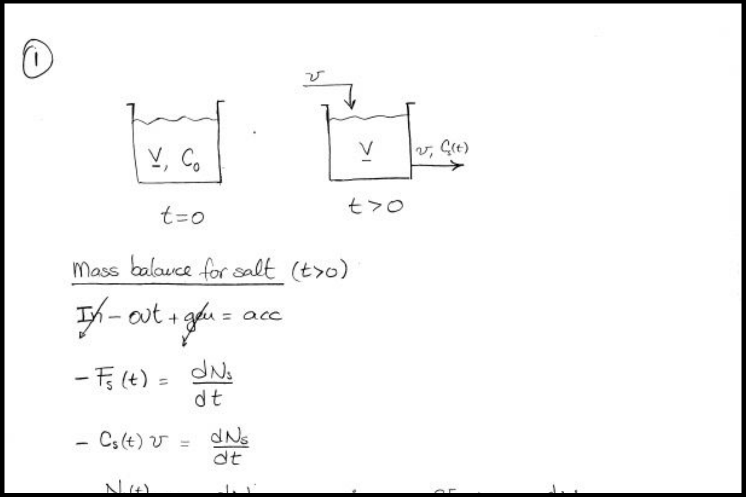
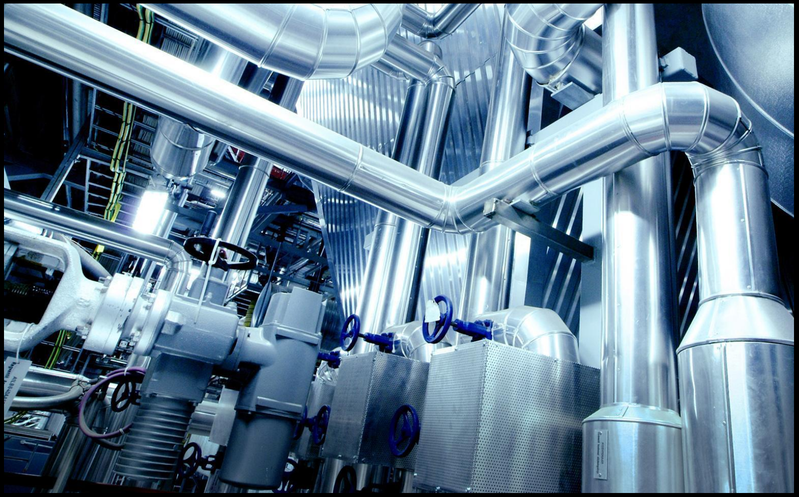
The Cooper Union for the Advancement of Science and Art





ChE352

Numerical Techniques for Chemical Engineers

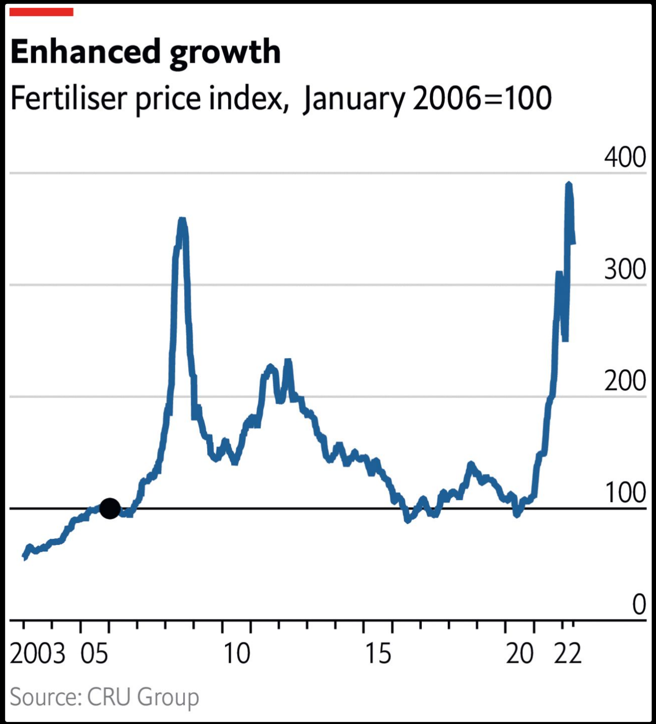
Professor Stevenson

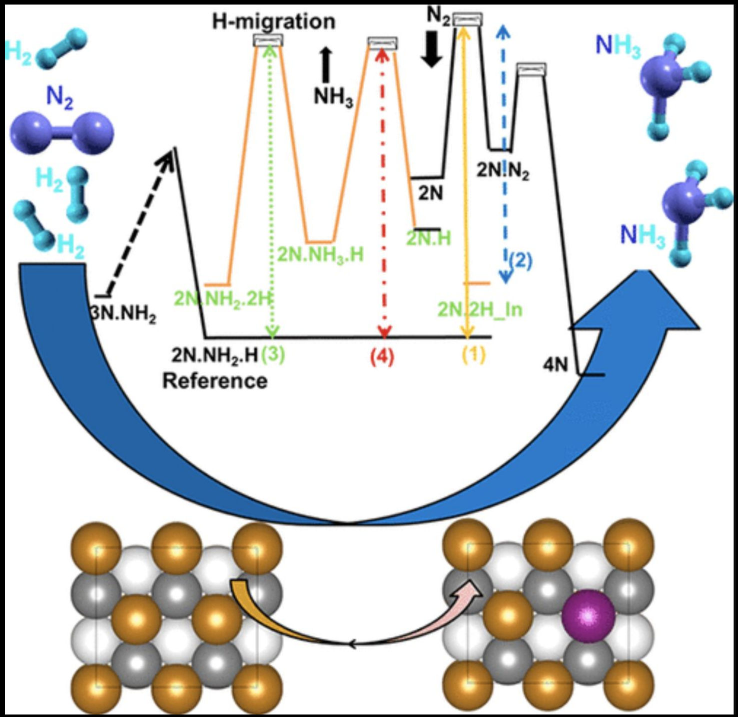
Lecture 5

Why is finding catalysts difficult?

- The Schrödinger equation lets us calculate energy barriers for any reaction
- But catalysts are still an open question - Why?

$$N_2 + 3\,H_2 \longrightarrow 2\,NH_3 \quad \Delta H^\circ = -91.8\ \mathrm{kJ/mol}$$

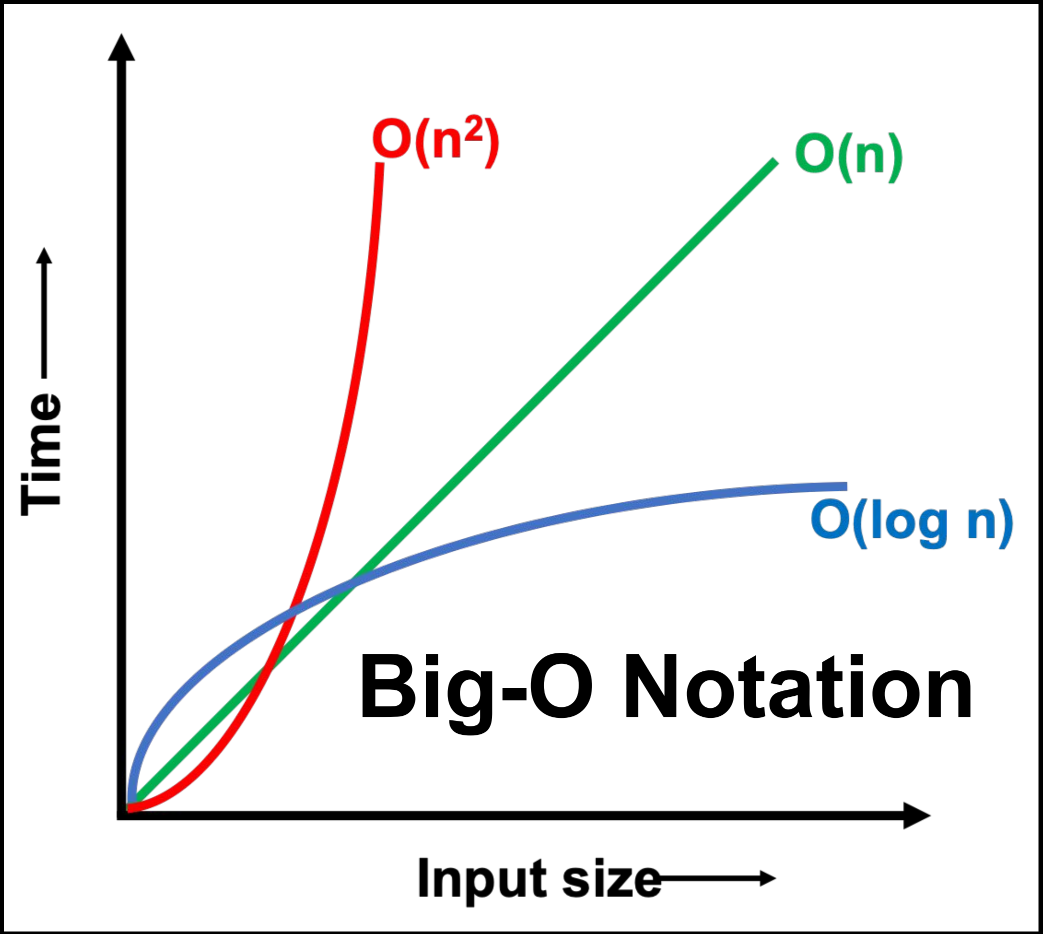




How can we reason about speed?

- Each computer's speed is different
- Even on the same computer, timing will vary randomly
- There is a way to describe speed that avoids all of this...

How can we reason about speed?



- Each computer's speed is different
- Even on the same computer, timing will vary randomly
- There is a way to describe speed that avoids all of this...

Big-O notation

Big-O simplifies an expression by discarding everything but the largest term in the limit of large N, also dropping constant prefactors

Run time	Big-O	Description
$t = N/2 + 999$	?	?
$t = 3N^2 + 5N$	?	?
$t = 7N^3 + 8N^2$	?	?
$t = 2^N + N^{9999}$	?	?

This gets rid of computer speed & noise

Big-O notation

Big-O simplifies an expression by discarding everything but the largest term in the limit of large N, also dropping constant prefactors

Run time	Big-O	Description
$t = N/2 + 999$	$O(N)$	Linear
$t = 3N^2 + 5N$	$O(N^2)$	Quadratic
$t = 7N^3 + 8N^2$	$O(N^3)$	Cubic
$t = 2^N + N^{9999}$	$O(2^N)$	Exponential

This gets rid of computer speed & noise

Big-O notation examples

O(N) algorithms act on each input a constant number of times

How might an $O(N^2)$ algorithm arise?

Big-O notation examples

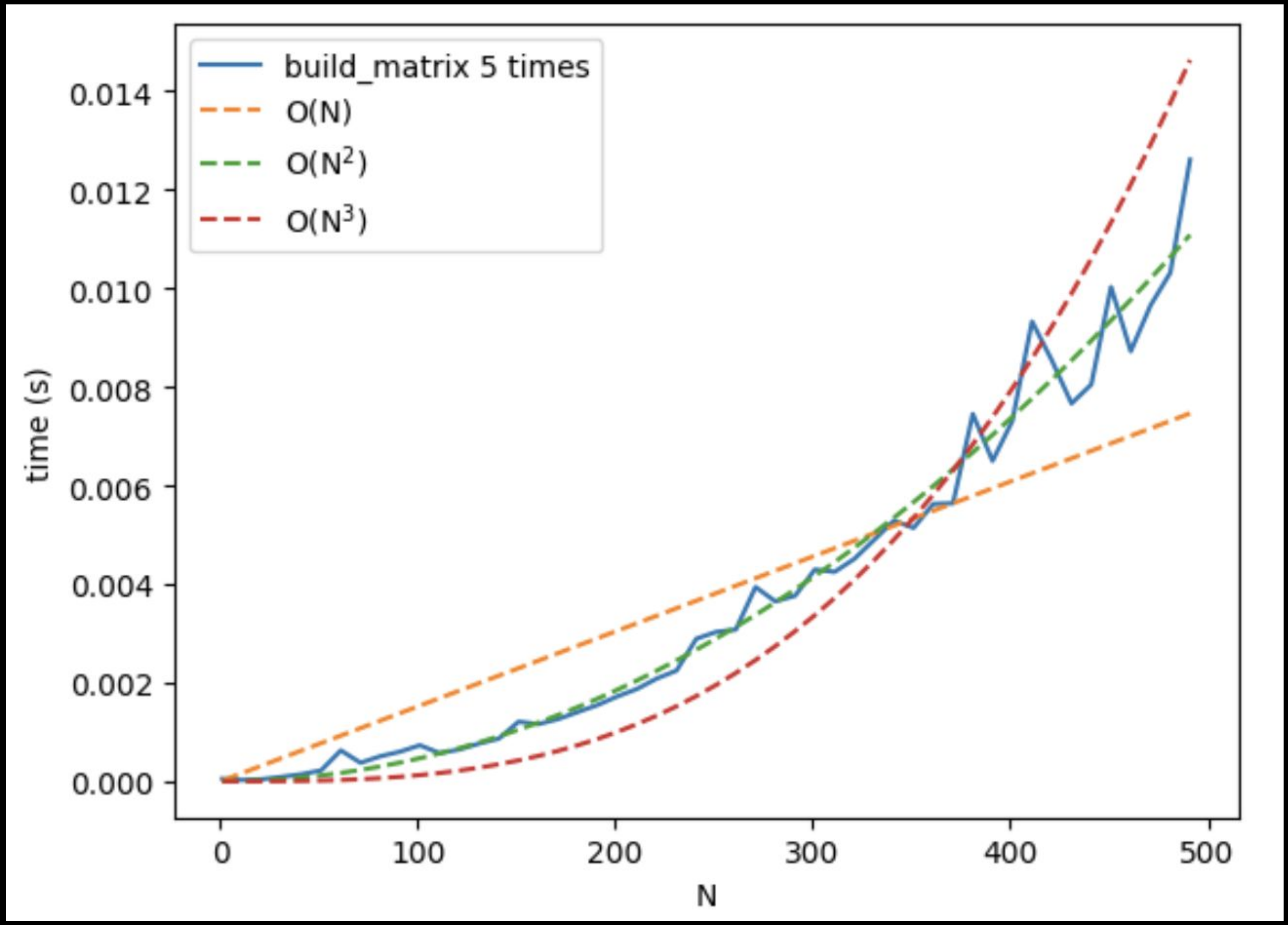
O(N) algorithms act on each input a constant number of times

How might an $O(N^2)$ algorithm arise?

1 ONE	2 TWO	3 THREE	4 FOUR	5 FIVE	6 SIX
1x1 = 1	2x1 = 2	3x1 = 3	4x1 = 4	5x1 = 5	6x1 = 6
1x2 = 2	2x2 = 4	3x2 = 6	4x2 = 8	5x2 = 10	6x2 = 12
1x3 = 3	2x3 = 6	3x3 = 9	4x3 = 12	5x3 = 15	6x3 = 18
1x4 = 4	2x4 = 8	3x4 = 12	4x4 = 16	5x4 = 20	6x4 = 24
1x5 = 5	2x5 = 10	3x5 = 15	4x5 = 20	5x5 = 25	6x5 = 30
1x6 = 6	2x6 = 12	3x6 = 18	4x6 = 24	5x6 = 30	6x6 = 36

Doing N*N things, like building an NxN matrix

Big-O notation examples



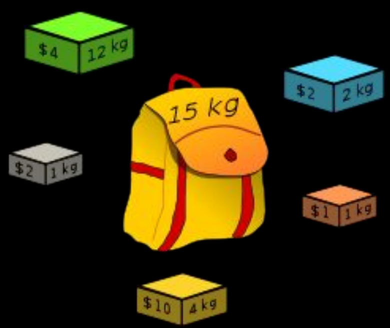
Doing N*N things, like building an NxN matrix

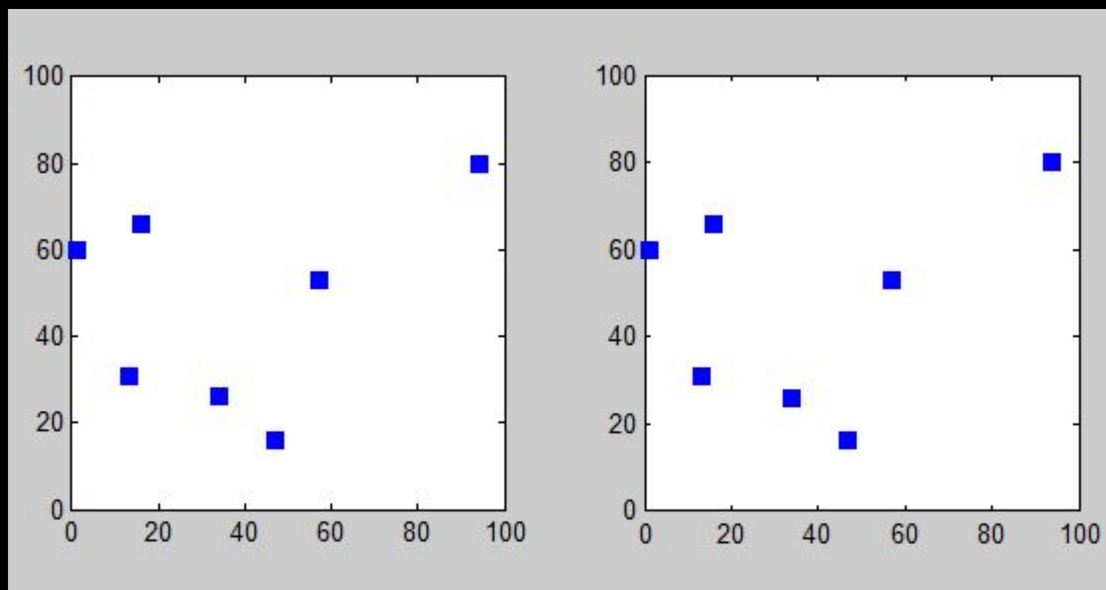
Big-O notation examples

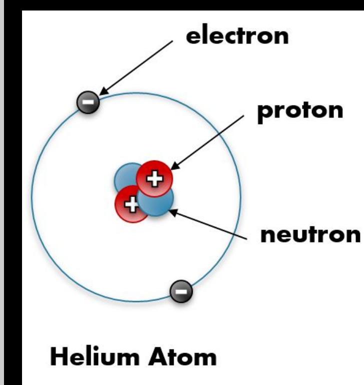
How might an $O(2^N)$, or worse, algorithm arise?

Big-O notation examples

How might an $O(2^N)$, or worse, algorithm arise?







Trying every permutation (Traveling Salesman, Knapsack Problem, <u>Schrödinger equation</u>)

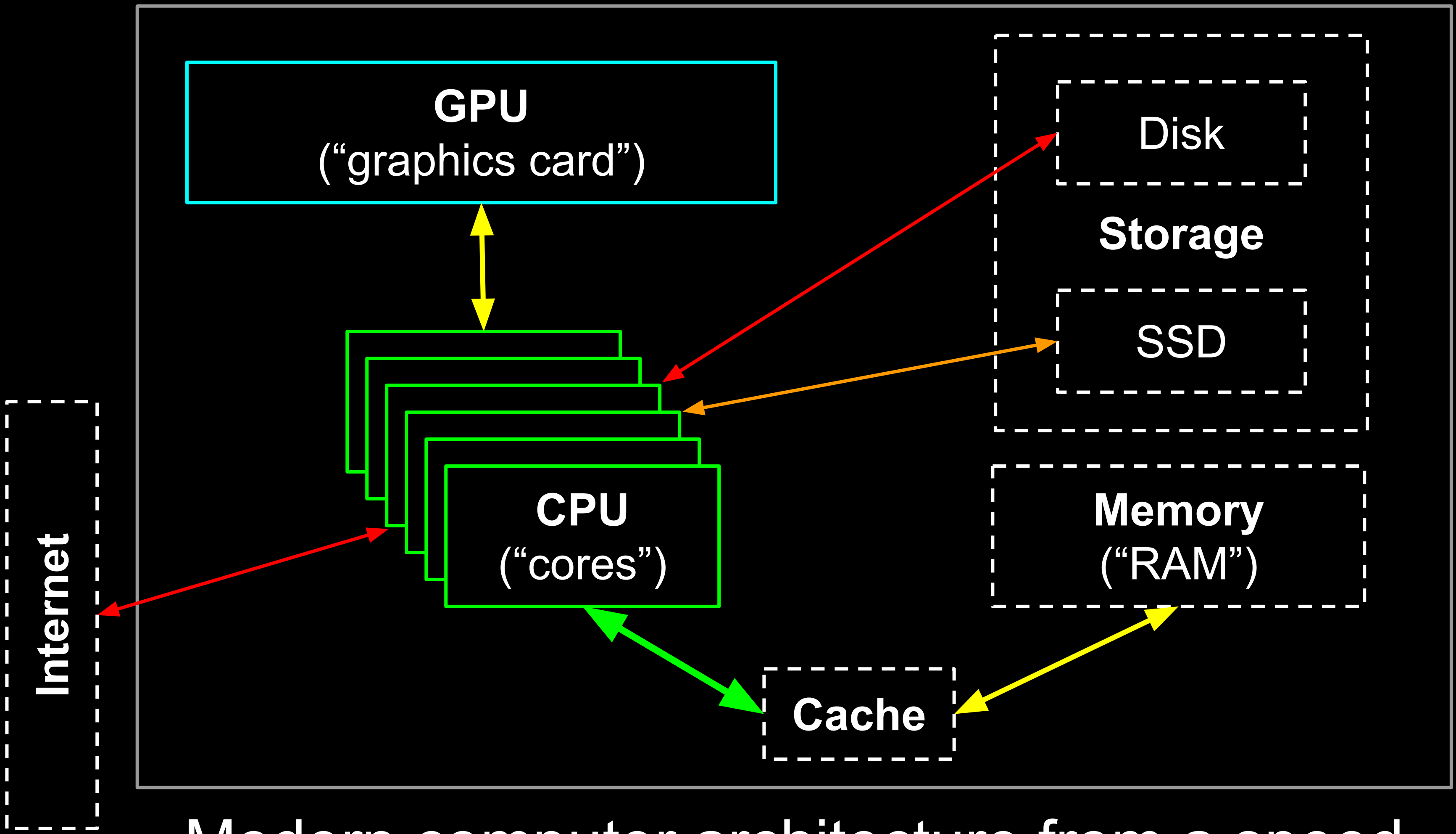
Constant factors

Big-O notation ignores constant factors:
If you wait an hour at every step,
your big-O stays the same!

Run time	Big-O
$t \propto 1{,}000{,}000\,N$	$O(N)$
$t \propto 1{,}000{,}000\,N^2$	$O(N^2)$

(You should still try not to do this)

What is fast and what is slow?



Modern computer architecture from a speed perspective (really fast, fast, medium, slow)

Why is code slow?

- Big-O is too high (bad algorithm)
  - How would you find big-O?
- Too much input/output (especially files)
  - Why is I/O slower than calculations?
- Too much task switching
  - Not *vectorized* (needs more numpy)
- *Convergence* is poor (algorithm issue)

Why is code slow?

```
import timeit

def arithmetic_answer():

    return 2 + 2

def file_answer():

    with open('data.txt', 'w') as f:

        f.write(f'{4}')

    with open('data.txt', 'r') as f:

        answer = int(f.read())

    return answer
```

What is "timeit"?

What do these two functions do?

Which one is faster?

Why is code slow?

```python
import timeit
def arithmetic_answer():
    return 2 + 2
def file_answer():
    with open('data.txt', 'w') as f:
        f.write(f'{4}')
    with open('data.txt', 'r') as f:
        answer = int(f.read())
    return answer
arithmetic_time = timeit.timeit(arithmetic_answer, number=1000)
file_time = timeit.timeit(file_answer, number=1000)
print(f'{arithmetic_time=:.1e}, {file_time=:.1e}')
```

Try this code in Colab

What % of file_time is the file writing vs the file reading?

Big-O for Getting Smaller

- Big-O also works for numbers getting smaller, such as error $\alpha - \alpha_n$ shrinking as n grows
  - $O(1/n + 1/n^2) =$ **?**

Big-O for Getting Smaller

- Big-O also works for numbers getting smaller, such as error $\alpha - \alpha_n$ shrinking as n grows
  - $O(1/n + 1/n^2) = O(1/n)$ as $n \to \infty$

For some K

$$\{\alpha_n\}_{n=1}^{\infty} \quad is \quad O\left(\frac{1}{n^{\beta}}\right) \leftrightarrow \boxed{\left|\alpha - \alpha_n\right| \le \frac{K}{n^{\beta}}} \quad (n\ large)$$

- Same idea if real-valued input *h* goes to zero:
  - $O(h^2 + h^4) =$ **?**

Big-O for Getting Smaller

- Big-O also works for numbers getting smaller, such as error α - α_n shrinking as n grows
  - $O(1/n + 1/n^2) = O(1/n)$ as $n \rightarrow \infty$

For some K

$$\{\alpha_n\}_{n=1}^{\infty} \quad is \quad O\left(\frac{1}{n^\beta}\right) \quad \leftrightarrow \quad \boxed{\left|\alpha - \alpha_n\right| \le \frac{K}{n^\beta}} \quad (n\; large)$$

- Same idea if real-valued input *h* goes to zero:
  - $O(h^2 + h^4) = O(h^2)$ as $h \rightarrow 0$

$$f(h) \quad is \quad O\left(h^\beta\right) \quad \leftrightarrow \quad \boxed{\left|f(h) - L\right| \le K h^\beta \quad as\; h \rightarrow 0}$$

For some K

Summary and Problems

- Open Python Numerical Methods, go to Chapter 8.4: Summary and Problems

https://pythonnumericalmethods.berkeley.edu/notebooks/chapter08.04-Summary-and-Problems.html

- Do the first three problems, starting with:
  - How would you define the size of the following tasks?
    - Solving a jigsaw puzzle.
    - Passing a handout to a class.
    - Walking to class.
    - Finding a name in dictionary.