COMP 282
Lecture 02: Asymptotic Analysis

Mahdi Ebrahimi

Announcement

- HW 01 (Linked List Review) due Thursday 7/16 11:59pm

Lecture Outline

- Overview: Algorithmic Analysis
- Code Modeling
- Asymptotic Analysis
- Big-O, Big-Omega, Big-Theta
- Case Study: Linear Search

Why Efficient Code?

- Computers are faster, have larger memories
  - So why worry about efficient code?
- And … how do we measure efficiency?

Example – Sum of First n Values

Consider the problem of summing

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$$

How would we code this?

Example – Sum of First n Values

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$$

Approach A:

```
sum = 0;
for (i = 1; i <= n; i++)
   sum = sum + i;
```

Approach B:

```
sum = 0;
for (i = 1; i <= n; i++)
   for (j = 1; j <= i; j++)
      sum = sum + 1;
```

Approach C:

```
sum = n * (n+1) / 2
```

Sum of First *n* Numbers

```
public static void main(String[] args){
      long n = 10000;
      long sum = 0;

      // Algorithm A
      for (long i = 1; i<= n; i++)
         sum = sum + i;
      System.out.println("Sum is " + sum);


      // Algorithm B
      sum = 0;
      for (long i = 1; i <= n; i++)
         for (long j = 1; j <= i; j++)
            sum = sum + 1;
      System.out.println("Sum is " + sum);


      // Algorithm C
      sum = n * (n + 1) / 2;
      System.out.println("Sum is " + sum);
   }
```

What is "best"?

- An algorithm has both time and space constraints – that is complexity
  - Time complexity
  - Space complexity
- This study is called analysis of algorithms

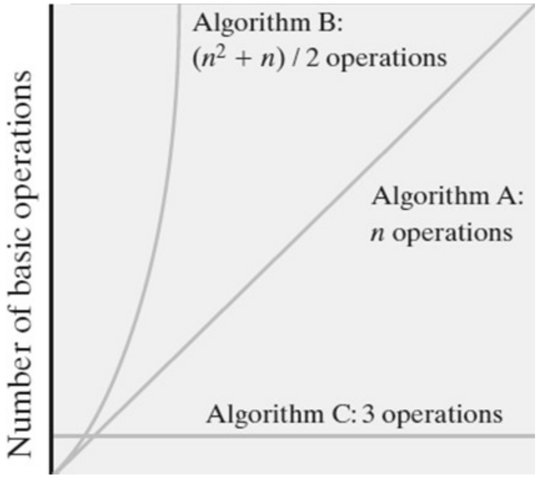
Counting Basic Operations

- A basic operation of an algorithm
  - The most significant contributor to its total time requirement
  - Number of required basic operations

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n+1)/2$	1
Multiplications			1
Divisions			1
Total basic operations	n	$(n^2+n)/2$	**3**

Counting Basic Operations

- Number of basic operations required by the algorithm



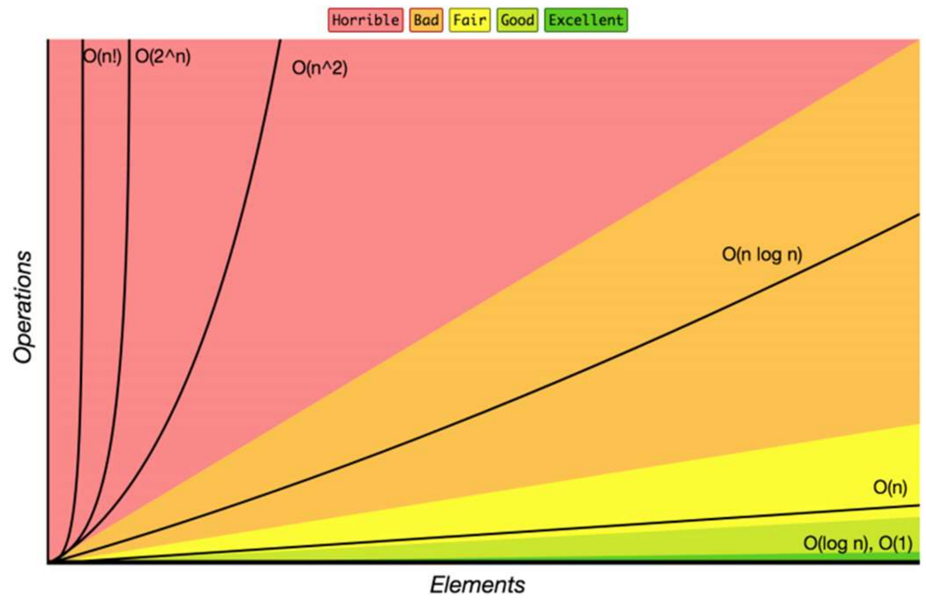
Typical growth-rate functions evaluated at increasing values of n

n	$\log(\log n)$	$\log n$	$\log^2 n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	2	3	11	10	33	10^2	10^3	10^3	10^5
10^2	3	7	44	100	664	10^4	10^6	10^{30}	10^{94}
10^3	3	10	99	1000	9966	10^6	10^9	10^{301}	10^{1435}
10^4	4	13	177	10,000	132,877	10^8	10^{12}	10^{3010}	$10^{19,335}$
10^5	4	17	276	100,000	1,660,964	10^{10}	10^{15}	$10^{30,103}$	$10^{243,338}$
10^6	4	20	397	1,000,000	19,931,569	10^{12}	10^{18}	$10^{301,030}$	$10^{2,933,369}$

Complexity Class

- **Complexity Class**: a category of algorithm efficiency based on the algorithm's relationship to the input size N

Complexity Class	Big-O	Runtime if you double N
constant	**O(1)**	**unchanged**
logarithmic	$O(\log_2 N)$	increases slightly
linear	**O(N)**	**doubles**
log-linear	$O(N \log_2 N)$	slightly more than doubles
quadratic	**$O(N^2)$**	**quadruples**
...	...	...
exponential	$O(2^N)$	multiplies drastically



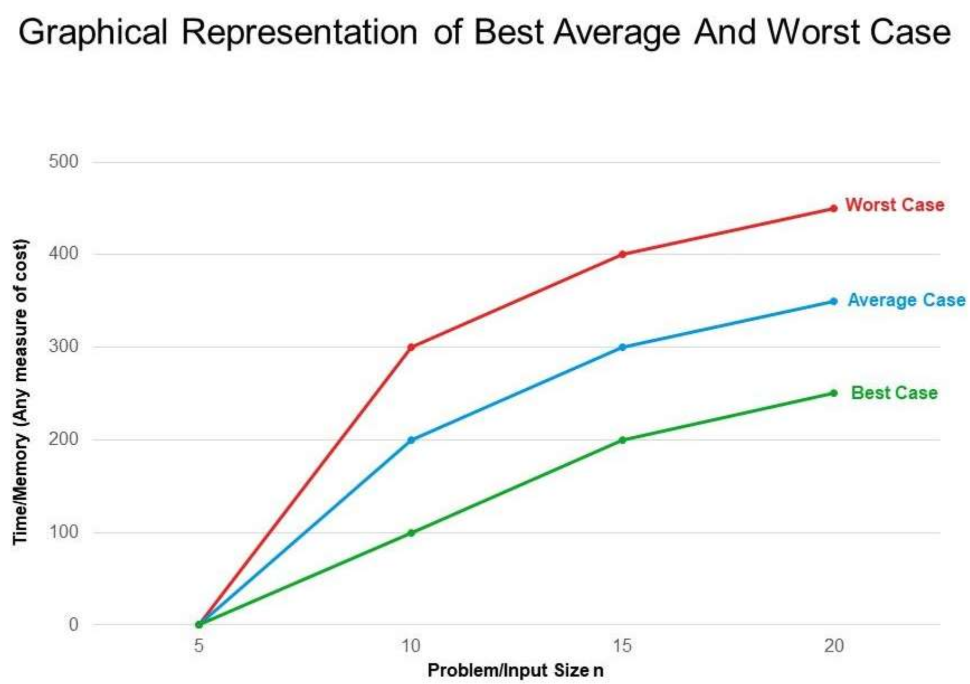
Big-Oh Analysis: Why?

	ArrayList	LinkedList
`add (front)`	O(n) linear	O(1) constant
`remove (front)`	O(n) linear	O(1) constant
`add (back)`	O(1) constant usually	O(n) linear
`remove (back)`	O(1) constant	O(n) linear
`get`	O(1) constant	O(n) linear
`Insert (anywhere)`	O(n) linear	O(n) linear

- Complexity classes help us differentiate between data structures
  - "Just change first node" vs. "Change every element" is clearly different
  - To *evaluate* data structures, need to understand impact of design decisions

Best, Worst, and Average Cases

- For some algorithms, execution time depends only on size of data set
- Other algorithms depend on the nature of the data itself
  - Here we seek to know best case, worst case, average case



Big-Oh Analysis: Why?

- We need a tool to analyze code, and we want it to be:

A **Simple**
We don't care about tiny differences in implementation, want the big picture result

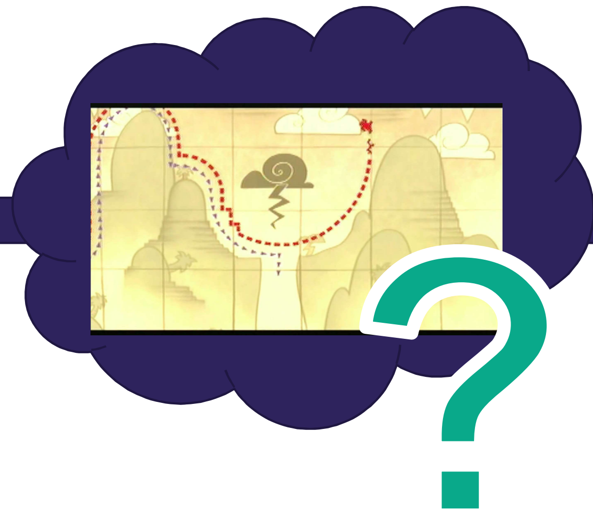
B **Mathematically Rigorous**
Use mathematical functions as a precise, flexible basis

C **Decisive**
Produce a clear comparison indicating which code takes "longer"

Big-Oh Analysis: ... How?!

CODE

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



BIG-OH

O(n)

- general patterns: “O(1) constant is no loops, O(n) is one loop, O(n^2) is nested loops”
  - This is still useful!
  - But in the future algorithm courses, you’ll go much more in depth: you will learn more about *why*, and how to handle more complex cases when they arise

Overview: Algorithmic Analysis


```
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

Talking About Code

- **Cost Model**: An analysis mindset to express the resource whose growth rate is being measured
- For simplicity, we'll discuss everything in terms of runtime today
  - But other cost models exist! For example, storage space is common
- This topic has a lot of details/relationships between concepts
  - We'll try to introduce things one at a time, but might take until next week for a "full"/satisfying picture to emerge

What is an operation?

- We don’t know exact runtime of every operation, but for now let’s try simplifying assumption: all basic operations take the same time

- Basics:
  - +, -, /, *, %, ==
  - Assignment
  - Returning
  - Variable/array access

- Function Calls
  - Total runtime in body
  - Remember: new calls a function (constructor)
- Conditionals
  - Test + time for the followed branch
    - Learn how to reason about branch
- Loops
  - Number of iterations * total runtime in condition and body

Code Modeling Example I

```
public void method1(int n) {
    int sum = 0;  +1
    int i = 0;  +1
    while (i < n) {  +1
        sum = sum + (i * 3);  +3
        i = i + 1;  +2
    }
    return sum;  +1
}
```

Loop runs n times

+6 *n

$f(n) = 6n + 3$

Code Modeling Example II

```
public void method2(int n) {
    int sum = 0;  +1
    int i = 0;  +1
    while (i < n) {  +1
        int j = 0;  +1
        while (j < n) {  +1
            if (j % 2 == 0) {  +2
                // do nothing
            }
            sum = sum + (i * 3) + j;  +4
            j = j + 1;  +2
        }
        i = i + 1;  +2
    } return sum;  +1
}
```

This inner loop runs n times

+9 *n

This outer loop runs n times

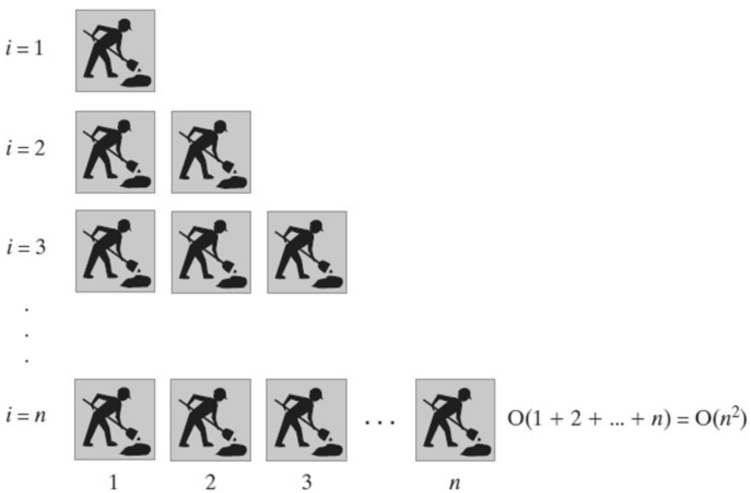
9n + 3 *n

f(n) = (9n+3)n + 3

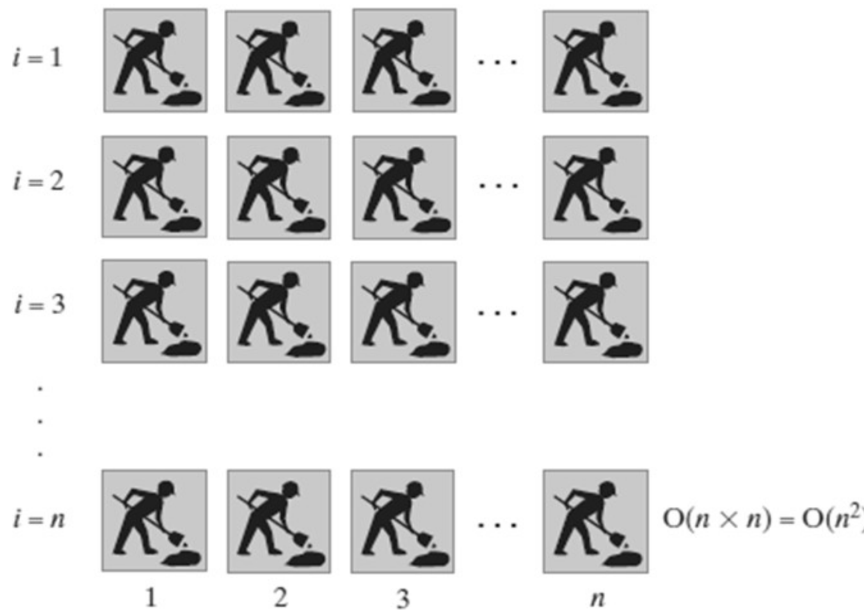
Picturing Efficiency – O(n) vs O(n^2)

```
for i = 1 to n
   sum = sum + i
```


```
for i = 1 to n
{  for j = 1 to i
       sum = sum + 1
}
```




```
for i = 1 to n
{  for j = 1 to n
       sum = sum + 1
}
```



Where are we?

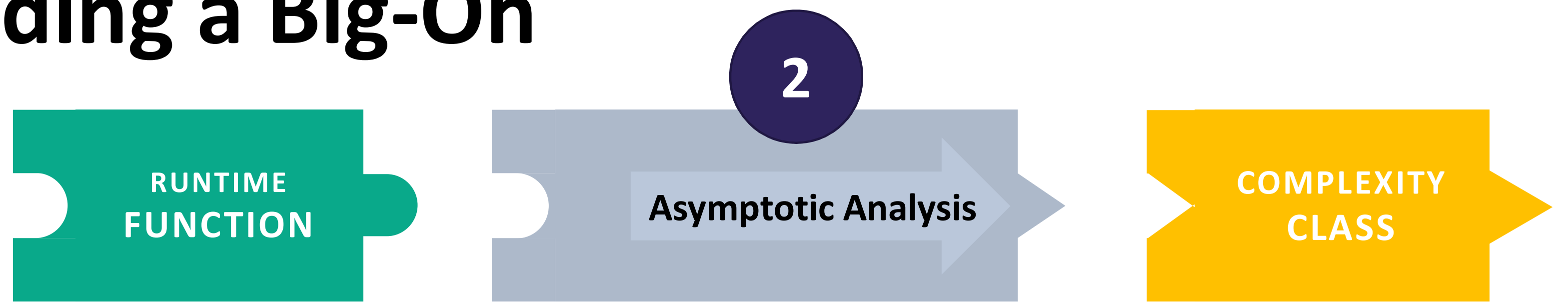

```
for (i = 0; i < n; i++) {
  a[i] = 1;
  b[i] = 2;
}
```

f(n) = 2n

O(n)

- We just turned a piece of code into a function!
  - We'll look at better alternatives for code modeling later
- Now to focus on step 2, asymptotic analysis

Finding a Big-Oh



- We have an expression for $f(n)$. How do we get the $O()$ that we've been talking about?

1. Find the "dominating term" and delete all others.
   - The "dominating" term is the one that is largest as n gets bigger. In this class, often the largest power of n.
2. Remove any constant factors.

$f(n) = (9n+3)n + 3$

$= 9n^2 + 3n + 3$

$\approx 9n^2$

$\approx n^2$

$f(n)$ is $O(n^2)$

Is it okay to throw away all that info?

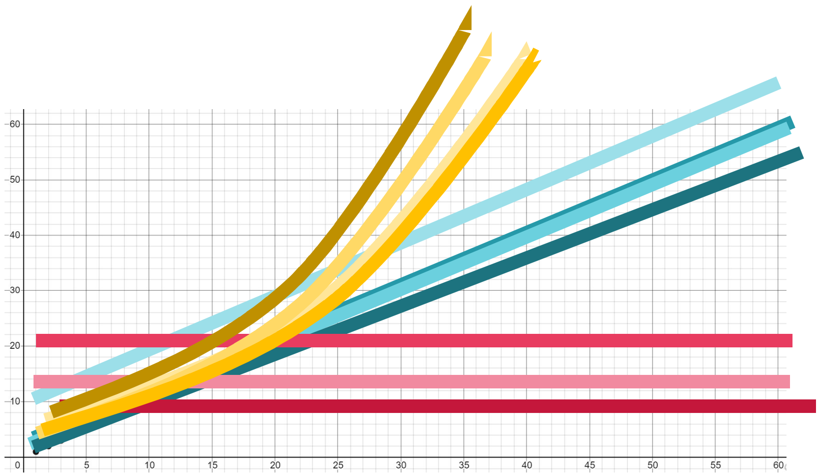
- Big-Oh is like the “significant digits” of computer science
- **Asymptotic Analysis**: Analysis of function behavior as its input approaches infinity
  - We only care about what happens when n approaches infinity
  - For small inputs, doesn’t really matter: all code is “fast enough”
  - Since we’re dealing with infinity, constants and lower-order terms don’t meaningfully add to the final result. The highest-order term is what drives growth!

Remember our goals:

A **Simple**
We don’t care about tiny differences in implementation, want the big picture result

C **Decisive**
Produce a clear comparison indicating which code takes “longer”

No seriously, this is really okay?



- There are tiny variations in these functions (2n vs. 3n vs. 3n+1)
  - But at infinity, will be clearly grouped together
  - We care about which *group* a function belongs in

- Let's convince ourselves this is the right thing to do:
  - https://www.desmos.com/calculator/t9qvn56yyb

Using Formal Definitions

- If analyzing simple or familiar functions, don't bother with the formal definition. You *can* be comfortable using your intuition!
- You're going to be making more subtle big-O statements in future classes like COMP 482.
  - We need a mathematical definition to be sure we know exactly where we are.
- You're going to learn how to use the formal definition, so if you come across a weird edge case, you know how to get your bearings.

B

Mathematically Rigorous
Use mathematical functions as a precise, flexible basis

Big-Oh Definition

- We wanted to find an upper bound on our algorithm's running time, but
  - We only care about what happens as n gets large.
  - We don't want to care about constant factors.

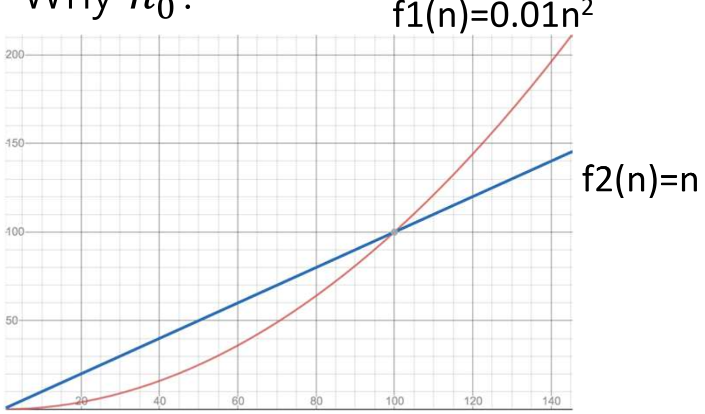
Big-Oh

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

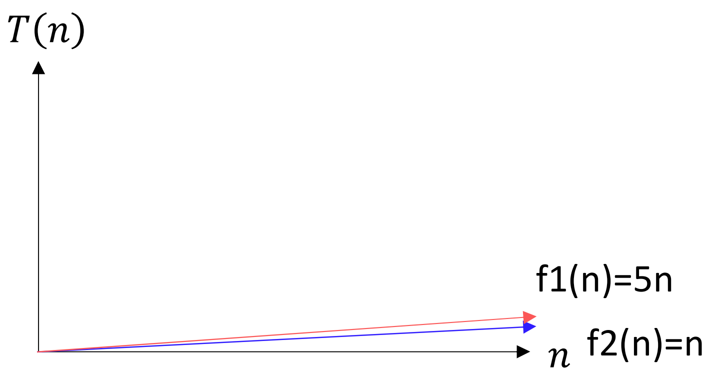
$$f(n) \leq c \cdot g(n)$$

We also say that $g(n)$ "dominates" $f(n)$

Why n_0?



Why c?



Big-Oh Proofs

Big-Oh

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

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

Show that $f(n) = 10n + 15$ is $O(n)$.

Apply definition term by term

$10n \leq c \cdot n$ when $c = 10$ for all values of n. $So\ 10n \leq 10n$

$15 \leq c \cdot n$ when $c = 15\ for\ n \geq 1.\ So\ 15 \leq 15n$

Add up all your truths

$10n + 15 \leq 10n + 15n = 25n$ for $n \geq 1$

10n + 15 <= 25n for n >= 1.

which is in the form of the definition

f(n) <= c * g(n)

where c = 25 and n_0 = 1.

Uncharted Waters: Prime Checking

```
boolean isPrime(int n) {
    int toTest = 2;  +1
    while(toTest < n) {  +1
        if (n % toTest == 0) {  +2
            return true;  +1
        } else {                      ~+5   *?
            toTest += 1;  +2
        }
    }
    return false;  +1
}
```

- Find a model $f(n)$ for the running time of this code on input n $\rightarrow$ 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

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"**
  - 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 $n \geq n_0$)

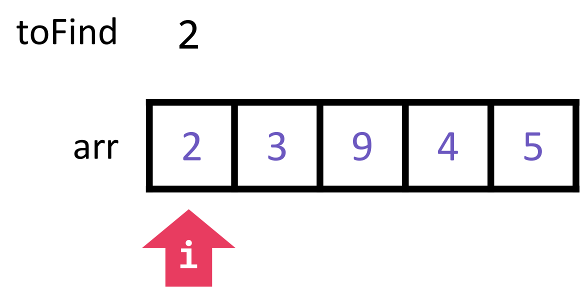
$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

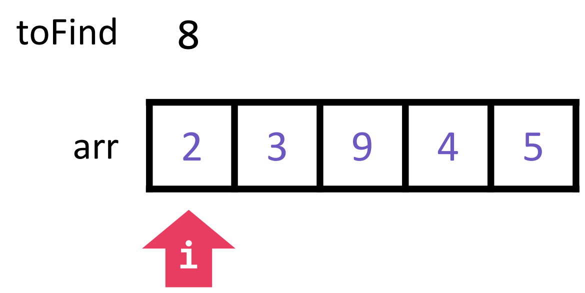
Case Study: Linear Search

- 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++) {   +2
      if (arr[i] == toFind) {   +1
         return i;   +1
      }
   }
   return -1;   +1
}
```

+4 *?

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

When would that happen?

- Suppose the loop runs n times?
  - $f(n) = 4n + 1$
- Suppose the loop only runs once?
  - $f(n) = 4$

toFind not present

These are key!

toFind at beginning

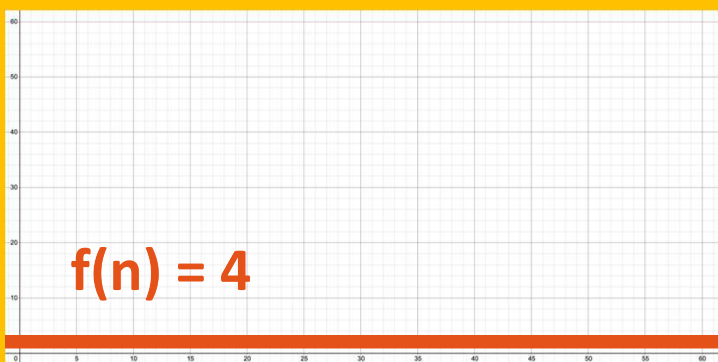
Best Case
On Lucky Earth

toFind 2

arr

2	3	9	4	5

i



After asymptotic analysis:

$O(1)$ $\Theta(1)$ $\Omega(1)$

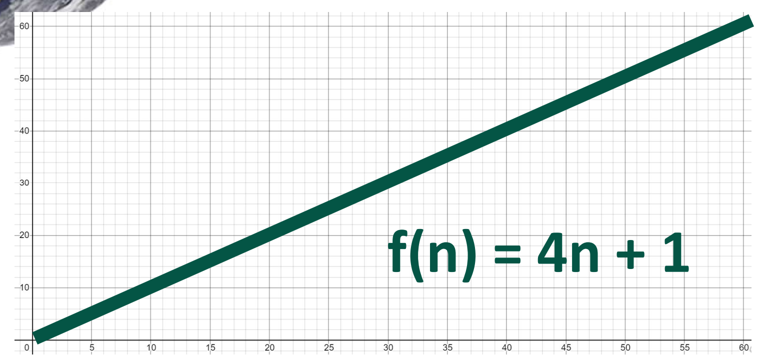
Worst Case
On Unlucky Earth

toFind 8

arr

2	3	9	4	5

i



Effect of Doubling the Problem Size

Growth-Rate Function for Size n Problems	Growth-Rate Function for Size $2n$ Problems	Effect on Time Requirement
1	1	None
$\log n$	$1 + \log n$	Negligible
n	$2n$	Doubles
$n \log n$	$2n \log n + 2n$	Doubles and then adds $2n$
n^2	$(2n)^2$	Quadruples
n^3	$(2n)^3$	Multiplies by 8
2^n	2^{2n}	Squares

Time Required To Process One Million Items

Growth-Rate Function *g*	**$g(10^6) / 10^6$**
$\log n$	0.0000199 seconds
n	1 second
$n \log n$	19.9 seconds
n^2	11.6 days
n^3	31,709.8 years
2^n	$10^{301,016}$ years

Rate is one million operations per second

YouTube Videos

Algorithms
Abdul Bari

https://www.youtube.com/watch?v=0IAPZzGSbME&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O

- Videos #:
  - 1,
  - 1.1, 1.2, **1.3 (start from 2:00)**, 1.4,
  - 1.5.1 – 1.5.3,
  - 1.6,
  - 1.7,
  - 1.8.1 - 1.8.2,
  - 1.9,
  - 1.10.1 - 1.10.2,
  - 1.11

End!

Algorithm Analysis

- Empirical vs. theoretical
- Space vs. time
- Worst case vs. Average case
- Upper, lower, or tight bound
- Determining the runtime of programs
- What about recursive programs?