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 vec is a vector of three ints
for (int x = 0; x < 3; x++)
   cout << vec[x];

# assume vec2 is a vector of six ints
for (int x = 0; x < 6; x++)
   cout << vec[x];
```

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

```
# assume vec is a vector of ints
for (int x = 0; x < vec.size(); x++)
   cout << vec[x];
```

If n = vec.size(), 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 as a basic operation.

```
# assume vec is a vector of ints
for (int x = 0; x < vec.size(); x++)
  if (vec[x] > 10)
    cout << vec[x];
```

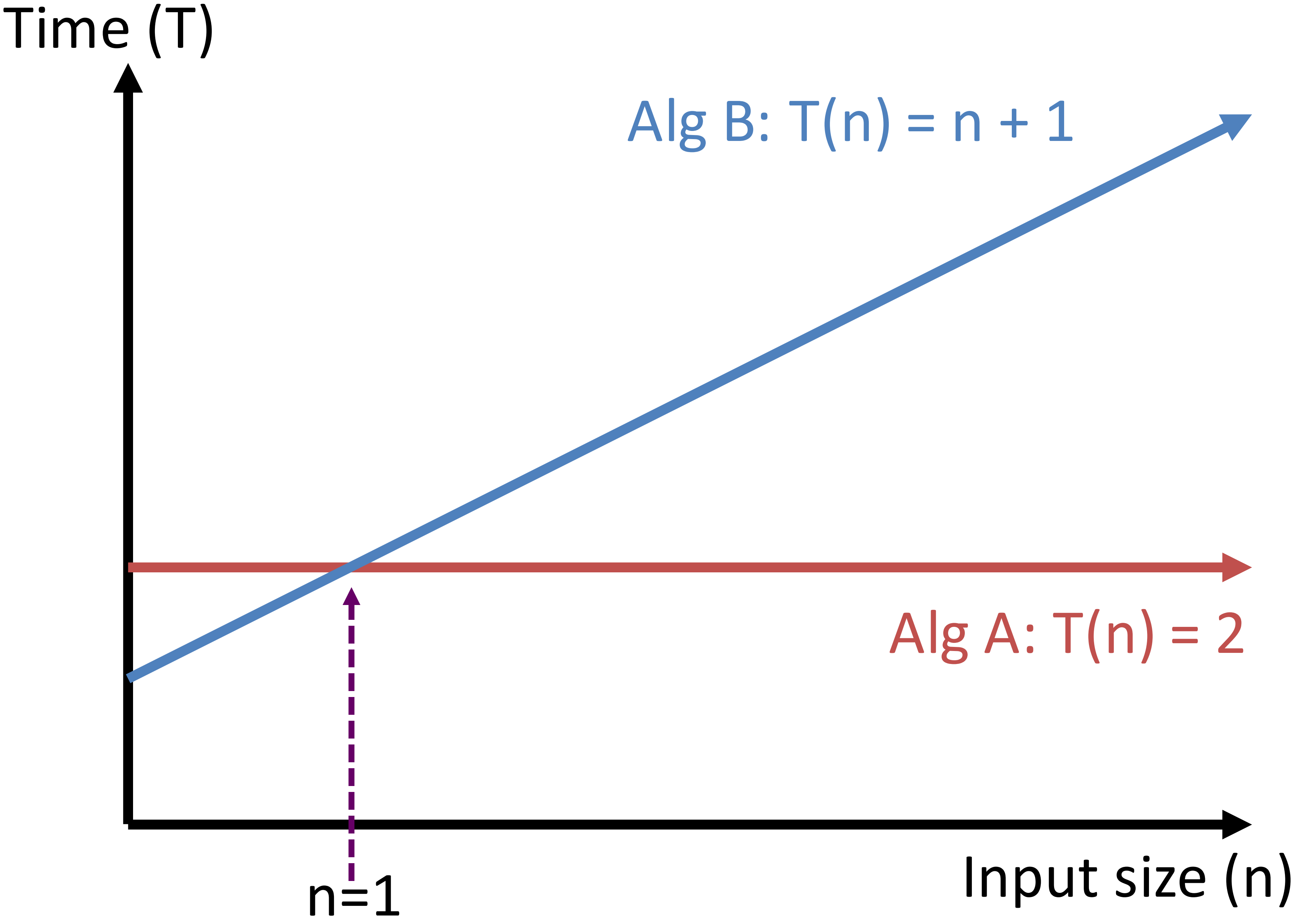
If n = len(L), what is a general formula for how long this algorithm takes, in terms of n, in the worst case?

- 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.
  - Sometimes we also care about ***average*** running time.
- 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.
   Basic ops are printing and adding. */

// algorithm A
var = var + n;
cout << var << endl;

// algorithm B
for (int x = 0; x < n; x++)
   var++;
cout << var << endl;
```


Time (T)
Alg B: T(n) = n + 1
Alg A: T(n) = 2
n=1
Input size (n)

Suppose we count comparisons:

```
double largest = vec[0];
for (int x = 0; x < vec.size(); x++)
{
  if (vec[x] > largest)   ← how many times?
    largest = vec[x]
}
```

Suppose we count comparisons:

```
double largest = -99999;
for (int x = 0; x < open.size(); x++)
{
  for (int y = 0; y < close.size(); y++)
  {
    if (close[y] – open[x] > largest)
      largest = close[y] – open[x]
  }
}
```

Suppose we count comparisons:

```
double largest = -99999;
for (int x = 0; x < open.size(); x++)
{
  for (int y = x; y < close.size(); y++)
  {
    if (close[y] – open[x] > largest)
      largest = close[y] – open[x]
  }
}
```

- 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;
cout << var << endl;
```

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

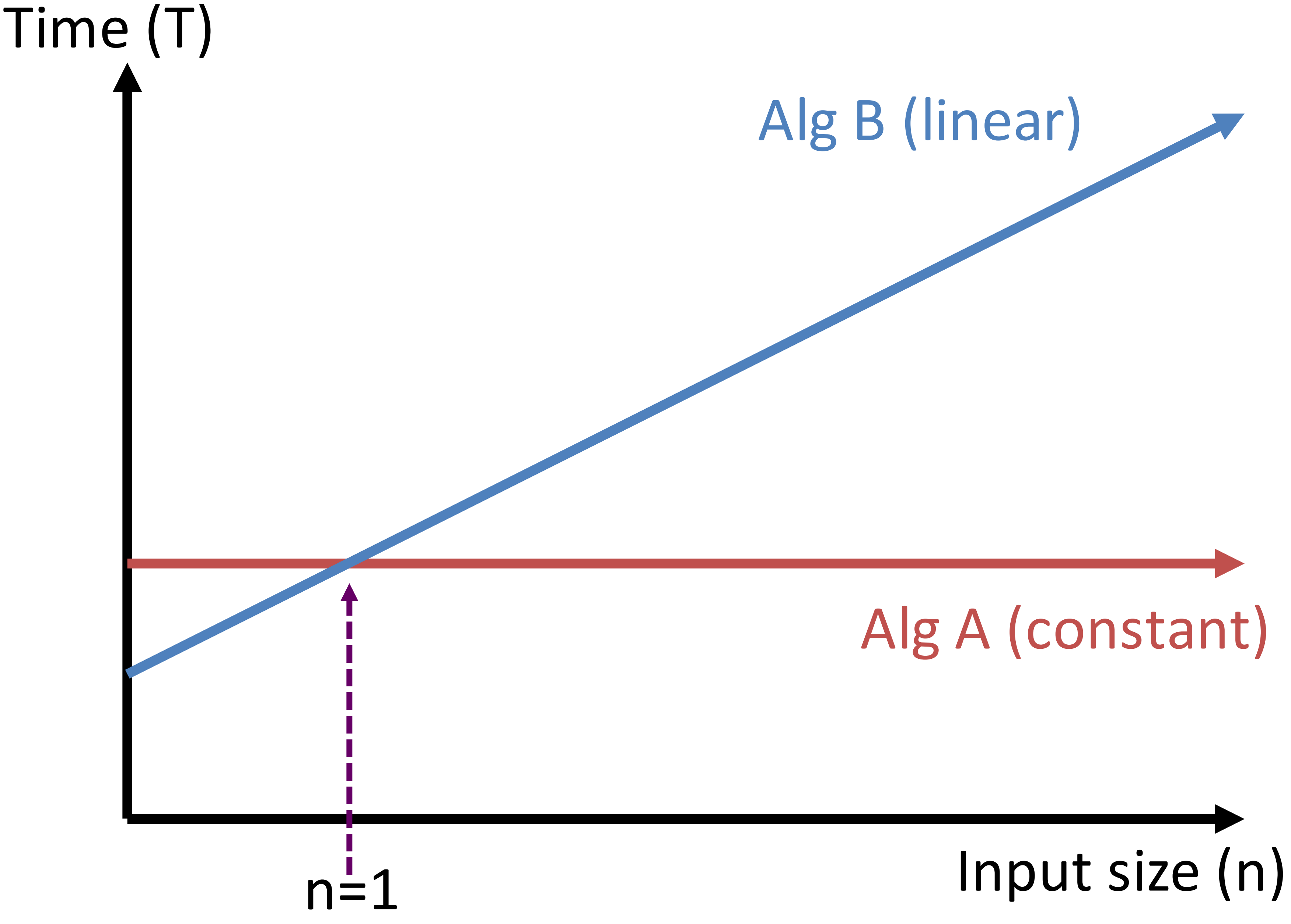
```
# algorithm B
for (int x = 0; x < n; x++)
   var++;
cout << var << endl;
```

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

Which is "better?"

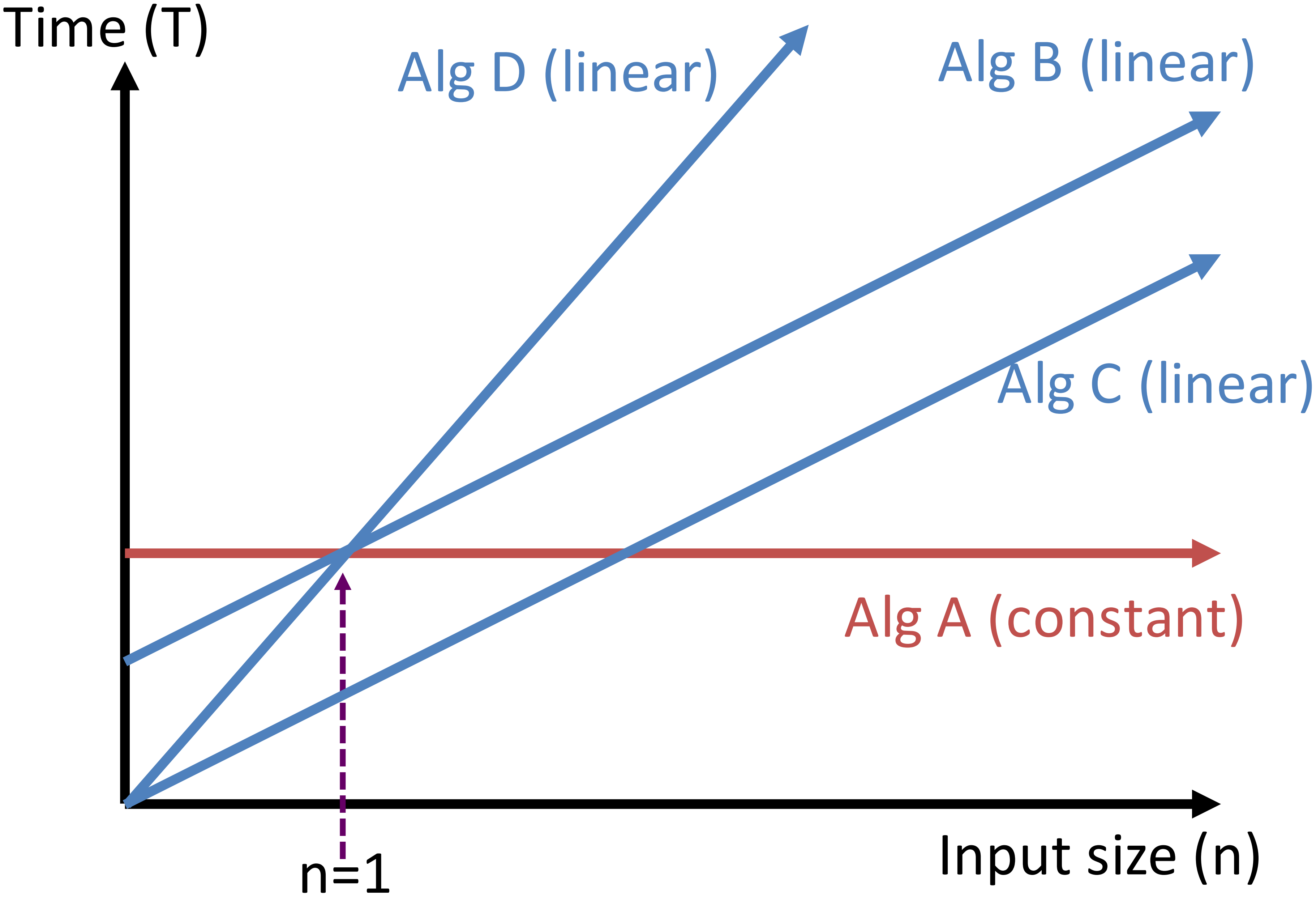
- In general, we prefer algorithms that run faster.
  - That is, as the algorithm's input size grows, the time required to run the algorithm should grow as slowly as possible.

- 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 ints in it
for (int x = 0; x < vec.size(); x++)
  cout << vec[x];

# algorithm D:
# assume vec has n ints in it
for (int x = 0; x < vec.size(); x++)
  if (vec[x] > 10)
    cout << vec[x];
```


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

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 (int x =
   for col in range(0, n):
      print(M[row][col])
```

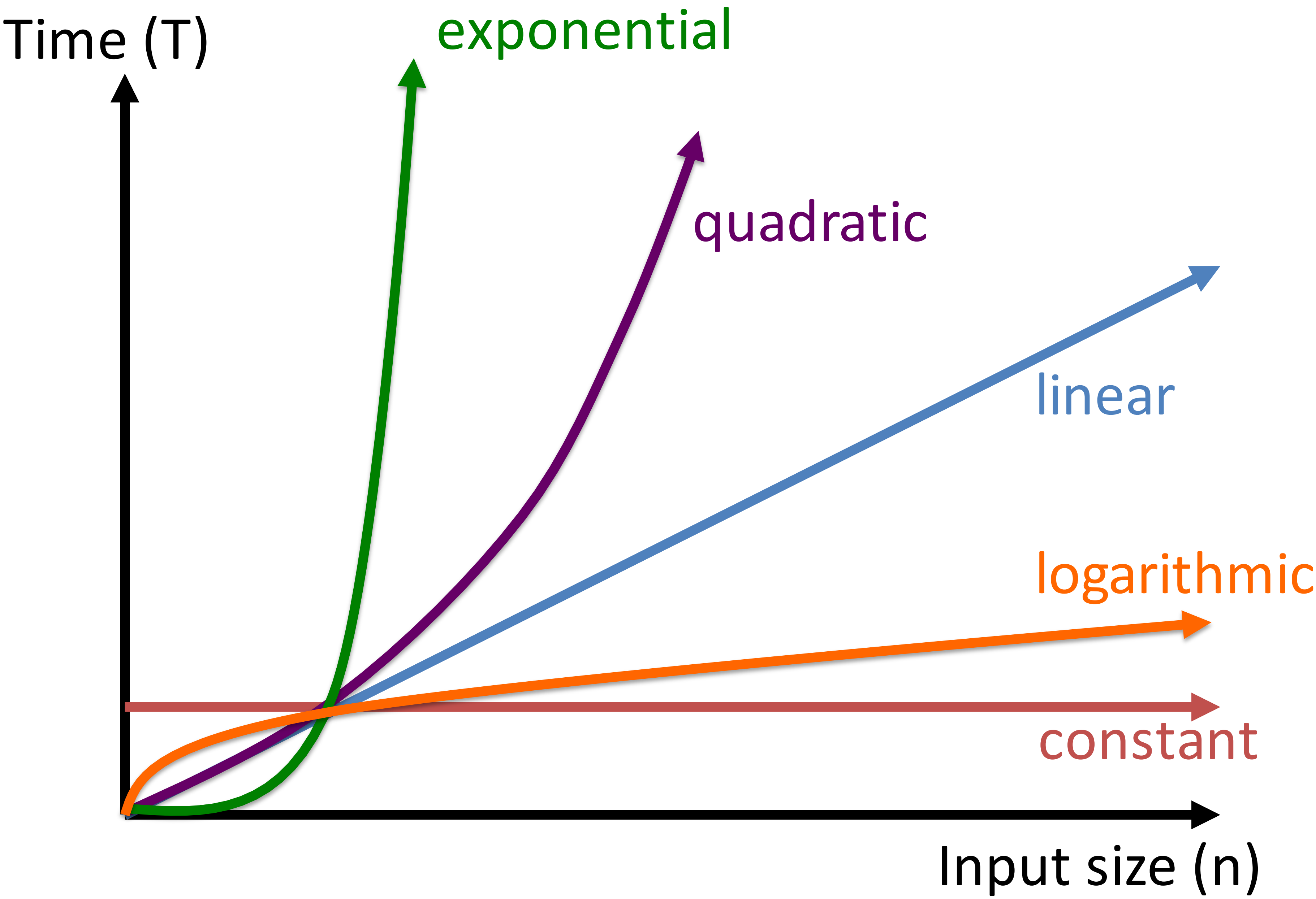
What is a general formula for how long this algorithm takes, in terms of n?

- Algorithm which doesn’t get slower as input size increases is a **constant-time** algorithm.
- Algorithm whose running time grows proportionally to input size is a **linear-time** algorithm.
- Algorithm whose running time grows proportionally to the square of the input size is a **quadratic-time** algorithm.
  - $O(n^2)$

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 situations, we ***depend*** on the fact that we don't have fast algorithms to solve problems.
  - Usually security situations involving breaking codes.


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

One million “basic” operations per second.

	logarithmic	linear	quadratic	exponential
n = 10				
n = 20				
n = 30				
n = 50				
n = 100				
n = 1,000				
n = 10,000				
n = 100,000				
n = 1,000,000				

One million "basic" operations per second.

	logarithmic	linear	quadratic	exponential
n = 10	0.0033 ms			
n = 20	0.0043 ms			
n = 30	0.0049 ms			
n = 50	0.0056 ms			
n = 100	0.0066 ms			
n = 1,000	0.0099 ms			
n = 10,000	0.0133 ms			
n = 100,000	0.0166 ms			
n = 1,000,000	0.0199 ms			

One million “basic” operations per second.

	logarithmic	linear	quadratic	exponential
n = 10	0.0033 ms	0.01 ms		
n = 20	0.0043 ms	0.02 ms		
n = 30	0.0049 ms	0.03 ms		
n = 50	0.0056 ms	0.05 ms		
n = 100	0.0066 ms	0.1 ms		
n = 1,000	0.0099 ms	1 ms		
n = 10,000	0.0133 ms	10 ms		
n = 100,000	0.0166 ms	0.1 sec		
n = 1,000,000	0.0199 ms	1 sec		

One million “basic” operations per second.

	logarithmic	linear	quadratic	exponential
n = 10	0.0033 ms	0.01 ms	0.1 ms	
n = 20	0.0043 ms	0.02 ms	0.4 ms	
n = 30	0.0049 ms	0.03 ms	0.9 ms	
n = 50	0.0056 ms	0.05 ms	2.5 ms	
n = 100	0.0066 ms	0.1 ms	0.01 sec	
n = 1,000	0.0099 ms	1 ms	1 sec	
n = 10,000	0.0133 ms	10 ms	1.67 min	
n = 100,000	0.0166 ms	0.1 sec	2.77 hours	
n = 1,000,000	0.0199 ms	1 sec	11.57 days	

One million "basic" operations per second.

	logarithmic	linear	quadratic	exponential
n = 10	0.0033 ms	0.01 ms	0.1 ms	1.024 ms
n = 20	0.0043 ms	0.02 ms	0.4 ms	
n = 30	0.0049 ms	0.03 ms	0.9 ms	
n = 50	0.0056 ms	0.05 ms	2.5 ms	
n = 100	0.0066 ms	0.1 ms	0.01 sec	
n = 1,000	0.0099 ms	1 ms	1 sec	
n = 10,000	0.0133 ms	10 ms	1.67 min	
n = 100,000	0.0166 ms	0.1 sec	2.77 hours	
n = 1,000,000	0.0199 ms	1 sec	11.57 days	

One million "basic" operations per second.

	logarithmic	linear	quadratic	exponential
n = 10	0.0033 ms	0.01 ms	0.1 ms	1.024 ms
n = 20	0.0043 ms	0.02 ms	0.4 ms	1.049 sec
n = 30	0.0049 ms	0.03 ms	0.9 ms	
n = 50	0.0056 ms	0.05 ms	2.5 ms	
n = 100	0.0066 ms	0.1 ms	0.01 sec	
n = 1,000	0.0099 ms	1 ms	1 sec	
n = 10,000	0.0133 ms	10 ms	1.67 min	
n = 100,000	0.0166 ms	0.1 sec	2.77 hours	
n = 1,000,000	0.0199 ms	1 sec	11.57 days	

One million "basic" operations per second.

	logarithmic	linear	quadratic	exponential
n = 10	0.0033 ms	0.01 ms	0.1 ms	1.024 ms
n = 20	0.0043 ms	0.02 ms	0.4 ms	1.049 sec
n = 30	0.0049 ms	0.03 ms	0.9 ms	17.9 min
n = 50	0.0056 ms	0.05 ms	2.5 ms	
n = 100	0.0066 ms	0.1 ms	0.01 sec	
n = 1,000	0.0099 ms	1 ms	1 sec	
n = 10,000	0.0133 ms	10 ms	1.67 min	
n = 100,000	0.0166 ms	0.1 sec	2.77 hours	
n = 1,000,000	0.0199 ms	1 sec	11.57 days	

One million "basic" operations per second.

	logarithmic	linear	quadratic	exponential
n = 10	0.0033 ms	0.01 ms	0.1 ms	1.024 ms
n = 20	0.0043 ms	0.02 ms	0.4 ms	1.049 sec
n = 30	0.0049 ms	0.03 ms	0.9 ms	17.9 min
n = 50	0.0056 ms	0.05 ms	2.5 ms	35.7 years
n = 100	0.0066 ms	0.1 ms	0.01 sec	
n = 1,000	0.0099 ms	1 ms	1 sec	
n = 10,000	0.0133 ms	10 ms	1.67 min	
n = 100,000	0.0166 ms	0.1 sec	2.77 hours	
n = 1,000,000	0.0199 ms	1 sec	11.57 days	

One million "basic" operations per second.

	logarithmic	linear	quadratic	exponential
n = 10	0.0033 ms	0.01 ms	0.1 ms	1.024 ms
n = 20	0.0043 ms	0.02 ms	0.4 ms	1.049 sec
n = 30	0.0049 ms	0.03 ms	0.9 ms	17.9 min
n = 50	0.0056 ms	0.05 ms	2.5 ms	35.7 years
n = 100	0.0066 ms	0.1 ms	0.01 sec	4×10^{16} years
n = 1,000	0.0099 ms	1 ms	1 sec	3×10^{287} years
n = 10,000	0.0133 ms	10 ms	1.67 min	----
n = 100,000	0.0166 ms	0.1 sec	2.77 hours	----
n = 1,000,000	0.0199 ms	1 sec	11.57 days	----