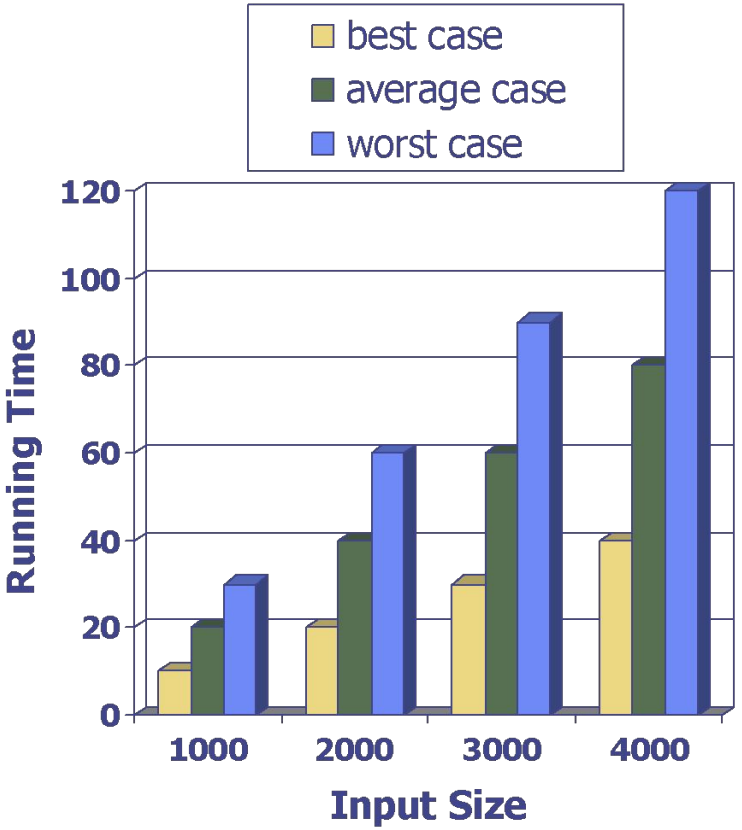
CSE 203: Asymptotic Analysis of Algorithms

Dr. Mohammed Eunus Ali
Professor
CSE, BUET

Some Slides from Tamassia & Goodrich

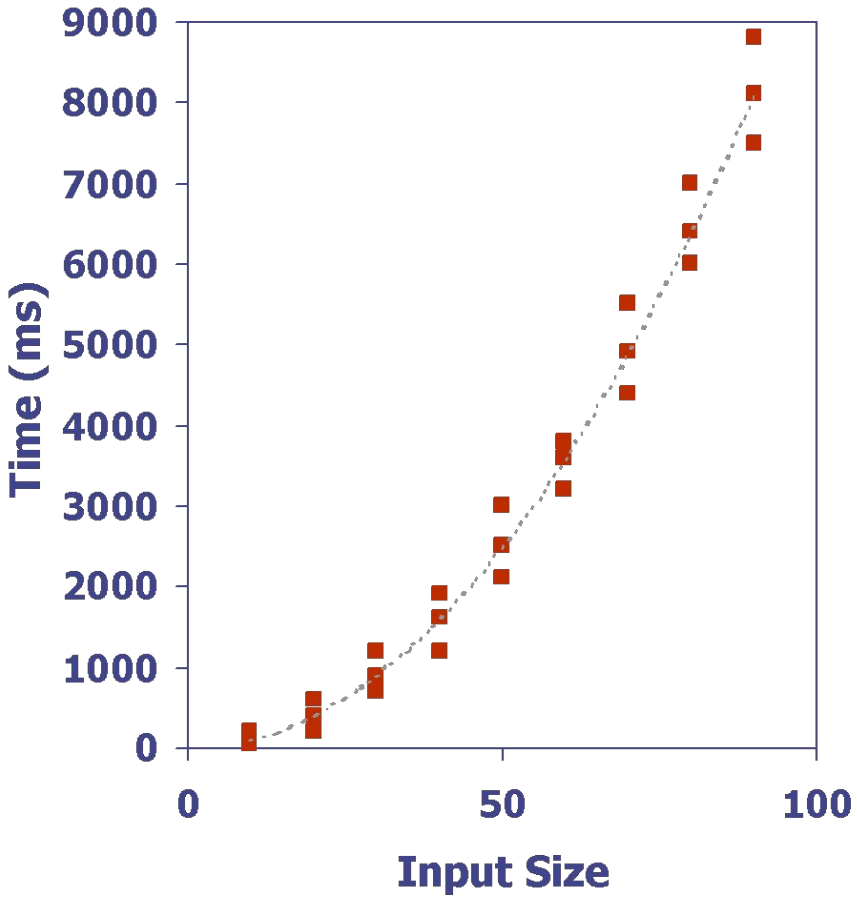
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

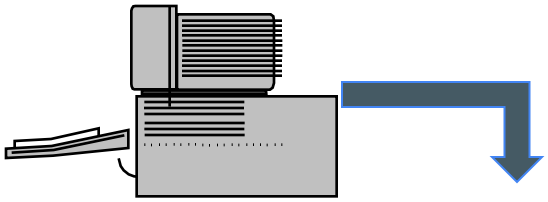
- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

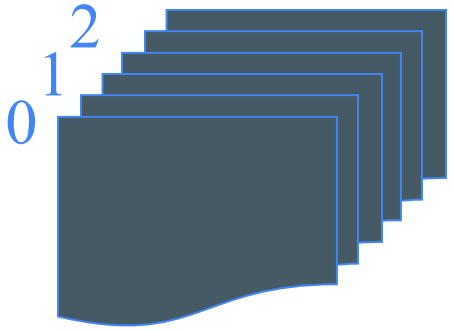
```
Algorithm arrayMax(A, n)
    Input array A of n integers
    Output maximum element of A

    currentMax ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > currentMax
then
            currentMax ← A[i]
    return currentMax
```

The Random Access Memory (RAM) Model

- A **CPU**



- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character



- Memory cells are numbered and accessing any cell in memory takes unit time.

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

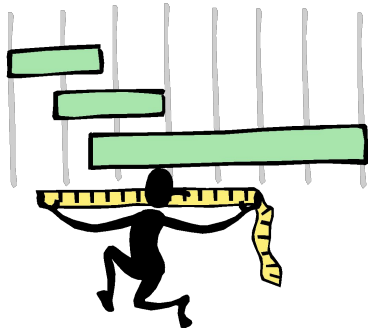
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm *arrayMax* (A, n)	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2n$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return *currentMax*	1
Total	$8n - 2$

Estimating Running Time



- Algorithm *arrayMax* executes $8n - 2$ primitive operations in the worst case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of *arrayMax*. Then
  $a(8n - 2) \leq T(n) \leq b(8n - 2)$
- Hence, the running time $T(n)$ is bounded by two linear functions

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***

Big Idea

Asymptotic (for large input size) Analysis

Algorithmic Analysis Roadmap


```
for (i = 0; i < n;
i++) {
  a[i] = 1;
  b[i] = 2;
}
```

f(n) = 2n

O(n)

- **Algorithmic Analysis**: The overall process of characterizing code with a complexity class, consisting of:
  - **Code Modeling**: Code → Function describing code's runtime
  - **Asymptotic Analysis**: Function → Complexity class describing asymptotic behavior

Asymptotic Analysis

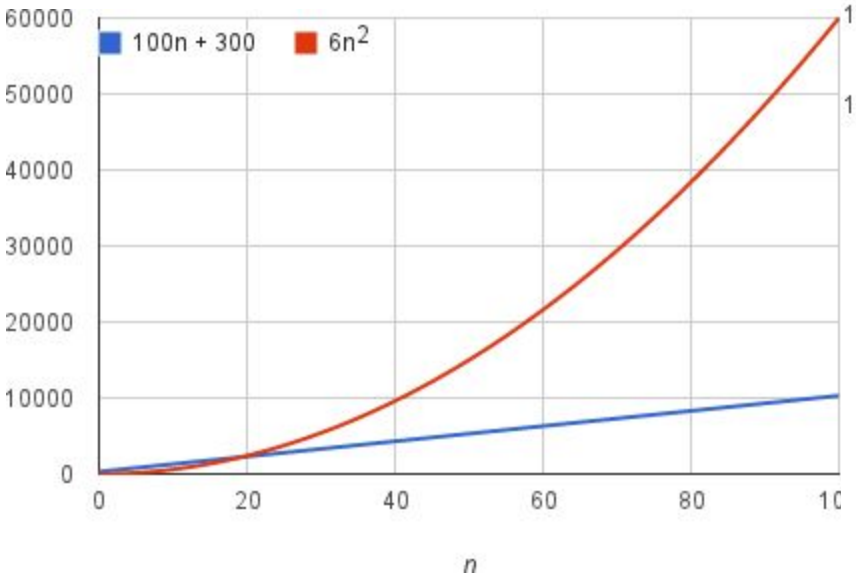
- First, we need to determine how long the algorithm takes, in terms of the size of its input.
  - the running me of the algorithm as a function of the size of its input.

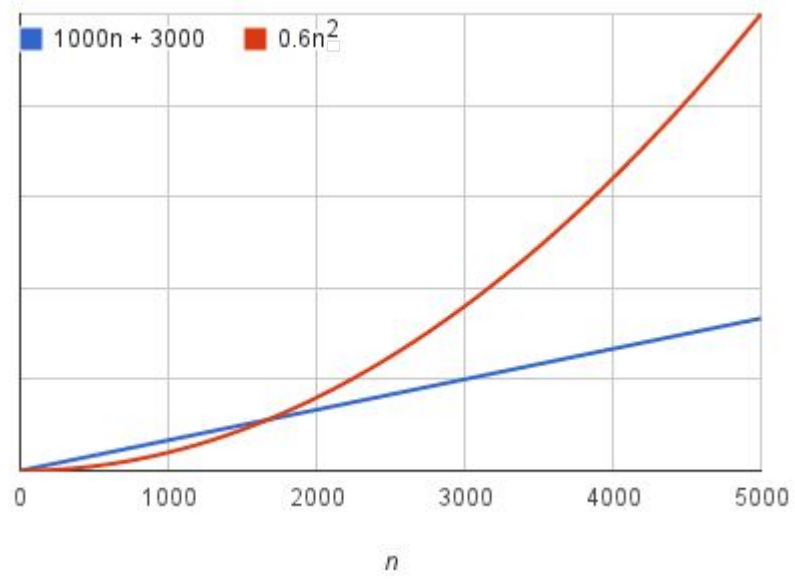
- The second idea is that we must focus on how fast a function grows with the input size.
  - We call this the rate of growth of the running me.

- By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running me—its rate of growth—without getting mired in details that complicate our understanding.
  - When we drop the constant coefficients and the less significant terms,

Examples

- f(n) = 6n^2 + 100n + 300
- Removing Lower order terms and constants





Asymptotic Notations: Big Oh, and Omega, and Theta

- Big-Oh is an **upper bound**
  - My code takes at most this long to run
- Big-Omega is a **lower bound**
  - My code takes at least this long to run
- Big Theta is **"equal to (tight bound)"**
  - My code takes "exactly"* this long to run
  - *Except for constant factors and lower order terms
  - Only exists when Big-Oh == Big-Omega!

Big-Oh

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \leq c \cdot g(n)$$

Big-Omega

$f(n)$ is $\Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$,

$$f(n) \geq c \cdot g(n)$$

Big-Theta

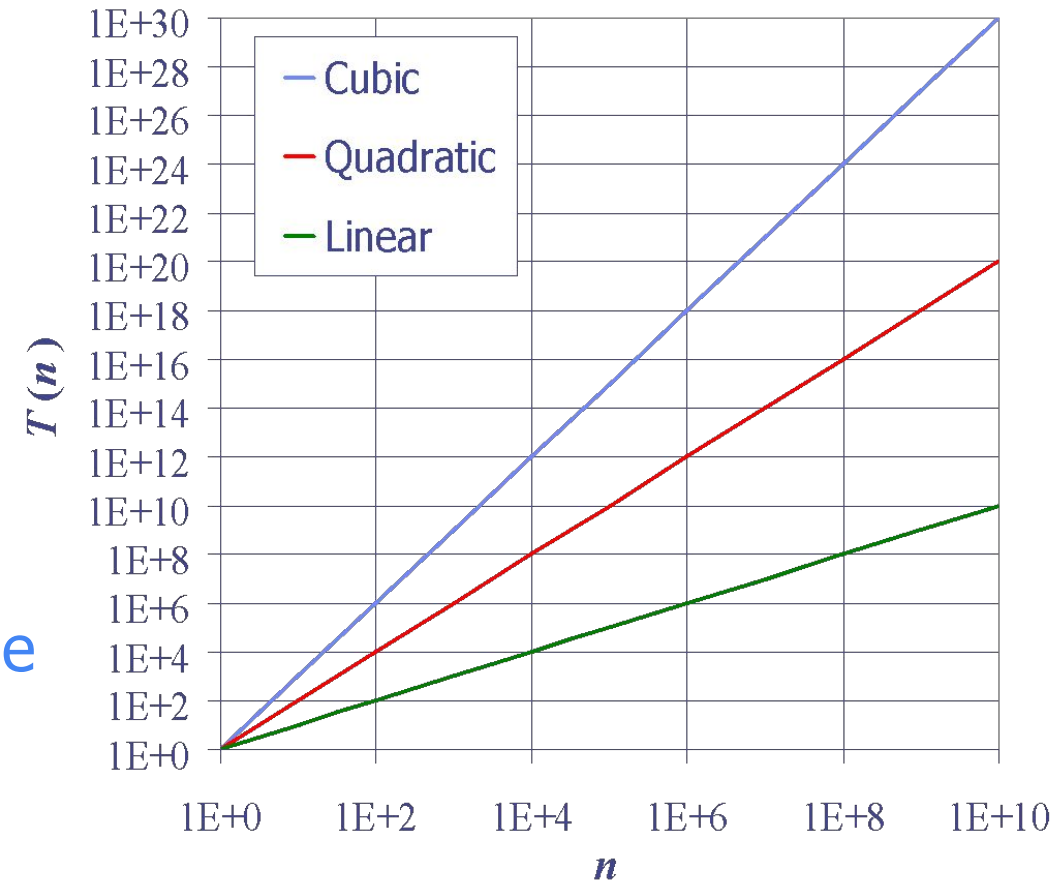
$f(n)$ is $\Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
(in other words: there exist positive constants $c1, c2, n_0$ such that for all $n \geq n_0$)

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Examples

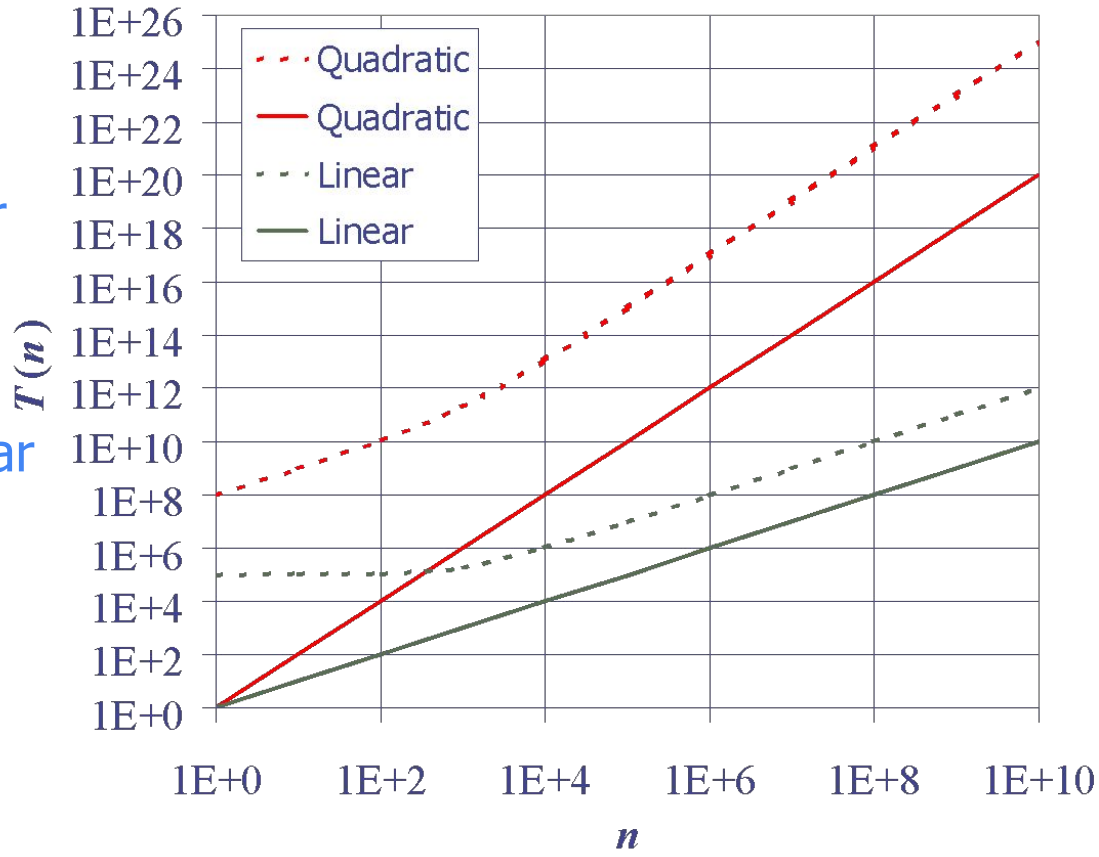
Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ log n
  - Linear ≈ n
  - N-Log-N ≈ n log n
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
- In a log-log chart, the slope of the line corresponds to the growth rate of the function



Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function

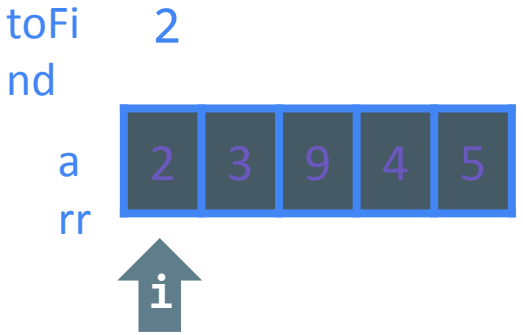


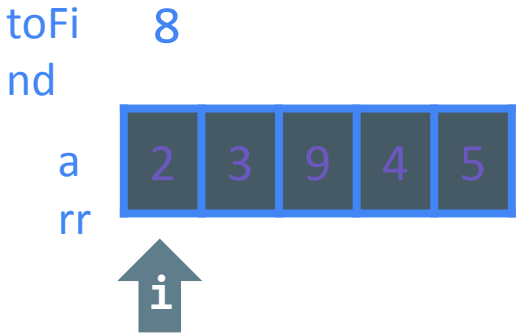
Case Study: Linear Search

- Let's analyze this realistic piece of code!

```
int linearSearch(int[] arr, int
toFind) {
   for (int i = 0; i <
arr.length; i++) {
    if (arr[i] == toFind) {
       return i;
      }
   }
   return -1;
}
```

- What's the first step?
  - We have code, so we need to convert to a function describing its runtime
  - Then we know we can use asymptotic analysis to get bounds





Let's Model This Code!

```
int linearSearch(int[] arr, int
toFind) {
    for (int i = 0; i <
arr.length; i++) {
    if (arr[i] == toFind) +1
        return i;    +1
       }
    }
    return -  +1
}
```

+2

+3

*Remember, these constants don't really matter (we'll start phasing them out soon)

*
?

Same problem as before:
How many times do loop run?

When would that happen?

toFind not present

toFind at beginning

These are key!

- Suppose the loop runs n times?
  - f(n) = 3n + 1
- Suppose the loop only runs once?
  - f(n) = 2

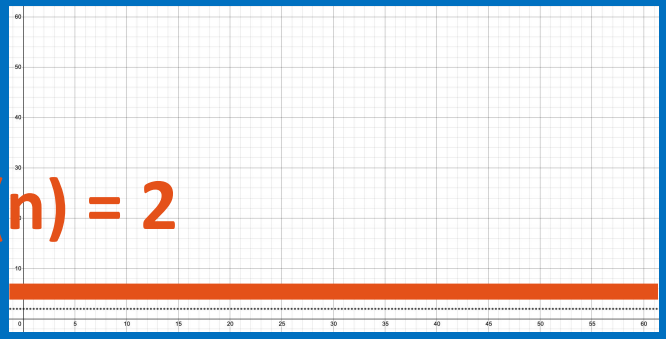
Best Case

toFind 2

arr

2	3	9	4	5

i



After asymptotic analysis:

O(1) Θ(1) Ω(1)

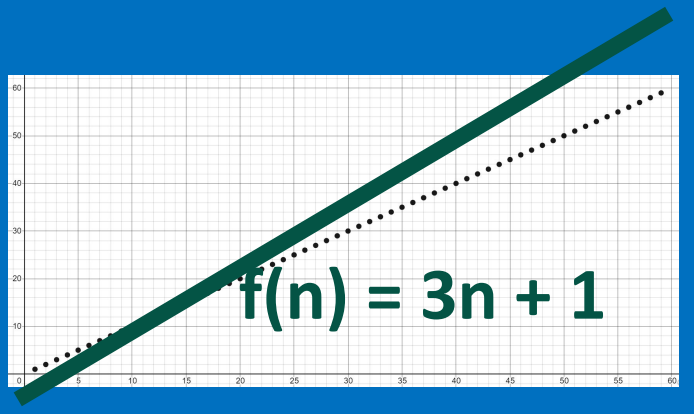
Worst Case

toFind 8

arr

2	3	9	4	5

i



After asymptotic analysis:

O(n) Θ(n) Ω(n)

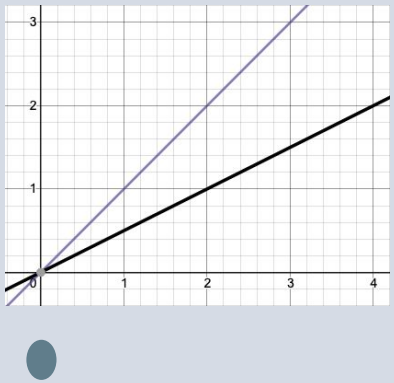
Big-Oh Definition

- Intuitively, f(n) is O(g(n)) if it's smaller than a constant factor of g(n), *asymptotically*
- To prove that, all we need is:
  - (**c**): What is the constant factor?
  - (**n_0**): From what point onward is f(n) smaller?

Big–Oh

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

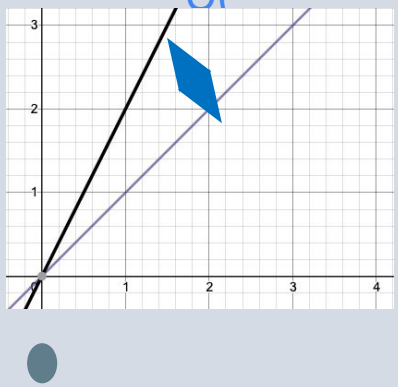
f(n) = 0.5n is O(g(n) = n)



Proof: **c=5** **n_0=0**

0.5n always ≤ n!
Straightforward O(n).

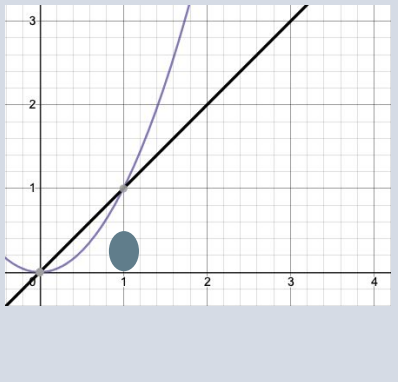
f(n) = 2n is O(g(n) = n)



Proof: **c=2** **n_0=0**

Just need to use constant factor
c=2 so $2n \leq c \cdot n$

f(n) = n is O(g(n) = n^2)



Proof: **c=4** **n_0=6**

$n \leq n^2$, but only after n=1.
Choose that as n_0.

Uncharted Waters: Prime Checking

```
boolean isPrime(int n) {
    int toTest = 2;
    while(toTest < n) {
        if (n % toTest == 0) {
            return false;
        } else {
            toTest += 1;
        }
    }
    return true;
}
```

+1

+1

+2

+1

+2

+1

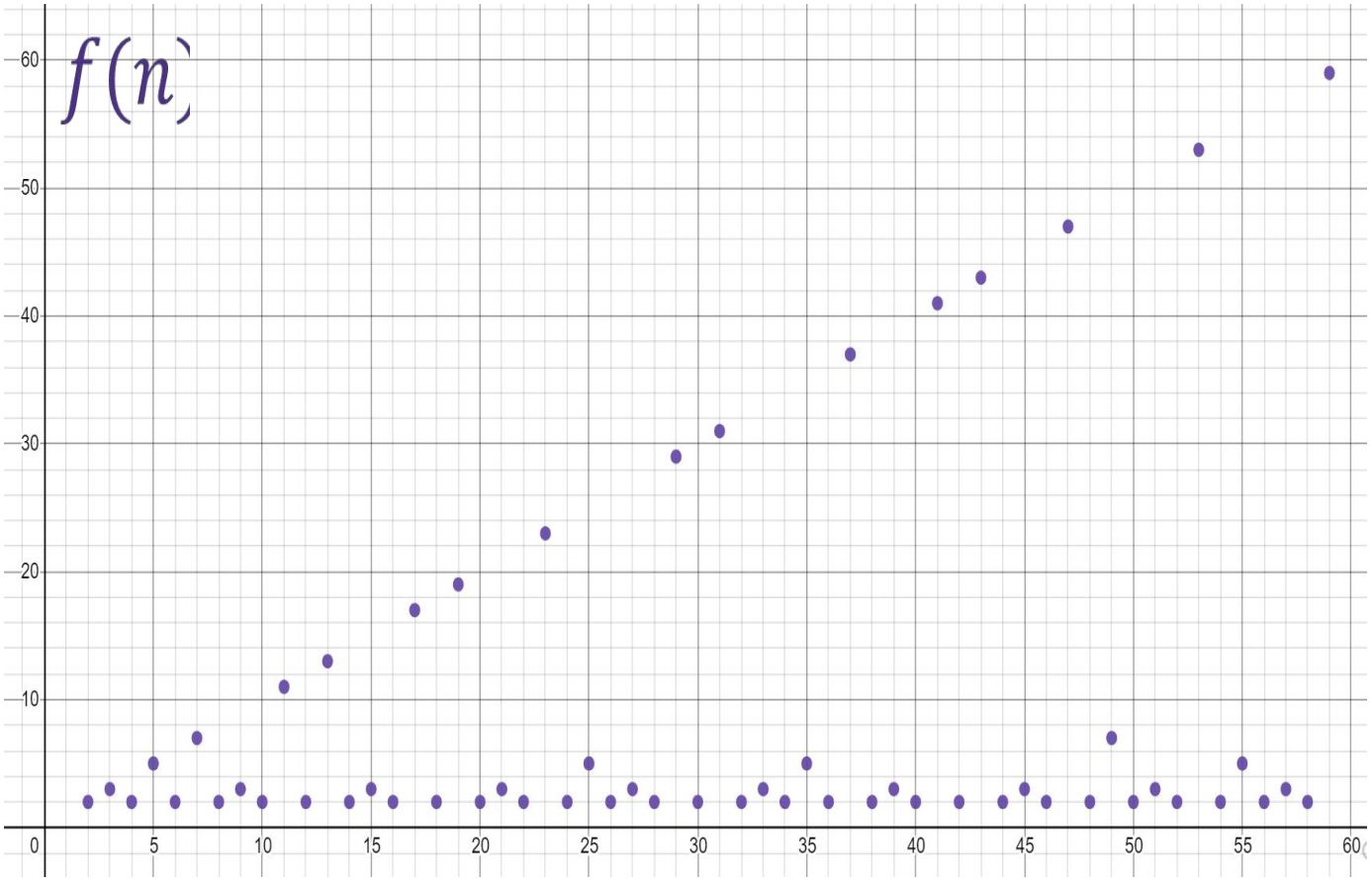
~+5

*?

- Find a model $f(n)$ for the running time of this code on input n → What's the Big-O?
  - We know how to count the operations
  - But how many times does this loop run?

- Sometimes it can stop early
- Sometimes it needs to run n times

Prime Checking Runtime

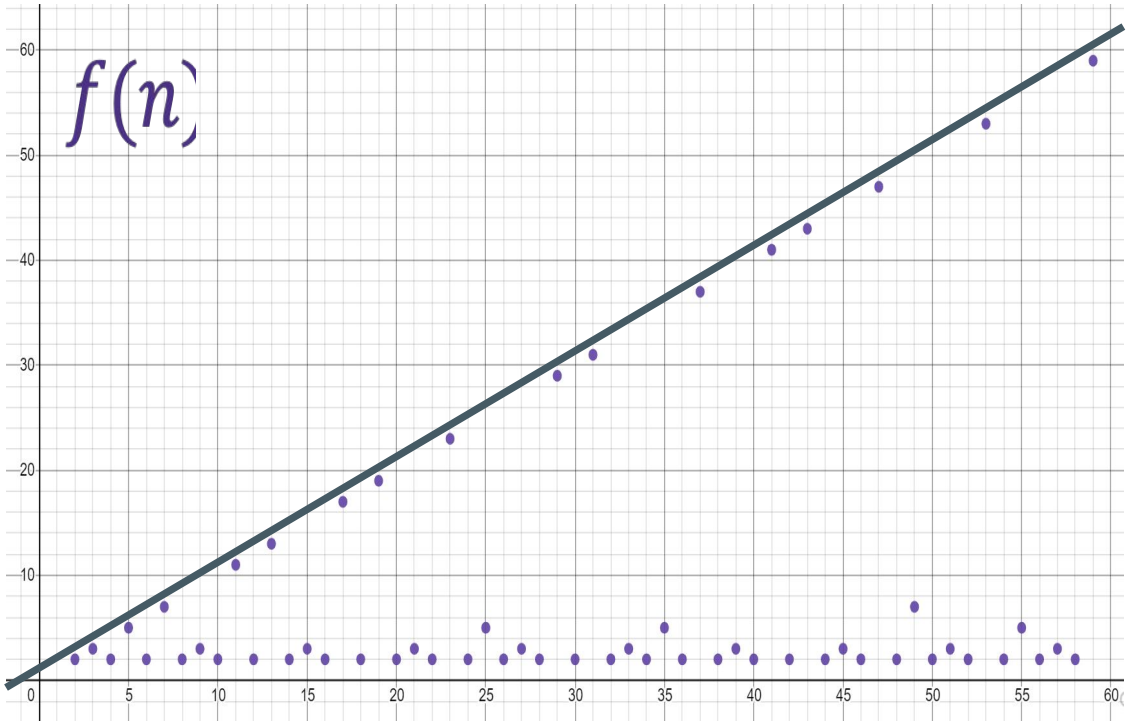


This is why we have definitions!

Is the runtime O(1) or O(n)?

More than half the time we need 3 or fewer iterations. Is it O(1)?

But we can always come up with another value of n to make it take n iterations. So O(n)?



Big-O

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

Using our definitions, we see it's O(n) and not O(1)

Is the runtime O(n)?

Can you find constants c and n_0?

How about $c = 1$ and $n_0 = 5$,

$f(n)$ =smallest divisor of $n \leq 1 \cdot n$ for $n \geq 5$

Is the runtime O(1)?

Can you find constants c and n_0?

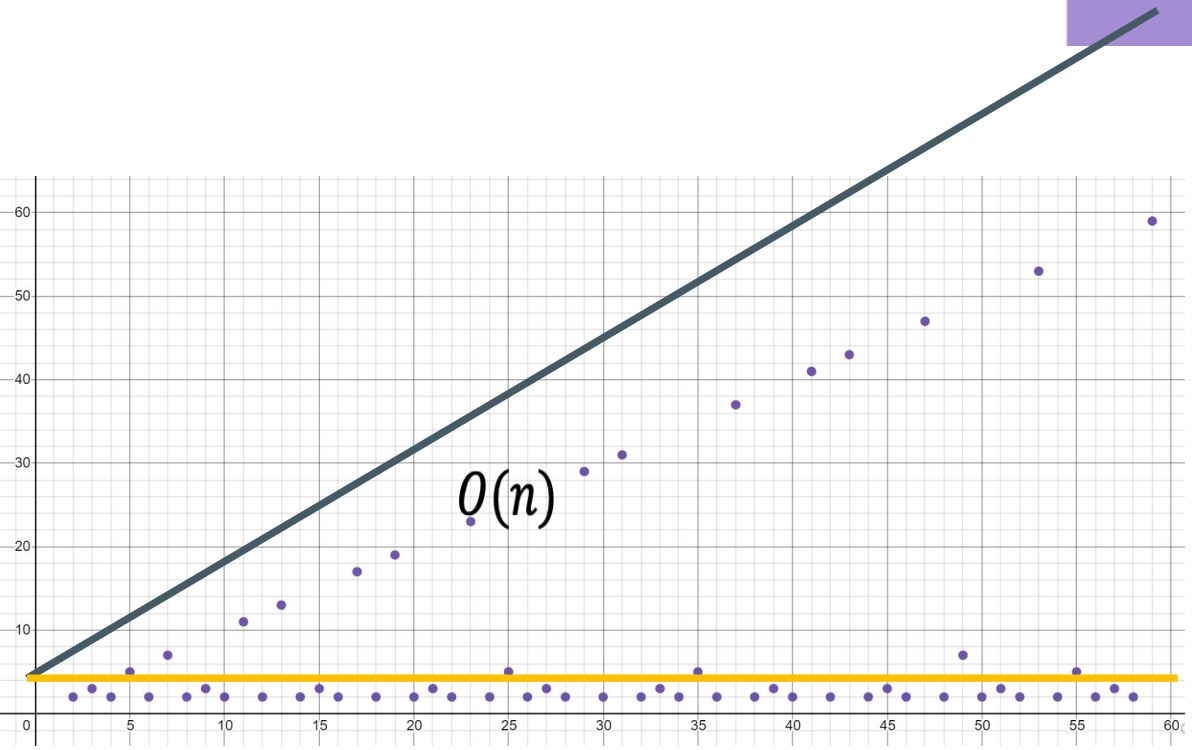
No! Choose your value of c. I can find a prime number k bigger than c.

And $f(k) = k > c \cdot 1$ so the definition isn't met!

Big-Ω [Omega]

Big-Omega

$f(n)$ is $\Omega(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \geq c \cdot g(n)$



Big-O

$f(n)$ is $O(g(n))$ if there exist positive constants c, n_0 such that for all $n \geq n_0$, $f(n) \leq c \cdot g(n)$

The formal definition of Big-Omega is the flipped version of Big-Oh!

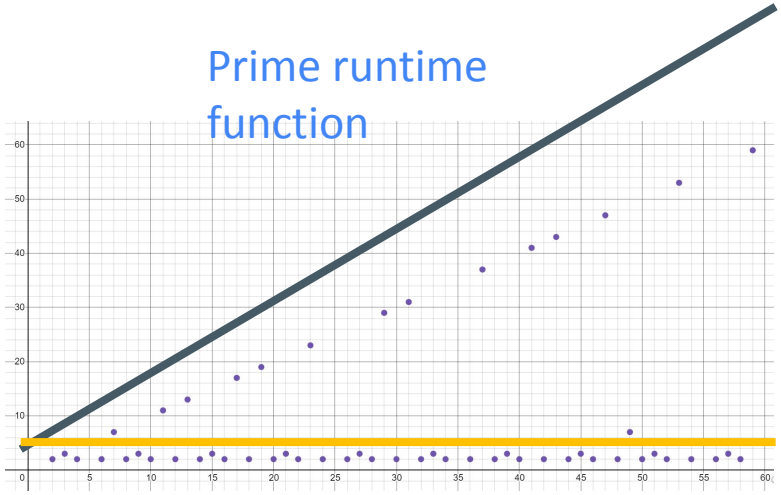
"f(n) is O(g(n))" : f(n) grows at most as fast as g(n)

"f(n) is Ω(g(n))" : f(n) grows at least as fast as g(n)

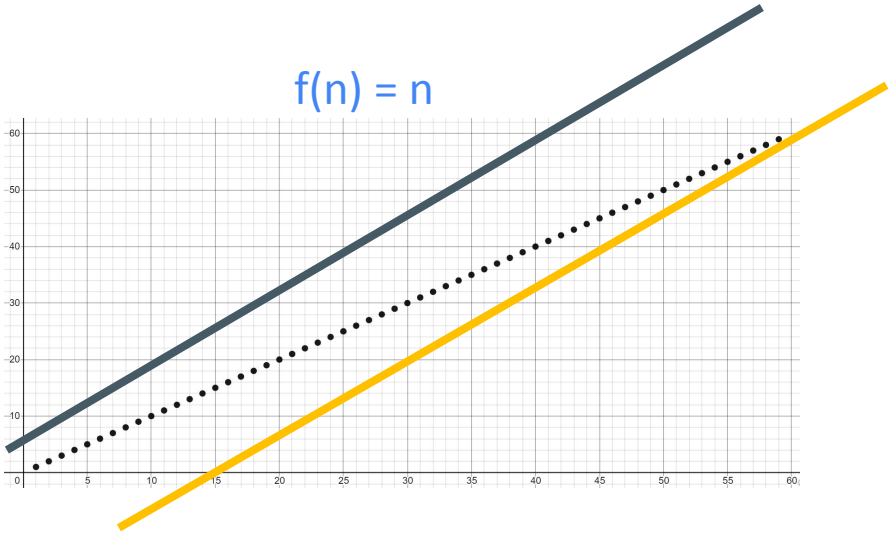
$\Omega(1)$

Tight Big-O and Big-Ω Bounds Together

Prime runtime function



f(n) = n



$O(n)$

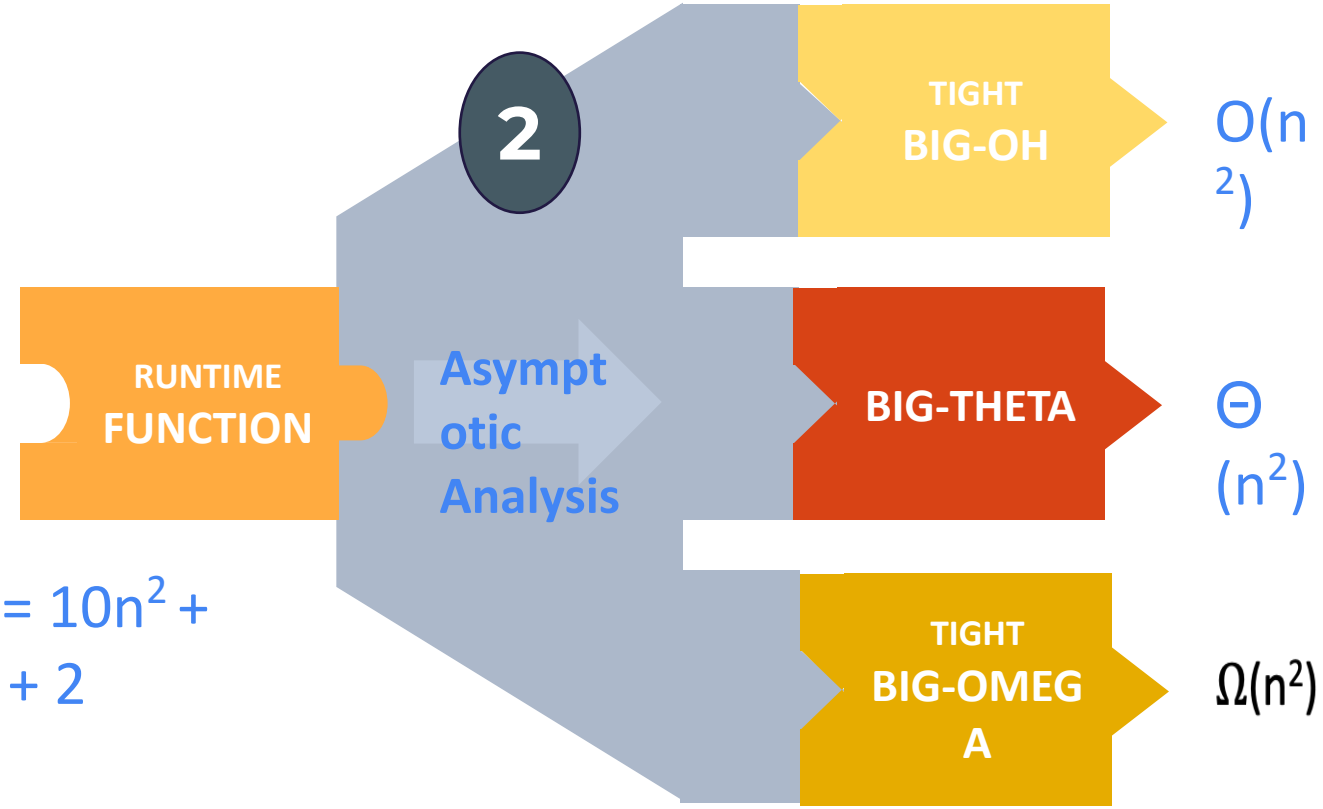
$\Omega(1)$

$O(n)$

$\Omega(n)$

Note: *most* functions look like the one on the right, with the same tight Big-Oh and Big-Omega bound. But we'll see important examples of the one on the left.

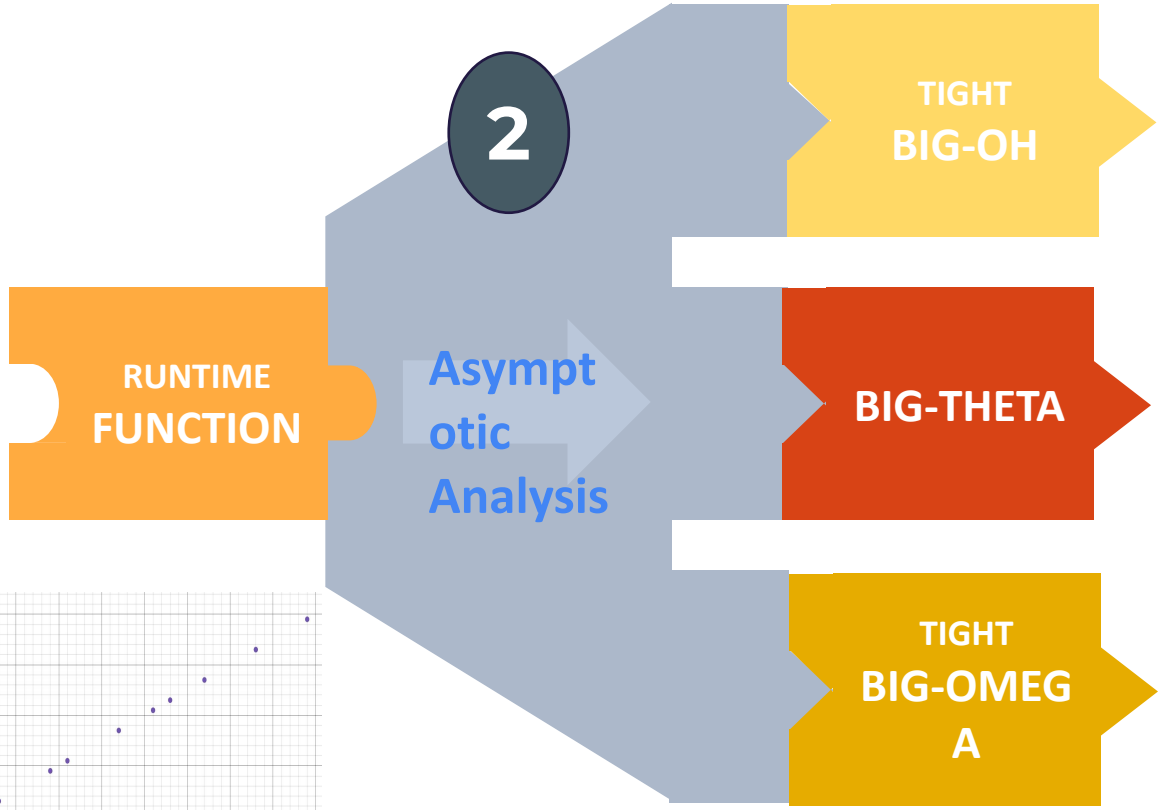
Our Upgraded Tool: Asymptotic Analysis

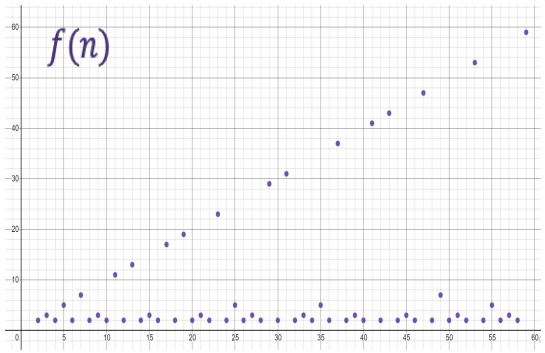


$f(n) = 10n^2 + 13n + 2$

We've upgraded our Asymptotic Analysis tool to convey more useful information! Having 3 different types of bounds means we can still characterize the function in simple terms, but describe it more thoroughly than just Big-Oh

Our Upgraded Tool: Asymptotic Analysis

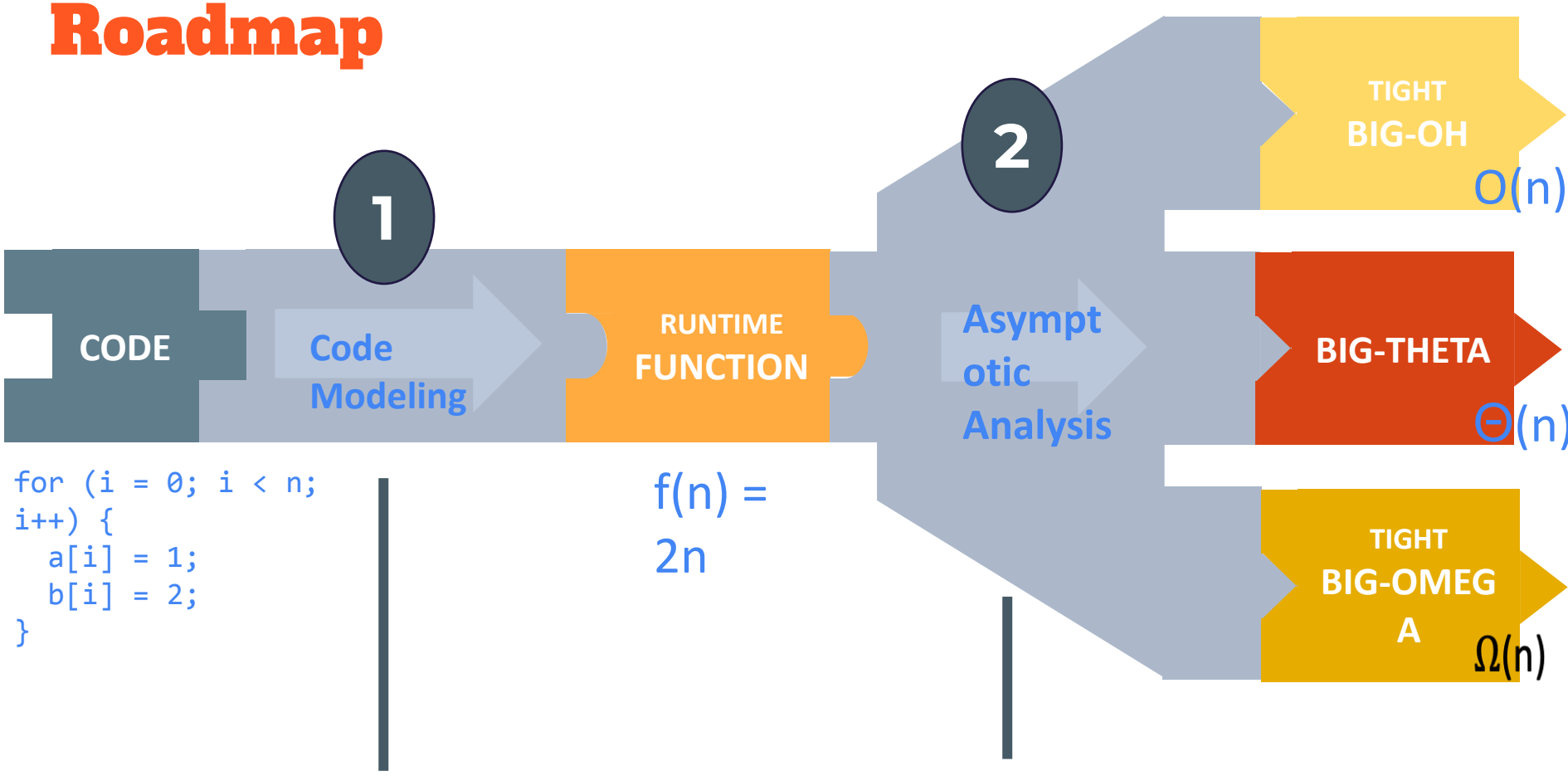




isPrime()

Big-Theta doesn't always exist for every function! But the information that Big-Theta doesn't exist can *itself* be a useful characterization of the function.

Algorithmic Analysis Roadmap



Now, let's look at this tool in more depth. How exactly are we coming up with that function?

We just finished building this tool to characterize a function in terms of some useful bounds!

Summarizing Big Theta, Oh, Omega