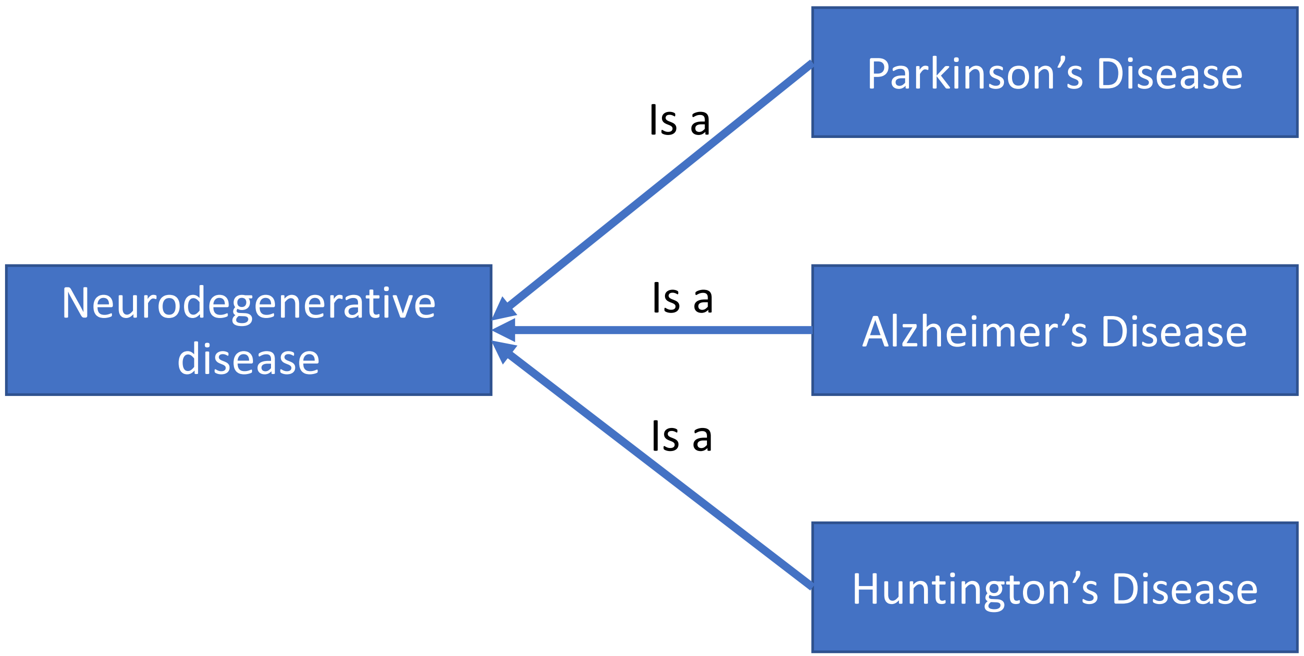
https://github.com/python/cpython/blob/master/Include/object.h

when Py_TRACE_REFS is not defined

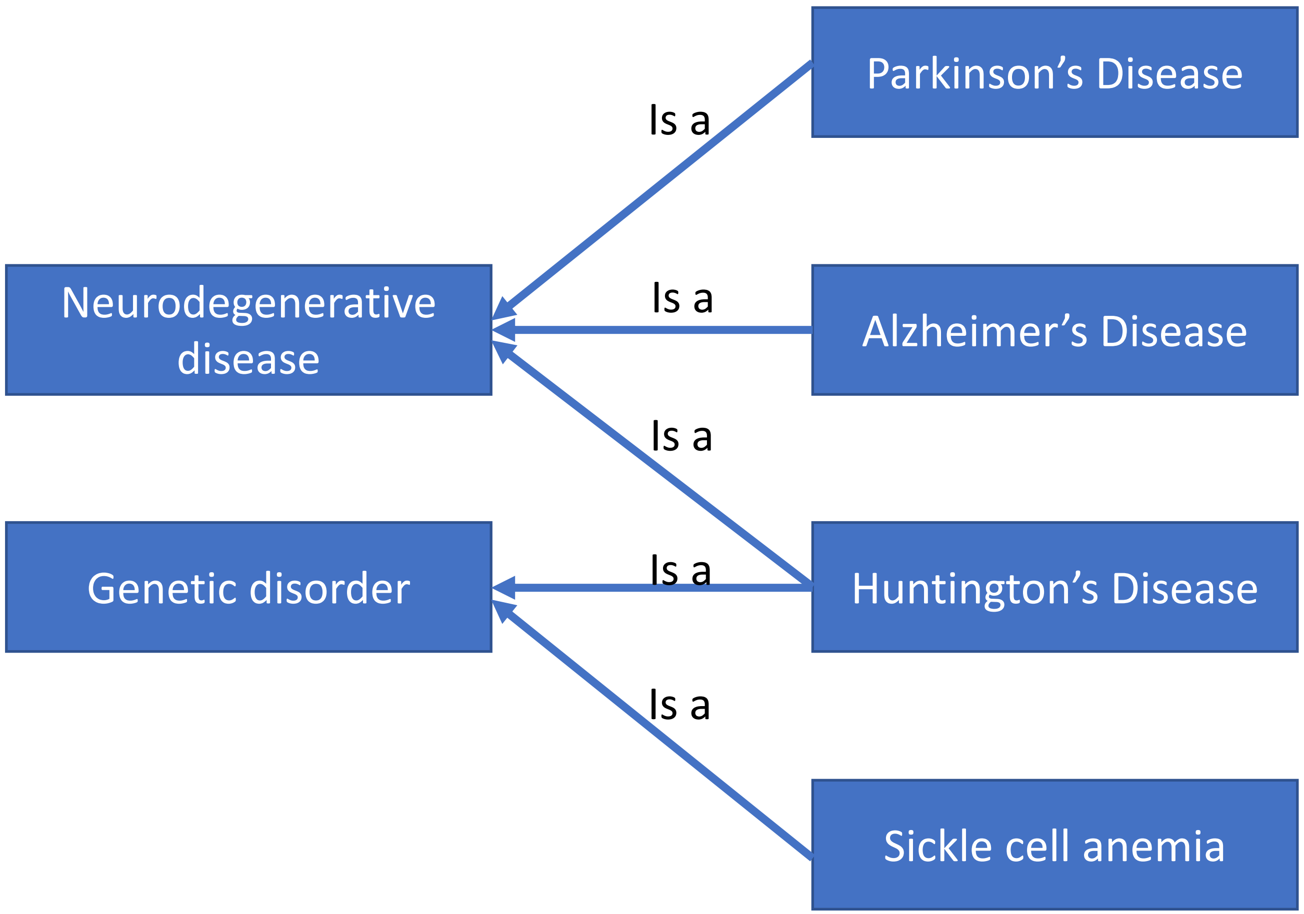
Review: Data Structures: Models of Numbers

```
>>> 9007199254740992.0 + 1
9007199254740992.0

>>> 1 + 2 ** -53 == 1
True

>>> 9007199254740992 + 1
9007199254740993
```

Review: Data Structures: Graph



Review: Data Structures: Graph



Alzheimer's disease (Q11081)

progressive, neurodegenerative disease characterized by memory loss — edit

Alzheimer | Alzheimers | AD | Alzheimer disease | Alzheimer's dementia | Alzheimers dementia | Alzheimer disease, familial | Alzheimer dementia | Alzheimer's Disease | Alzheimers disease

▾ In more languages

Configure

Language	Label	Description	Also known as
English	Alzheimer's disease	progressive, neurodegenerative disease characterized by memory loss	Alzheimer Alzheimers AD Alzheimer disease Alzheimer's dementia Alzheimers dementia Alzheimer disease, familial Alzheimer dementia Alzheimer's Disease Alzheimers disease
Spanish	enfermedad de Alzheimer	enfermedad neurodegenerativa caracterizada por la pérdida de memoria	Alzheimer Alzheimers ANUNCIO Demencia de Alzheimer Enfermedad de Alzheimer, famili…
Traditional Chinese	阿茲海默症	人類疾病	
Chinese	阿兹海默病	人類疾病	阿爾哈瑪病 阿茲海默症 老年痴呆 早老性痴呆

All entered languages

Statements

instance of	disease ▸ 1 reference	edit
	rare disease	edit

symptoms	misplacing things and losing the ability to retrace steps ▸ 1 reference	edit
	dementia ▸ 1 reference	edit
		+ add value

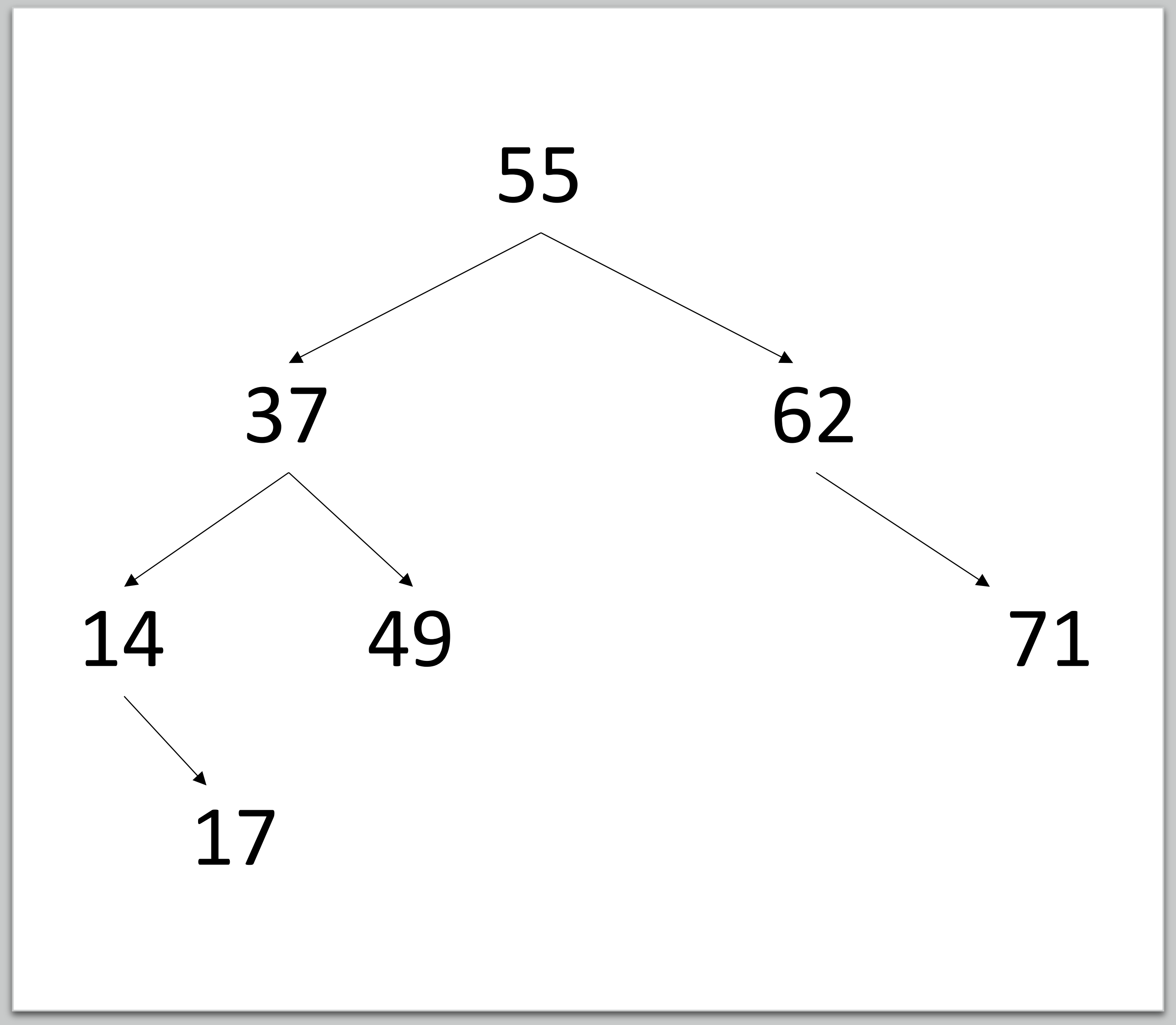
medical examinations	neurological diagnostic techniques ▸ 1 reference	edit
	medical history ▸ 1 reference	edit
	magnetic-resonance imaging ▸ 1 reference	edit
	psychological test ▾ 0 references	edit
		+ add reference
		+ add value

possible treatment	memantine ▸ 1 reference	edit
	donepezil ▸ 1 reference	edit
	rivastigmine	edit

Abridged from https://www.wikidata.org/wiki/Q11081, accessed 2020-09-08

Review: Data Structures: Binary search trees

- When adding or looking for items, move left if the new item is smaller, right if bigger than the current vertex.
- If a tree of N items is balanced, takes about $\log_2 N$ comparisons to test if an item is present.
- Can be used to sort items.
- E.g. build a tree with:
  55, 62, 37, 49, 71, 14, 17



Analysis of Algorithms

9 September 2021

Overview

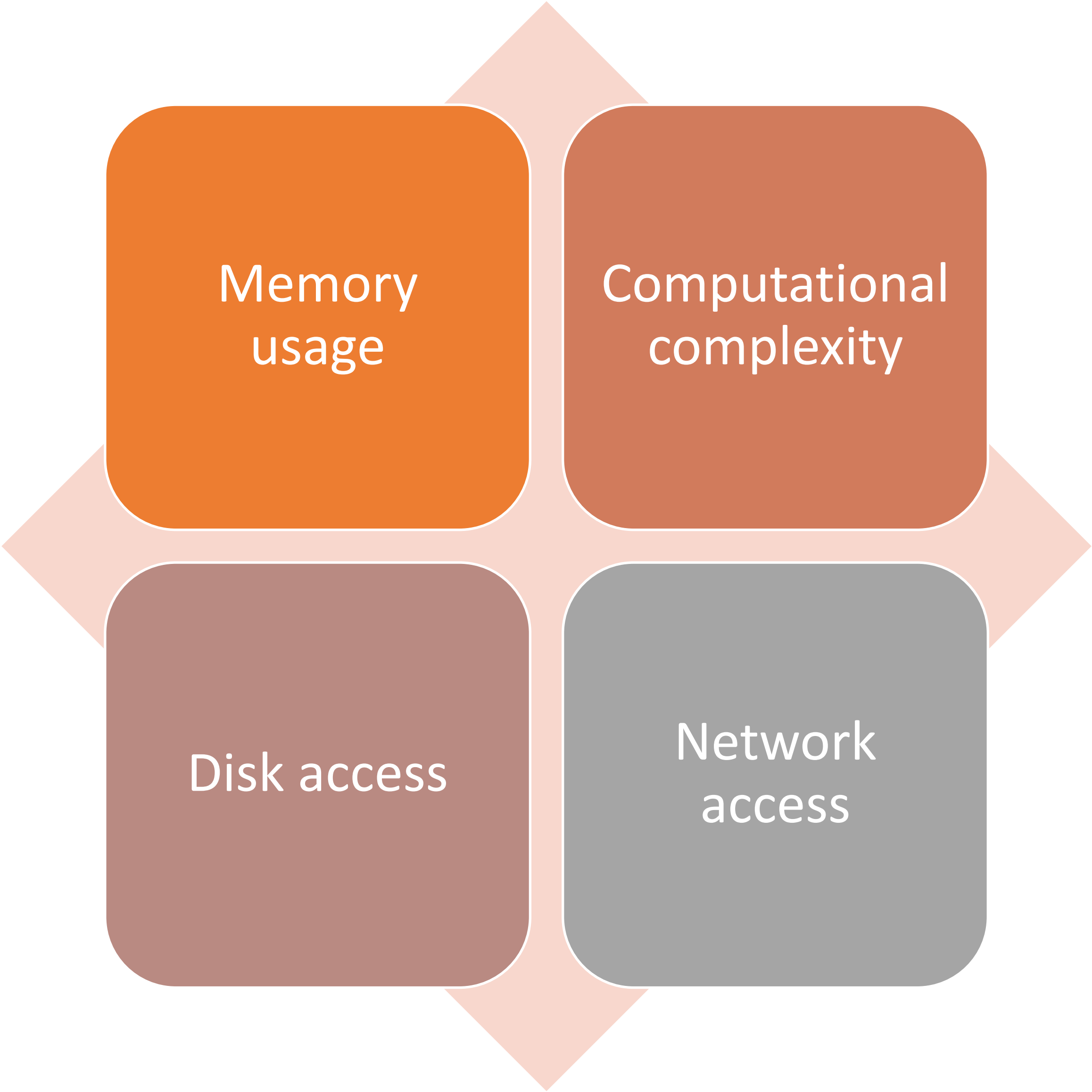
- Motivating example.
- Mathematical concepts: big-O, little-o, big-Θ.
- Examples.

Motivating example: finding a patient

- How much work is it to find a specific patient: Rodriguez, Kathy

- If I have n total patients stored as:
    - an unsorted list?
    - a list sorted by name?
    - a binary search tree sorted by name?
    - a dictionary keyed by name?

Performance considerations



Mathematical Review

Big *O*, Little *o*, and Big Θ

Big O

For a given $f(x)$, we say

$$f(x) = O\big(g(x)\big)$$

if and only if there exists x_0 and M such that

$$|f(x)| \leq Mg(x)$$

for all $x \geq x_0$.

Example

$$3x^5 + 200x^4 + 7 = O(x^5)$$

Example

$3x^5 + 200x^4 + 7 = O(x^5)$

Proof:

For $x \geq 201$,

$$x^5 = x \cdot x^4 \geq 201x^4 = 200x^4 + x^4 \geq 200x^4 + 201^4 > 200x^4 + 7$$

Therefore for $x \geq 201$,

$$4x^5 = 3x^5 + x^5 > 3x^5 + 200x^4 + 7 = |3x^5 + 200x^4 + 7|$$

That is, $|3x^5 + 200x^4 + 7| < Mx^5$ for all $x \geq x_0$ where $M = 4$ and $x_0 = 201$. The claim follows immediately, by definition of "Big O".

Lemma

If

$$f(x) = \sum_{i=0}^{m} a_i x^i = a_0 + a_1 x + \cdots + a_m x^m$$

then

$$f(x) = O(x^m)$$

That is, a polynomial of degree m is $O(x^m)$.

Warning

- The “=” symbol in Big O notation is a symbol denoting a non-symmetric relationship; it is not equality.

- For this reason, sometimes people prefer to write

$$f(x) \in O(g(x))$$

Little o

For a given $f(x)$, we say

$$f(x) = o\big(g(x)\big)$$

if and only if for every $\varepsilon > 0$, there exists N such that

$$|f(x)| \leq \varepsilon g(x)$$

for all $x \geq N$.

Example

$$2x = o(x^2)$$

Example

$$2x = o(x^2)$$

Proof:

Let $\varepsilon > 0$ be given. Define $N = \frac{2}{\varepsilon}$.
Then for any $x \geq N$, (multiplying both sides by x)

$$x^2 \geq \frac{2}{\varepsilon} x$$

so

$$\varepsilon x^2 \geq 2x = |2x|$$

where the last equality holds because $\varepsilon > 0$ implies $N > 0$ and thus $x \geq \mathrm{N} > 0$.

Lemma

If

$$f(x) = o(g(x))$$

then

$$f(x) = O\big(g(x)\big).$$

The converse *does not hold*.

Big Θ

For a given $f(x)$, we say

$$f(x) = \Theta\big(g(x)\big)$$

if and only if there exists x_0, M_1 and M_2 such that

$$M_1 g(x) \leq |f(x)| \leq M_2 g(x)$$

for all $x \geq x_0$.

$O(1)$

$O(\log\log n)$

$O(\log n)$

$O(n^a), 0 < a < 1$

$O(n)$

$O(n \log n)$

$O(n^a), a > 1$

$O(a^n), a > 1$

$O(n!)$

smaller

larger

Back to informatics

In data science and informatics…

… we usually want to quantify memory usage and computational complexity in terms of the number of data points, n.

Example

If we have n glucose readings, what is the memory usage and compute time of the following algorithm, in terms of Big Θ?

```
max_glucose = 0
for glucose in patients_glucose:
    if glucose > max_glucose:
        max_glucose = glucose
```

Example

How long will this take to run in the big O sense? Assume `data` is a Python `list` of length n.

```
total = 0
for i in range(100):
    for j in range(i, 100):
        total += i * j * len(data)
```

Example

If we have n nmers all of length 15, what is the memory usage and compute time of the following algorithm, in terms of Big O?

```
for nmer in nmers:
    hashed_values = [
        (((int(sha256(nmer.lower().encode()).hexdigest(), 16)
          % bits_48) * a) % p) & scale
        for a in range(100)
    ]
```

Observation

- You can reduce runtime significantly for specific datasets without changing the Big O, little o, or Big Θ.
- If the size of your data can change, then reducing the Big O, little o, or Big Θ is more important.
  - Speedups for fixed-size datasets will come “for free” with improvements in computer technology.

Example

If we have n glucose readings, what is the memory usage and compute time of the following algorithm, in terms of Big O?

```
distinct_readings = []
for glucose in patients_glucose:
    for old_value in distinct_readings:
        if glucose == old_value:
            break
    else:
        distinct_readings.append(glucose)
```

Example

If we have n glucose readings, what is the memory usage and compute time of the following algorithm, in terms of Big O?

```
distinct_readings = set()
for glucose in patients_glucose:
    distinct_readings.add(glucose)
distinct_readings = list(distinct_readings)
```

Timing

`time.perf_counter()`

Provides a high-resolution timer measured in seconds.

Time 0 is undefined and system dependent, but the difference of two times can be used to calculate runtime.

(You could do something similar with `time.time()`, which returns the number of seconds since January 1, 1970 00:00 GMT, but due to the finite precision of floating-point numbers, this is less precise than using `time.perf_counter`.)

Example: timing (bubble sort)

```
$ python bubblesort.py
Runtime: 10.022397994995117 s
$ python bubblesort.py
Runtime: 9.96274995803833 s
$ python bubblesort.py
Runtime: 10.286312341690063 s
```

```python
import random
import time

# ensure consistent random numbers
random.seed(1)
weights = [random.normalvariate(170, 30)
           for patient in range(10_000)]

def time_algorithm(algorithm, original_data):
    data = list(original_data)
    start = time.perf_counter()
    algorithm(data)
    return time.perf_counter() - start

def sort(data):
    changes = True
    while changes:
        changes = False
        for i in range(len(data) - 1):
            if data[i + 1] < data[i]:
                data[i], data[i + 1] = data[i + 1], data[i]
                changes = True
    return data

print(f'Runtime: {time_algorithm(sort, weights)} s')
```

Example: timing (bubble sort)

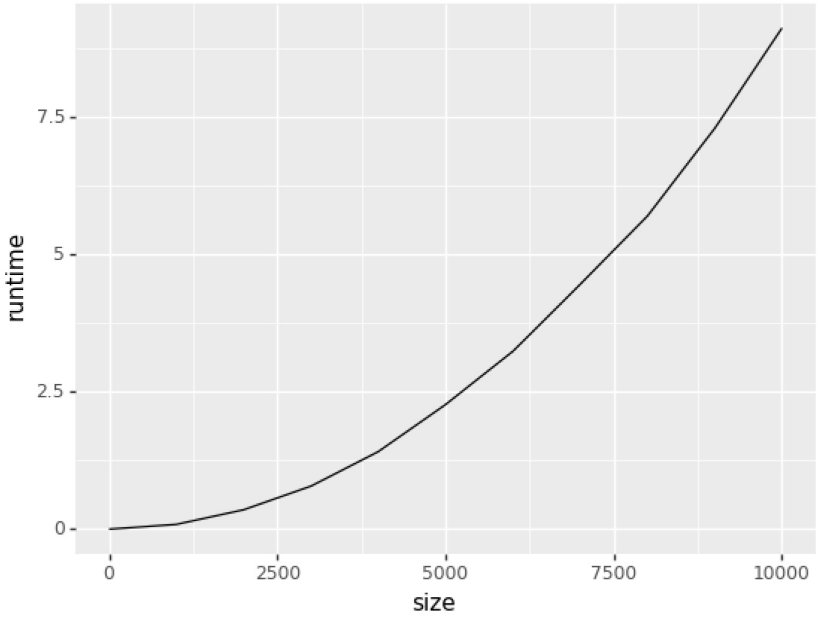
```
import random
import time
# ensure consistent random numbers
random.seed(1)
weights = [random.normalvariate(170, 30)
           for patient in range(10_000)]
def time_algorithm(algorithm, original_data):
    times = []
    for attempt in [1, 2, 3]:
        data = list(original_data)
        start = time.perf_counter()
        algorithm(data)
        times.append(time.perf_counter() - start)
    return min(times)
def sort(data):
    changes = True
    while changes:
        changes = False
        for i in range(len(data) - 1):
            if data[i + 1] < data[i]:
                data[i], data[i + 1] = data[i + 1], data[i]
                changes = True
    return data

print(f'Runtime: {time_algorithm(sort, weights)} s')
```

Modified to make multiple attempts and report the minimum

```
$ python bubblesort2.py
Runtime: 9.966766119003296 s
$ python bubblesort2.py
Runtime: 9.92915391921997 s
$ python bubblesort2.py
Runtime: 9.93209195137024 s
```

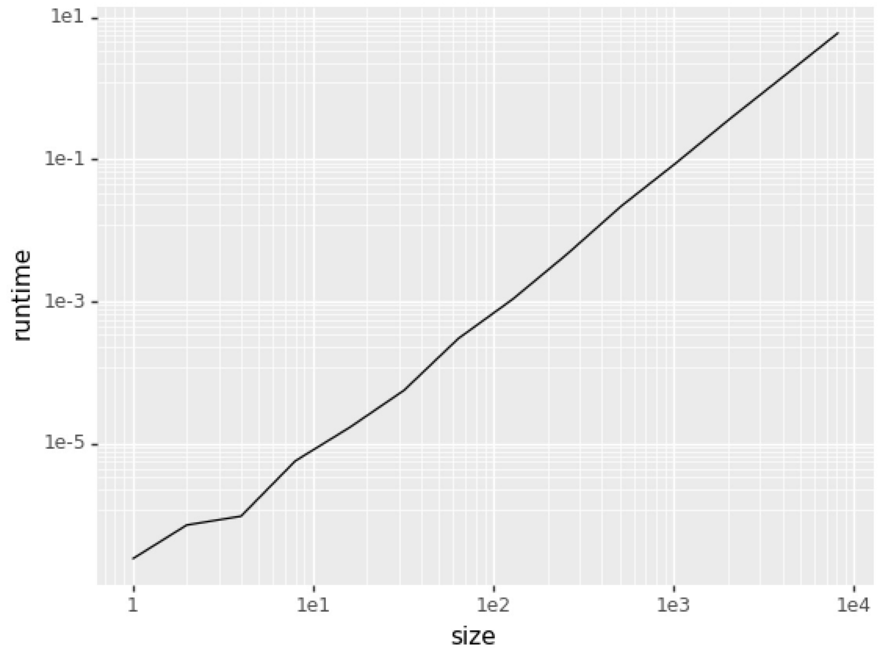
Example: timing (bubble sort)

```
import plotnine as p9
import pandas as pd

sizes = [1, 1000, 2000, 3000, 4000, 5000, 6000, 7000,
         8000, 9000, 10000]

times = [time_algorithm(sort, weights[:size])
         for size in sizes]
print(
    p9.ggplot()
    + p9.geom_line(
        pd.DataFrame({'size': sizes, 'runtime': times}),
        p9.aes(x='size', y='runtime'))
)
```



Example: timing (bubble sort)

```python
import plotnine as p9
import pandas as pd

sizes = [2 ** i for i in range(14)]

times = [time_algorithm(sort, weights[:size])
         for size in sizes]
print(
    p9.ggplot()
    + p9.geom_line(
        pd.DataFrame({'size': sizes, 'runtime': times}),
        p9.aes(x='size', y='runtime'))
    + p9.scale_x_log10()
    + p9.scale_y_log10()
)
```

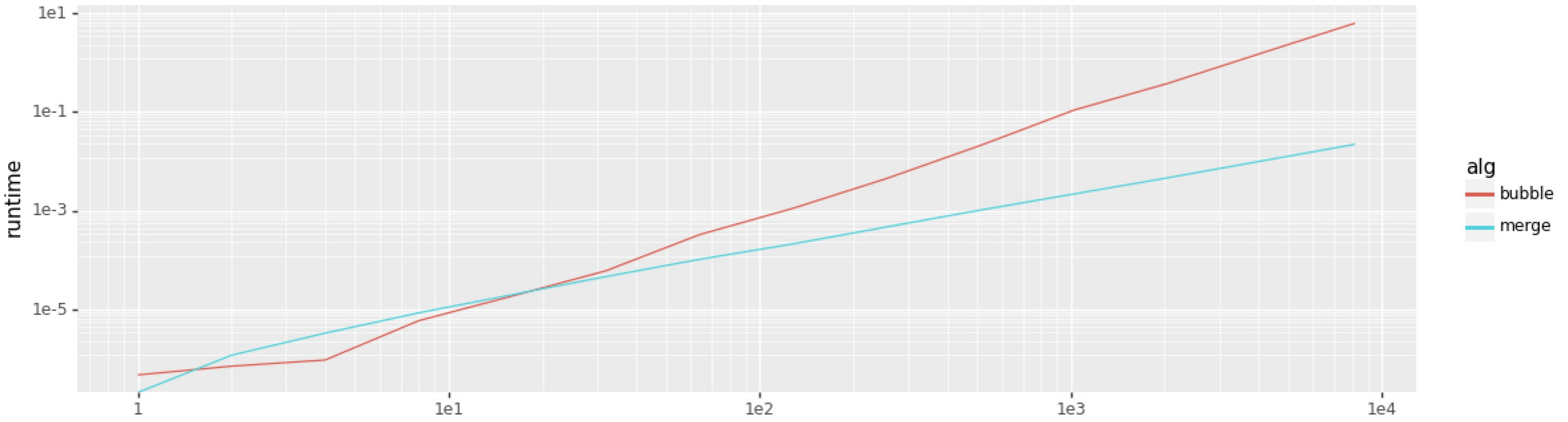


Example: merge sort

```python
def merge_sort(data):
  if len(data) <= 1:
    return data
  else:
    split = len(data) // 2
    left = iter(merge_sort(data[:split]))
    right = iter(merge_sort(data[split:]))
    result = []
    # note: this takes the top items off the left and right piles
    left_top = next(left)
    right_top = next(right)
    while True:
      if left_top < right_top:
        result.append(left_top)
        try:
          left_top = next(left)
        except StopIteration:
          # nothing remains on the left; add the right + return
          return result + [right_top] + list(right)
      else:
        result.append(right_top)
        try:
          right_top = next(right)
        except StopIteration:
          # nothing remains on the right; add the left + return
          return result + [left_top] + list(left)
```

Comparison of merge and bubble sorts

```
import plotnine as p9
import pandas as pd

sizes = [2 ** i for i in range(14)]
bubble_times = [time_algorithm(sort, weights[:size])
                for size in sizes]
merge_times = [time_algorithm(merge_sort, weights[:size])
               for size in sizes]
print(
    p9.ggplot()
    + p9.geom_line(
        pd.concat([
            pd.DataFrame({'size': sizes, 'runtime': bubble_times, 'alg': 'bubble'}),
            pd.DataFrame({'size': sizes, 'runtime': merge_times, 'alg': 'merge'})
        ]),
        p9.aes(x='size', y='runtime', color='alg'))
    + p9.scale_x_log10()
    + p9.scale_y_log10()
)
```



Python operations

- Looking up or setting an item in a `list` or `dictionary` by index.
  - $O(1)$
    - For dictionaries, this is assuming the add doesn’t require reallocating storage.
- Checking if an item is in a list
  - $O(n)$
- Checking if an item is in a set
  - $O(1)$

Computational Complexity of Common Informatics Tasks

- Comparison sort
  - $O(n \log n)$
- Binary tree insertion/searching/deletion
  - Arbitrary tree: $O(n)$
  - Balanced tree: $O(\log n)$
- Matrix multiplication
  - Usual algorithm: $O(n^3)$
  - Strassen algorithm: $O(n^{2.807355})$
  - Coppersmith-Winograd: $O(n^{2.375477})$

Computational Complexity of Common Informatics Tasks

- Traveling salesman problem
  - Via dynamic programming: $2^{O(n)}$
  - Brute force: $O(n!)$
- PCA, n data points, p features
  - $O(\min(p^3, n^3))$
- Fast Fourier transform (time series analysis)
  - $O(n \log n)$
- Naïve k-nearest neighbors (with p features)
  - $O(np + kn)$ or $O(knp)$ depending on implementation
- Lloyd's algorithm (k-means; p features; i iterations)
  - $O(nkpi)$ but for convergence $i = 2^{\Omega(\sqrt{n})}$

Coming up soon...

Tue. 14 Sep.

Lecture on Data Standards

Wed. 15 Sep

Discussion on Convolutional Neural Networks

Possibly another discussion presentation?

Thu. 16 Sep.

Lecture on Big Data.

Homework 1 due.