Runtime and Big-O Notation

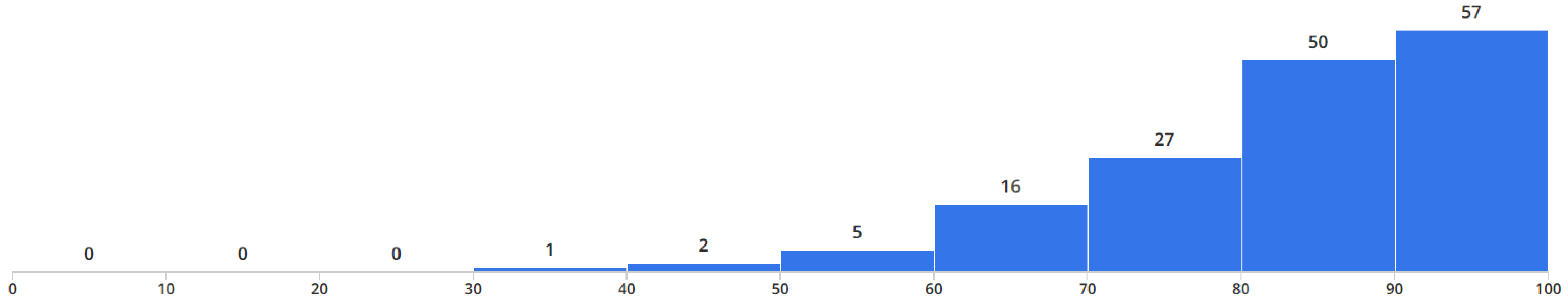
15-110 – Monday 02/27

Announcements

- Hw3 was due today
  - How did it go?

- **No Check4** due to spring break. Hw4 is extra-large instead – start early!

- **Exam1** grades have been released! Median = 85; well done!!



Learning Objectives

- 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**

Efficiency = Time = Money

We talk about efficiency a lot in this unit. Why do we care?

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.

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

Comparing Search Algorithms

Comparing Linear vs. Binary Search

Recall our comparisons of linear search vs. binary search in the previous lectures. How can we compare these algorithms at an **abstract** level to see which is more efficient?

We could run them on the same input and time them. However, how quickly a program runs varies based on lots of factors (the implementation, the machine, which other programs are running, etc.)

Instead, we'll choose some **meaningful action** that occurs in the program and count the number of actions the program takes on a given input.

Counting the number of actions

What actions might we count? Some lines of code may compose multiple operations into one line, and some actions may take longer than others to execute on the computer's hardware.

Instead of trying to count every action the computer takes, we can choose some specific action and count how many times the algorithm runs that action based on the **size of the input**.

For example, in linear or binary search we can count the total number of **comparisons** that the algorithms make to find an item based on the number of items in the list.

Linear vs. Binary Search: Search for 66

```
def linSearch(lst, target):
  if len(lst) == 0:
    return False
  elif lst[0] == target:
    return True
  else:
    return linSearch(lst[1:], target)
```

```
def biSearch(lst, target):
  if lst == [ ]:
    return False
  else:
    mid = len(lst) // 2
    if lst[mid] == target:
      return True
    elif target < lst[mid]:
      return biSearch(lst[:mid], target)
    else: # lst[mid] < target
      return biSearch(lst[mid+1:], target)
```

How many list elements are compared to 66?

linear search: 9 elements
binary search: 4 elements

							1st	4th	3rd		2nd			
12	**25**	**32**	**37**	**41**	**48**	**58**	**60**	**66**	**73**	**74**	**79**	**83**	**91**	**95**

Best Case, Worst Case

Best Case and Worst Case

To truly compare the algorithms, it isn't enough to test them on a random example. We want to know how they'll do in the **best case** and in the **worst case**. Those cases are defined based on the properties of the **input** to the function.

Best case: an input of size n that results in the algorithm taking the **least steps possible** for that size.

Worst case: an input of size n that results in the algorithm taking the **most steps possible** for that size.

Best Case and Worst Case – Linear Search

What's the **best case** for linear search?

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

What's the **worst case** for linear search?

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

Best Case and Worst Case – Binary Search

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

Best Case/Worst Case Actions

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

For both linear search and binary search, there's just **one comparison** – when you find the item with the first comparison, you can exit the function immediately.

How many actions in the **worst case**?

In linear search, we have to check **every single element**. If the list has n elements, we do **n comparisons** before knowing the element is not in the list.

What about binary search?

Worst Case Action Count – Binary Search

Each call to binary search compares one item of the list. How many recursive calls (and therefore comparisons) do we make to binary search for different length lists?

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	~$\log_2(n)$

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!**

Sidebar: Calculating Efficiency

Our implementation of binary search only looks better than our implementation of linear search because we **only count comparisons**.

Slicing a list also takes additional work, as the computer needs to create a copy of the list. Our recursive implementations of linear and binary search both slice the list on every call.

This is **inefficient** – we're doing more work than we need to! A better approach would be to pass the **reference** of the original list (no copies created) and change the indexes checked instead of changing the list itself.

Function Families

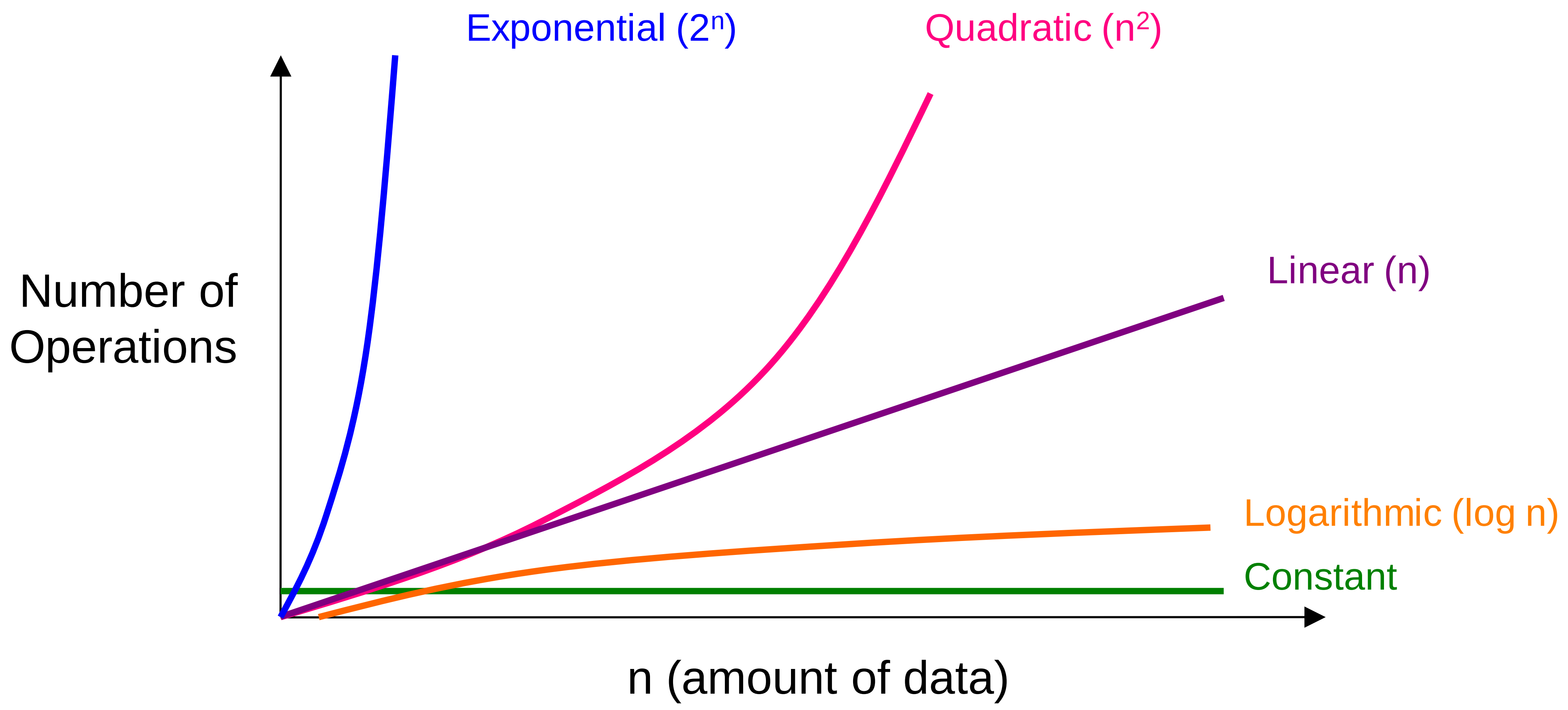
Function Families

When we count the actions taken by algorithms, we don't really care about one-off operations; we care about actions that are related to the **size of the input**. For example, if we set `y = 2x + 10`, **x** is the input size.

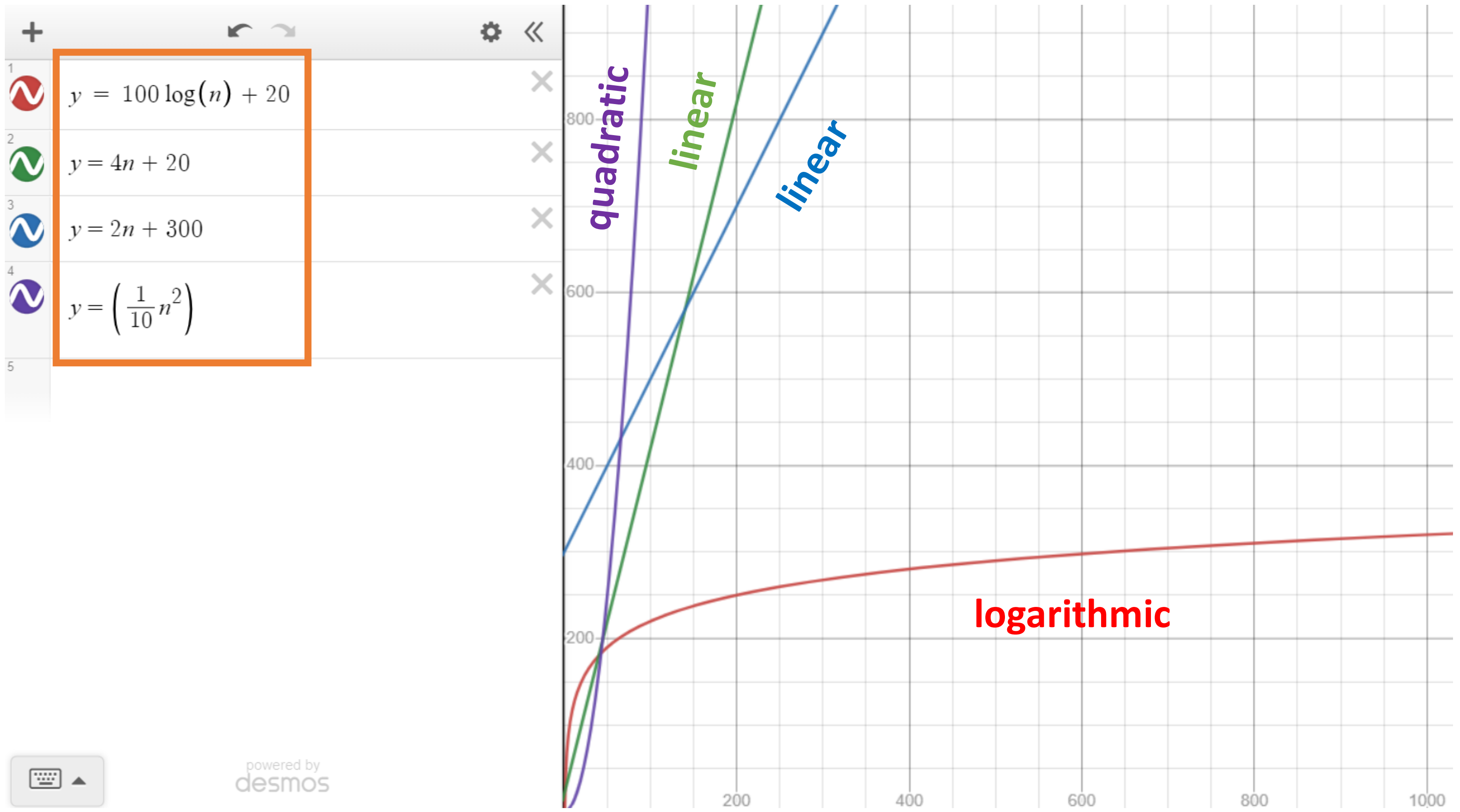
In math, a **function family** is a set of equations that all grow at the same rate as their inputs grow. For example, an equation might grow linearly or quadratically.

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


$y = 100\log(n) + 20$

$y = 4n + 20$

$y = 2n + 300$

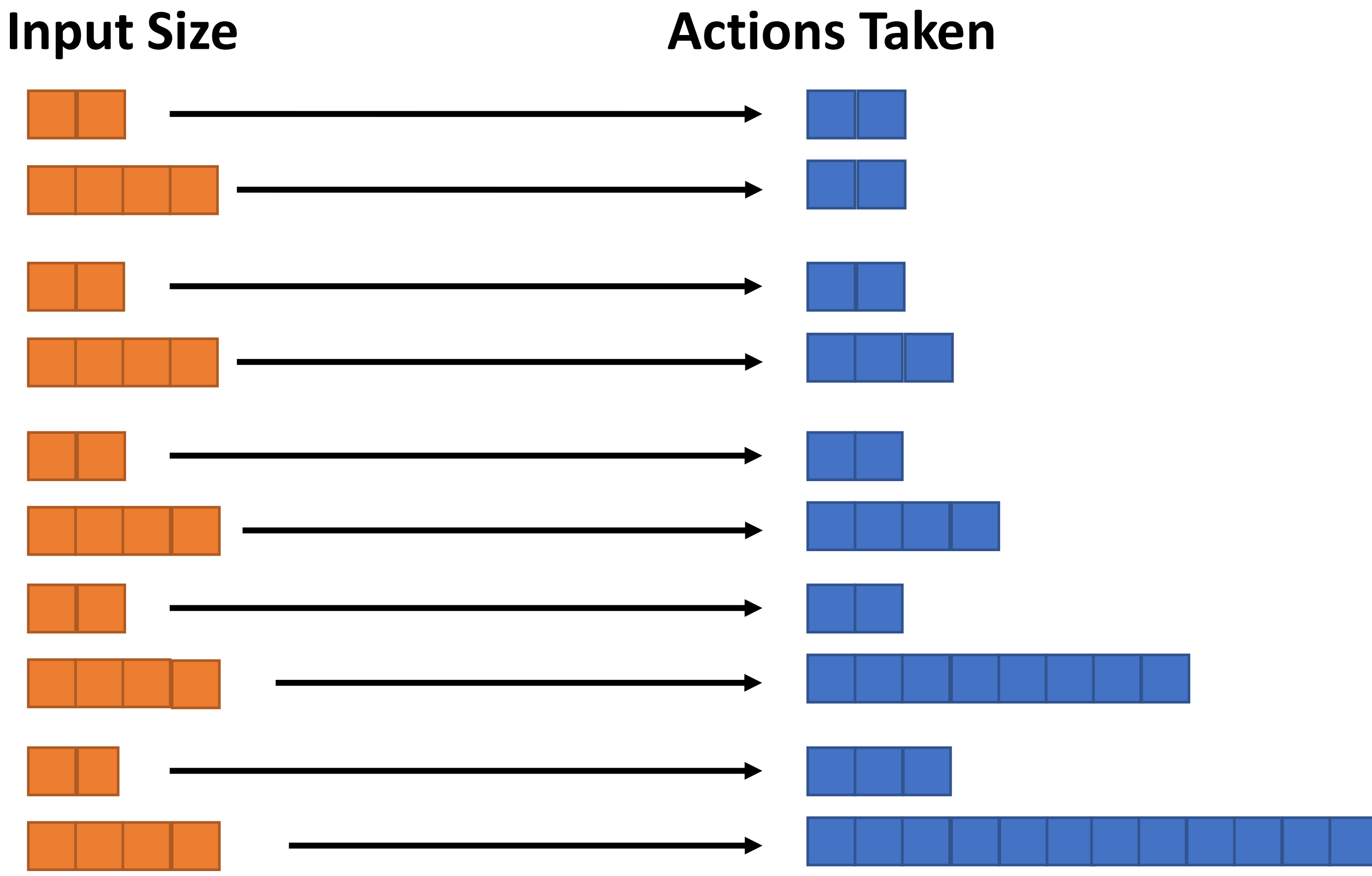
$y = \left(\frac{1}{10}n^2\right)$

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 we determine a program or algorithm's 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.**

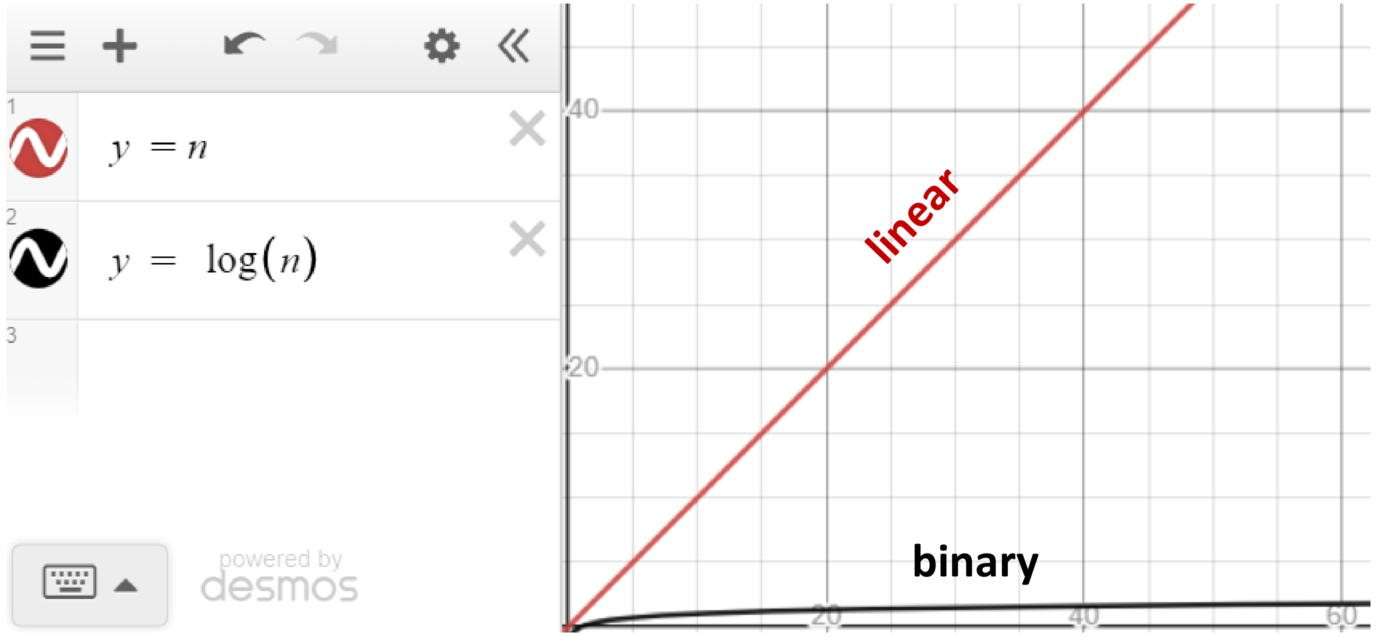
f(n)	**Big-O**
n	O(n)
32n + 23	O(n)
$5n^2 + 6n - 8$	$O(n^2)$
18 log(n)	O(log n)

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 of Linear Search / Binary Search

Because runtime for linear search is proportional to the length of the list in the worst case, it is O(n). Every time we double the length of the list, binary search does just one more comparison in the worst case; it is O(log n).



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

Big-O Calculation Strategy

We'll often need to calculate the Big-O of an algorithm or a piece of code to determine how efficient it is and whether we can make it better.

We can determine an algorithm's Big-O by determining how many actions take place based on the size of the input. We can often do a rough estimate of actions by just counting the number of statements that will run. This can be affected by **function calls** and **loops**.

Sequential Statements are Added

First, if you have statements that run sequentially, the runtime for each statement is **added** to the total runtime. This happens because each new statement adds another amount of time.

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

Functions Have Their Own Runtimes

What if we call a function? We need to add **that function's runtime** to the overall runtime. Note that 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 Multiply Runtimes

Most non-constant runtimes come from some use of loops (or recursion). This is because loops let us **repeat actions**, so we must **multiply** the runtime of the loop body by the number of times the loop iterates.

```
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)
```

Conditionals are Sequential

If we multiply loop bodies by the number of repetitions, do we do the same thing to conditionals?

No! Conditionals act more **sequentially** – in the worst case you run the Boolean test expression, then you run the body. So you should add the two together.

```
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)
```

Be Careful of Built-in Runtimes!

Functions we define aren't the only ones that can have non-constant runtimes; some of the built-in Python functions (and operations) have non-constant runtimes too!

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

We'll let you know on assignments and exams when a built-in method or operation is not constant time. Except for `in`, which just implements linear search; therefore `in` applied to a list or string will run in O(n) where n is the length of that list/string!

Common Big-O Runtimes

O(1) is Constant Time

```
# n = len(lst)
def swap(lst, i, j):
    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)**.

Why? O(log 2n) = O(log n) + 1 - you add one action per doubling of the input.

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 = 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 Objectives

- 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**