Algorithmic Complexity
We looked at the cost of an algorithm by counting how many times lines are executed. We can see from this process that some lines in an algorithm have a fixed number of executions, regardless of the input. For example, in this algorithm, the index = -1 will only ever execute once. But, the number of times that the for loop executes depends on the size of A.

Pseudocode	**Worst-case cost***
`findItem(A, v)`	
`  index = -1`	1
`  for i=1 to A.length`	A.length + 1
`    if A[i] == v`	A.length
`      index = i`	A.length
`  return index`	1

There are also simplifications here. The actual cost of these computations varies by computer and programming language. A faster computer will have a lower cost, and functionality can be 1 line of code in 1 language and 10 lines of code in another. Ultimately, what we care about when we talk about the cost of an algorithm is what contributes the most to its runtime.

In the *findItem* algorithm, when A.length is small, e.g. 10, then the constants are a significant contributor to the cost. However, as A.length grows, they become less significant. For example, if A.length is 10,000, then 10,000 + 3 is not that different than 10,000.

Asymptotic Analysis (bounds on running time or memory)
Suppose an algorithm for processing a retail store's inventory takes 10,000 milliseconds to read the initial inventory from disk, and then 10 milliseconds to process each transaction (items acquired or sold). Processing *n* transactions takes (10,000 + 10 n) ms.

Even though 10,000 > 10, the "10 n" term will be more important if the number of transactions is very large. The values 10,000 and 10 are coefficients that will change if we buy a faster computer or disk drive, or use a different language or compiler. We want a way to express the theoretical speed of an algorithm that is independent of a specific implementation on a specific machine – specifically, we want to ignore constant factors (which get smaller and smaller as technology improves).

Big-Oh Notation (upper bounds on a function's growth)
We use what's called Big-Oh notation to compare how quickly two functions grow as *n -> infinity*.

Let *n* be the size of a program's (algorithm's) input. (The input is in bits, numbers, words, or any data type.).

Let *T(n)* be a function that represents the algorithm's precise running time in milliseconds, given an input of size *n*. This includes the specific instructions and the actual runtime of each instruction.

Let *f(n)* be a simple growth function such as *f(n) = n,* a function that grows at a rate of *n*.

We can express the growth rate of *T(n)* by relating it to another growth function. If *T(n)* grows no faster than *f(n)*, then we say:

T(n) is in O(f(n))
or
T(n) is in O(n)

If and only if

T(n) <= c f(n)

Whenever *n* is big, for some large constant *c*.

That doesn't sound very specific, to use "big" and "large".
How big is "big" for *n*?

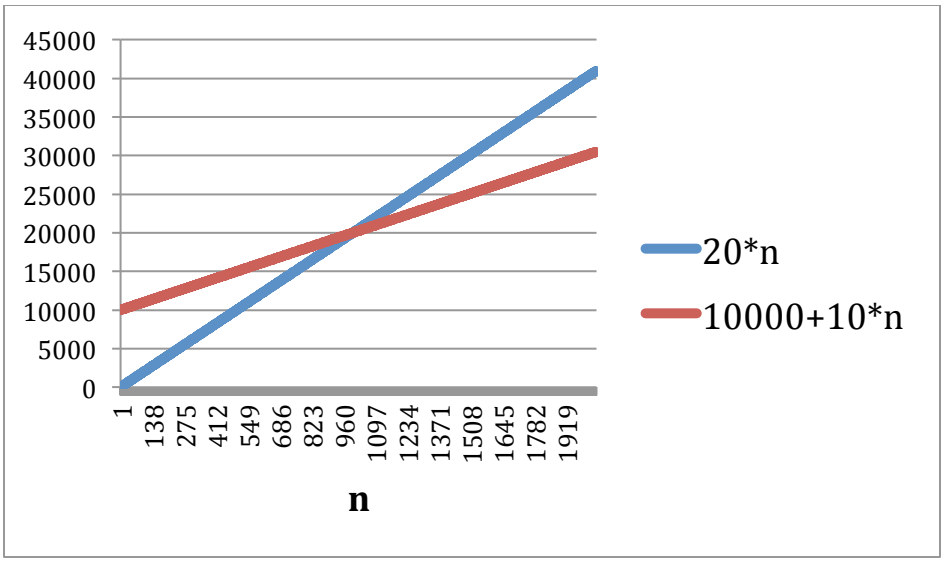
Big enough to make *T(n)* fit under *c f(n)* curve.

How large is "large" for c?

Large enough to make *T(n)* fit under *c f(n)* curve.

Inventory Example:
Given the function T(n) = 10,000 + 10n from the inventory example above. We can choose c to be as large as we want. We want to make T(n) fit underneath c f(n), so pick c = 20. We get something that looks like the following graph. This is the asymptotic behavior showing that 20*n will grow faster than 10,000*10n, so above a certain value of n, the size of the data is a bigger contributor to runtime than the startup cost.

Another way to think of it is that you're trying to prove that one function is asymptotically bounded by another [T(n) is in O(f(n))]. Since you're concerned with behavior as n -> infinity, you're allowed to multiply the functions by positive constants, as this doesn't affect the asymptotic behavior. You can also move the vertical line (N) as far to the right as you like, N is the value where the function values cross. For the inventory example, using c=20, N = 1000.



What this means, practically speaking
Returning to the initial example of the 10,000ms startup and the transaction cost of 10. If there are only a few transactions, then the 10,000ms startup might not be worth it and a less-efficient algorithm could be a better choice. However, with many, many transactions, the startup cost becomes less of a contributor to the overall cost.

Algorithmic Complexity
Big-oh notation provides a theoretical upper bound on an algorithm's growth rate. This is also referred to as the complexity of the algorithm.
You can go from the cost to the complexity by applying the following rules:

- If T(n) includes multiple terms, keep the term with the largest growth rate, and discard the others. For example, if $T(n) = 5n^3 + 3n^2 + n + 5$, the $5n^3$ has the largest growth rate and will dominate the other terms as n grows sufficiently large. The $3n^2$ and n terms can be discarded.
- Any constants in $T(n)$ can be omitted. For example, $5n^3$ has a constant of 5 that can be omitted, leaving n^3.

Therefore, $T(n) = 5n^3 + 3n^2 + n + 5$ is in $O(n^3)$.

In the *findItem* algorithm, $n = A.length$, and $T(n) = 3n + 3$. The dominant term is $3n$, and after removing the 3, we're left with $T(n)$ is in $O(n)$.

(Note: There are O(1) operations, which means that the operation is independent of the input size, but not that it's only 1 operation.)