GLIS 691 | Karyn Moffatt

Computer Programming for Information Professionals

Complexity

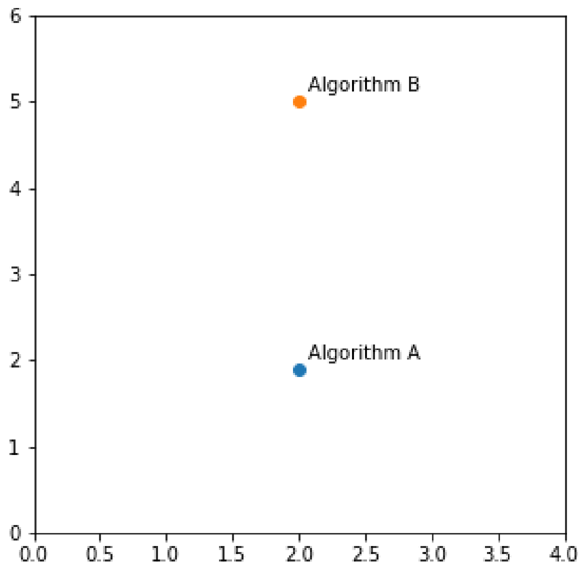
What makes a good algorithm?

Suppose we have two algorithms ...

- We've tested both algorithms thoroughly, they both work
- Maybe we choose the "faster" one
- So we try both algorithms on a particular input...

Which is better?

In [102]:
```
import matplotlib.pyplot as plt; import numpy as np; x = np.linspace(2, 2, 1); y = .1 * x ** 2+1.5; y2 = 2.5*x; fig = plt.figure(figsize = (5, 5)); plt.axis([0, 4, 0, 6]); plt.plot(x, y, 'o'); plt.plot(x, y2, 'o',);
plt.annotate('Algorithm A', (x,y), textcoords="offset points", xytext=(5,5), ha='left'); plt.annotate('Algorithm B', (x,y2), textcoords="offset points", xytext=(5,5), ha='left')
plt.show()
```

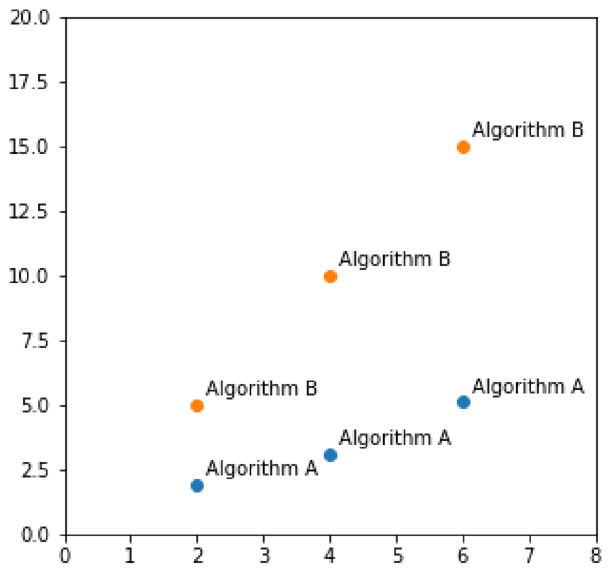


But what if that's just one case?

Need to see how the two algorithms fare on different input

- Notably, how does performance changes as a function of **input size**
- Running the algorithms again on a few more inputs...

In [103]:
```
import matplotlib.pyplot as plt; import numpy as np; x = np.linspace(2, 6, 3); y = .1 * x ** 2+1.5; y2 = 2.5*x; fig = plt.figure(figsize = (5, 5)); plt.axis([0, 8, 0, 20]); plt.plot(x, y, 'o'); plt.plot(x, y2, 'o',)
for a,b in zip(x,y): plt.annotate('Algorithm A', (a,b), textcoords="offset points", xytext=(5,5), ha='left')
for a,b in zip(x,y2): plt.annotate('Algorithm B', (a,b), textcoords="offset points", xytext=(5,5), ha='left')
plt.show()
```



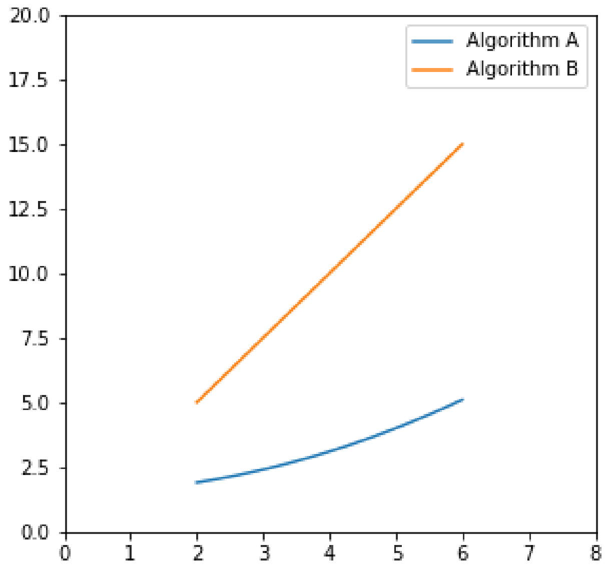
Connecting the dots...

Algorithm A is looking a lot faster

And the gap is widening...

Or is it?

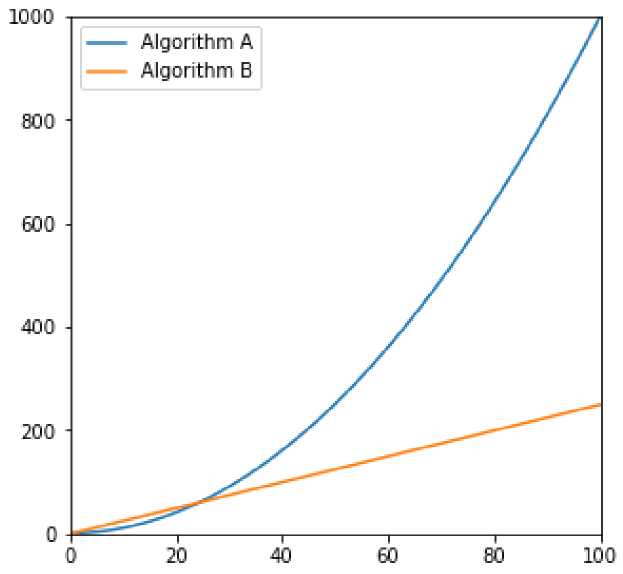
In [104]:
```
import matplotlib.pyplot as plt; import numpy as np
x = np.linspace(2, 6, 200); y = .1 * x ** 2+1.5; y2 = 2.5*x; fig = plt.fig
ure(figsize = (5, 5))
plt.axis([0, 8, 0, 20]); plt.plot(x, y); plt.plot(x, y2); plt.legend(["Alg
orithm A", "Algorithm B"]); plt.show()
```



What happens with *large* inputs?

Let's try some substantially larger inputs

In [105]:
```
import matplotlib.pyplot as plt; import numpy as np
x = np.linspace(0, 100, 1000); y = .1 * x ** 2+1.5; y2 = 2.5*x
fig = plt.figure(figsize = (5, 5)); plt.axis([0, 100, 0, 1000]); plt.plot(
x, y); plt.plot(x, y2); plt.legend(["Algorithm A", "Algorithm B"]); plt.sh
ow()
```



Performance on small inputs can be different than for large

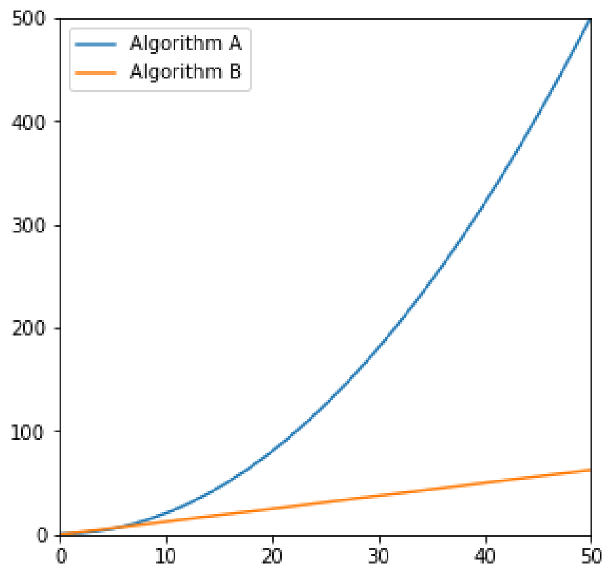
- Only really care about performance at scale (i.e., on large inputs)
- We want to be able to talk about performance as a function of input size

Faster Computer?

- We also want to be careful about how we quantify speed
- So far we've looked at seconds
    - But that depends on the computer
    - And what else is running on the computer?
- A faster computer will change the exact timings
    - But the shape of the curves remain the same

It's this shape that we care about

In [101]:
```
import matplotlib.pyplot as plt; import numpy as np
x = np.linspace(0, 50, 1000); y = .2 * x ** 2+.75; y2 = 1.25*x
fig = plt.figure(figsize = (5, 5)); plt.axis([0, 50, 0, 500]); plt.plot(x,
y); plt.plot(x, y2); plt.legend(["Algorithm A", "Algorithm B"]); plt.show
()
```



And what about space (i.e., memory)?

- We already know that space is a limited resource
- So we may also want to compare algorithms based on memory usage
- Like time measuring space is tricky
    - Want to think about the number of *objects* created rather than absolute memory usage
- Often a trade-off
    - E.g. Pre-compute values (e.g., Word scores in assignment 2)
        - Better time usage on subsequent look ups
        - But requires more space (to store all the pre-computed values)

Best Case, Worst Case, Average Case

Consider sorting a deck of cards by hand...

- If the deck is already sorted, that will be a lot faster than if it is all mixed up

When comparing algorithms, we also need to think about how they work and which situations (or kinds of input) will result in better or worse performance

- See bubble sort example

Want to understand efficiency

Challenges in understanding efficiency of solution to a computational problem:

- Many different **implementations**
- Only a handful of different **algorithms**
- Want to separate choice of implementation from choice of algorithm

Orders of Growth

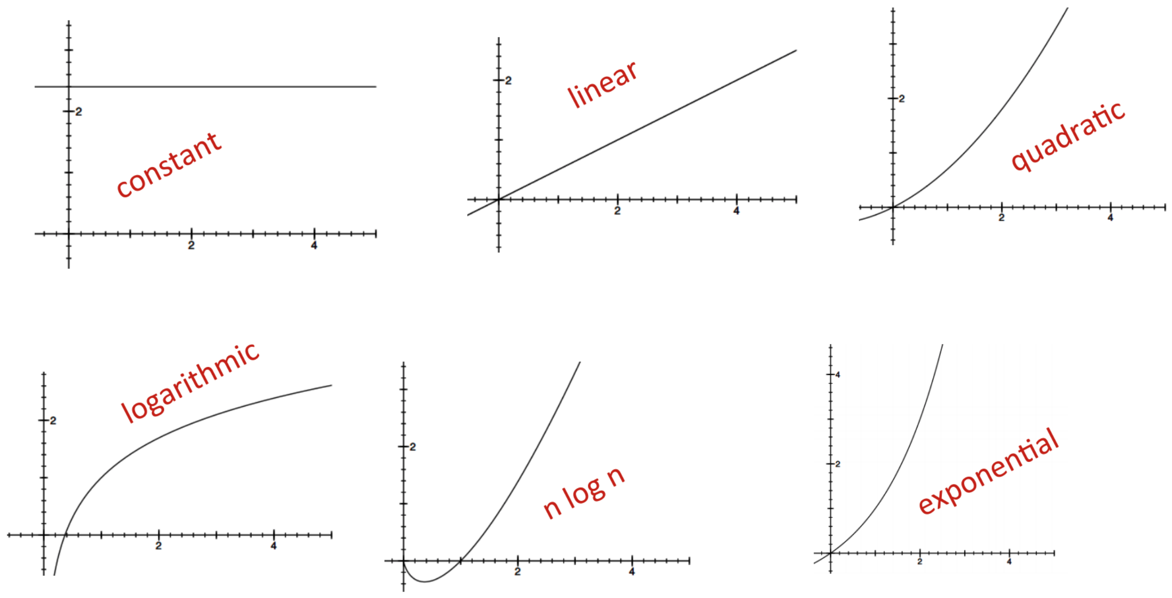
Goals:

- Evaluate efficiency when input is very **big**
- Express growth as a **function of input**
- Put an **upper bound** on growth — as tightly as possible
- Do **not** need to be **precise** 'order of' not 'exact' growth
- Care really about the largest factors (what parts take the longest)
- Generally care about worst case

Big Oh Notation

- An upper bound on asymptotic growth (order of growth)
- Big Oh or `O()` is used to describe the worst case
  - part of the program that causes the bottle neck
  - express rate of growth relative to the input size
  - evaluate the **algorithm** not the machine or the implementation

Types of orders of growth



Complexity Classes

- $O(1)$ → denotes constant running time
- $O(log(n))$ → denotes logarithmic running time
- $O(n)$ → denotes linear running time
- $O(nlog(n))$ → denotes log-linear running time
- $O(n^c)$ → denotes polynomial running time (c is a constant)
- $O(c^n)$ → denotes exponential running time (c is a constant)

Examples

```
In [84]: def printFirst(someList):
             print(someList[0])
```

In [88]:
```python
def printElements(someList):
    for i in someList:
        print(i)
```

In [94]:
```python
def printTriangle(size):
    for i in range(1, size+1):
        output = ""
        for j in range(i):
            output += "*"
        print(output)
```

Recap

- Focus on the big picture, only care about what will dominate at scale
- Watch for hidden complexity in built-in functions, E.g

```python
max([someList])
```

- Lots of resources to help with decision making
  - Your job is aprreciate the importance of considering complexity
  - Can look up complexity of common algorithms to guide decision making

Have a nice summer!