Running time of algorithms

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])
```

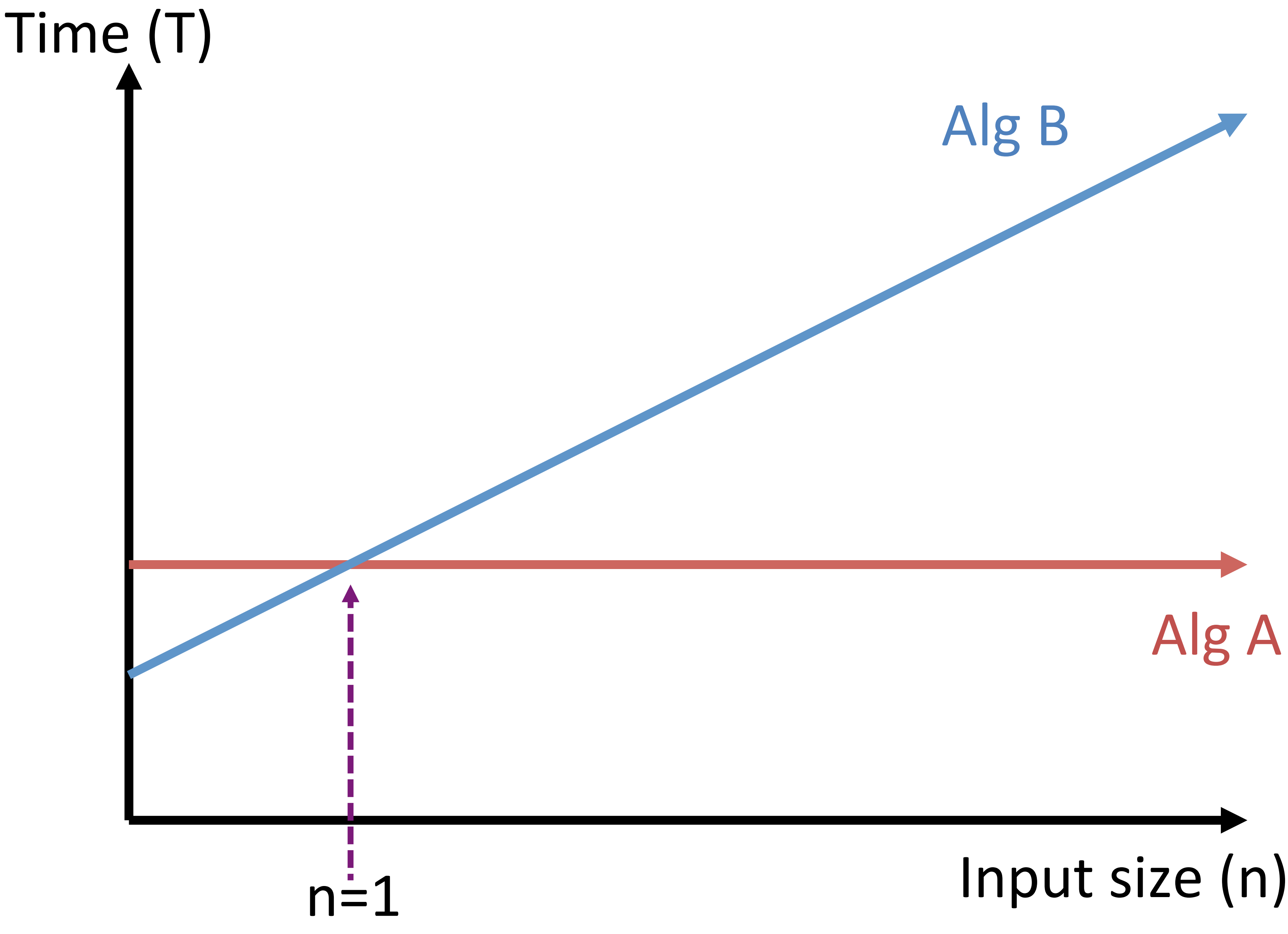
If n = len(L), what is a general formula for how long this algorithm takes, in terms of n, in the worst case?

- 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.

```
# 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)
```


Time (T)
Alg B
Alg A
n=1
Input size (n)

- 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 (n has no effect on the algorithm's speed), we say the running time is **constant**.
- If the running time grows proportionally to n, we say the running time is **linear**.
  - 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)
```

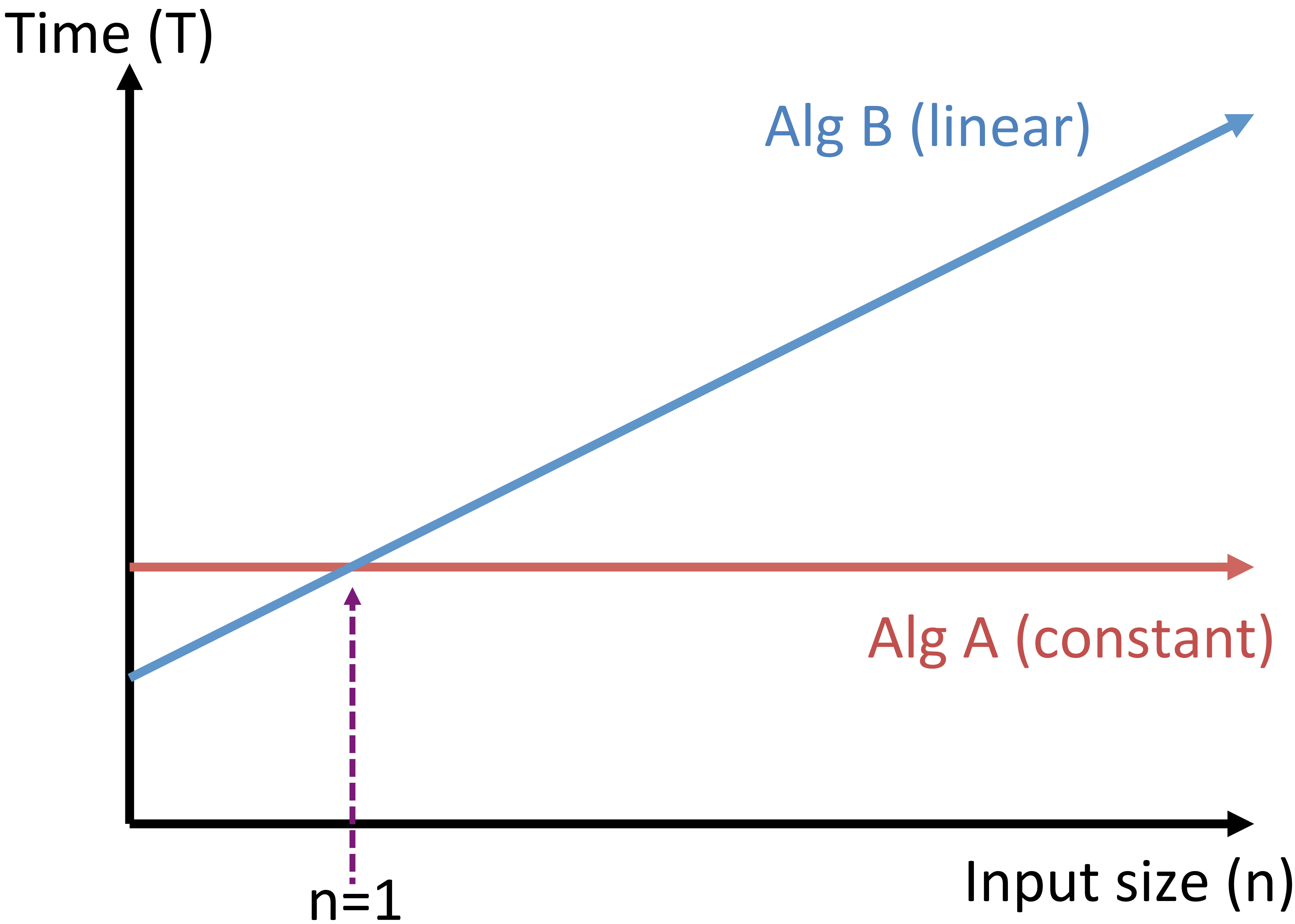
What class does algorithm A fall into? [constant or linear]

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

What class does algorithm B fall into? [constant or linear]

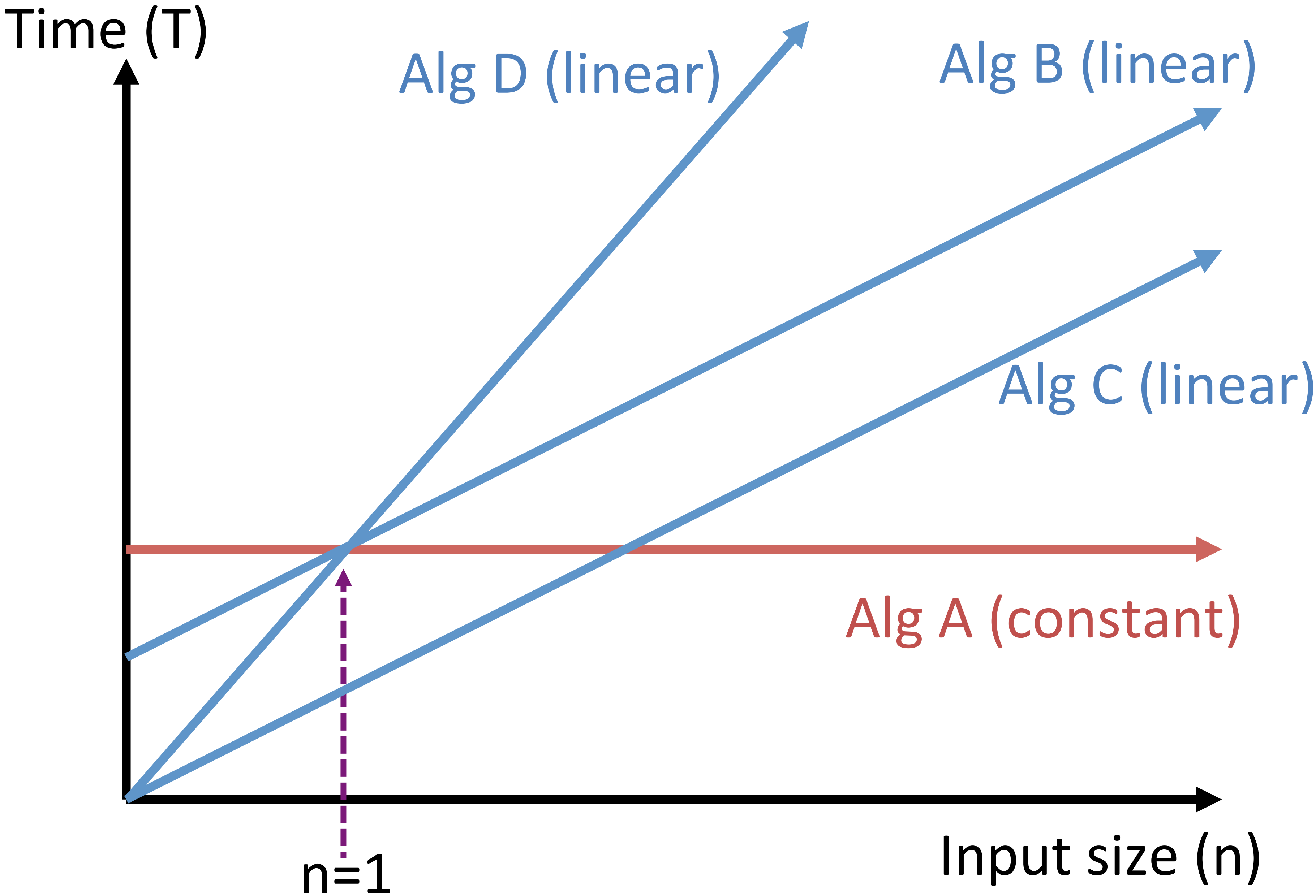
Which is "better?"

- In general, prefer algorithms that run faster.
  - That is, take less time for bigger and bigger input sizes.
- Therefore, an algorithm that runs in constant time is "generally" preferred over a linear-time algorithm.


Time (T)
Alg B (linear)
Alg A (constant)
n=1
Input size (n)

```
# algorithm C:
# assume L has n numbers in it
for pos in range(0, len(L)):
    print(L[pos])

# algorithm D:
# assume L has n numbers in it
for pos in range(0, len(L)):
    if L[pos] > 10:
        print(L[pos])
```


Time (T)
Alg D (linear)
Alg B (linear)
Alg C (linear)
Alg A (constant)
n=1
Input size (n)

- 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?

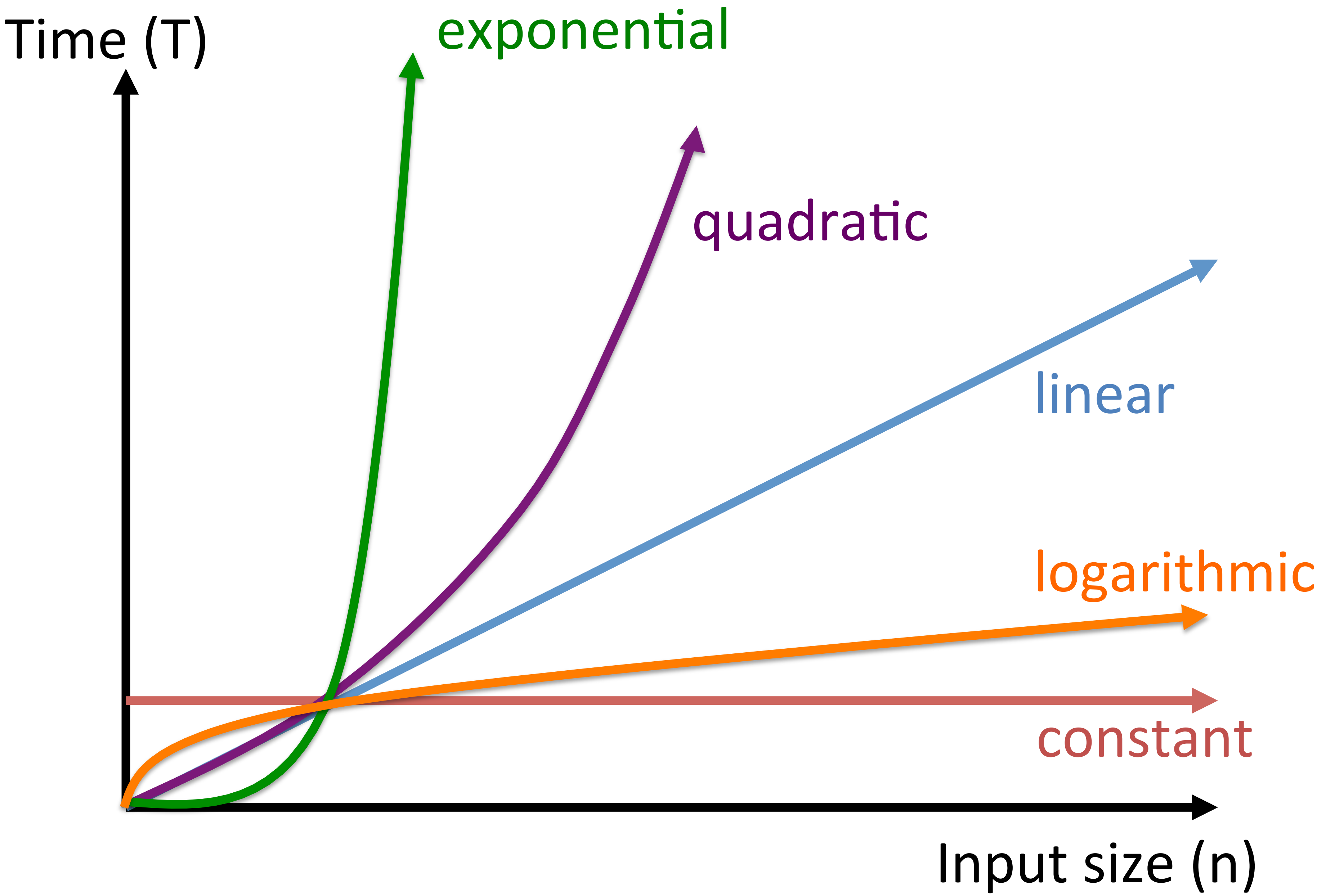
Common running times

- Algorithm which doesn't get slower as input size increases is a **constant-time** algorithm.
- Algorithm which grows proportionally to input size a **linear-time** algorithm.
- Algorithm which grows proportionally to the square of the input size is a **quadratic-time** algorithm.

Watch Phil Tear A Phone Book in Half

- If a list is sorted, you can use the binary search algorithm to find the position of an element in the list.
  - Takes logarithmic time.
- If a list is not sorted, you can't use binary search; you have to use sequential search.
  - Takes linear time.

- Some problems have algorithms that run even more slowly than quadratic time.
  - Cubic time (n^3), higher polynomials, …
  - Exponential time (2^n) is even slower!
- In some cases, we ***depend*** on the fact that we don't have fast algorithms to solve problems.


Time (T)
exponential
quadratic
linear
logarithmic
constant
Input size (n)

One million “basic” operations per second.

	log.	linear	quadratic	expo.
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.

	log.	linear	quadratic	expo.
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.

	log.	linear	quadratic	expo.
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.

	log.	linear	quadratic	expo.
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	----