How can we measure the running time of algorithms?

- Idea: Use a stopwatch.
  - What if we run the algorithm on a different computer?
  - What if we code the algorithm in a different programming language?
  - Timing the algorithm doesn't (directly) tell us how it will perform in other cases besides the ones we test it on.

How can we measure the running time of algorithms?

- Idea: Count the number of "basic operations" in an algorithm.
  - "Basic operations" are things the computer can do "in a single step," like
    - Printing a single value (number or string)
    - Comparing two values
    - (simple) math, like adding, multiplying, powers
    - Assigning a variable a value

- How many basic operations are done in this algorithm?
  - Only count printing as a basic operation.

```
# assume L is a list of three numbers
for pos in range(0, 3):
    print(L[pos])

# assume L2 is a list of six numbers
for pos in range(0, 6):
    print(L2[pos])
```

- How many basic operations are done in this algorithm?
  - Only count printing as a basic operation.

```
# assume L is a list of numbers
for pos in range(0, len(L)):
    print(L[pos])
```

If n = len(L), what is a general formula for how long this algorithm takes, in terms of n?

- How many basic operations are done in this algorithm, ***in the worst possible case***?
  - Only count printing and comparing as a basic operations.

```
# assume L is a list of numbers
for pos in range(0, len(L)):
    if L[pos] > 10:
        print(L[pos])
```

If n = len(L), what is a general formula for how long this algorithm takes, in terms of n, in the worst case?

- Computer scientists often consider the running time for an algorithm in the worst case, since we know the algorithm will never be slower than that.
- We express the running time of an algorithm as a function in terms of “*n*,” which represents the size of the input to the algorithm.
- For an algorithm that processes a list, *n* is the length of the list.

```python
# Assume for both algorithms, var and n are
already defined as positive integers.

# algorithm A
var = var + n
print(var)

# algorithm B
for x in range(0, n):
   var = var + 1
print(var)
```

- We group running times together based on how they grow as *n* gets really big.
- If the running time stays exactly the same as n gets big, we say the running time is **constant**.
- If the running time grows proportionally to n, we say the running time is **linear in n**.
  - If the input size doubles, the running time roughly doubles.
  - If the input size triples, the running time roughly triples.

```
# algorithm A
var = var + n
print(var)
```

What class does algorithm A fall into?

```
# algorithm B
for x in range(0, n):
    var = var + 1
print(var)
```

What class does algorithm B fall into?

```
# algorithm C:
# assume L is a list of numbers
for pos in range(0, len(L)):
    print(L[pos])

# algorithm D:
# assume L is a list of numbers
for pos in range(0, len(L)):
    if L[pos] > 10:
        print(L[pos])
```

Classes have special names, which use big-O notation.

Constant time algorithm: O(1)

Read as “big-oh of 1” or “oh of 1”

Linear time algorithm: O(n)

Read as “big oh of n” or “oh of n”

These classes give us a rough estimate of how fast an algorithm runs, without worrying about details.

- How many basic operations are done in this algorithm?
  - Only count printing as a basic operation.

```
# assume M is a n by n matrix of numbers
for row in range(0, n):
    for col in range(0, n):
        print(M[row][col])
```

What is a general formula for how long this algorithm takes, in terms of n?

Common running times

- Algorithm which doesn't get slower as input size increases is O(1).
- Algorithm which grows proportionally to input size is O(n) [linear].
- Algorithm which grows proportionally to the square of the input size is $O(n^2)$ [quadratic].

Watch Phil Tear A Phone Book in Half

One million "basic" operations per second.

	O(log n)	O(n)	O(n^2)	O(2^n)
n = 10	0.003 ms			
N = 20	0.004 ms			
N = 40	0.005 ms			
N = 80	0.007 ms			
N = 1,000	0.009 ms			
N = 10,000	0.013 ms			

One million "basic" operations per second.

	O(log n)	O(n)	O(n^2)	O(2^n)
n = 10	0.003 ms	0.01 ms		
N = 20	0.004 ms	0.02 ms		
N = 40	0.005 ms	0.04 ms		
N = 80	0.007 ms	0.08 ms		
N = 1,000	0.009 ms	1 ms		
N = 10,000	0.013 ms	10 ms		

One million "basic" operations per second.

	O(log n)	O(n)	O(n^2)	O(2^n)
n = 10	0.003 ms	0.01 ms	0.1 ms	
N = 20	0.004 ms	0.02 ms	0.4 ms	
N = 40	0.005 ms	0.04 ms	1.6 ms	
N = 80	0.007 ms	0.08 ms	6.4 ms	
N = 1,000	0.009 ms	1 ms	1 second	
N = 10,000	0.013 ms	10 ms	100 seconds	

One million “basic” operations per second.

	O(log n)	O(n)	O(n^2)	O(2^n)
n = 10	0.003 ms	0.01 ms	0.1 ms	1 ms
N = 20	0.004 ms	0.02 ms	0.4 ms	1 sec
N = 40	0.005 ms	0.04 ms	1.6 ms	305 hours
N = 80	0.007 ms	0.08 ms	6.4 ms	3.81 x 10^10 years
N = 1,000	0.009 ms	1 ms	1 second	----
N = 10,000	0.013 ms	10 ms	100 seconds	----