Algorithmic Complexities

Lecture 7, Feb 10

601.**226** Data Structures- Spring 2020

Motivation

For a given algorithm, we want to quantify how the algorithm's running time grows as the input (of size n) grows.

Worst-case Analysis

Usually, we are interested in knowing the **worst-case running time**, as it provides a guaranteed upper-bound on the runtime.

Runtime is not “time”!

We want to produce a notion of running time which is independent of features such as processor speed, etc.

Counting the steps!

A step in an algorithm refers to performing a **basic operation** like assigning a value to a variable, performing arithmetic/logical operation, looking up one entry in an array, etc.

Put an example here of a simple two layered for loop to get an explicit intuitive understanding of how to count the steps. Then students complete the clicker question, which is a two layered for loop.

CLICKER QUESTION #1

Goal

Given an algorithm (in pseudo-code) such as the Linear search algorithm, specify the running time (in steps) as a function of the input size n (size of the search space).

Linear Search

```
define linearSearch (list[], target)
    index gets 0
    for each value in list
        if value equals target
            return index
        otherwise
            add 1 to index
    return NOT_FOUND
```

Linear Search

```
define linearSearch (list[], target)
    index gets 0                           // 1
    for each value in list                 // N
        if value equals target             // N
            return index                   // 1
        otherwise                          // N (or 0?)
            add 1 to index                 // N
    return NOT_FOUND                       // 1
```

Total: $4N + 3$

Get rid of this and ask students as a clicker question.
By the way, 4N + 3 is wrong after reviewing with
Joanne in more detail.

CLICKER QUESTION #2 Correct the answer to be 3N + 2

Worksheet

Do #1

Asymptotic analysis of code.

How costly is a “step”?

As our pseudo-code provides a high-level description of the algorithm, a step in it may correspond to K low-level machine instructions when an implementation of the algorithm is compiled on a computer with a particular architecture.

How costly is a “step”?

- Linear search total steps: $T(N) = 4N + 3$
- Each “step” is
  - 25 low-level machine instruction: $T(N) = 100N + 75$
  - 38 low-level machine instruction: $T(N) = 152N + 114$

Preferably, we would want to state that the running time grows like N (up to constant factors).

Emphasize the notion that sometimes we aren’t even sure exactly what we are counting. Thus, we want a more general picture (Big O) when we are counting steps. This is why asymptotic analysis is good.

The need for asymptotes!

To measure and compare the running time of algorithms in a way that is independent of the hardware and a good reflection of the features of the algorithm on pseudo-code level.

Could move this slide before the previous two slides

Asymptotic order of growth

Given an algorithm in pseudo-code which requires at most $T(n)$ steps to complete for an input of size $n \in \mathbb{N}$, $T : \mathbb{N} \to \mathbb{R}_+$

Asymptotic upper bounds O (big o/oh)

O(f(n)): The set of functions that grow **no faster** than f(n)

Asymptotic upper bounds O (big o/oh)

- $T(n) \in O(f(n))$, i.e., $T(n)$ is in $O(f(n))$
- $T(n) = O(f(n))$, i.e., $T(n)$ is $O(f(n))$

Both means the same thing, while the latter is a slight abuse of notation.

CLICKER QUESTION #3

Asymptotic upper bounds O (big o/oh)

- Find a function $f : N \rightarrow R_+$ such that

$$T(n) \leq c \cdot f(n), \;\; \forall n \in N, n \geq n_0 , c \in R$$

- In English: find a function $f : N \rightarrow R_+$ which, if multiplied by a constant, positive factor $c \in R$, $c > 0$, provides an **upper bound** to $T(n)$ for sufficiently large n, i.e. for all $n \in N$, $n \geq n_0 \in N$.

Get rid of this “in English” definition.
Add: “We’re looking for a function that can be multiplied by a positive, real, constant so that the product c*f(n) is always greater than or equal to the true run time.”

Asymptotic upper bounds O (big o/oh)

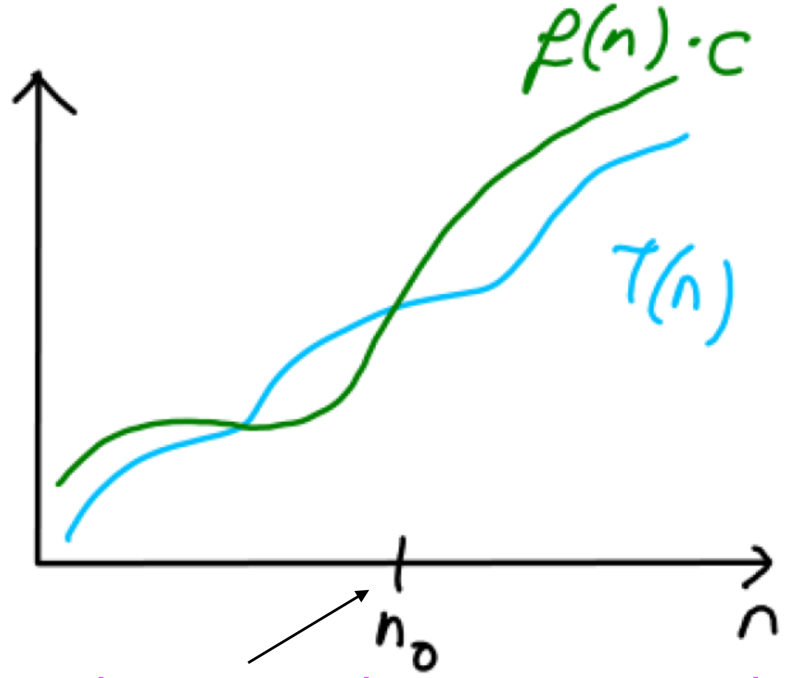
- Find a function $f : \mathbb{N} \rightarrow \mathbb{R}_+$ such that

$$T(n) \leq c \cdot f(n),$$

$$\forall n \in \mathbb{N},\ n \geq n_0,\ c \in \mathbb{R}$$

Note: c is a constant,

i.e. does not depend on n.



This is the point where our upper bound stays our upper bound

Example: $T(n) = 7n^2 + 5$

Would $f(n) = n^2$ work, i.e. $T(n) = O(n^2)$?

Rewrite this equation stuff nicer

$$T(n) = 7n^2 + 5 \leq 7n^2 + 5n^2 = 12n^2 \text{ for } \forall n \in N, n \geq 1$$

$$\Rightarrow T(n) \leq c \cdot f(n) \text{ for } \forall n \in N, n \geq n_0 = 1 \text{ and } c = 12 \in R$$

Make sure the wording of the question re-enforces good habits of thinking about these functions as elements in a set

CLICKER QUESTION #4
—> have two answers that are correct so that students understand that there can be more than one solution for n_0 and c

Knowing the definition, now we can write proofs for big-Oh.

The key is finding n_0 and c

The choice of n_0 and c is not unique.

Example: $T(n) = 7n^2 + 5$

Would $f(n) = n^3$ work, i.e. $T(n) = O(n^3)$?

$T(n) = 7n^2 + 5 \leq 7n^2 + 5n^2 = 12n^2 \leq 12n^3$ for $\forall n \in N, n \geq 1$

$\Rightarrow T(n) \leq c \cdot f(n)$ for $\forall n \in N, n \geq n_0 = 1$ and $c = 12 \in R$

Note: $O(\cdot)$ only expresses an upper bound but does not necessarily provide a precise description of the worst-case running time, i.e. a tight upper bound.

Add: If the growth rate of a function, f(n), is greater than the growth rate of the most dominant term of the actual run-time, then that function can act as an upper bound, but it's not necessarily a tight upper bound.

Asymptotic lower bounds Ω (greek "Omega")

Ω(f(n)): The set of functions that grow **no slower** than f(n)

Asymptotic lower bounds Ω (greek "Omega")

- Find a function $f : N \rightarrow R_+$ such that

$$T(n) \geq c \cdot f(n), \forall n \in N, n \geq n_0 , c \in R$$

- In English: find a function $f : N \rightarrow R_+$ which, if multiplied by a constant, positive factor $c \in R$, $c > 0$, provides a **lower bound** to $T(n)$ for sufficiently large n, i.e. for all $n \in N$, $n \geq n_0 \in N$.

Get rid of this "in English" definition.
Add: "We're looking for a function that can be multiplied by a positive, real, constant so that the product c*f(n) is always less than or equal to the true run time."

Asymptotic lower bounds Ω (greek "Omega")

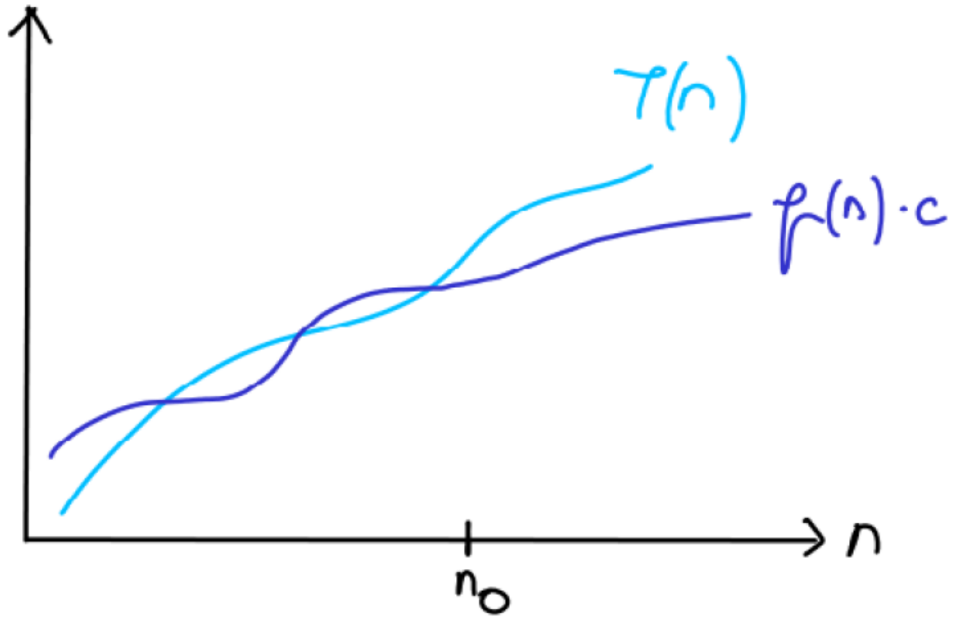
- Find a function $f : N \rightarrow R_+$ such that

$T(n) \geq c \cdot f(n)$,

$\forall n \in N, n \geq n_0 , c \in R$

Note: c is a constant,

i.e. does not depend on n.



Example: $T(n) = 7n^2 + 5$

Would $f(n) = n^2$ work, i.e. $T(n) = \Omega(n^2)$?

$T(n) = 7n^2 + 5 \geq 7n^2$ for $\forall n \in N$

$\Rightarrow T(n) \geq c \cdot f(n)$ for $\forall n \in N$, $n \geq n_0 = 0$ and $c = 7 \in R$

Example: $T(n) = 7n^2 + 5$

Would $f(n) = n^1$ work, i.e. $T(n) = \Omega(n)$?

$T(n) = 7n^2 + 5 \geq 7n^2 \geq 7n$ for $\forall n \in N, n \geq 1$

$\Rightarrow T(n) \geq c \cdot f(n)$ for $\forall n \in N, n \geq n_0 = 0$ and $c = 7 \in R$

Note: similar to O(.), Ω(.) only express a lower bound, which is not necessarily a tight bound.

CLICKER QUESTION #5

Often, for a given algorithm, we do not only want to know an upper and a lower bound to the worst-case running time of the algorithm, but we would like a **tight bound** which gives us a precise description of the algorithm's running time.

Asymptotically tight bounds Θ (greek “Theta”)

$\Theta(f(n))$: The set of functions that grow **no faster and no slower** than $f(n)$

Add: Theta will reflect the most “dominant” term in your run time. This will become evident through examples.

Asymptotically tight bounds Θ (greek “Theta”)

- Find a function f : N → R_+ such that $c'f(n) \leq T(n) \leq cf(n), \forall n \in N, n \geq n_0, \{c, c'\} \in R$

- f(n) is an asymptotically tight bound for T(n) or that T(n) is Θ(f(n)) if T(n) = O(f(n)) = Θ(f(n)), where T, f : N → R_+

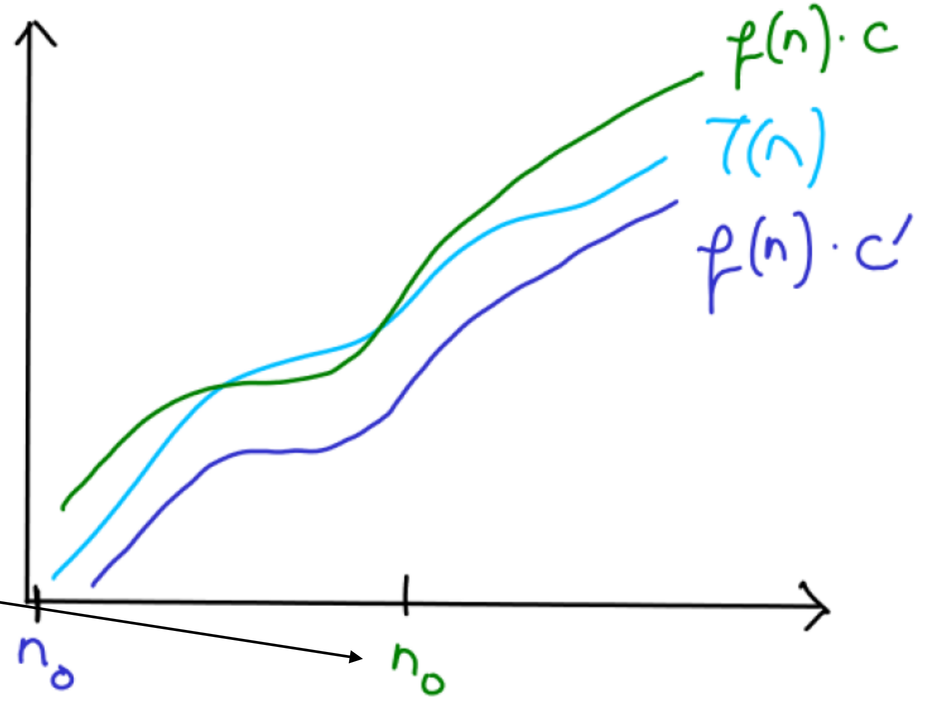
Asymptotically tight bounds Θ (greek "Theta")

- Find a function f : N → R_+ such that $c'f(n) \leq T(n) \leq cf(n), \forall n \in N, n \geq n_0, \{c, c'\} \in R$

Note: c and c' are constants,

i.e. do not depend on n.

After some number of input values, your run time is SANDWICHED between two functions that differ only in their constant coefficients



Example: $T(n) = 7n^2 + 5$

Would $f(n) = n^2$ work, i.e. $T(n) = \Theta(n^2)$?

Earlier we showed $T(n) = O(n^2)$ and $T(n) = \Omega(n^2)$ therefore $T(n) = \Theta(n^2)$

Add: Since we can sandwich 7n^2+5 in between 7n^2 and 12n^2, we see that we can trap our run time using the most dominant term of n^2.

Asymptotic Notation

Good summary! Make sure to give students enough time to absorb this.

Big Oh (upper bound)	$T(N)$ is $O(f(N))$ if there exists c, n such that $T(N) \le cf(N)$ for all $N \ge n$.
Big Omega (lower bound)	$T(N)$ is $\Omega(g(N))$ if there exists c, n such that $T(N) \ge cg(N)$ for all $N \ge n$.
Big Theta (tight bound)	$T(N)$ is $\Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

Sandwich Big Theta between O and Omega

From Functions to Complexity

What is the asymptotic complexity of this function:

$$T(N) = 3\log N^2 + (\log N)^2 + (N+1)^2 \log(4N)$$

Restate the goal: Simplify this expression as much as possible to find the most asymptotically dominant term.

From Functions to Complexity

What is the asymptotic complexity of this function:

$$
\begin{aligned}
T(N) &= 3\log N^2 + (\log N)^2 + (N+1)^2 \log(4N) \\
&= 3\times 2\log N + \log^2 N + \left(N^2 + 2N + 1\right)\left(\log 4 + \log N\right) \\
&= 6\log N + \log^2 N + \left(N^2 + 2N + 1\right)\left(2 + \log N\right) \\
&= 6\log N + \log^2 N + 2N^2 + 4N + 2 + N^2\log N + 2N\log N + 2\log N \\
&= N^2\log N + 2N^2 + 2N\log N + 4N + \log^2 N + 8\log N + 2
\end{aligned}
$$

Note that this is the most dominant term

$N^2 \log N \le T(N) \le c.N^2 \log N$ for some constant c

$T(N)$ is $\Omega(N^2 \log N)$, $O(N^2 \log N)$ and therefore $\Theta(N^2 \log N)$

Math Review (laws of exponents & logs)

$$x^{-a} = \frac{1}{x^a}$$

$$x^a x^b = x^{(a+b)}$$

$$\frac{x^a}{x^b} = x^{(a-b)}$$

$$\left(x^a\right)^b = \left(x\right)^{ab} = \left(x^b\right)^a$$

$$\log_a b = x \Leftrightarrow a^x = b$$

$$\log\left(ab\right) = \log a + \log b$$

$$\log\left(\frac{a}{b}\right) = \log a - \log b$$

$$\log\left(a^b\right) = b \log a$$

$$\log^b a = \left(\log a\right)^b \neq \log\left(a^b\right)$$

$$\log_a b = \frac{\log_c b}{\log_c a}$$

CLICKER QUESTION #6

Helpful Summation Identities

Useful for finding closed form solutions for the running times of loops

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$$

$$\sum_{i=0}^{N} 2^i = 2^{(N+1)} - 1$$

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6}$$

$$\sum_{i=0}^{N-1} c^i = \frac{1-C^N}{1-C} = \frac{C^N - 1}{C - 1}$$

For more, see

- http://en.wikipedia.org/wiki/Summation
- https://www.wolframalpha.com

From Code to Running Time

- What is the running time of this code:

```
int sum = 0;
for (int i = N; i > 0; i /= 2) {
    for (int j = N; j > 0; j -= 2) {
        sum++;
    }
}
```

From Code to Running Time

- What is the running time of this code:

```
int sum = 0;
for (int i = N; i > 0; i /= 2) {
    for (int j = N; j > 0; j -= 2) {
        sum++;
    }
}
```

`int sum = 0;`	1 (or 2?)
`int i = N;`	1 (or 2?)
`i > 0;`	log N (or log N + 1?)
`i /= 2`	log N
`int j = N`	1 * outer loop reps
`j > 0;`	(N/2 + 1) * outer reps
`j -= 2`	(N/2) * outer reps
`sum++;`	1 * inner * outer reps

From Code to Running Time

- What is the running time of this code:

```
int sum = 0;
for (int i = N; i > 0; i /= 2) {          // log N outer loops
    for (int j = N; j > 0; j -= 2) {      // N/2 inner loops
        sum++;
    }
}
```

$$T(N) = 1 + 2 + 2\log N + (2+N)\log N + {}^{N}\!/_{2}\log N = 3 + 4\log N + 1.5N\log N$$

$$\left.\begin{aligned} T(N) \le 2N\log N \Rightarrow \mathrm{O}(N\log N) \\ T(N) \ge N\log N \Rightarrow \Omega(N\log N) \end{aligned}\right\} T(N) \in \Theta(N\log N)$$

Exercise

```java
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.println(“*”);
    }
}
```

Exercise

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.println(“*”);
    }
}
```

`int i = 1;`	1
`i <= N;`	N
`i++`	N
`int j = 1`	1 * outer loop reps
`j <= i;`	depends on i
`j++`	depends on i
`print`	1 * inner * outer reps

Add: “We can see that the inner loop runs once, then twice, then three times, etc. Maybe we can use this pattern to our advantage.”

CLICKER QUESTION #7

Exercise

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.println("*");
    }
}
```

`int i = 1;`	1
`i <= N;`	N
`i++`	N
`int j = 1`	1 * outer loop reps
`j <= i;`	depends on i
`j++`	depends on i
`print`	1 * inner * outer reps

$$T(N) = \frac{N(N+1)}{2} = \Theta(N^2) \quad (\text{where } c = \frac{1}{2})$$

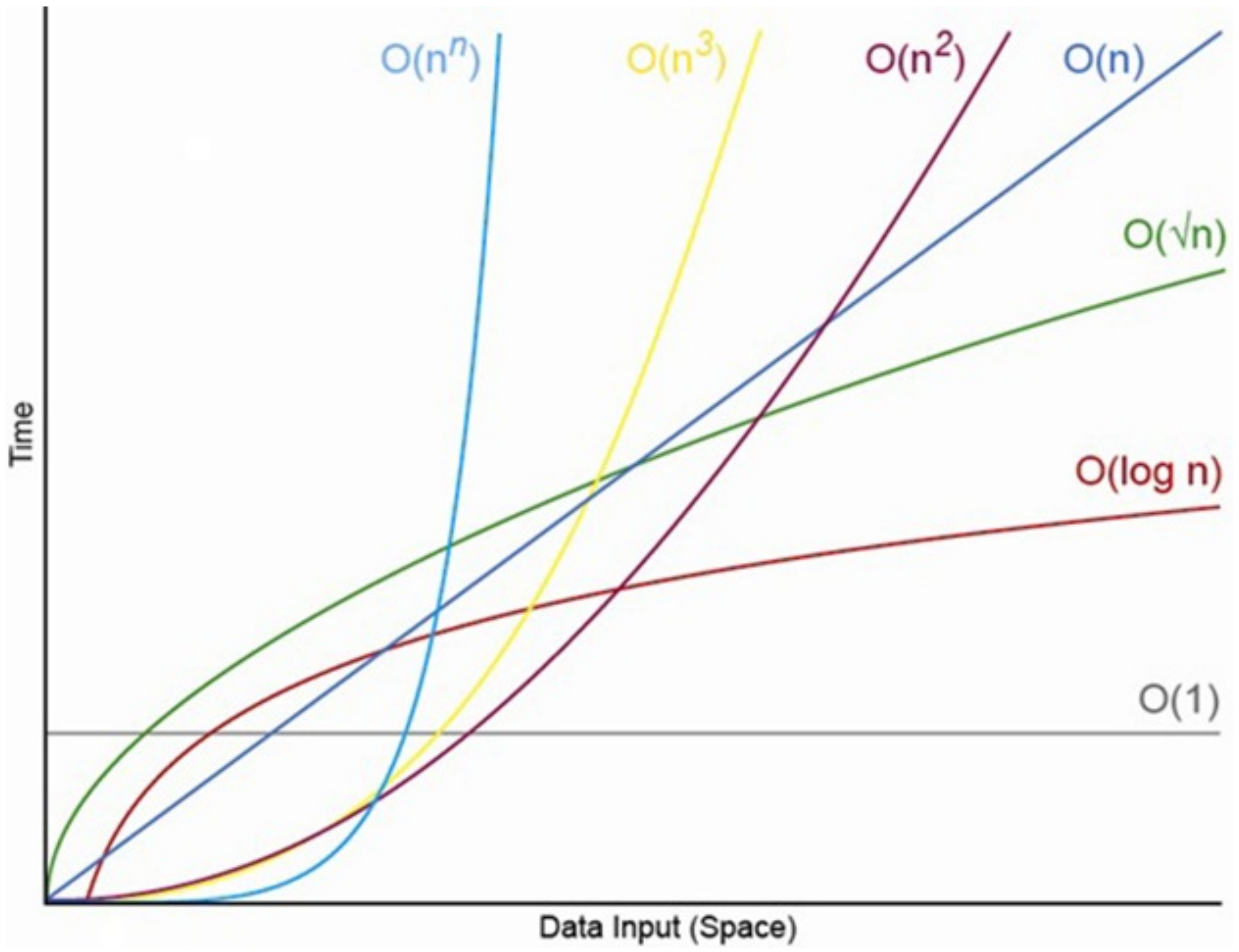
We say N^2 is a "tight bound" for this code.

Add: The tight bound can be hard to count exactly, but we can try to figure out what it is if we can
identify lower and upper bounds.
E.g. Lower is N and upper is N^2 here, so we can use that to
find theta here

Good summary! Give students enough time to absorb this. Emphasize the importance of the dominant term.

Common Running Times

- Constant: $O(1)$
- Logarithmic: $O(\log n)$
- Linear: $O(n)$
- Log-linear: $O(n \log n)$
- Quadratic: $O(n^2)$
- Polynomials: $O(n^c)$
- Exponential: $O(c^n)$
- Factorial: $O(n!)$



Reference: https://apelbaum.wordpress.com/2011/05/05/big-o/

Worksheet

Do #2

Asymptotic analysis of code.

Types of Analysis

- Worst/best case: Running time for an input of size n that makes the algorithm take the maximum/minimum number of steps.

- Average-case: Expected running time for a random input of size n.

Types of Analysis

- Amortized: Worst-case running time for any sequence of n operations.
- Probabilistic: Expected running time of a randomized algorithm.

Survey of common running times

Worksheet

Do #2.5

Asymptotic analysis of code.

Linear time: O(n)

- Running time is proportional to input size.
- Example: Computing the maximum number in a list of numbers.

Constant time: O(1)

- Running time does not depend on the input size.

- It does not mean it only takes one step.

Provide an example of a O(1) operation that takes more than one step.

Linearithmic time: O(n log n)

- Arises in divide-and-conquer algorithms.
- Example: Merge sort and heapsort.

Quadratic time: O(n^2)

- Enumerate all pairs of elements.
- Ex: Bubble/Insertion/Selection sort

Sublinear time: O(log n)

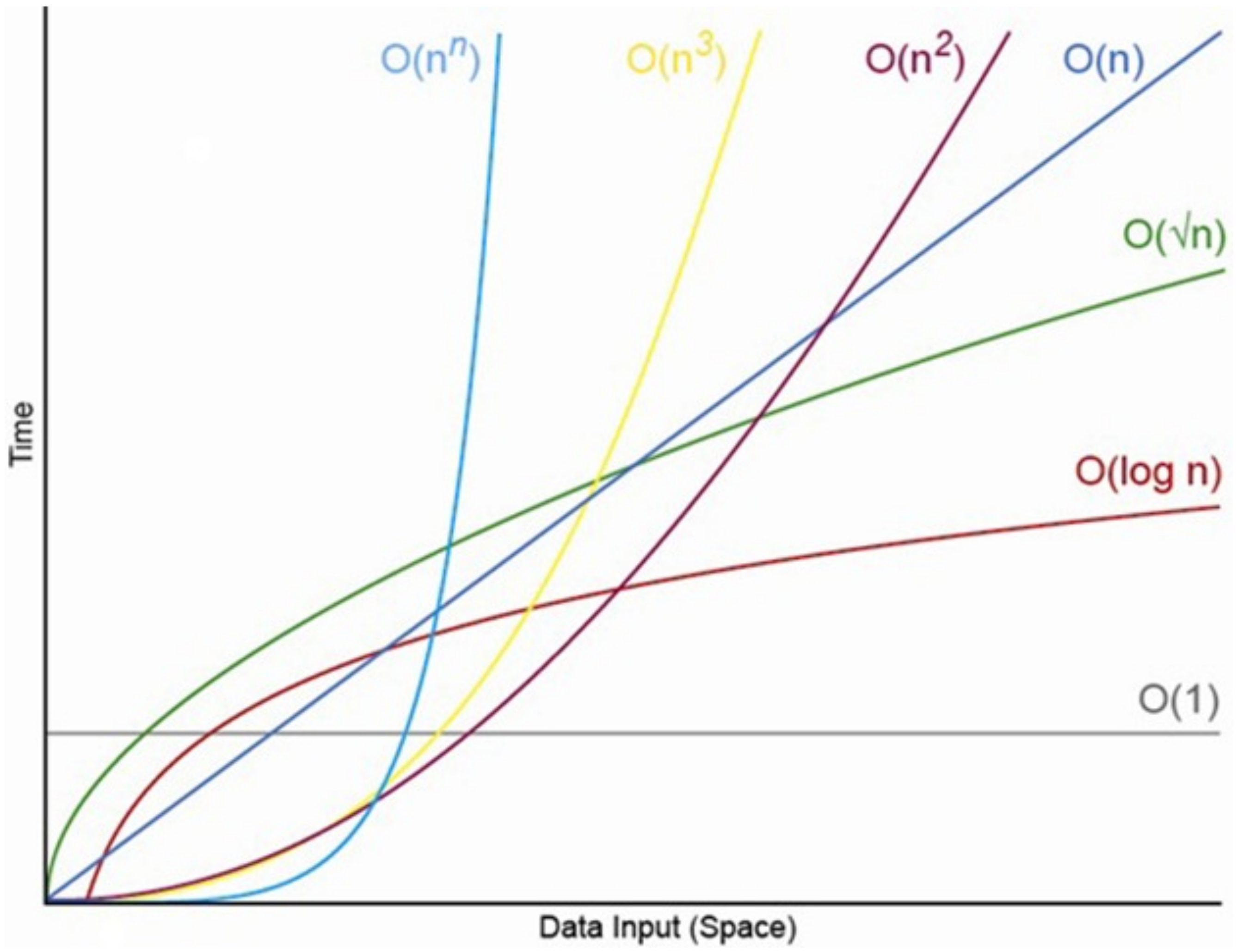
- Runtime grows slower than the size of the problem

- Example: Binary Search

Good summary! Give students enough time to absorb this. Emphasize the importance of the dominant term.

Common Running Times

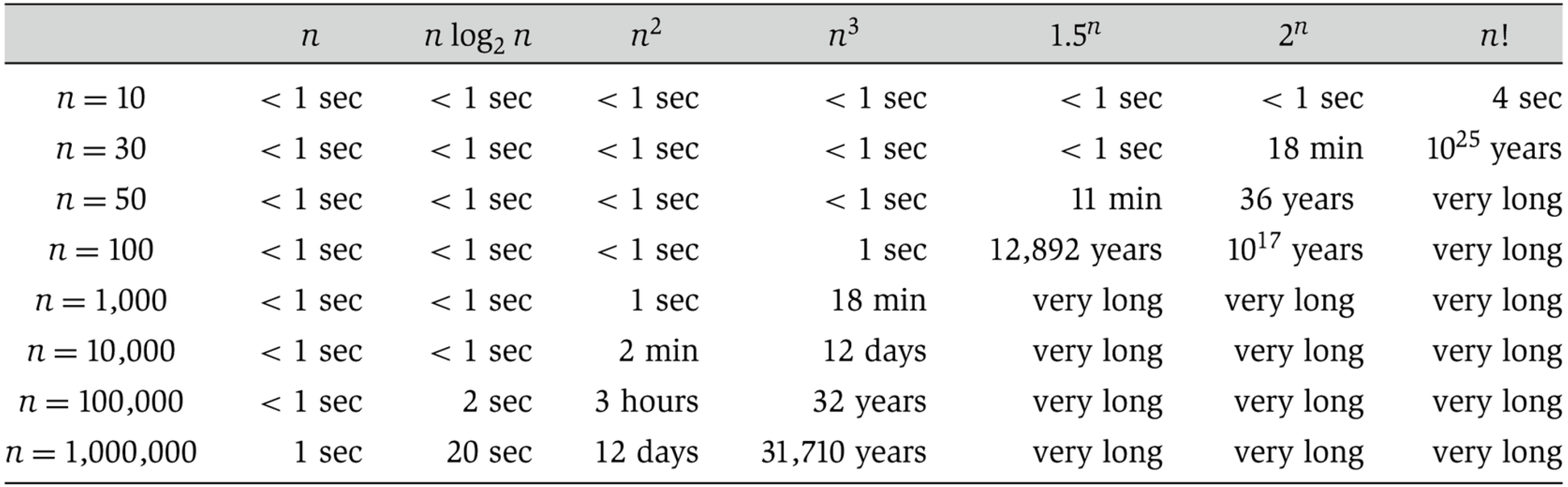
- Constant: $O(1)$
- Logarithmic: $O(\log n)$
- Linear: $O(n)$
- Log-linear: $O(n \log n)$
- Quadratic: $O(n^2)$
- Polynomials: $O(n^c)$
- Exponential: $O(c^n)$
- Factorial: $O(n!)$



Reference: https://apelbaum.wordpress.com/2011/05/05/big-o/

Why it matters?

- For a processor performing 10^6 instruction per second



	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Worksheet

Do #3

Asymptotic analysis of code.

Algorithmic efficiencies

<u>Time Complexity</u>

- how many steps will an algorithm take based on the input size?

<u>Space Complexity</u>

- how much (extra) memory will the algorithm use based on the input size?

We mostly focus on time complexities

- But we need to pay attention to space as well.
- Careful: recursive methods can be deceptively poor in space (and time).