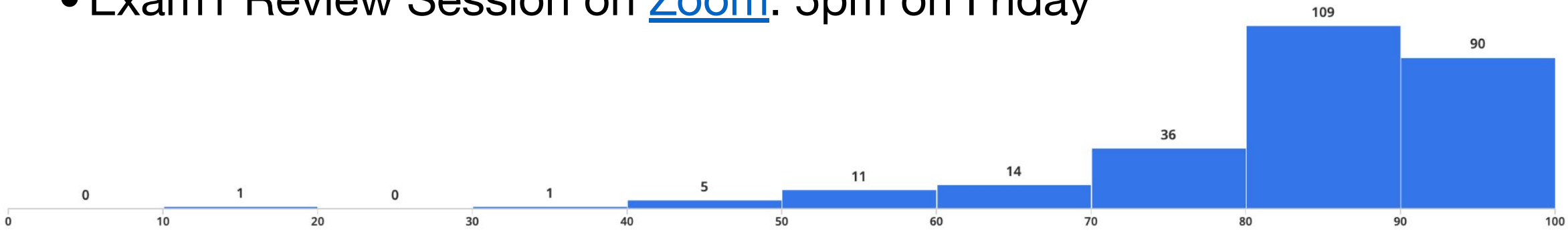
Runtime and Big-O Notation

15-110 – Monday 10/07

Announcements

- **Hw3** was due today
- **No Check4** due to fall break. Be ready to start on HW4 right after break!
- **Exam1** grades have been released! Median = 86.5; well done!
- Exam reflection due Friday at 11:59pm
- Exam1 Review Session on Zoom: 5pm on Friday



Minimum	Median	Maximum	Mean	Std Dev
19.5	86.5	99.0	83.67	--

Learning Goals

- Identify the **worst case** and **best case** inputs of functions
- Compare the **function families** that characterize different functions
- Calculate a specific function or algorithm's efficiency using **Big-O notation**

A major goal of computer scientists is not just to make algorithms that work, but algorithms that work **efficiently**.

Computers are fast, but they can still take time to do complex actions. Faster algorithms can save lives, increase company profits, and reduce user frustration.

Resources we care about:

- **Running time: how long does it take for the algorithm to run?**
- Space: how much memory does the algorithm use?
- Energy: how much energy does running it consume?
- Connectivity: how many network connection does it make?

Search Algorithm Efficiency

How can we **compare the efficiency** of linear and binary search?

To compare running time in an abstract way, we count the number of **actions/steps** it takes for the algorithm to run based on the **size of the input**.

Which steps are we counting? Simple operations on the input. For searching, this is comparisons between number we are trying to find and the list we are searching.

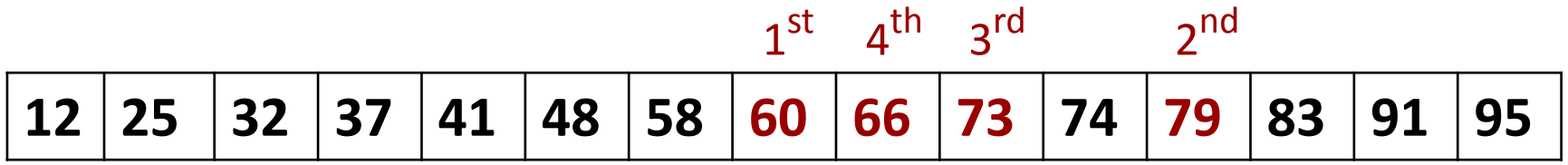
What is the size of the input? The number of items in the input.

For searching, this is the list we are searching.

How many comparisons for linear search vs. binary search to find 66?

Linear search: **9 comparisons**

Binary search: **4 comparisons**



This is for a specific input.
How do we compare running times across inputs?

Running Time: Best Case, Worst Case

We can compare algorithms based on their **best case and worst case running times**.

Best case input:

An input of size `n` that results in the algorithm taking the **least steps possible.**

Worst case input:

An input of size `n` that results in the algorithm taking the **most steps possible**.

Linear search: Best and worst case input

What is the best case for linear search?

Answer: A list where the item we search for is in the first position

What is the worst case for linear search?

Answer: A list where the item we search for is not in the list.

Binary search: Best and worst case input

You do: what's the **best case** input and **worst case** input for binary search if we are counting comparisons?

Given the best case and worst case input (size n), we can determine **how many actions it takes to run the algorithm** on those inputs.

How many actions do we perform in the **best case**?

Linear search: 1 comparison

Binary search: 1 comparison

How many actions do we perform in the **worst case**?

Lineary search: n comparisons (we have to check every element)

Binary search: ??

How many actions do we perform in the **worse case in binary search**?

List size	Number of recursive calls
1	1
2^2-1 = 3	2
2^3-1 = 7	3
2^4-1 = 15	4
2^5-1 = 31	5
2^k -1	k
n	$\log_2(n)$

Each recursive call does 1 comparison, so **how many recursive calls**?

When the input length doubles for linear search, it does twice as many comparisons.

But, **when the input length doubles for binary search, it does just one more comparison!**

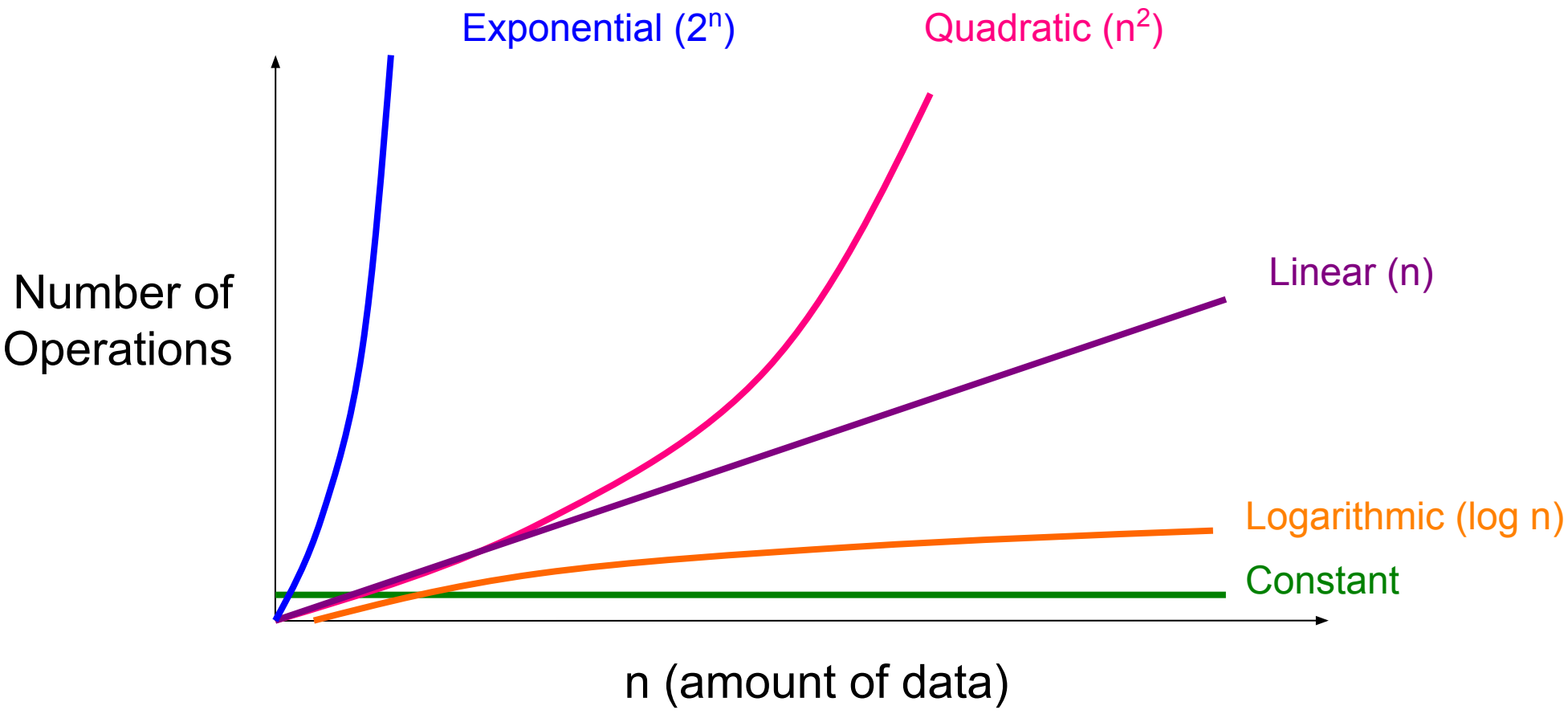
Function Families

When comparing algorithms, we care about **how the number of steps grows with the size of the input**.

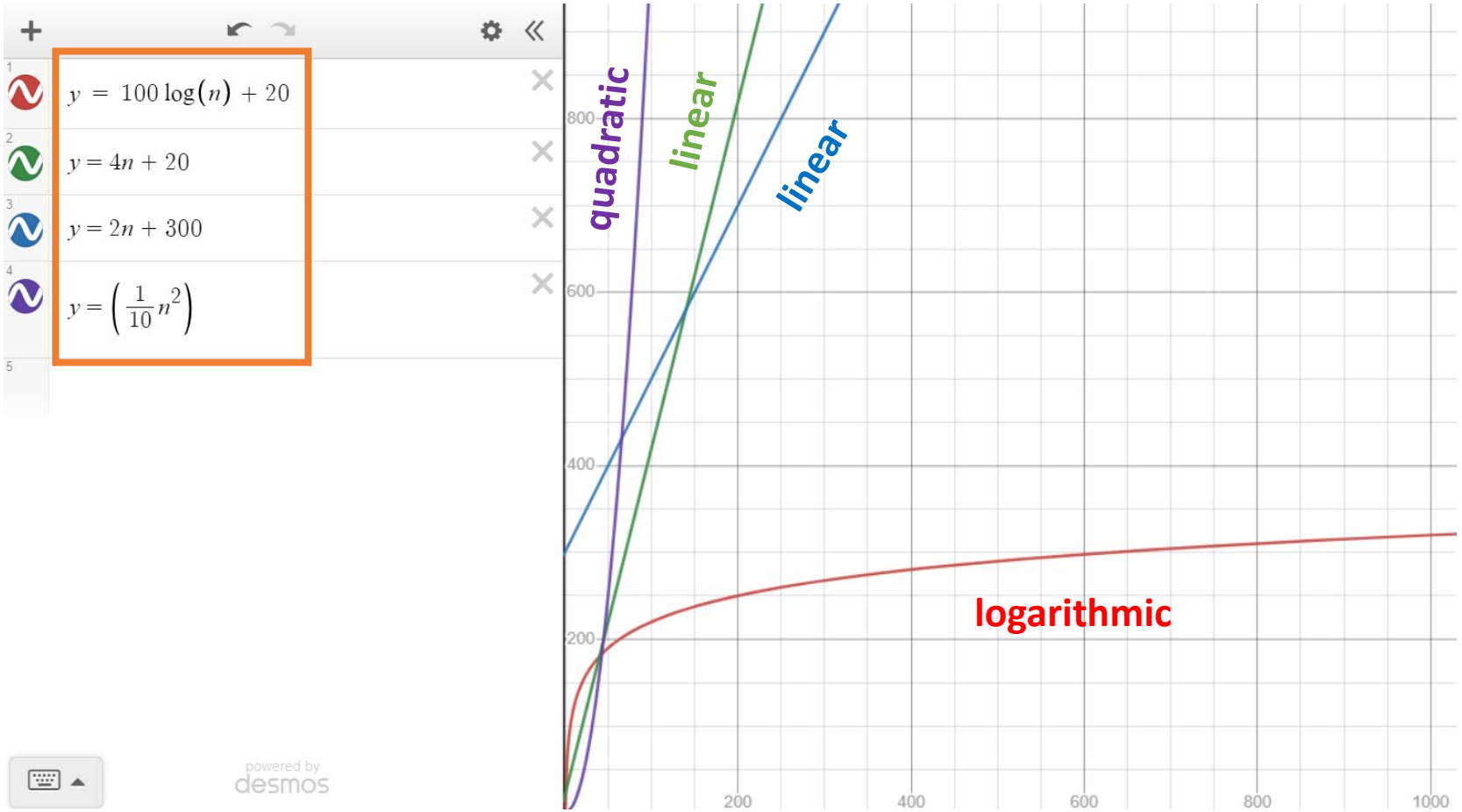
In math, a **function family** is a set of equations that all grow at the same rate as their inputs grow. For example, an equation might grow linearly or quadratically.

When determining which function family represents the actions taken by an algorithm, we say that **n is the size of the input**. For a list, that's the number of elements; for a string, the number of characters.

Common Function Families



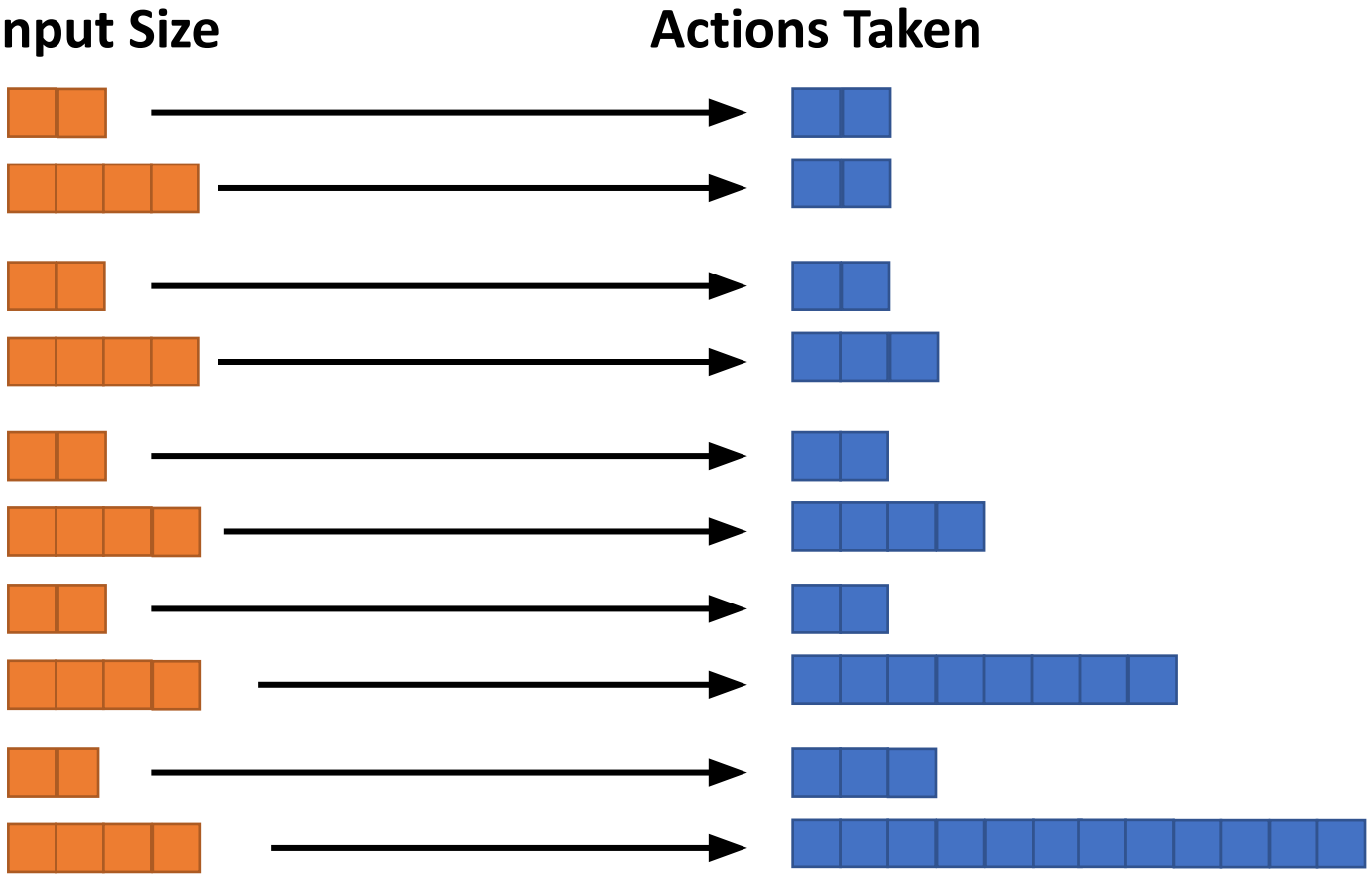
Function Families and Constants



Notice that as n grows, the function family becomes much more important than the constants, and functions with the same function family behave similarly.

Alternate Visualization

Here's another way to think about the function families. Consider what happens when you **double** the size of the input.



Constant — **double input, no change in actions**

Logarithmic — **double input, +1 action**

Linear — **double input, double actions**

Quadratic — **double input, quadruple actions**

Exponential — **double input, many many more actions!**

Big-O Notation

Big-O Notation: When determining an algorithm's running time, we **denote the function family and ignore constant factors**.

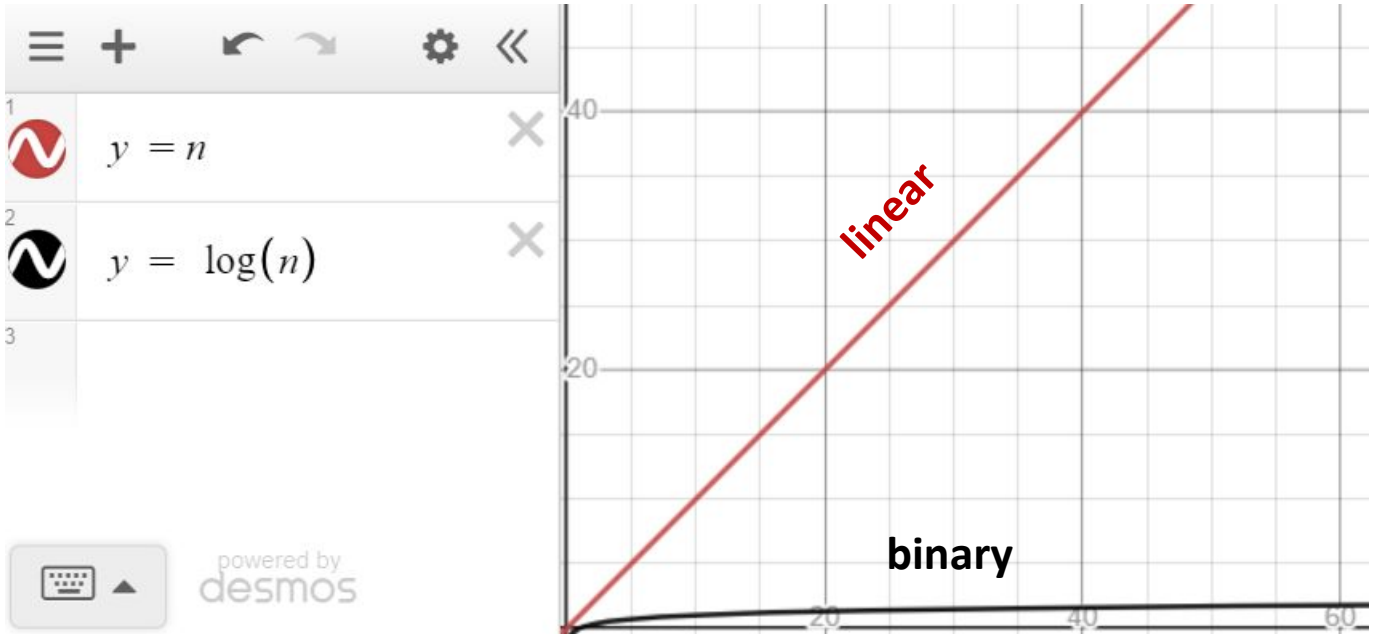
When we determine a program or algorithm runtime, we ignore constant factors and smaller terms. All that matters is the function family, the dominant term (highest power of n). That is the idea of Big-O notation.

f(n)	Big-O
n	O(n)
32n + 23	O(n)
$5n^2 + 6n - 8$	$O(n^2)$

Unless specified otherwise, the Big-O of an algorithm refers to its runtime in the **worst case** (computer scientists are pessimists).

Caveat: this is a simplified definition. If you take other CS classes, you'll learn more about how Big-O actually works.

Big-O for Linear Search vs. Binary Search



Linear search:

Double the input, double the comparisons:

O(n)

Binary search:

Double the input, +1 comparisons:

O(log n)

Binary search is incredibly fast. Linear search is much slower in the worst case!

Calculating Big-O Runtimes

If there are statements that **run sequentially**, the runtime for each statement is **added to the total runtime**.

```
def example(x): # n = x
    x = x + 5 # O(1)
    y = x + 2 # O(1)
    print(x, y) # O(1)
# Total: O(1)
```

Note: If a step's runtime does not depend on the size of the input, it is O(1)!

Conditionals run sequentially.

```
def firstOrLast(s): # n = len(s)
    if len(s) % 2 == 0: # O(1)
        return s[0] # O(1)
    else: # O(1)
        return s[len(s)-1] # O(1)
# Total: O(1)
```

If there are functions, need to **add that function's runtime** to the overall runtime. (It may not be O(1)!)

```
def printContains(lst, x): # n = len(lst)
    result = linearSearch(lst, x) # O(n)
    print(x, "in lst?", result) # O(1)
# Total: O(n)
```

Loops repeat actions, so **multiply the runtime of the loop body** by the number of times the loop repeats*.

```
def addThemAll(lst): # n = len(lst)
    result = 0 # O(1)
    for i in range(len(lst)): # n repetitions
        result = result + lst[i] # O(1)
    return result # O(1)
# Total: O(n)
```

* caveat: multiplying only works if the work done in each loop is constant each iteration

Some **built-in Python functions and operations** have non-constant runtimes!

```
def countAll(lst): # n = len(lst)
    for i in range(len(lst)): # n repetitions
        count = lst.count(i) # O(n)
        print(i, "occurs", count, "times") # O(1)
# Total: O(n^2)
```

We will usually let you know on assignments and exams when a built-in method/functions or operation is not constant time.
Question: What is the runtime of the `in` operator? `x in lst`

Common Big-O Runtimes

O(1) is Constant Time

```
def swap(lst, i, j): # n = len(lst)
    tmp = lst[i]
    lst[i] = lst[j]
    lst[j] = tmp
```

Does the runtime of this algorithm depend on the number of items in the list?

Answer: No.

This algorithm is **constant time** or **O(1)**; its time does not change with the size of the input.

O(log n) is Logarithmic Time

```
def countDigits(n): # n = n
    count = 0
    while n > 0:
        n = n // 10
        count = count + 1
    return count
```

Every time you increase n by a factor of 10, you run the loop one more time. All the operations in the loop are constant time. Similar to binary search, the algorithm is **logarithmic time**, or **O(log n)**.

Multiplying input by 10, adds one more iteration to the loop.

Even though this is $\log_{10}(n)$, we don't include the base in the Big-O notation because a change of base is just a multiplicative factor.

O(n) is Linear Time

```
def countdown(n): # n = n
    for i in range(n, -1, -5):
        print(i)
```

This code will loop n/5 times overall. If we double the size of n, how many more times do we go through the loop?

Answer: We double the number of times through the loop. That is **linear time**, or **O(n)**, as it is proportional to the size of n. Stepping by 5 doesn't change the function family.

$O(n^2)$ is Quadratic Time

```
def multiplicationTable(n): # n = n
    for i in range(1, n+1):
        for j in range(1, n+1):
            print(i, "*", j, "=", i*j)
```

This seems tricky at first, but note that **every iteration** of the outer loop will do **all the work** of the inner loop.

The inner loop does n total iterations (with O(1) work in its body). This is repeated n times by the outer loop. Therefore, the entire runtime is $O(n^2)$.

$O(2^n)$ is Exponential Time

```
def move(start, tmp, end, num): # n = num
    if num == 1:
        return 1
    else:
        moves = 0
        moves = moves + move(start, end, tmp, num - 1)
        moves = moves + move(start, tmp, end, 1)
        moves = moves + move(tmp, start, end, num - 1)
        return moves
```

This is Towers of Hanoi. Every time we add 1 disc we double the number of moves. That's **exponential time**, or **$O(2^n)$**.

$O(2^{n+1}) = O(2^n) + O(2^n)$

For Recursion, Look at the Number of Calls

Is all recursion exponential? Not necessarily! It depends on the **number of recursive calls** the function will need to make.

```
def countdown(n): # n = n
    if n <= 0:
        print("Finished!")
    else:
        print(n)
        countdown(n - 5)
```

Consider the example above. If you call the function on 100, it will make the next call on 95, then 90, etc; 20 total calls will be made. If you double the input, 40 calls will be made. The function is O(n).

Activity: Calculate the Big-O of Code

Activity: predict the Big-O runtime of the following piece of code.

```
def sumEvens(lst): # n = len(lst)
    result = 0
    for i in range(len(lst)):
        if lst[i] % 2 == 0:
            result = result + lst[i]
    return result
```

[if time] Complex Big-O Example

Let's look at a more complex example together:

```
1: def example(lst): # n is len(lst)
2:     result = []
3:     for i in range(0, len(lst), 2):
4:         if lst[i] != lst[i+1]:
5:             average = (lst[i] + lst[i+1]) / 2
6:             if average in lst:
7:                 result.append(average)
8:     return count
```

Runtime: constant + n/2 * (constant + constant + n + constant) = constant + constant * (n^2) + constant * n = **O(n^2)**

Line 3 iterates n/2 times – we should multiply that by the work done by the loop body.

Line 4 is a conditional with a constant check – add it to the rest of the loop body.

Line 6 is a conditional with a O(n) check – add n to the rest of the body.

Lines 2, 5, 7, and 8 don't depend on the size of the input; they're constant actions.

Additional Learning: High-Speed Trading

Want more examples of how efficiency impacts real life? Check out this podcast episode on high-speed computer trading (where milliseconds make the difference between profit and loss):

https://radiolab.org/episodes/267124-speed

Learning Goals

- Identify the **worst case** and **best case** inputs of functions
- Compare the **function families** that characterize different functions
- Calculate a specific function or algorithm's efficiency using **Big-O notation**