Advanced CS #3 – Efficiency Analysis

CS SCHOLARS – PROGRAMMING

Learning Objectives

Recognize **linear search** and **binary search** when reading and writing code to search for items in sorted lists

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

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

Recognize the requirements for building a good **hash function** and a good **hashtable** that lead to **constant-time search**

Efficiency = Time = Money

Why should we care about how fast our code is?

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**.

Two Ways to Search

Searching for Items

**Search** is one of the most common tasks a computer needs to do.
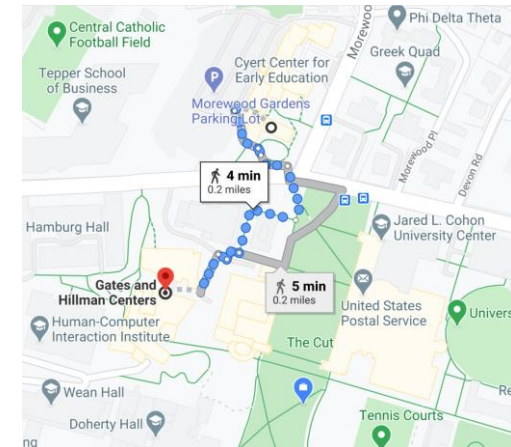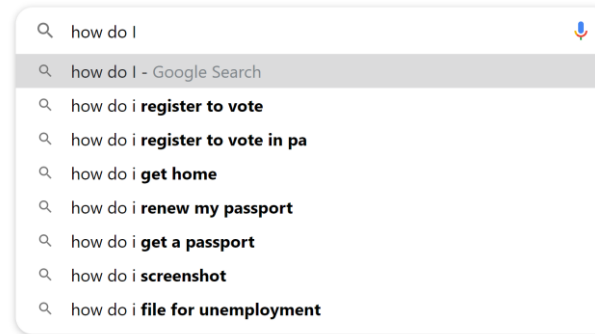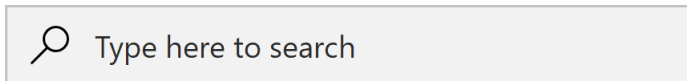
Suppose we want to determine whether a list contains a specific value. We know that the `in` operator can check this for us, but what algorithm does `in` implement?
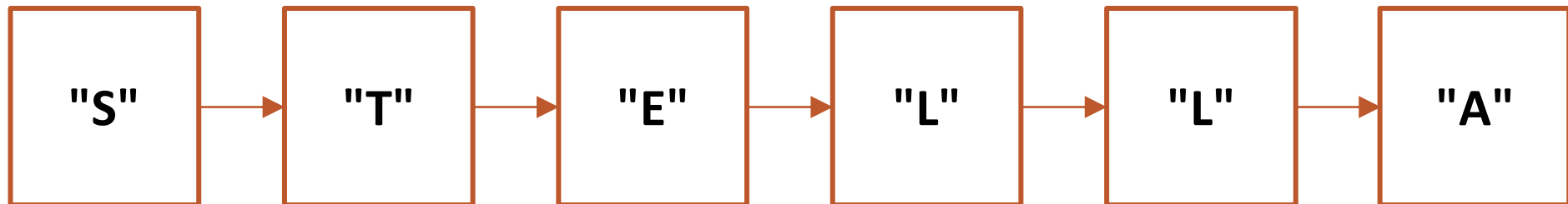
We'll need to think about this from a computer's perspective...

How Computers See Lists

If we ask a computer to check if a value is in a list, it sees the whole list as a series of not-yet-known values:
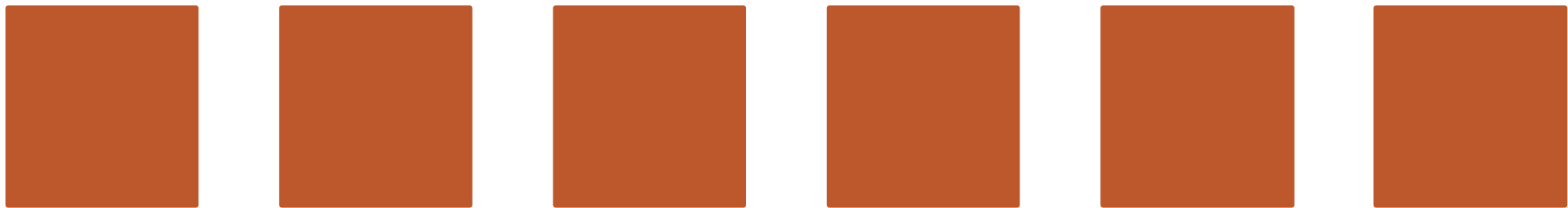
In order to determine if the value is one of them, it needs to check each item in turn.

"S" → "T" → "E" → "L" → "L" → "A"

For Loop Search Function

We can use a for loop to implement this approach as code. We call this **linear search**, because it searches all items in a linear order.

```python
def linearSearch(lst, target):
    for i in range(len(lst)):
        if lst[i] == target:
            return True
    return False
```

Note that we can return True as soon as we find the target value, but we can't return False until we've examined all the values.

Question: If target appears more than once in lst, which value will cause the function to return?

Answer: The first one!

Recursive Linear Search Algorithm

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

> Answer: an empty list. The item can't possibly be in an empty list, so the result is `False`.

> Also: a list where the first element is what we're searching for, so the result is `True`.

How do we make the problem **smaller**?

> Answer: call the linear search on all but the first element of the list.

How do we **combine** the solutions?

> Answer: no combination necessary. The recursive call returns whether the item occurs in the rest of the list; just return that result unmodified.

Recursive Linear Search Code

```python
def recursiveLinearSearch(lst, target):

    if lst == [ ]:

        return False

    elif lst[0] == target:

        return True

    else:

        return recursiveLinearSearch(lst[1:], target)


print(recursiveLinearSearch(["dog", "cat", "rabbit", "mouse"], "rabbit"))

print(recursiveLinearSearch(["dog", "cat", "rabbit", "mouse"], "horse"))
```

Alternative to Linear Search

Linear Search is a nice, straightforward approach to searching a set of items. But that doesn't mean it's the only way to search.

Assume you want to search a dictionary to find the definition of a word you just read. Would you use linear search, or a different algorithm?
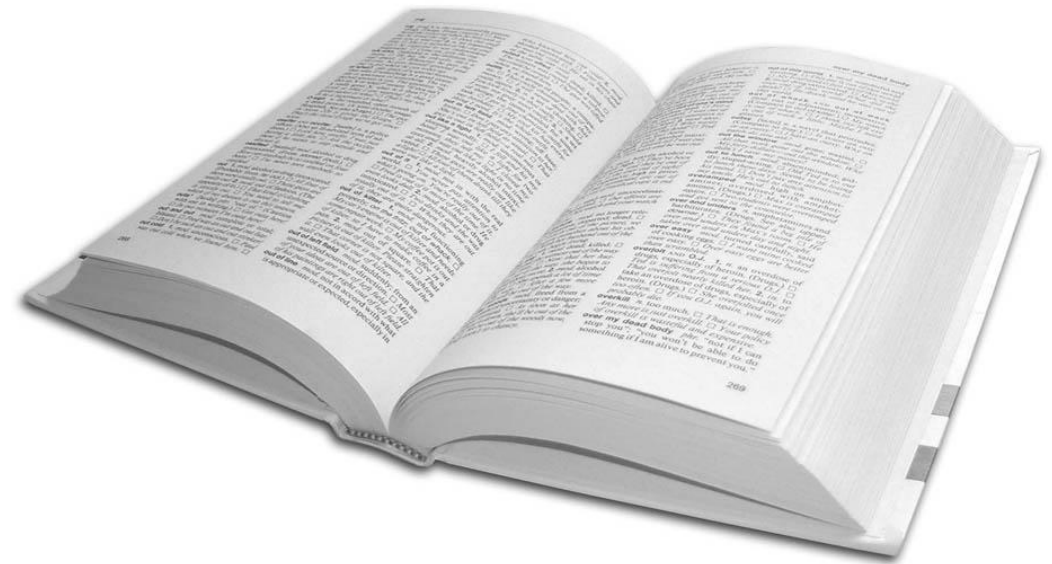
Can we take advantage of dictionaries being **sorted**?

Binary Search Divides the List Repeatedly

In **Linear Search**, we start at the beginning of a list and check each element in order. So if we search for 98 and do one comparison...

Start here

| 2 | 5 | 10 | 20 | 42 | 56 | 67 | 76 | 89 | 95 |
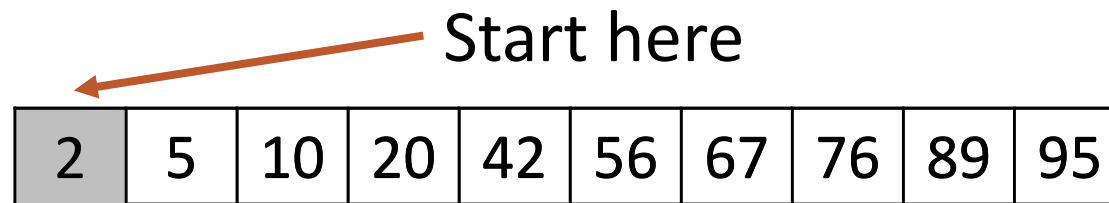
In **Binary Search** on a **sorted list**, we'll start at the **middle** of the list and **eliminate** half the list based on the comparison we do. When we search for 98 again...

Start here

| 2 | 5 | 10 | 20 | 42 | 56 | 67 | 76 | 89 | 95 |

Many more #s have been eliminated!

Algorithm for Binary Search

Algorithm for Binary Search:

1. Find the middle element of the list.

2. Compare the middle element to the target.
   a) If they're equal – you're done!
   b) If the item is **smaller** – recursively search to the **left** of the middle.
   c) If the item is **bigger** – recursively search to the **right** of the middle.

Example 1: Search for 73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	**60**	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
~~12~~	~~25~~	~~32~~	~~37~~	~~41~~	~~48~~	~~58~~	~~60~~	66	73	74	**79**	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
~~12~~	~~25~~	~~32~~	~~37~~	~~41~~	~~48~~	~~58~~	~~60~~	66	**73**	74	~~79~~	~~83~~	~~91~~	~~95~~

Found: return True

Example 2: Search for 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

Not found: return False

Activity: Trace Binary Search

You do: determine the correct trace for the following call to binary search. Which numbers are visited?

Note that when there are an even number of elements, we'll break ties to the right.

```
binarySearch([2, 7, 11, 18, 19, 32, 45, 63, 84, 95, 97], 95)
```

Activity Answer

32, 84, 97, 95

Base Case and Recursive Case of Binary Search

What are the **base cases** for binary search?

       Answer: an empty list. The target can't possibly be in an empty list, so the result is `False`.

       Also: a list where the target is the middle element. Then we can stop searching and immediately return `True`.

How do we make the problem **smaller**?

       Answer: get rid of the half of the list we know the target isn't in (which half?).

How do we **combine** the solutions?

       Answer: no need to combine anything. Simply return the result of the recursive function call.

Binary Search in Code

Now we just need to translate the algorithm to Python.

```python
def binarySearch(lst, target):
    if ____ # base case
        return _____
    else:
        # Find the middle element of the list.
        # Compare middle element to the target.
            # If they're equal – you're done!
            # If the item is smaller, recursively search
            #     to the left of the middle.
            # If the item is bigger, recursively search
            #     to the right of the middle.
```
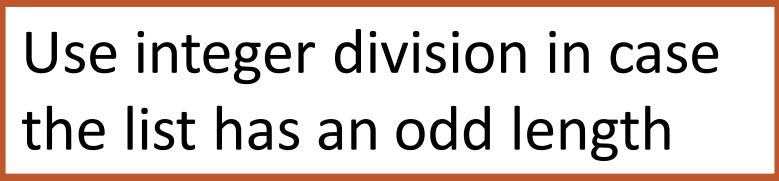
Binary Search in Code – Base Case

The first **base case** is the empty list, and return False

```python
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    else:
        # Find the middle element of the list.
        # Compare middle element to the target.
            # If they're equal – you're done!
            # If the item is smaller, recursively search
            #     to the left of the middle.
            # If the item is bigger, recursively search
            #     to the right of the middle.
```

Binary Search – Middle Element

To get the middle element, use **indexing** with half the length of the list.

```python
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    else:
        midIndex = len(lst) // 2
        # Compare middle element to the target.
            # If they're equal – you're done!
            # If the item is smaller, recursively search
            #     to the left of the middle.
            # If the item is bigger, recursively search
            #     to the right of the middle.
```

Use integer division in case the list has an odd length

Binary Search – Base Case

The second **base case** occurs when we find the target. Return True.

```python
def binarySearch(lst, target):

    if lst == [ ]:

        return False

    else:

        midIndex = len(lst) // 2

        if lst[midIndex] == target:

            return True

        # If the item is smaller, recursively search
        #     to the left of the middle.

        # If the item is bigger, recursively search
        #     to the right of the middle.
```

Binary Search – Comparison

Use an if/elif/else statement to decide which side to use for the smaller problem.

```
def binarySearch(lst, target):
    if lst == [ ]:
        return False
    else:
        midIndex = len(lst) // 2
        if lst[midIndex] == target:
            return True
        elif target < lst[midIndex]:
            _____ # recursively search to the left of the middle
        else: # lst[midIndex] < target
            _____ # recursively search to the right of the middle
```

Binary Search – Recursive Calls

Use **slicing** to make the recursive call and return the result immediately.

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

Linear Search vs. Binary Search

Why should we go through the effort of writing this more-complicated search method?

Answer: **efficiency**. Binary search is **vastly** more efficient than linear search, as it performs a lot fewer comparisons to find the same item.

Comparing Linear vs. Binary Search

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 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 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)
```

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

```
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:
            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, 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 inputs to the function.

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 if we're counting comparisons?

Best Case and Worst Case – Binary Search

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

Answer: a list where the item we search for is right in the middle

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

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

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**. 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 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, **linear search** does **twice** as many comparisons.

But, when the input length doubles, **binary search** 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 and change the indexes checked instead of changing the list itself.
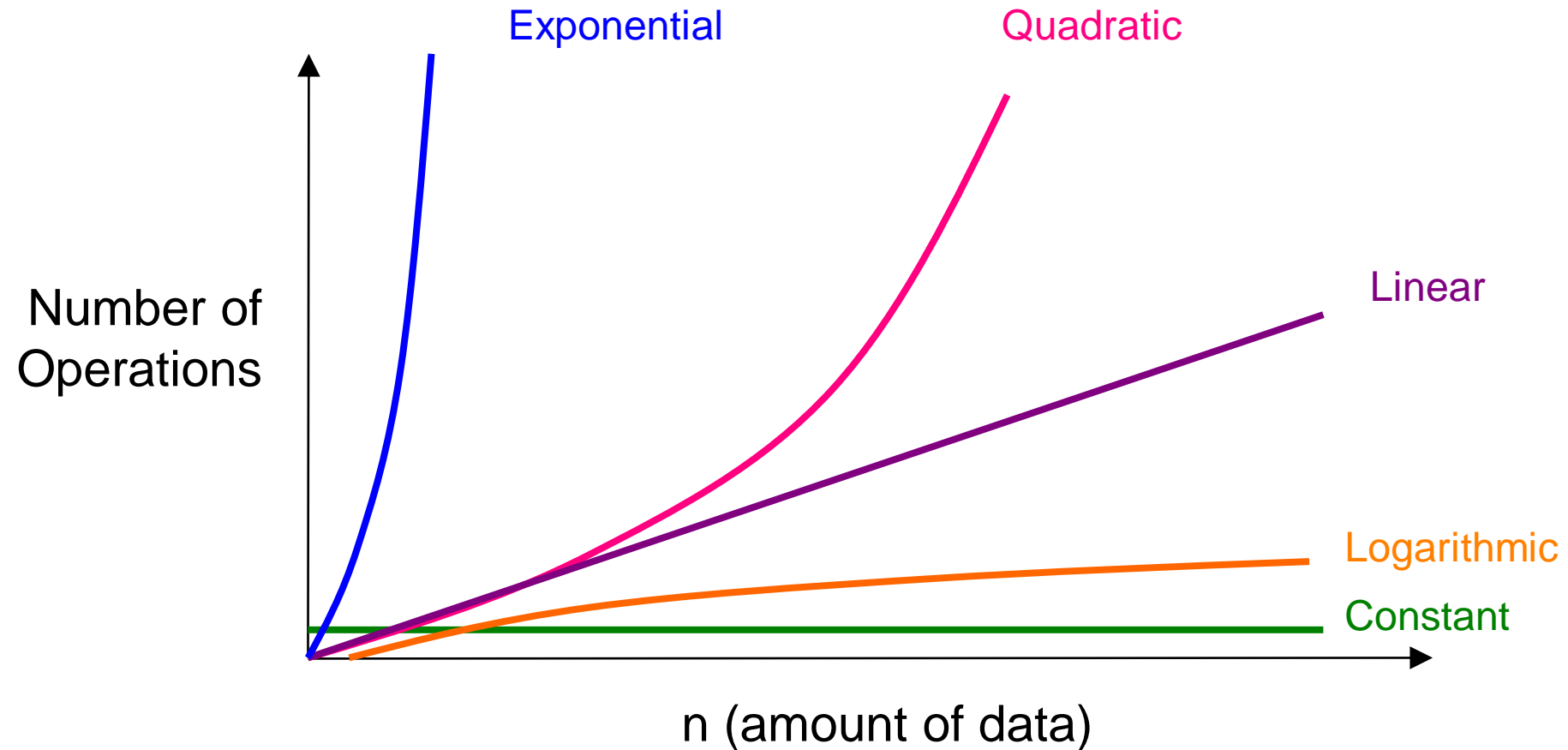
Function Families

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**.

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.

Common Function Families

Number of Operations

n (amount of data)

Exponential
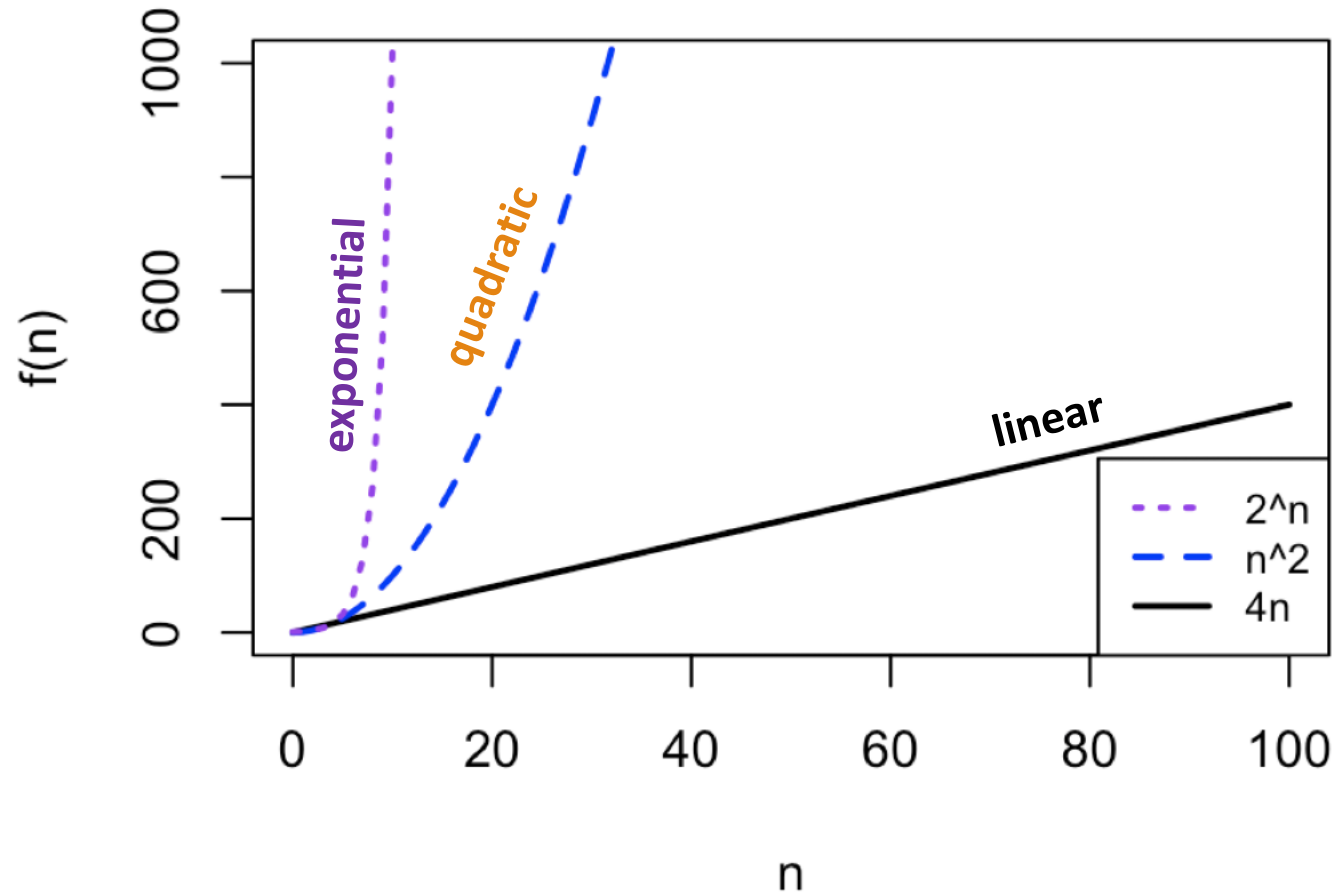
Quadratic

Linear

Logarithmic

Constant

Function Families and Constants
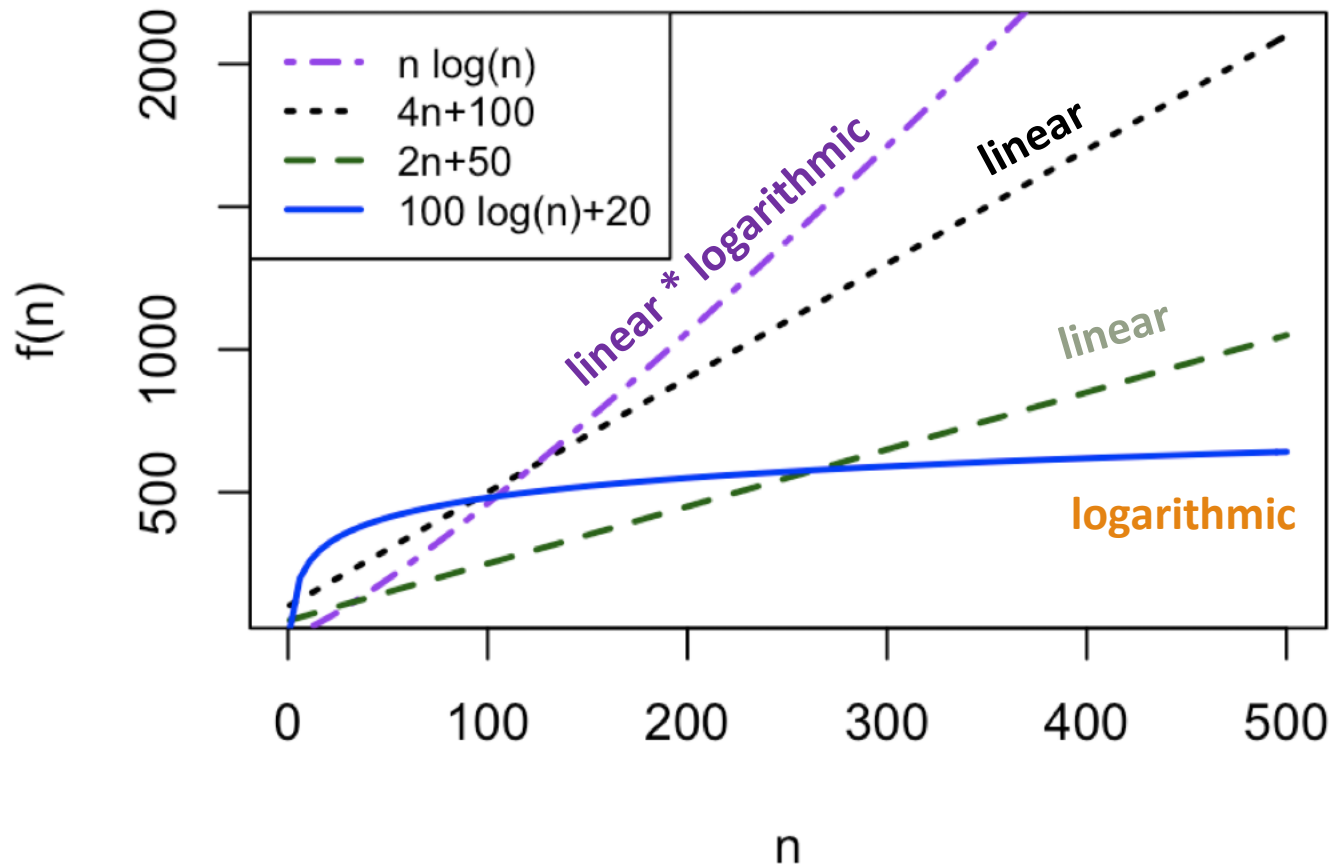
Notice that as n grows, the two linear functions become larger than the logarithmic function and the linear * logarithmic function becomes larger than both linear functions, regardless of the constants.
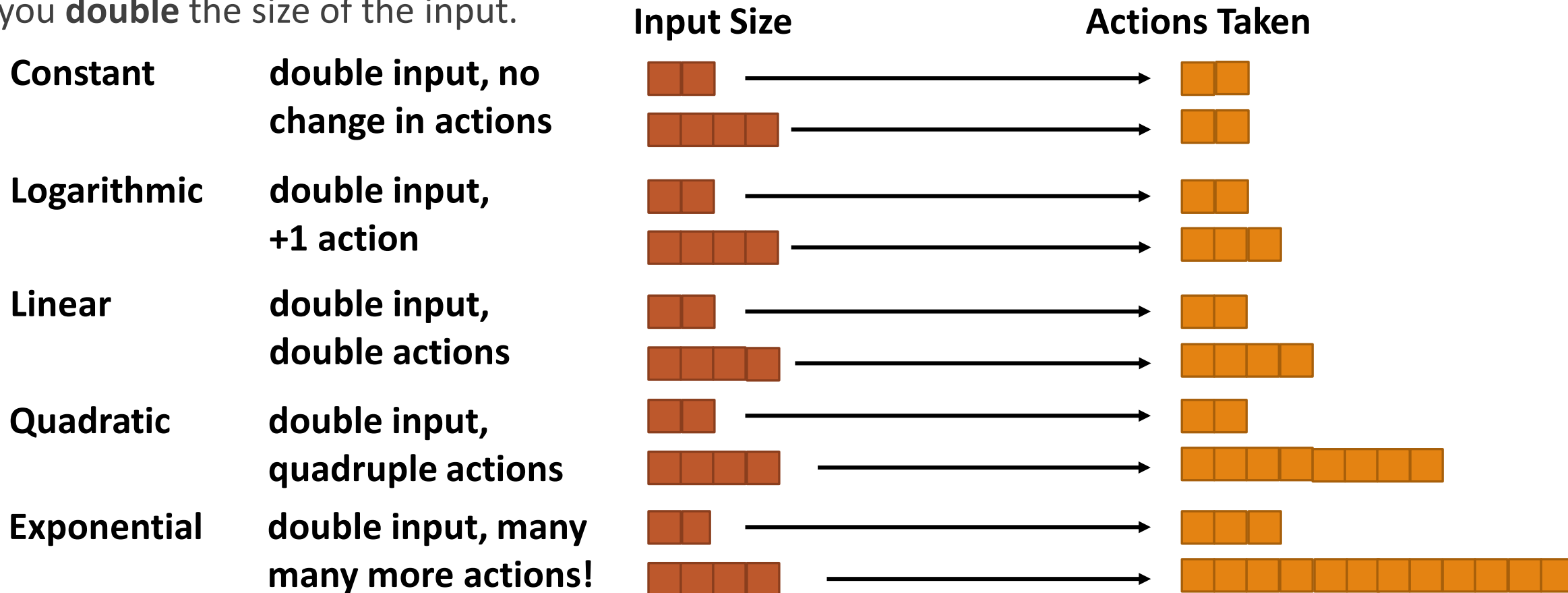
Function Family Comparisons

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. Consider what happens when you **double** the size of the input.

Input Size   **Actions Taken**

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

Big-O Notation

When we determine a program or algorithm's runtime, we **ignore constant factors and smaller terms**. All that matters is the dominant term (the highest power of n), the function family. 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 in the worst case; it is O(log n).
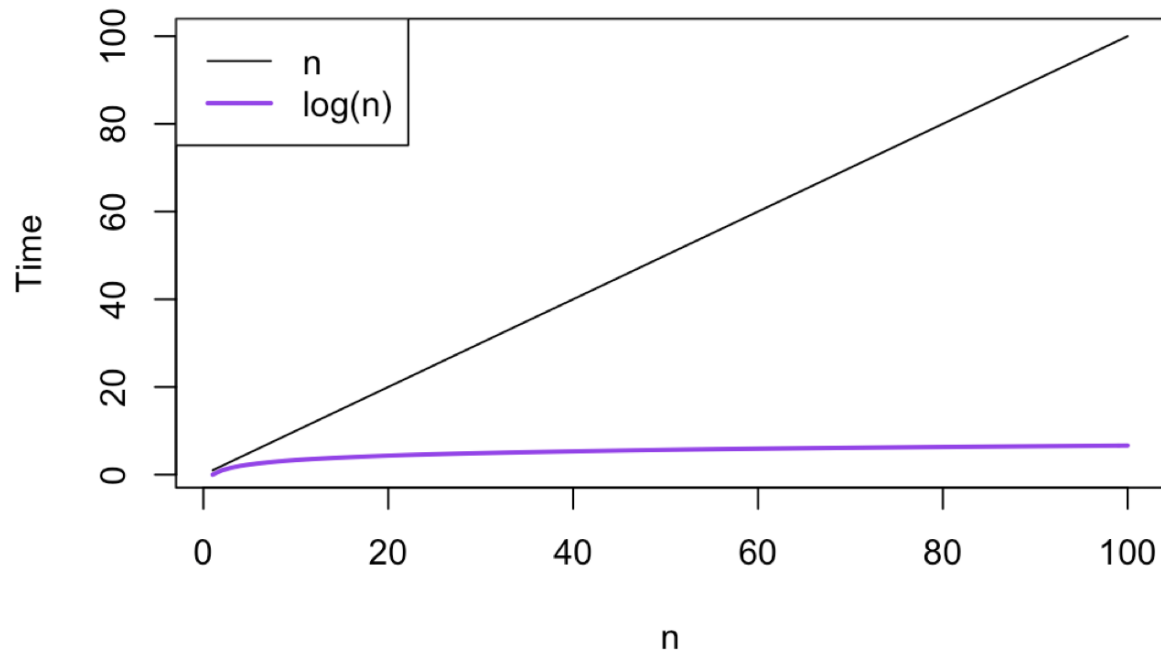
Except for very small n, binary search is blazingly faster. Linear search is exponentially slower in the worst case!

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 are **added** if we increase the size of the input. We can often do a rough estimate of actions by just counting the number of statements that will run.

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** or **O(1)** when its time does not change with the size of the input.

O(log n) is Logarithmic Time

```
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 **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

```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**, or **O(n)**, as it is proportional to the size of n. Stepping by 5 doesn't change the function family.

Note that O(2n) = O(n) + 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. This is **quadratic time**, or **O(n^2)**.

Every time you add a new element, 1 action is added to each iteration of the inner loop and 1 iteration is added to the outer loop (n+1 actions). That's 2n+1 new actions added. O((n+1)^2) = O(n^2) + 2n + 1.

O(2^n) is Exponential Time

```
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 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.

```python
def countdown(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).

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.

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
```

Activity Answer

O(n)

Optimizing Search

Increase Efficiency by Cutting Extra Work

We've talked about how to determine the efficiency of an algorithm, but we haven't addressed a more important question. How can we *design* algorithms to make them more efficient?

Sometimes making a program more efficient is easy; you just need to look for unnecessary actions (statements that aren't used, loops that repeat work already done) and cut them.

```python
def findLargest(lst):
    largest = lst[0]
    for i in range(len(lst)):
        for j in range(len(lst)):
            if lst[i] > largest and \
                lst[i] > lst[j]:
                largest = lst[i]
    return largest


# could be

def findLargest(lst):
    largest = lst[0]
    for i in range(1, len(lst)):
        if lst[i] > largest:
            largest = lst[i]
    return largest
```

Increase Efficiency by Thinking Differently

More often we increase the efficiency of an algorithm by **thinking about the problem in a different way**.

The obvious solution to a problem isn't always the most efficient. We can often make a faster solution by using a different data structure or an entirely different algorithmic approach.

Improving Search

We've discussed linear search (which runs in O(n)), and binary search (which runs in O(log n)).

We use search all the time, so we want to search as quickly as possible. **Can we search for an item in O(1) time?**

We can't *always* search for things in constant time, but there are certain circumstances where we can.

Search in Real Life – Post Boxes

Consider how you receive mail at college. Your mail is sent to the post box in a central location. Do you have to check every box to find your mail?

No- just check the one assigned to you.

This is possible because your mail has an **address** on the front that includes your mailbox number. Your mail will only be put into a box that has the same number as that address, not other random boxes.
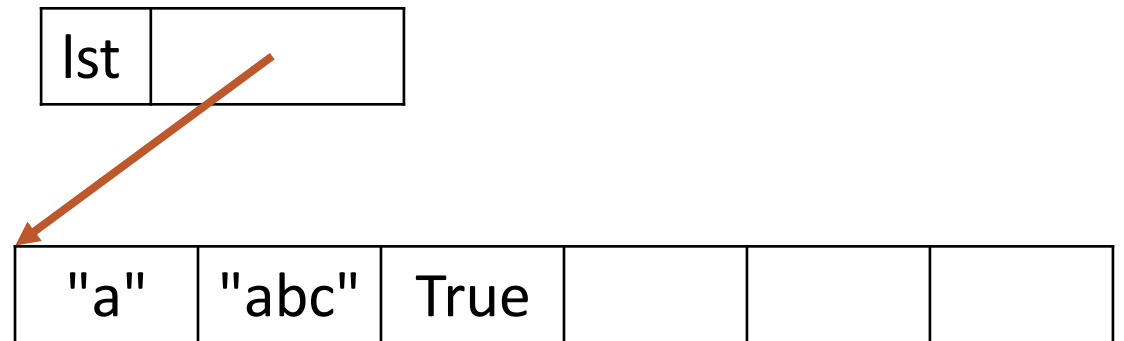
Picking up your mail is a O(1) operation!

Search in Programming – List Indexes

We can't search a list for an item in constant time, but we **can** look up an item based on an index in constant time.

Reminder: Python stores lists in memory as a series of **adjacent parts**. Each part holds a single value in the list, and all these parts use the **same amount of space**.

Example:

```
lst = ["a", "abc", True]
```

lst	

| "a"	"abc"	True			

Search in Programming – List Indexes

We can calculate the exact starting location of a list index's memory address based on the first address where `lst` is stored. If the size of a part is N, we can find an index's address with the formula:

```
start + N * index
```

Example: in the list to the right, each part is 8 bytes in size and the memory values start at `0800`. To access `lst[2]`, compute:

```
0800 + 8 * 2 = 0816
```

Given a memory address, we can get the value from that address in constant time. Looking up an index in a list is O(1)!

lst

"A"   "B"   "C"   "D"   "E"

0800   8 bytes   8 bytes   8 bytes   8 bytes   8 bytes
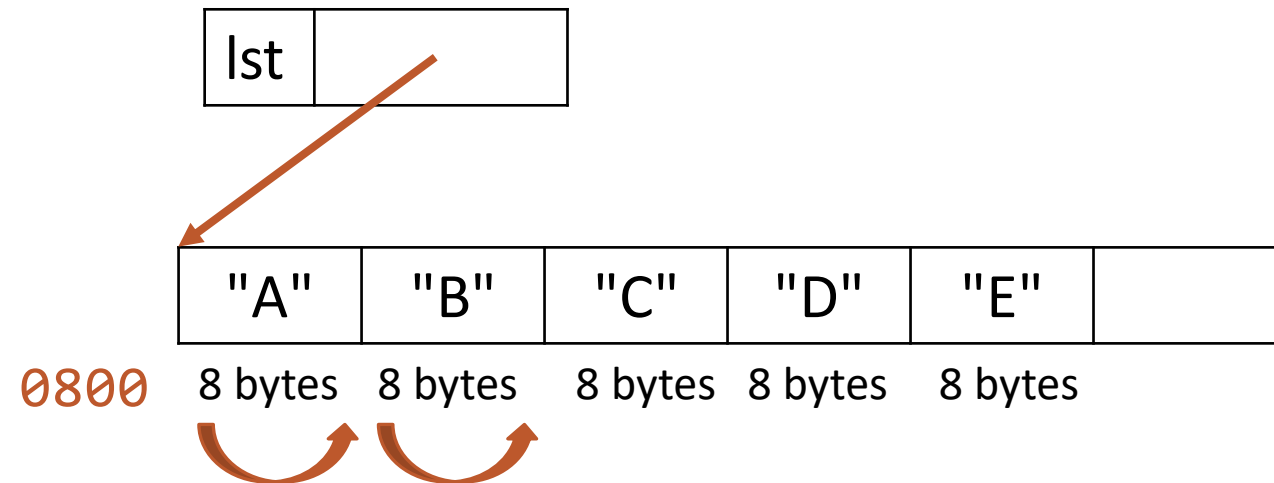
Combine the Concepts

To implement constant-time search, we want to combine the ideas of post boxes and list index lookup. Specifically, we want to determine which index a value is stored in **based on the value itself**.

If we can calculate the index based on the value, we can retrieve the value in constant time.

Hash Functions Map Values to Integers

In order to determine which list index should be used based on the value itself we'll need to **map values to indexes** (integers).

We call a function that maps values to integers a **hash function**. This function must follow two rules:

- Given a specific value `x`, `hash(x)` must **always** return the same output `i`

- Given two different values `x` and `y`, `hash(x)` and `hash(y)` should **usually** return two different outputs, `i` and `j`

Built-in Hash Function

We don't need to write our own hash function most of the time- Python already has one!

```
x = "abc"
```

```
hash(x) # some giant number
```

`hash()` works on integers, floats, Booleans, strings, and some other types as well.

Hashtables Organize Values

Now that we have a hash function, we can use it to organize values in a special data structure.

A **hashtable** is a list with a fixed number of indexes. When we place a value in the list, we put it into an index **based on its hash value** instead of placing it at the end of the list.

We often call these indexes 'buckets'. For example, the hashtable to the right has four buckets. **Important:** actual hashtables have far more buckets than this.

index 0	index 1	index 2	index 3
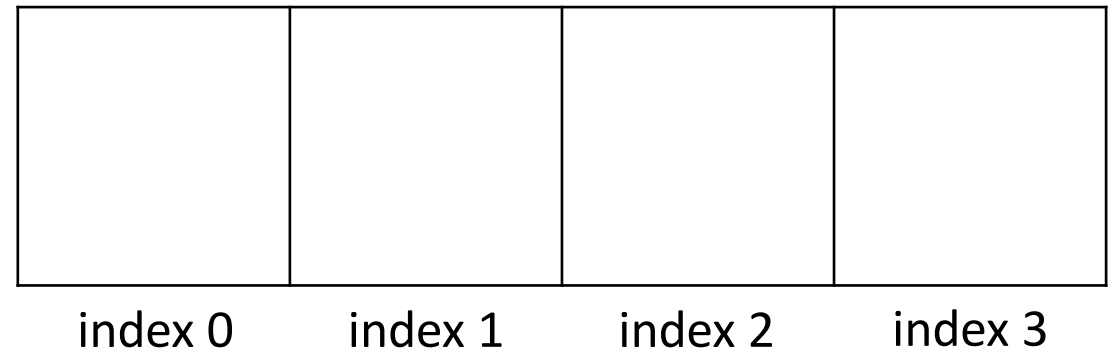
Adding Values to a Hashtable

For simplicity, let's say this hashtable uses a hash function that maps strings to indexes using the first letter of the string, as shown to the right. (This is not a good hash function, but it will serve as an example).

First, add "book" to the table. hash("book") is 1, so we'll put the value in bucket 1.

Next, add "yay". hash("yay") is 24, which is outside the range of our table. How do we assign it?

Use value % tableSize to map integers larger than the size of the table to an index. 24 % 4 = 0, so we put "yay" in bucket 0.

```
def hash(s):
    # number of letters between
    # first letter and 'a'
    return ord(s[0]) - ord('a')
```

"yay"	"book"		

| index 0 | index 1 | index 2 | index 3 |

Dealing with Collisions

When you add lots of values to a hashtable, two elements may **collide**. This happens if they are assigned to the same index. For example, if we try to add both **"cmu"** and **"college"** to our table, they will collide.

Hashtables are designed to handle collisions. One algorithm for handling collisions is to put the collided values in a list and put that list in the bucket. *If* your table size is reasonably big and the indexes returned by the hash function are reasonably spread out, each bucket will normally hold a **constant** number of values.

Our example hash function is not good because it only looks at the first letter. A function that uses all the letters would be better.

```
def hash(s):
    return ord(s[0]) - ord('a')
```

"yay"	"book"	"cmu" "college"	
index 0	index 1	index 2	index 3

You Do: Search a Hashtable

Let's say that we want to algorithmically check whether the string **`"friday"`** is in our hashtable.

```
def hash(s):
    return ord(s[0]) - ord('a')
```

You do: Which buckets does the algorithm need to check?

"yay"	"book"	"cmu" "college"	
index 0	index 1	index 2	index 3

Activity Answer

You only have to check one index – index 1.

Searching a Hashtable is O(1)!

To search for a value, call the hash function on it and mod the result by the table size. **The index produced is the only index you need to check!**

For example, we can check if `"book"` is in the table just by checking bucket 1.

If the value is in the table, it will be at that index. If it isn't, it won't be anywhere else either. To check for `"stella"` just look in in bucket 2.

Because we only need to check one index and each index holds a constant number of items, finding a value is O(1).

```
def hash(s):
    return ord(s[0]) - ord('a')
```

"yay"	"book"	"cmu" "college"	
index 0	index 1	index 2	index 3

Caveat: Don't Hash Mutable Values!

What happens if you try to put a list in a hashtable? Let's set `lst = ["a", "z"]` and use the given hash to add `lst`.

This might seem fine at first, but it will become a problem if you change the list before searching. Let's say we set `lst[0] = "d"`.

When we hash the list again, the hashed value is `3`, not `0`. But the list isn't stored in bucket 3! We can't find it reliably.

For this reason, **we don't put mutable values into hashtables**. If you try to run the built-in `hash` on a list, it will crash.

```
def hash(s):
    return ord(s[0]) - ord('a')
```

"yay" ["a", "z"]	"book"	"cmu" "college"	
index 0	index 1	index 2	index 3

Learning Objectives

Recognize **linear search** and **binary search** when reading and writing code to search for items in sorted lists

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

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

Recognize the requirements for building a good **hash function** and a good **hashtable** that lead to **constant-time search**