Lecture 5: Case Analysis

CSE 373 – Data Structures and Algorithms

Warm Up!

Big-O

$f(n) \in O(g(n))$ 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

$f(n) \in \Omega(g(n))$ 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

$f(n) \in \Theta(g(n))$ if
$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Which of the following is in $O(n^2)$? $\Omega(n^2)$? $\Theta(n^2)$?

a. $f(n) = 42$

$f(n) \in O(n^2)$

b. $f(n) = 5n + 100$

$f(n) \in O(n^2)$

c. $f(n) = nlog_2(3n)$

$f(n) \in O(n^2)$

d. $f(n) = 4n^2 - 2n + 10$

$f(n) \in O(n^2)$ $f(n) \in \Omega(n^2)$ $f(n) \in \Theta(n^2)$

e. $f(n) = 2^n$

$f(n) \in \Omega(n^2)$

Take 2 Minutes

1. www.pollev.com/cse373activity for participating in our active learning questions.
2. https://www.pollev.com/cse373studentqs to ask your own questions

Simplified, tight big-O

In this course, we'll essentially use:

- Polynomials (n^c where c is a constant: e.g. $n, n^3, \sqrt{n}, 1$)
- Logarithms $\log n$
- Exponents (c^n where c is a constant: e.g. $2^n, 3^n$)
- Combinations of these (e.g. $\log(\log(n))$, $n \log n$, $\left(\log(\mathrm{n})\right)^2$)

For **this course**:

- A "tight big-O" is the slowest growing function among those listed.
- A "tight big-Ω" is the fastest growing function among those listed.
- (A Θ is always tight, because it's an "equal to" statement)
- A "simplified" big-O (or Omega or Theta)
  - Does not have any dominated terms.
  - Does not have any constant factors – just the combinations of those functions.

Administrivia

- Project 0 due 11:59pm PST tonight
- Project 1 out by midnight tonight, due Wednesday April 15th
- New post-lecture extra credit
  - Available on website before 6pm PST day of lecture
  - Closes 48 hours later
- collaboration policy (please don’t share code with not-your-partner – but you can talk about it at a high level)
- come to OH! We’re lonely and Piazza is getting filled with debugging questions that are easier to talk about in real-time.

Project notes

How to effectively work on partner projects:

Pair program! See the [document on the webpage].

- Two brains is better than one when debugging
- We expect you to understand the full projects, not just half of the projects.

Meet in real-time with your partner (screen-share over a zoom call!).

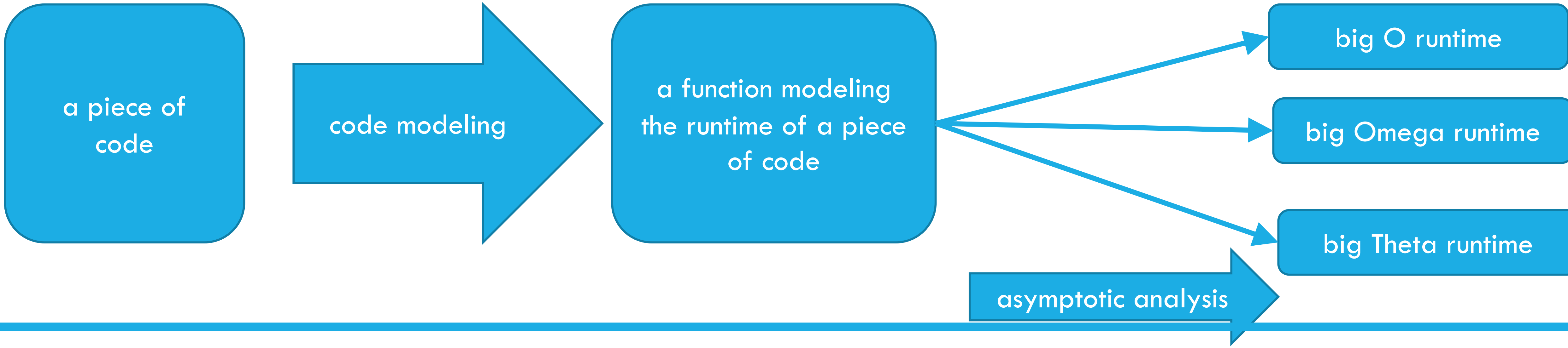
Please don't:

- Come to office hours and say "my partner wrote this code, I don't understand it. Please help me debug it."
- Just split the project in-half and each do half (or alternate projects)
- Be mean to your partner. Working with someone else will probably take some patience but the result is usually awesome if we're all respectful.

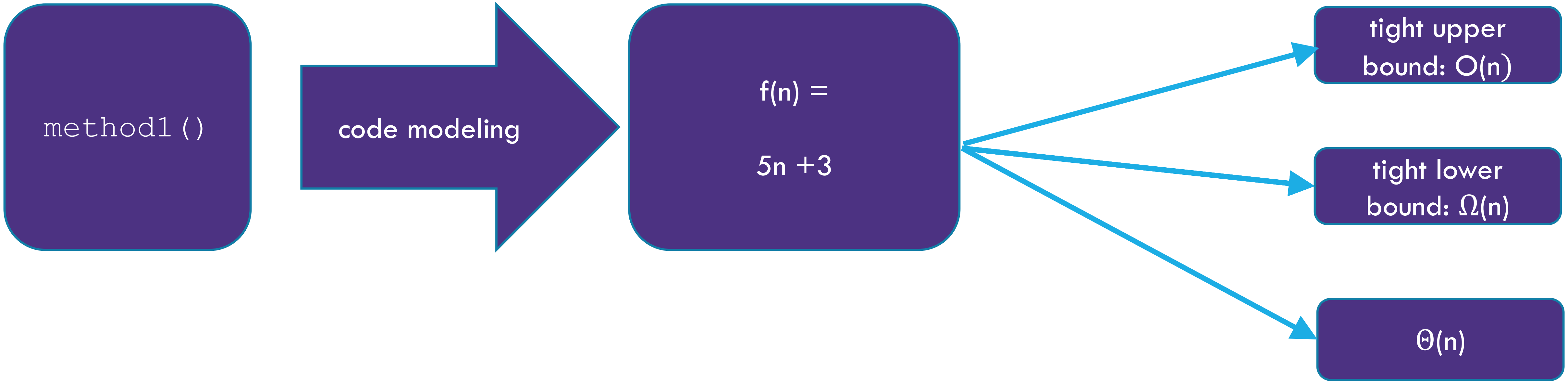
Questions

Case Analysis

Where we left off last time:



An example of the process



```java
public void print(int n){
    for(int i=0; i < n; i++){
        System.out.println(i);
    }
}
```

What's the code model for the runtime of `print` in terms of n? (Assume System.out.println takes constant runtime)

```java
 public void print(int n){
   for(int i=0; i < n; i++){
       System.out.println(i);
   }
}
```

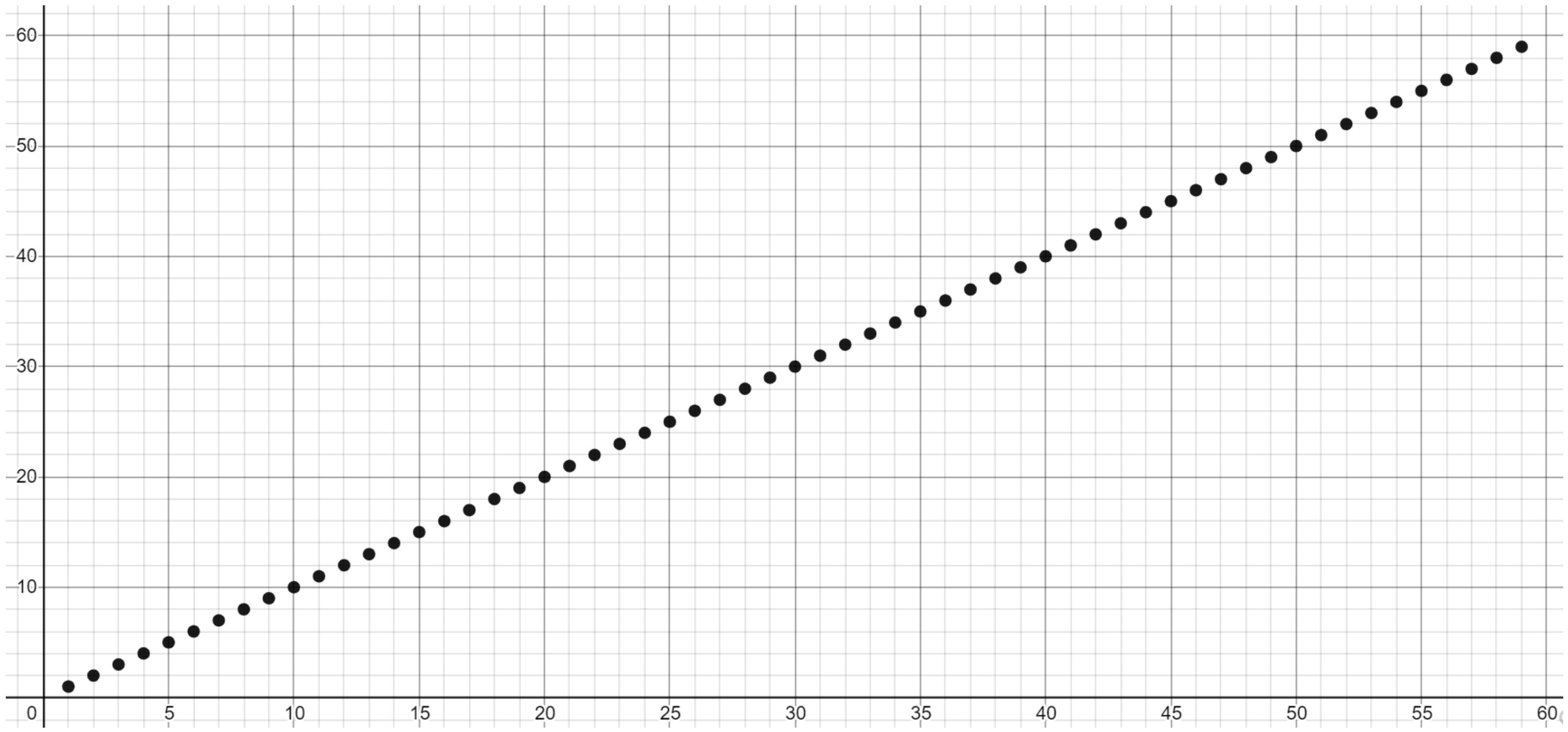
What's the code model for the runtime of `print` in terms of n? (Assume System.out.println takes constant runtime)

Maybe something like $f(n) = 3n + 2$ (note: this is made up, since the details of the constants don't matter)

```
public void print(int n){
   for(int i=0; i < n; i++){
       System.out.println(i);
   }
}
```

What's the code model for the runtime of `print` in terms of n? (Assume System.out.println takes constant runtime)

Maybe something like f(n) = 3n + 2 (note: this is made up, since the details of the constants don't matter)



Cases

We defined $f(n)$ to be (our model for) the number of operations the code does on an input of size n.

For the code we've seen so far, how the variable n (the size of the input) affects the code has been the only thing determining what the code model looks like. We're now going to take a step and start looking at some code that has more factors than just n.

Linear Search

/* given an array and int toFind, return index where toFind is located, or -1 if not in array.*/

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

How should we model this code's runtime as a mathematical function?

linearSearch Models: formulas

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

How should we model this code's runtime as a mathematical function?

Unlike before, the number of steps for this piece of code does not depend solely on the input size, n (length of the array). In this case, there's another factor which is: where does `toFind` appear in the input array (if at all)?

linearSearch Models: formulas

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

How should we model this code's runtime as a mathematical function?

Unlike before, the number of steps for this piece of code does not depend solely on the input size, n (length of the array). In this case, there's another factor which is: where does `toFind` appear in the input array (if at all)?

If `toFind` is in `arr[0]`, we'll only need one iteration. One specific example input: arr = [1, 2, 4, 9, 16, 25, 36, 49, 64] and toFind = 1

$$f_1(n) = 4$$

(note: the constants here were made-up since they don't affect anything for this analysis)

linearSearch Models: formulas

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

How should we model this code's runtime as a mathematical function?

Unlike before, the number of steps for this piece of code does not depend solely on the input size, n (length of the array). In this case, there's another factor which is: where does `toFind` appear in the input array (if at all)?

If `toFind` is in `arr[0]`, we'll only need one iteration. One specific example input: arr = [1, 2, 4, 9, 16, 25, 36, 49, 64] and toFind = 1

$$f_1(n) = 4$$

If `toFind` is not in `arr`, we'll need n iterations. One specific example input : arr = [1, 2, 4, 9, 16, 25, 36, 49, 64] and toFind = -5

$$f_2(n) = 9n + 1$$

(note: the constants here were made-up since they don't affect anything for this analysis)

linearSearch Models: formulas

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

How should we model this code's runtime as a mathematical function?

Unlike before, the number of steps for this piece of code does not depend solely on the input size, n (length of the array). In this case, there's another factor which is: where does `toFind` appear in the input array (if at all)?

If `toFind` is in `arr[0]`, we'll only need one iteration. One specific example input: arr = [1, 2, 4, 9, 16, 25, 36, 49, 64] and toFind = 1

$$f_1(n) = 4$$

If `toFind` is not in `arr`, we'll need n iterations. One specific example input : arr = [1, 2, 4, 9, 16, 25, 36, 49, 64] and toFind = -5

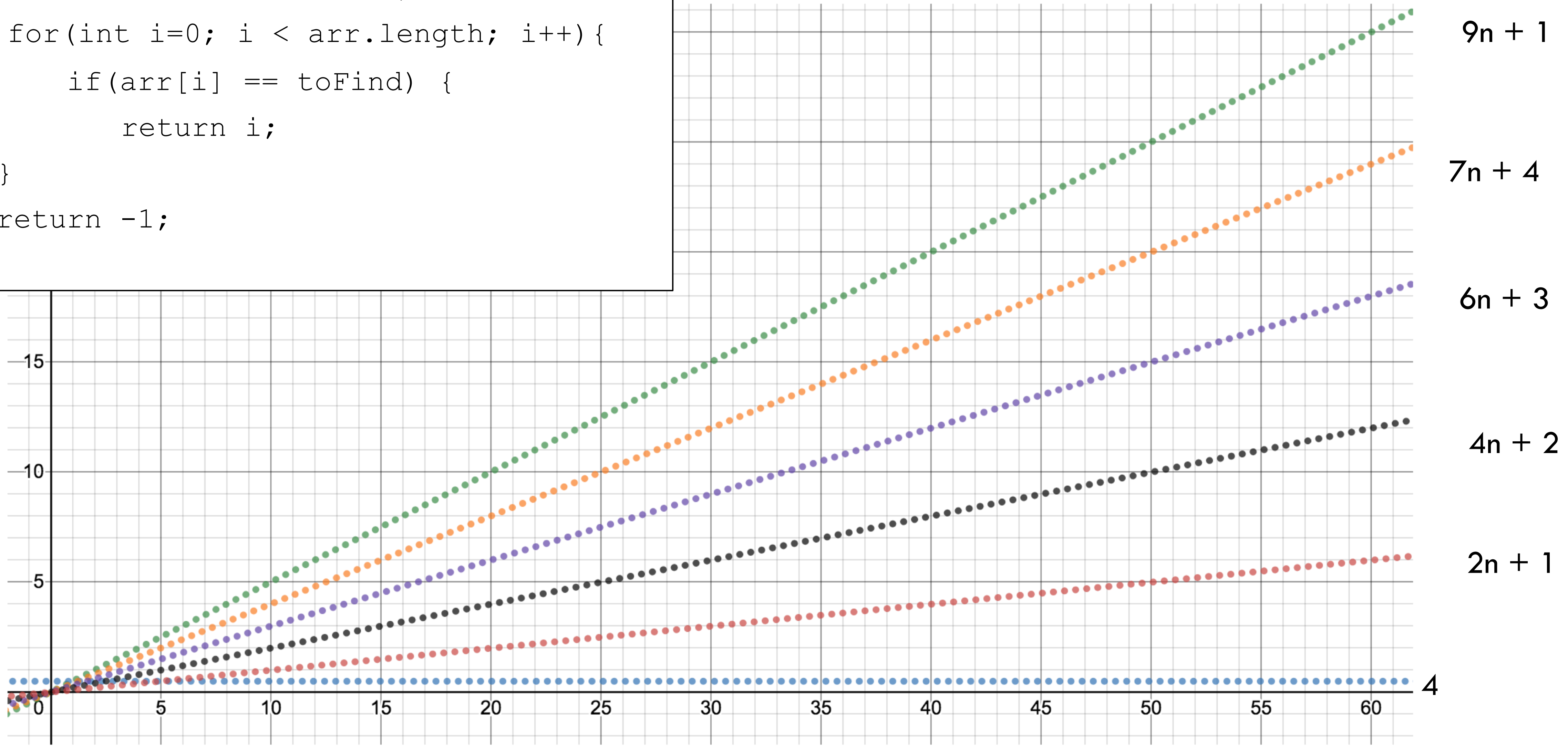
$$f_2(n) = 9n + 1$$

And what about in-between if toFind is somewhere inbetween? What if it's at index 5? What if it's index n – 5? Every one of these situations deserves its own runtime function.

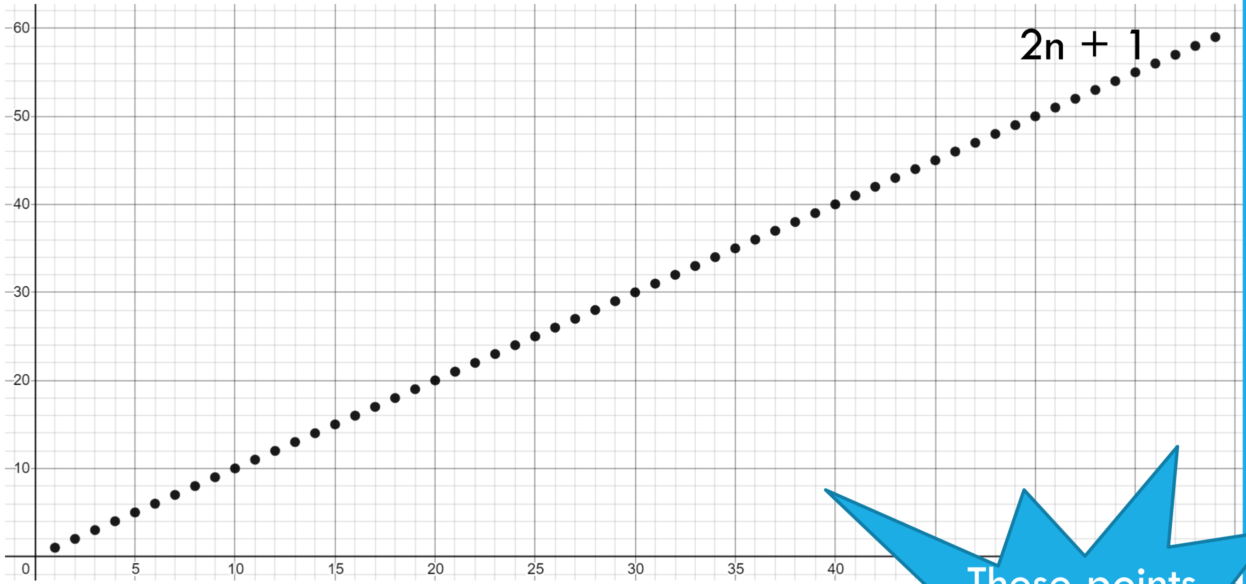
(note: the constants here were made-up since they don't affect anything for this analysis)

linearSearch Models: visually

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



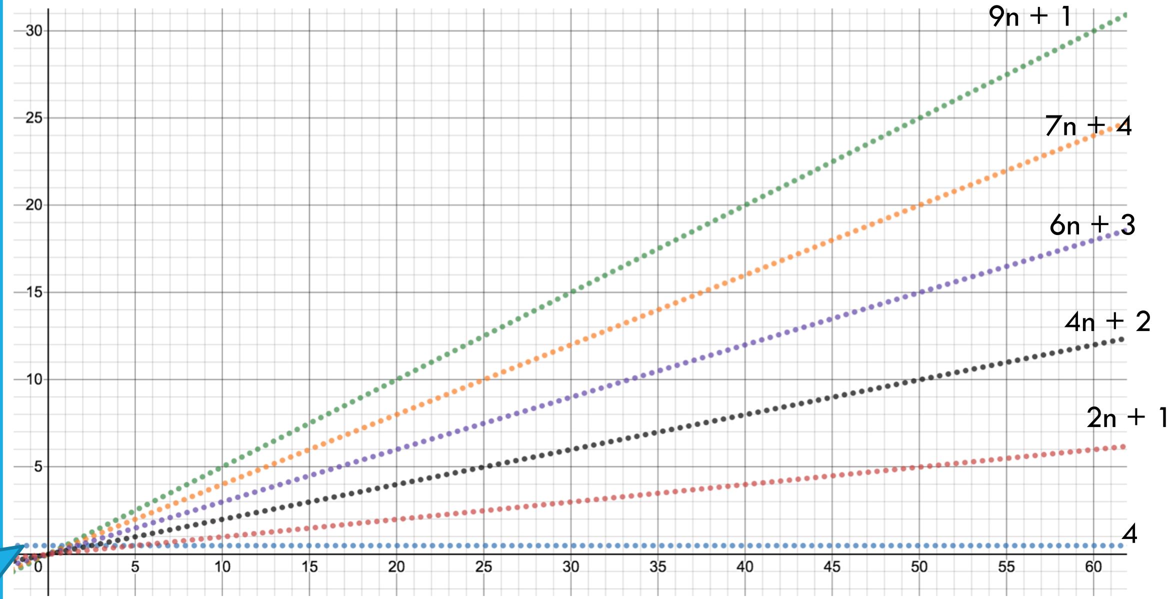
```java
public void print(int n){
    for(int i=0; i < n; i++){
        System.out.println(i);
    }
}
```



Code we've looked at before: the `print` method only had one model/runtime function. The runtime function will ALWAYS be 2n + 1.

These points are big ideas!

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



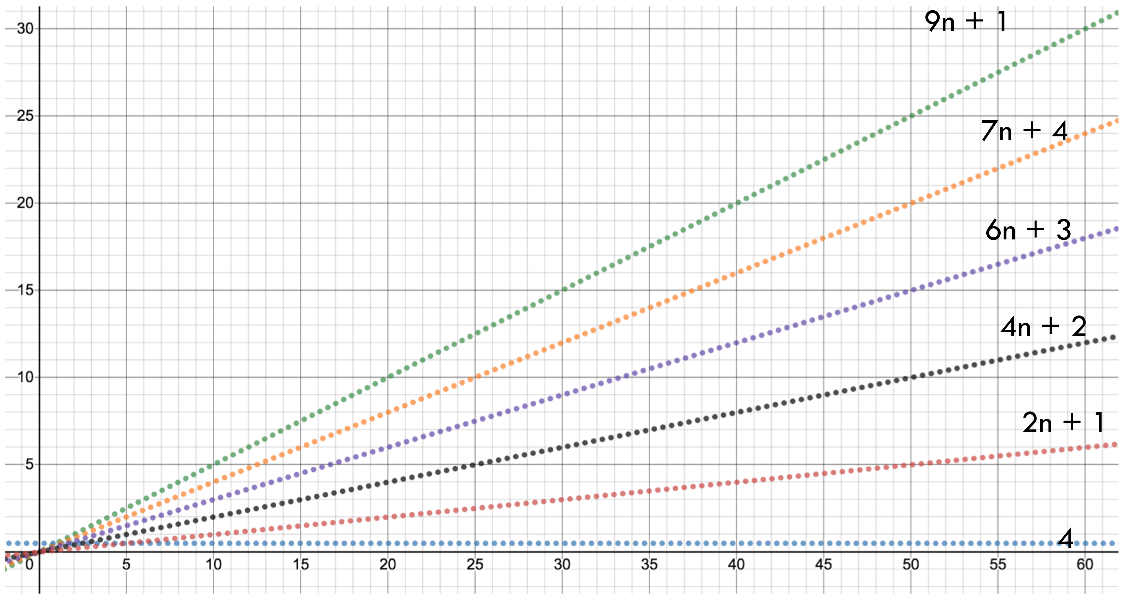
Code we're looking at now: the `linearSearch` method has multiple possible models/runtime functions. Which runtime function is applicable depends on deciding which case we want to talk about.

Usually we care about the longest our code could run on an input of size n.

This is **worst-case** analysis

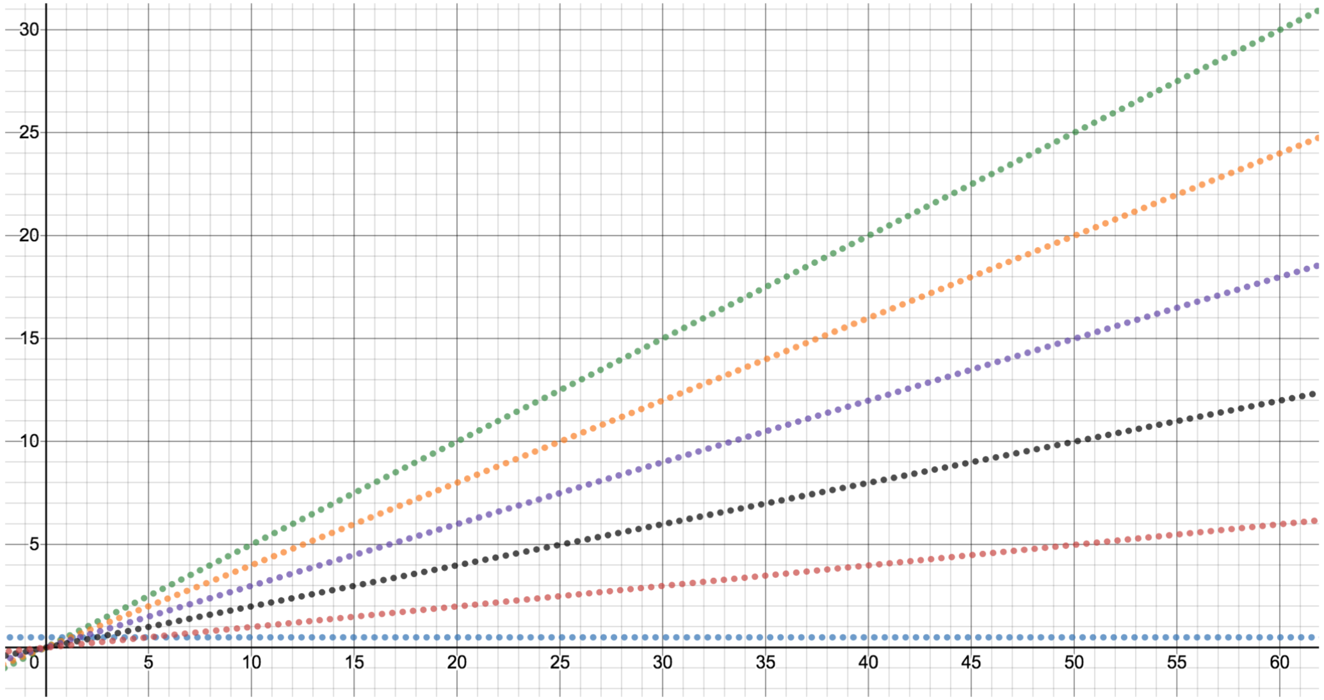
But sometimes we care about the fastest our code could finish on an input of size n.

This is **best-case** analysis



For linearSearch,
- the model for the worst case is $f(n) = 9n + 1$
- the model for the best case is $f(n) = 4$.

Note: the best case and worst case situations for this method actually have different complexity class runtimes!



<u>**Case**</u> (rough definition): a description of the state of parameters and relevant state for a method/algorithm that is specific enough that a code model (runtime function) can be determined whose only parameters are the input size(s).

example cases from before:

- `toFind` is in `arr[0]` `(best case)` → `4`
- `toFind` is not in `arr` `(worst case)` → `9n + 1`

So every different possible runtime function (see graph) that comes from some particular state of the parameters/data structure is a different case!

Usually we'll only ask explicitly about best/worst in this course, but there are plenty of other useful cases to discuss, however (next slide).

Other useful types of cases (beyond best/worst)

"Assume X won't happen case"

- Assume our array won't need to resize is the most common.

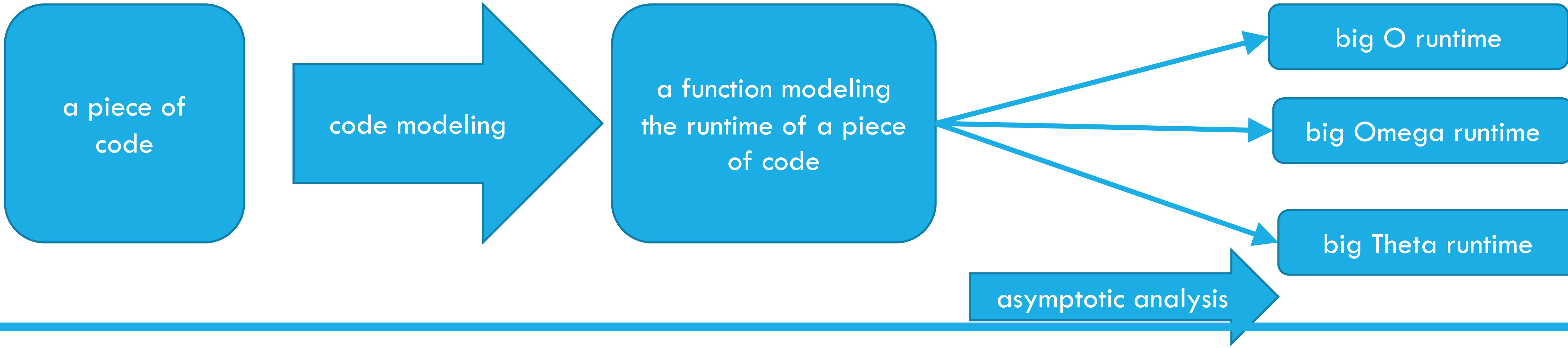
"Average case"

- Assume your input is random
- Need to specify what the possible inputs are and how likely they are.
- $f(n)$ is now the **average** number of steps on a **random** input of size n.

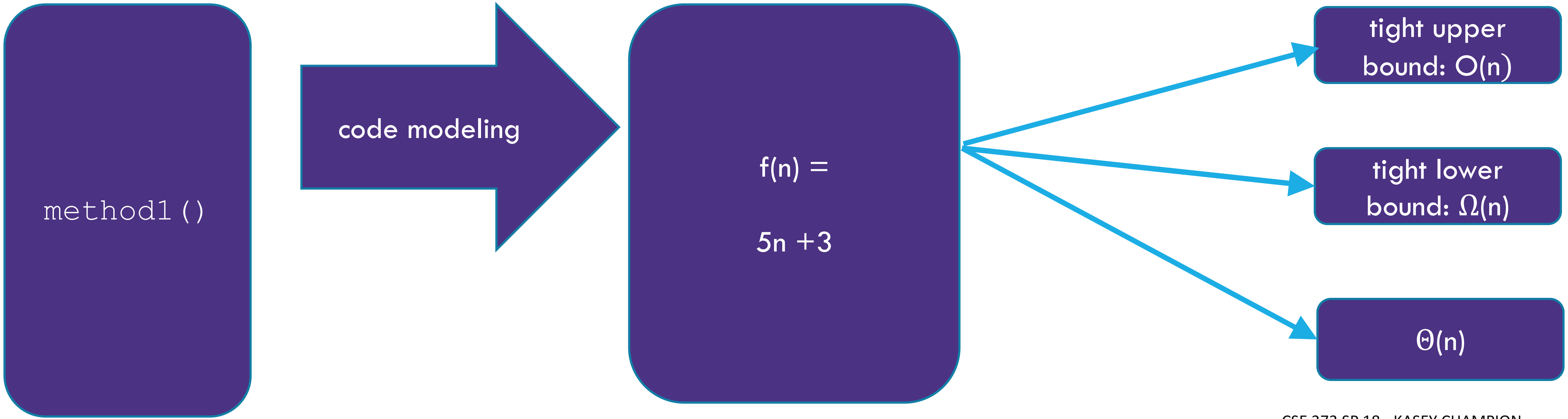
"In-practice case"

- This isn't a real term. (We just made it up)
- Make some reasonable assumptions about how the real-world is probably going to work
  - We'll tell you the assumptions, and won't ask you to come up with these assumptions on your own.
- Then do worst-case analysis under those assumptions.

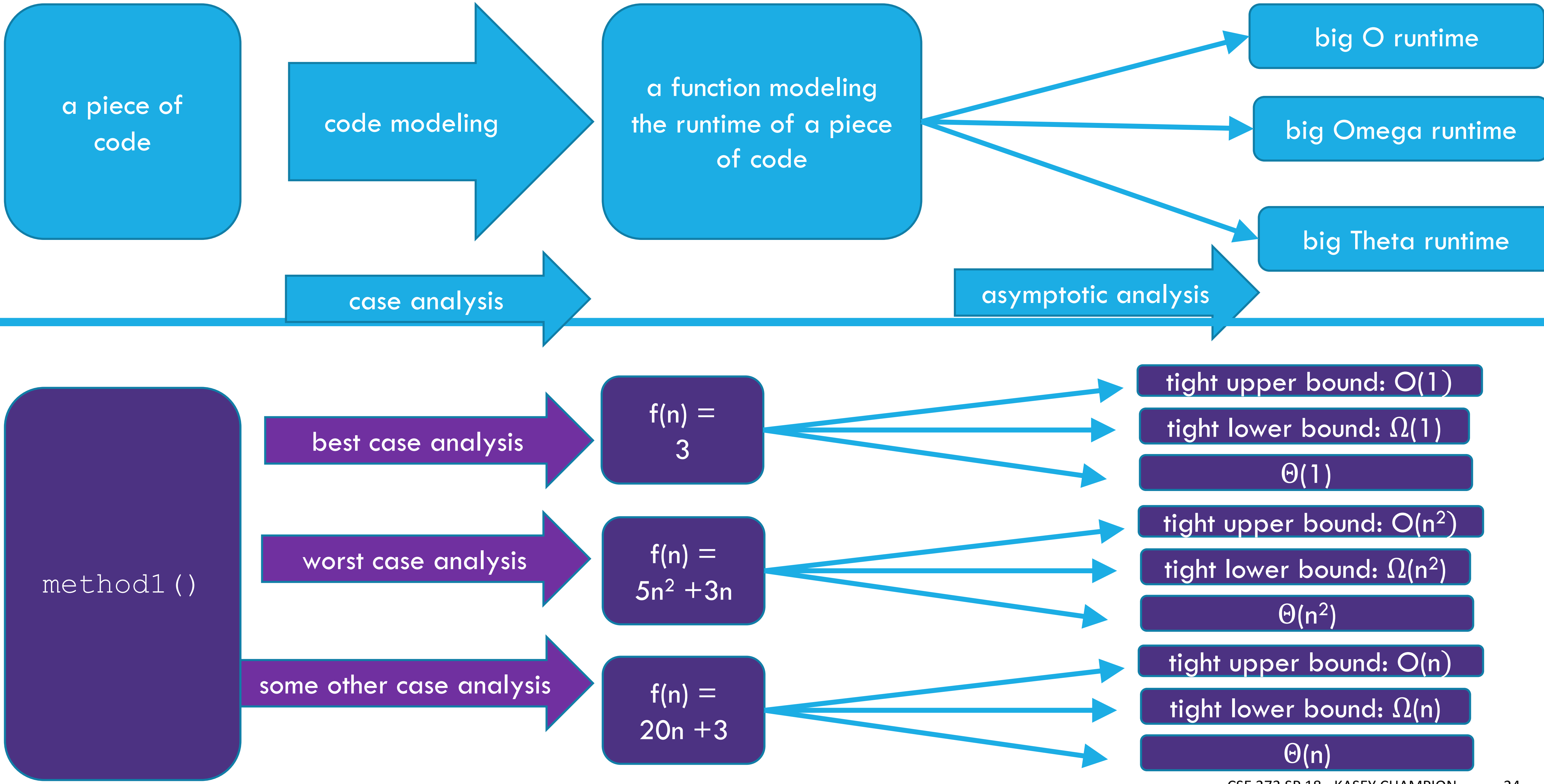
Where we left off last time:



An example of the process



Where we left off last time:



Questions?

Common Questions

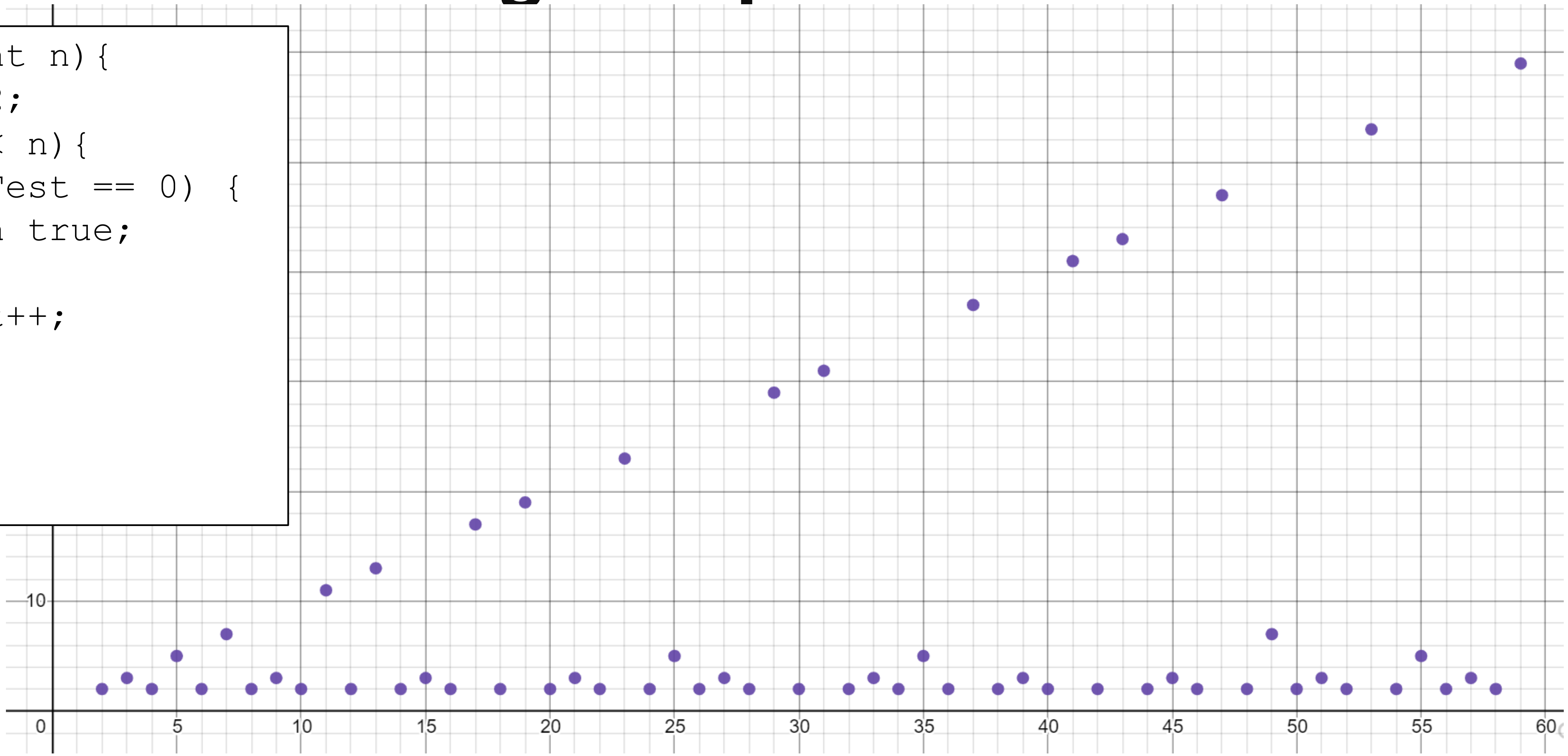
How can you tell if there's a different best/worst case code model for a given piece of code?

How does this relate to big O / big Omega / big Theta?

Can you choose n = 0 to be the best case? Can we choose n = infinity to be our worst case?

How can you tell if there's a different best/worst case code model for a given piece of code?

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

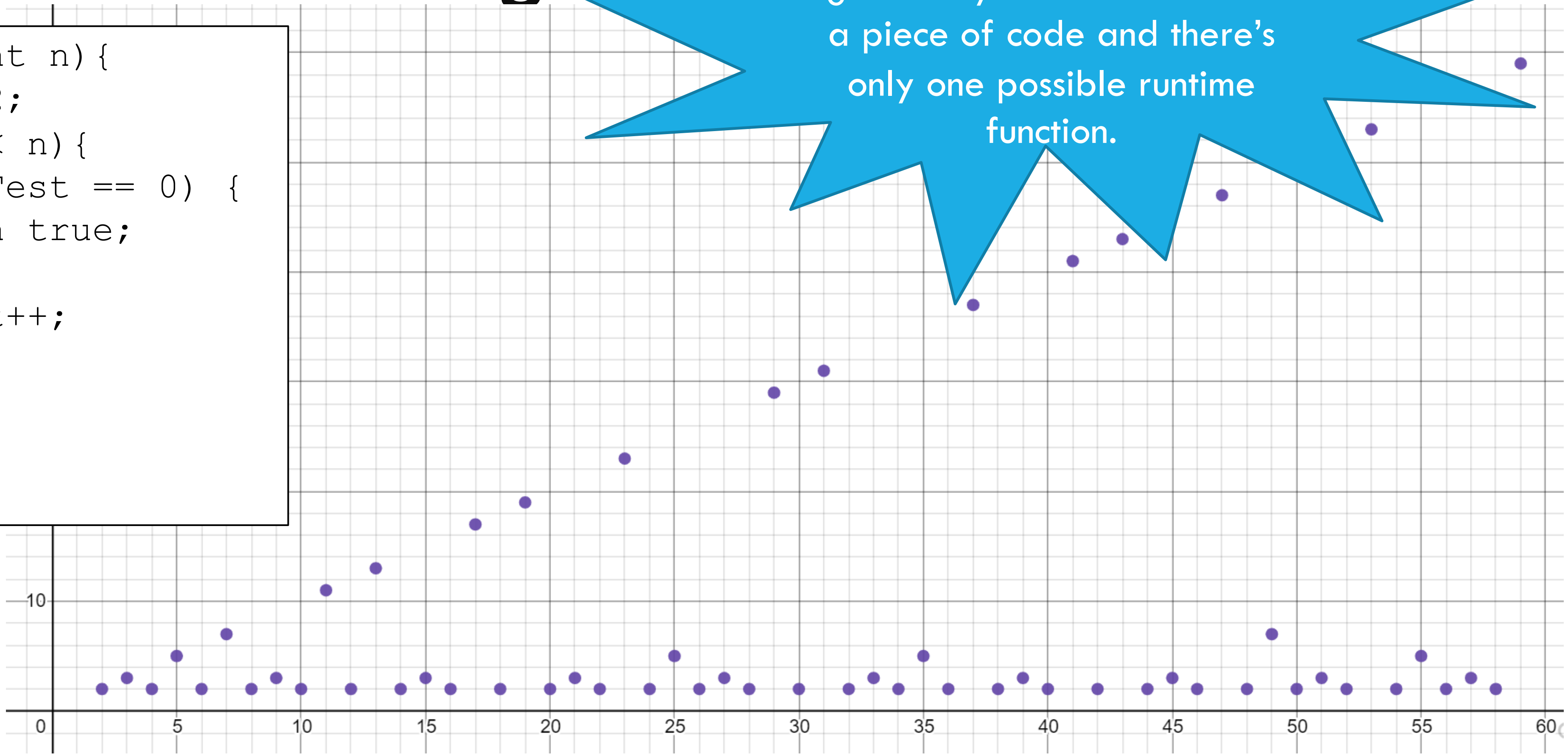


Are there other possible code models for this piece of code?

In other words: if n is given, are there still other factors that determine the runtime?

How can you tell if there's a differ… …st/worst case code model for a given …

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



Sometimes, there aren't significantly different cases for a piece of code and there's only one possible runtime function.

Are there other possible code models for this piece of code?

In other words: if n is given, are there still other factors that determine the runtime?

No! This is actually pretty similar to the print method we saw earlier – the only variable / possible thing that can affect the runtime is the input parameter number, n.

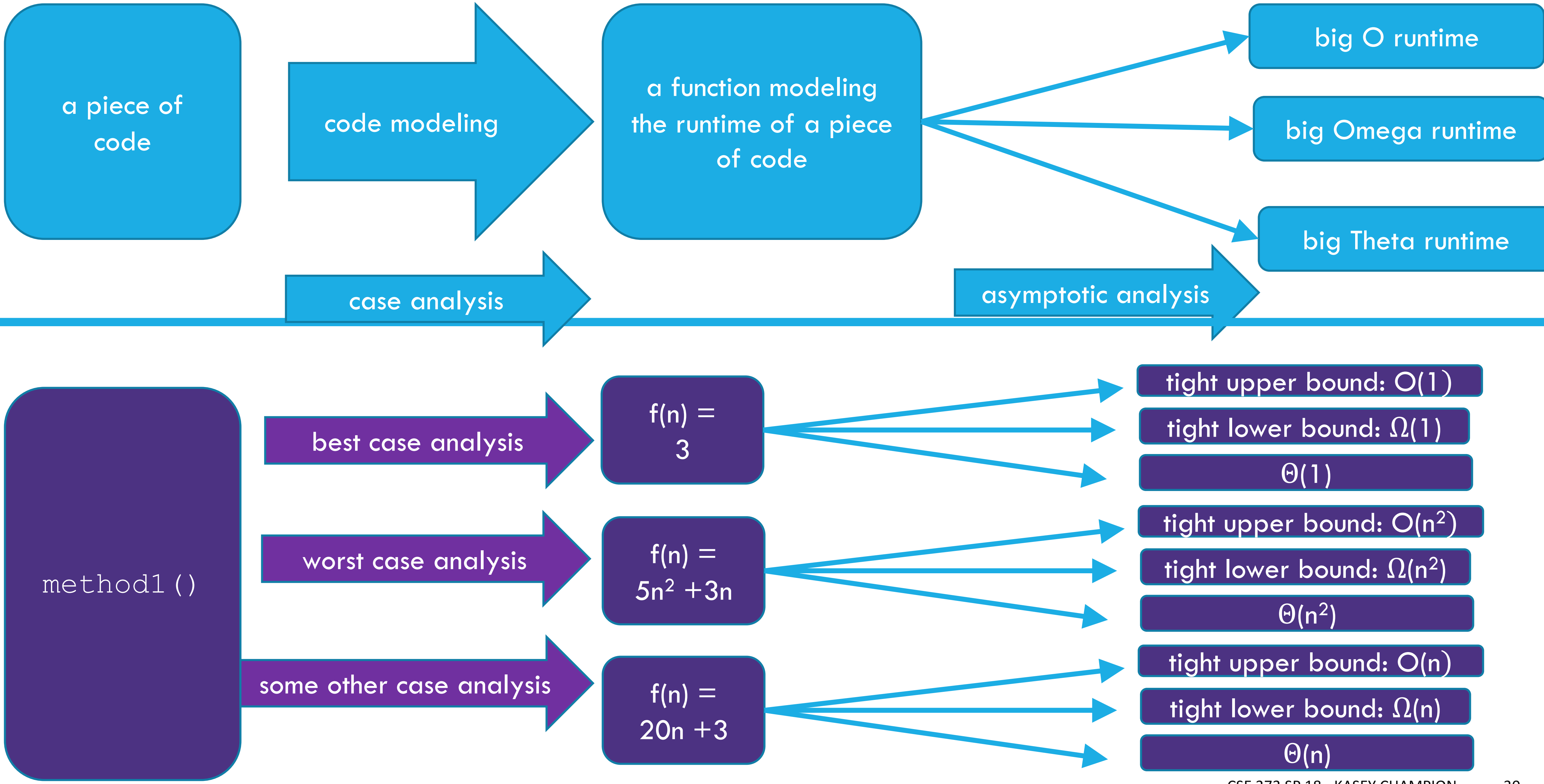
Common Questions

How can you tell if there's a different best/worst case code model for a given piece of code?

How does case analysis relate to asymptotic analysis (big O / big Omega / big Theta)?

Can you choose n = 0 to be the best case? Can we choose n = infinity to be our worst case?

Where we left off last time:



Common Questions

How can you tell if there's a different best/worst case code model for a given piece of code?

How does case analysis relate to asymptotic analysis (big O / big Omega / big Theta)?

Can you choose n = 0 to be the best case? Can we choose n = infinity to be our worst case?

Common Questions

How can you tell if there's a different best/worst case code model for a given piece of code?

How does case analysis relate to asymptotic analysis (big O / big Omega / big Theta)?

Can you choose n = 0 to be the best case? Can we choose n = infinity to be our worst case?

Because this is all in the lens of code → mathematical function, this mathematical function has to be defined for all values of n. If you just scope it to the n = 0, what does your function look like? Basically you can't decide on a specific input size since that's supposed to be the x-axis for our graph. Doing so would be like honing in on one specific x/y point instead of defining a full function that can plug in any n.

How to do case analysis

1. Determine if there are actually significantly different cases.
   - Are there any other variables/parameters that could affect the runtime other than the input size? If the input/ data structure size is the only factor, then there aren't any other significant cases (think isPrime and print).

2. Look at the code, and try to figure out more specifically how things could change depending on the input (except for the input size).
   - How can you exit loops early (for determining the best case)?
     - Conversely, How can you make sure loops run for as long as possible (for determining the worst case)?
   - Can you return (exit the method) early? (for determining the best case)
   - Are some if/else branches much slower than others?

3.. Figure out what inputs can cause you to hit the (best/worst) parts of the code. (e.g. what does the input array look like? What parameter values/ combinations of values trigger the expensive logic?)

Some previous data structure runtimes / code snippets + something new

ArrayList

- size

- insert(key, value) // ignore resizing

LinkedDictionary

- get

Bubble Sort code

We're going to try breakouts again for these! See [this google doc] for the problems and instructions for breakouts!

solutions ([link])

Breakout Instructions

1. Instructor will trigger breakout rooms
2. Accept the invite that pops up
3. Work with your partners to answer the question on slide 16
4. TAs will be coming in and out. Fill out this form to request a TA's assistance: https://forms.gle/b9NiC1s11FKBcpm89
5. Instructor will end the breakouts in 5 minutes

For detailed instructions on how breakouts work: https://docs.google.com/presentation/d/15HiAPu6yYz2WWbkonRejBtUcq_FFhmoWFyT2l25G06o/edit#slide=id.g8289eae46a_0_694

Another example together: ArrayList insert

```
data; // a field for the array that stores all the values
size; // a field to keep track of the number of valid values


// inserts the given value at the given index
public void insert (index, value) {
    for (int i = size; i > index; i--) {
       data[i] = data[i - 1];
    }
    data[index] = value;
    size++;
}
```

What are the different code models (are there multiple? is there just one?) if n = the size of the ArrayList.

Some previous quarter slides that say the stuff we talked about in a different way

Caution

Keep separate the ideas of best/worse case and O, Ω, Θ.

Big-O is an upper bound, regardless of whether we're doing worst or best-case analysis.

Worst case vs. best case is a question **once we've fixed $\boldsymbol{n}$** to choose the state of our data that decides how the code will evolve.

 What is the exact state of our data structure, which value did we choose to insert?

O, Ω, Θ are choices of how to summarize the information in the model.

	Big-O	Big-Omega	Big-Theta
Worst Case	No matter what, as n gets bigger, the code takes at most this much time	Under certain circumstances, as n gets bigger, the code takes at least this much time	On the worst input, as n gets bigger, the code takes precisely this much time (up to constants).
Best Case	Under certain circumstances, even as n gets bigger, the code takes at most this much time.	No matter what, even as n gets bigger, the code takes at least this much time.	On the best input, even as n gets bigger, the code takes precisely this much time (up to constants)

“worst input”: input that causes the code to run slowest.

	Big-O	Big-Omega	Big-Theta
Worst Case	No matter what, as n gets bigger, the code takes at most this much time	Under certain circumstances, as n gets bigger, the code takes at least this much time	On the worst input, as n gets bigger, the code takes precisely this much time (up to constants).
Best Case	Under certain circumstances, even as n gets bigger, the code takes at most this much time.	No matter what, even as n gets bigger, the code takes at least this much time.	On the best input, even as n gets bigger, the code takes precisely this much time (up to constants)

"worst input": input that causes the code to run slowest.

	Big-O	Big-Omega	Big-Theta
Worst Case	No matter what, as n gets bigger, the code takes at most this much time	Under certain circumstances, as n gets bigger, the code takes at least this much time	On the worst input, as n gets bigger, the code takes precisely this much time (up to constants).
Best Case	Under certain circumstances, even as n gets bigger, the code takes at most this much time.	No matter what, even as n gets bigger, the code takes at least this much time.	On the best input, even as n gets bigger, the code takes precisely this much time (up to constants)

"worst input": input that causes the code to run slowest.

Some more notes: Simplified, tight big-O

Why not always just say $f(n)$ is $O\left(f(n)\right)$.

It's always true! (Take $c = 1, n_0 = 1$).

The goal of big-O/Ω/Θ is to group similar functions together.

We want a simple description of f, if we wanted the full description of f we wouldn't use O

Simplified, tight big-O

In this course, we'll essentially use our complexity classes (the different orders of growth):

- Polynomials (n^c where c is a constant: e.g. $n, n^3, \sqrt{n}, 1$)
- Logarithms $\log n$
- Exponents (c^n where c is a constant: e.g. $2^n, 3^n$)
- Combinations of these (e.g. $\log(\log(n))$, $n \log n$, $\left(\log(\mathrm{n})\right)^2$)

For **this course**:

- A "tight big-O" is the slowest growing function among those listed.
- A "tight big-Ω" is the fastest growing function among those listed.
- (A Θ is always tight, because it's an "equal to" statement)
- A "simplified" big-O (or Omega or Theta)
  - Does not have any dominated terms.
  - Does not have any constant factors – just the combinations of those functions.