Runtime and Big-O Notation

15-110 - Wednesday 10/07

Learning Objectives

- Compare the **function families** that different functions run in

- Identify the **worst case** and **best case** inputs of functions

- Calculate a specific function's efficiency using **Big-O notation**

Efficiency = Time = Money

We'll 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**.

Linear Search vs. Binary Search

Comparing Linear vs. Binary Search

Recall our comparison of linear search vs. binary search in the previous lecture. How can we compare these two algorithms at an **abstract** level?

We could run both 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 count the number of **actions** the program takes on a given input.

Counting the number of actions

What actions might we count? Recall that some actions, such as adding two numbers, become several steps when translated to bytecode. And some actions may take longer than others to execute.

Instead of trying to count every action, we count how many times the algorithm processes **each element in the input**.

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

Linear vs. Binary Search: Search for 66

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

How many list elements are compared to 66?
        linear search: 9 comparisons
        binary search: 4 comparisons

```python
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)
```

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

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**, based on the inputs.

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

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

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?

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** – a list of any length in which it finds the item with the first comparison.

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

> In linear search, we have to check **every single element**. So if the list has n elements, we do n **comparisons**.

> What about binary search?

Worst Case Action Count – Binary Search

Each call to binary search compares one item of the list. How many calls do we make to binary search for different length lists?

List size	Number of 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, **linear search** does **twice** as many comparisons.

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

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, and change the indexes checked instead of the list itself.

Big-O Notation

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**.

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 equation 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.

Function Families in Graphs
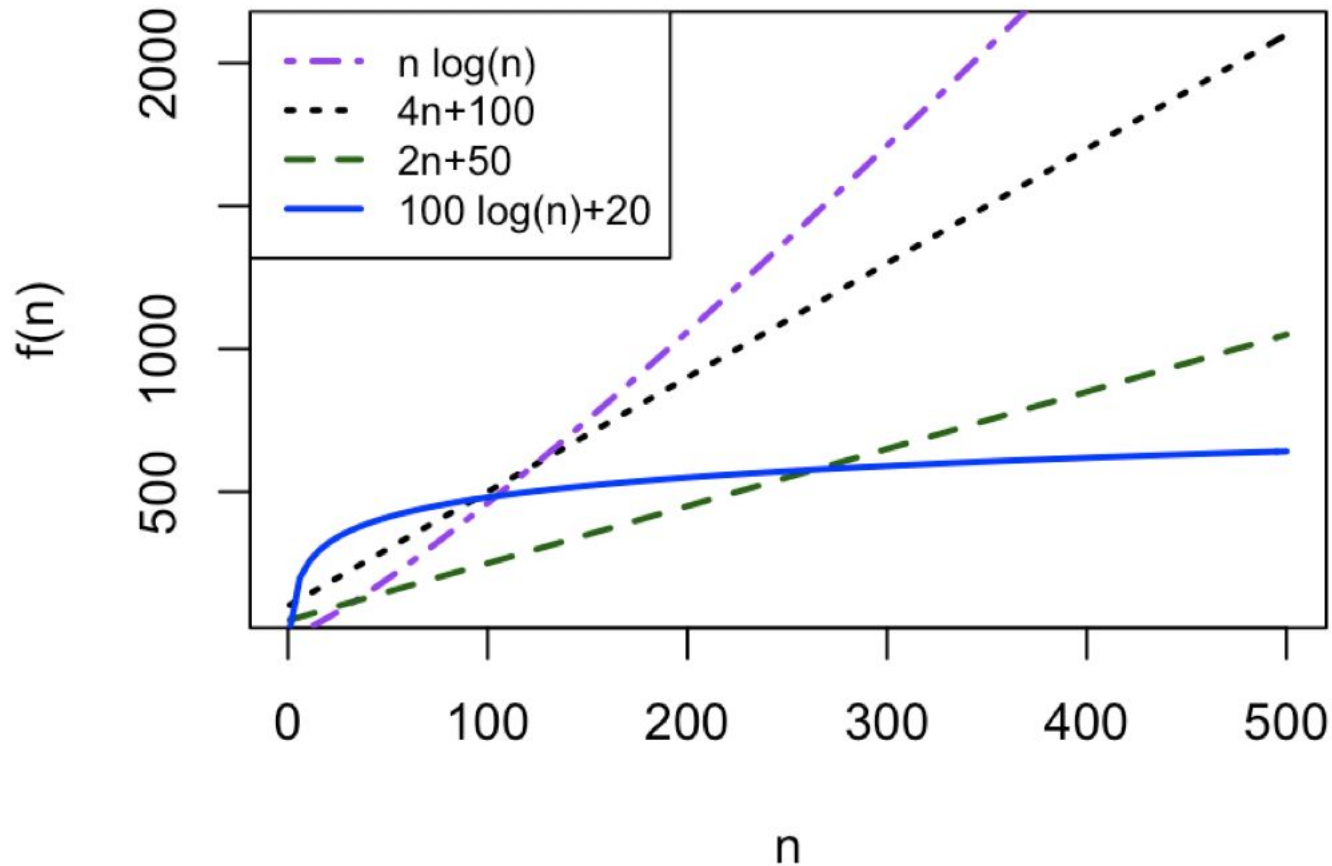
Notice that as n grows, the two linear functions become larger than the log(n) function, and the n log(n) function becomes larger than both linear functions, regardless of the constants.
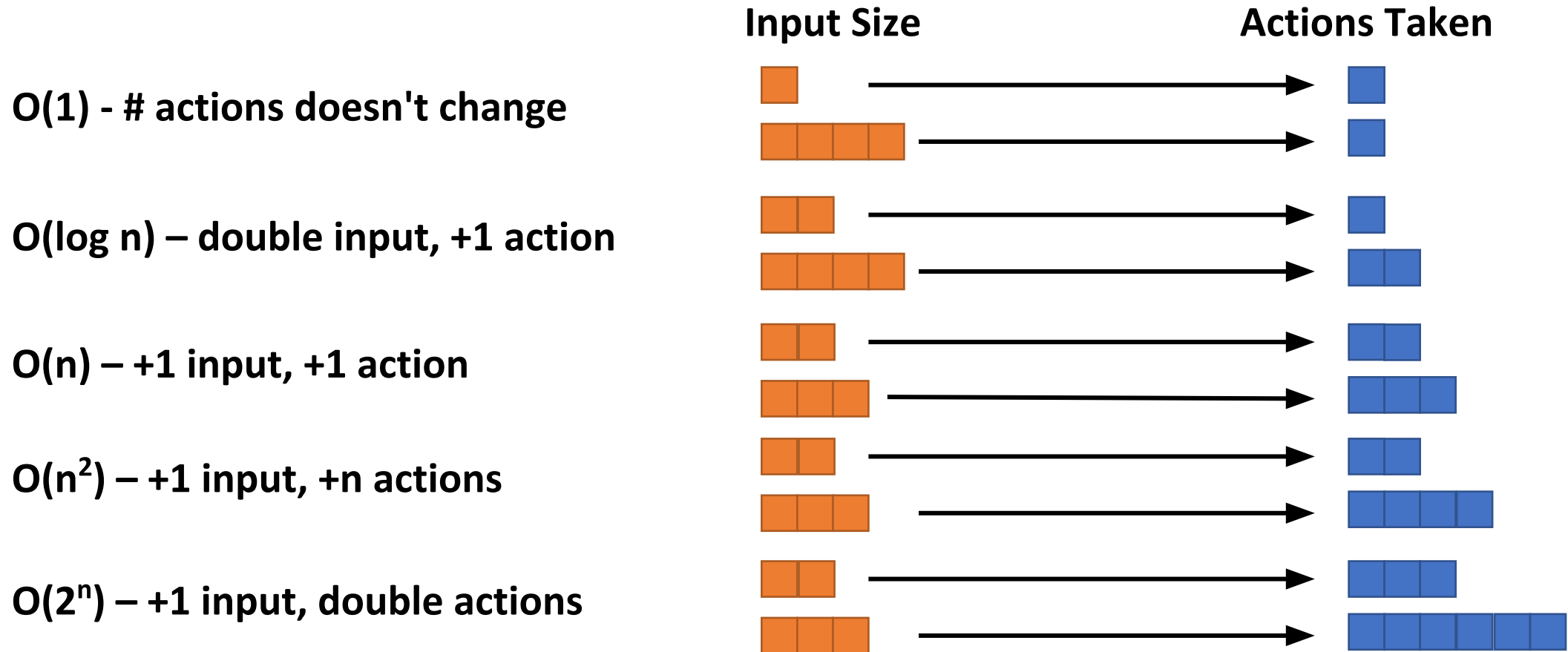
Function Families in Graphs
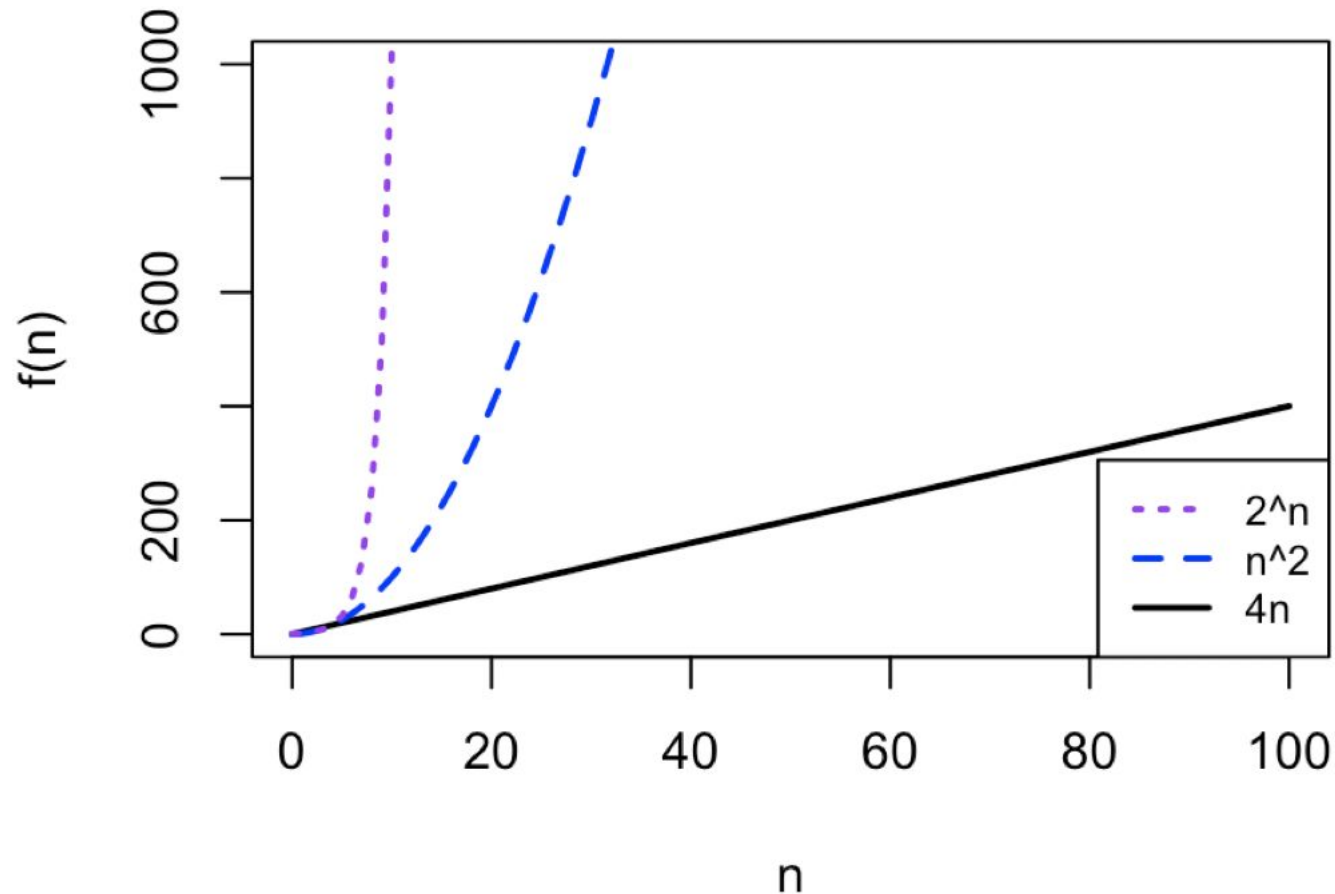
Even for small n, exponential functions quickly skyrocket and quadratic functions grow rapidly compared to linear functions.

Alternate Visualization

Here's another way to think about the function families.

Input Size   **Actions Taken**

O(1) - # actions doesn't change

O(log n) – double input, +1 action

O(n) – +1 input, +1 action

O(n^2) – +1 input, +n actions

O(2^n) – +1 input, double actions

Big-O Notation

When we determine an equation's function family, we **ignore constant factors and smaller terms**. All that matters is the dominant term (the highest power of n). That is the idea of **Big-O notation.**
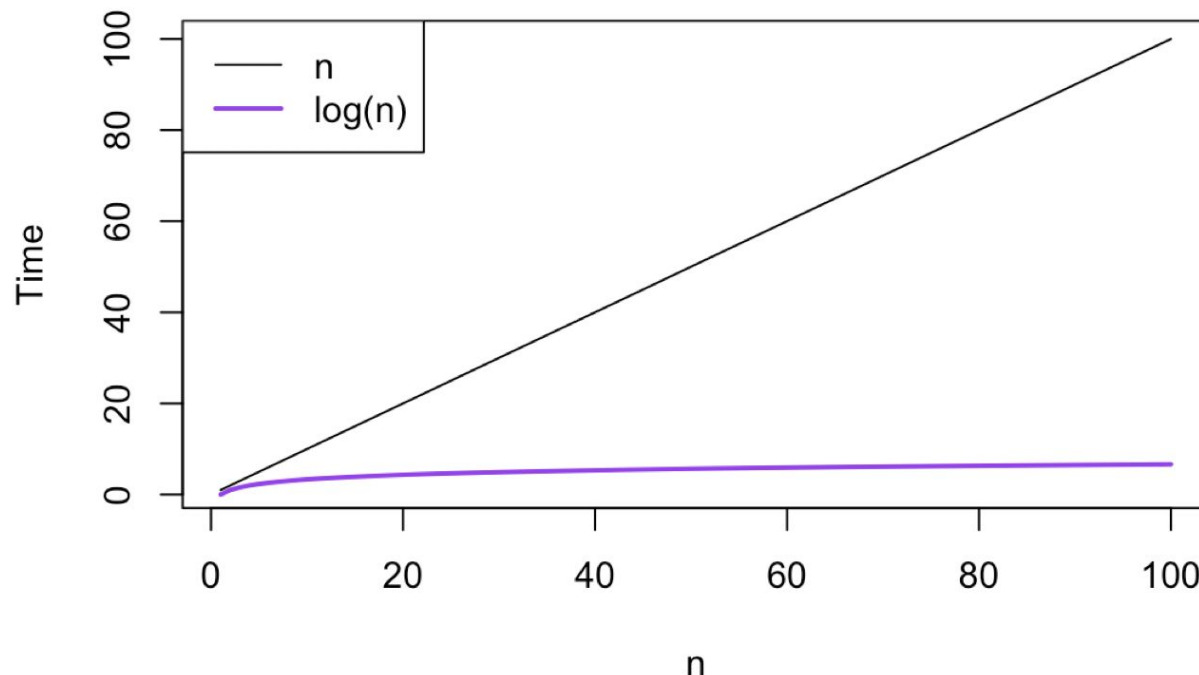
f(n)	Big-O
n	O(n)
32n + 23	O(n)
$5n^2 + 6n - 8$	$O(n^2)$
18 log(n)	O(log n)

Unless specified otherwise, the Big-O of an algorithm refers to its **worst case** run time (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; it is O(log n).

Except for very small n, binary search is blazingly fast. Linear search is exponentially slower in the worst case!

Big-O Calculation

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 a function's Big-O by determining how many actions are **added** if we increase the size of the input.

Let's go through a bunch of examples to demonstrate.

O(1) is Constant Time

```
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.

We say that an algorithm is **constant time** when its time does not change with the size of the input.

O(log n) is Logarithmic Time

```python
def countDigits(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 do the loop one more time. All the operations in the loop are constant time. Analogous to binary search, the algorithm is O(log n).

Even though it 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

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

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, as it is proportional to the size of n. Note that stepping by 5 doesn't change that it is O(n).

O(n^2) is Quadratic Time

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

If we double the size of n, we execute the outer loop twice as many times. And for each time we execute the outer loop, we execute the inner loop twice as many times. Generating the table takes 4 times as long.

If we triple the size of n, generating the table takes 9 times as long. The runtime is proportional to n^2.

O(2^n) is Exponential Time

```python
def move(start, tmp, end, 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 one disc, we double the number of moves. That's exponential time, or 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.

```python
def countdown(n):
    if n <= 0:
        print("Finished!")
    else:
        print(start)
        countdown(start - 5, end)
```

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).

Be Careful of Built-in Runtimes!

```python
def countAll(lst):
    for i in range(len(lst)):
        count = lst.count(i)
        print(i, "occurs", count, "times")
```

This is actually O(n^2), because each call to `lst.count(i)` takes O(n) time.

We'll let you know on assignments and exams when a built-in operation is not constant time.

Activity: Calculate the Big-O of Code

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

```python
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
```

Learning Objectives

- Compare the **function families** that different functions run in

- Identify the **worst case** and **best case** inputs of functions

- Calculate a specific function's efficiency using **Big-O notation**

- **Feedback: https://bit.ly/110-feedback**