Bonus Slide

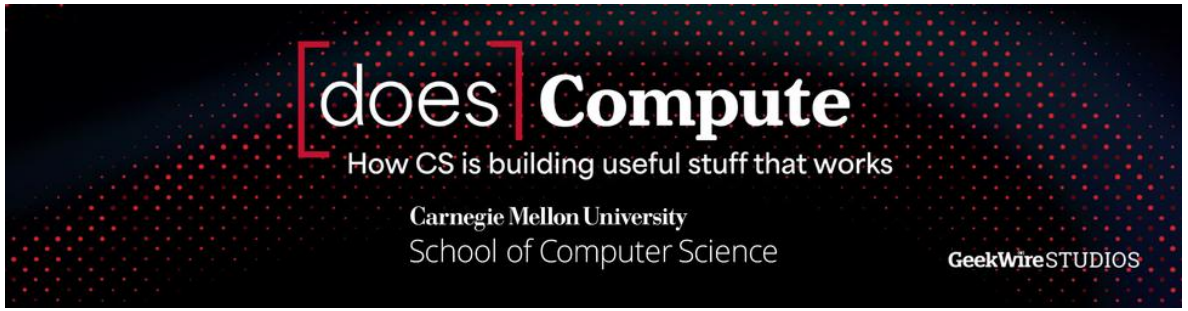
does Compute Podcast

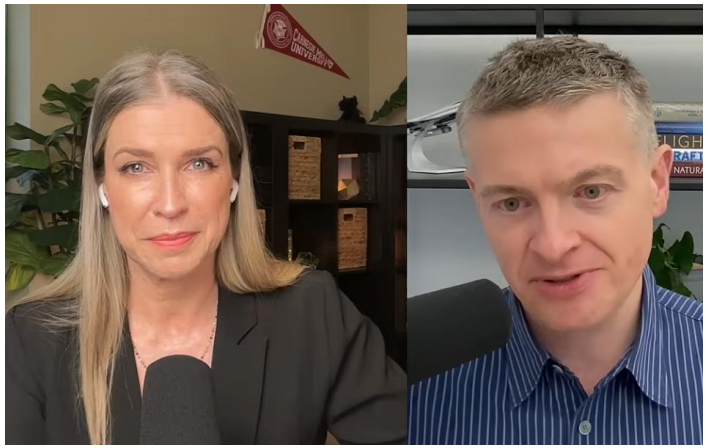
The School of Computer Science collaborates with GeekWire to make a podcast about interesting computer science topics.

Each episode features an interview with an SCS faculty member, alumni, or student who discusses the project they're working on in depth.

Check it out:

https://www.geekwire.com/does-compute/





Season 3 Episode 1
"Skeptics Don't Create the Future"
developing supersonic jets with simulation!

Unit 3: Comparing Algorithms

Search Algorithms

15-110 – Wednesday 11/04

Quizlet7

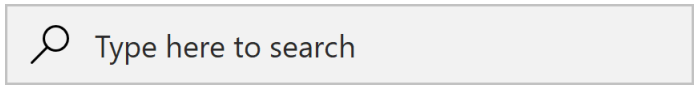
Learning Objectives

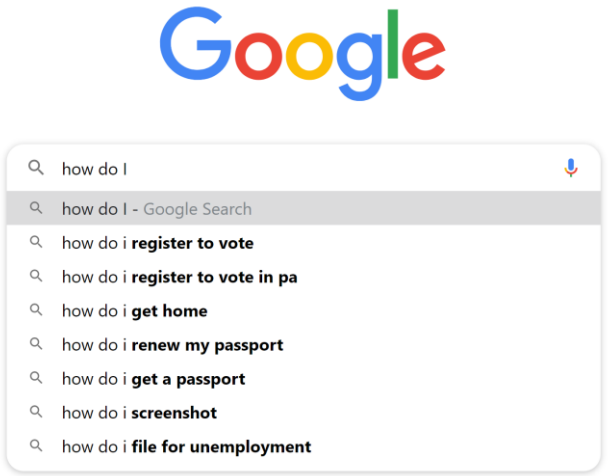
- Recognize and use **linear search** and **binary search** to search for items in given lists
- Recognize the requirements for building a good **hash function** and a good **hashtable** that lead to **constant-time search**
- Recognize whether or not two algorithms implement the same **functionality**.
- Identify any **restrictions** placed on the inputs of an algorithm to ensure it functions correctly.
- Compare similar algorithms to determine which is **more efficient**.

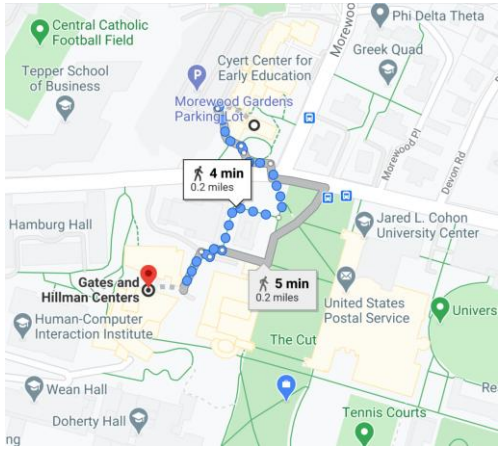
Linear Search

Searching for Items

Search is one of the most common tasks a computer needs to do. We'll discuss it in depth in this unit.







You use search in real life all the time too! Every time you manually look through papers or other physical documents for information, you conduct a search algorithm of your own.

Implementing Search

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?

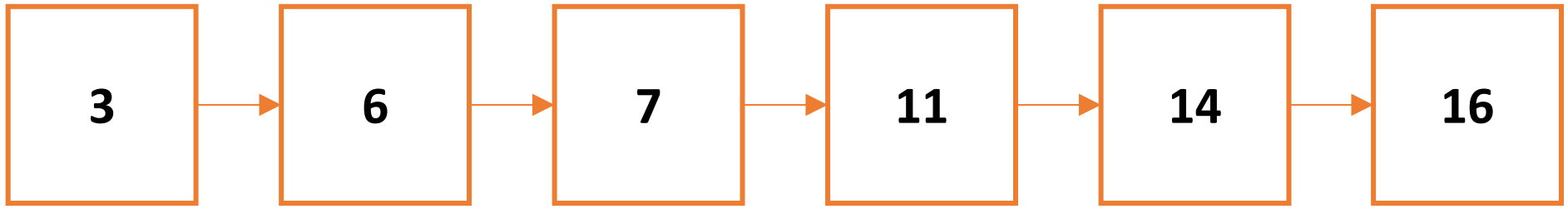
We'll need to think about this from a computer's perspective...

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.



Analogy: Searching a Stack of Books

This is like looking for a particular book in a stack of books.

You need to repeatedly check each book until you find the right title, or until you've checked them all.



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.

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

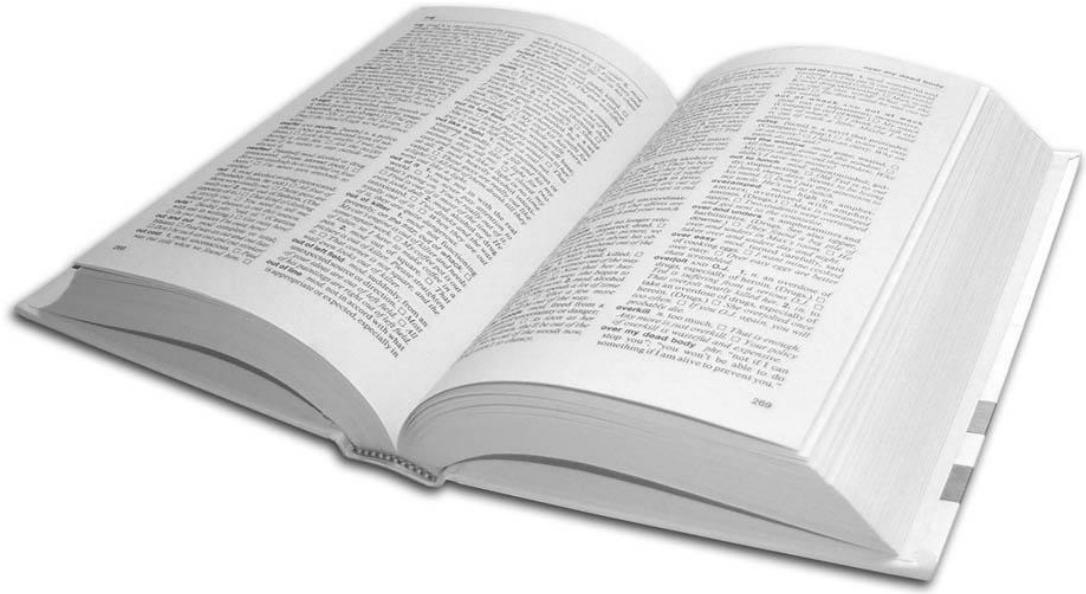
Note that we can return `True` as soon as we find the target value, but we can't return `False` until we've examined all the values.

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

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?

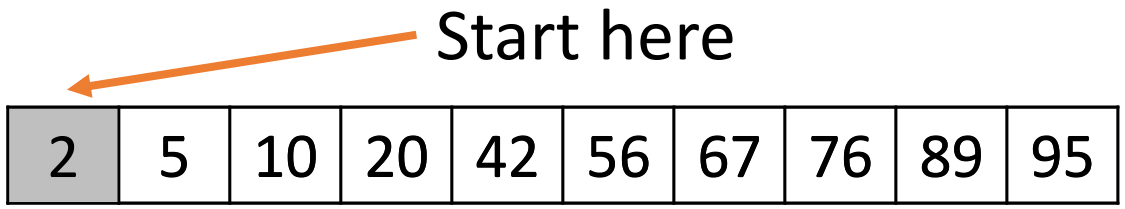


Can we take advantage of dictionaries being **sorted**?

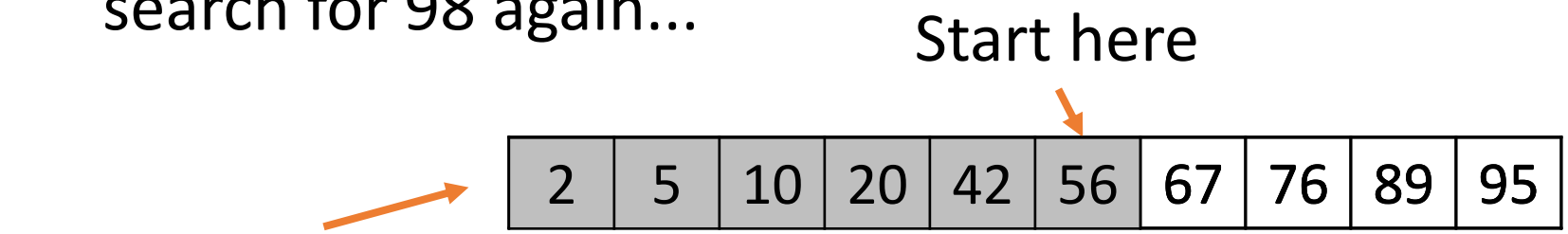
Binary Search

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...



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...



Analogy: Searching in a Library

If you're looking for a particular book in a library, you don't have to check every single book.

You can navigate to the right location because the books are **sorted** and you know your book's **author** already.

You can use existing information to speed up your algorithm!



Binary Search – Iterative

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

Binary Search – Iterative

1. Initialize two variables, `low` and `high`, marking the range of the list to still search

```
def binarySearch(lst, target):
    low = 0
    high = len(lst) - 1
```

Binary Search – Iterative

1. Initialize two variables, `low` and `high`, marking the range of the list to still search
2. Loop while the range is valid `low <= high`:

```
def binarySearch(lst, target):
    low = 0
    high = len(lst) - 1
    while low <= high:




    return False
```

Binary Search – Iterative

1. Initialize two variables, `low` and `high`, marking the range of the list to still search
2. Loop while the range is valid `low <= high`:
   a) Compute the middle index

What if there are an even number of elements? We'll break ties to the left.

```
def binarySearch(lst, target):
    low = 0
    high = len(lst) - 1
    while low <= high:
        mid = (low + high) // 2




    return False
```

Binary Search – Iterative

1. Initialize two variables, `low` and `high`, marking the range of the list to still search
2. Loop while the range is valid `low <= high`:
   a) Compute the middle index
   b) Compare the middle element to `target`:
      i. If they're equal, the target is found, return `True`

What if there are an even number of elements? We'll break ties to the left.

```
def binarySearch(lst, target):
    low = 0
    high = len(lst) - 1
    while low <= high:
        mid = (low + high) // 2
        if lst[mid] == target:
            return True



    return False
```

Binary Search – Iterative

1. Initialize two variables, `low` and `high`, marking the range of the list to still search
2. Loop while the range is valid `low <= high`:
   a) Compute the middle index
   b) Compare the middle element to `target`:
      i. If they're equal, the target is found, return `True`
      ii. If target is smaller, the target must be in the left half of the list thus we narrow the range by setting `high = mid - 1`.

What if there are an even number of elements? We'll break ties to the left.

```
def binarySearch(lst, target):
    low = 0
    high = len(lst) - 1
    while low <= high:
        mid = (low + high) // 2
        if lst[mid] == target:
            return True
        elif target < lst[mid]:
            high = mid - 1


    return False
```

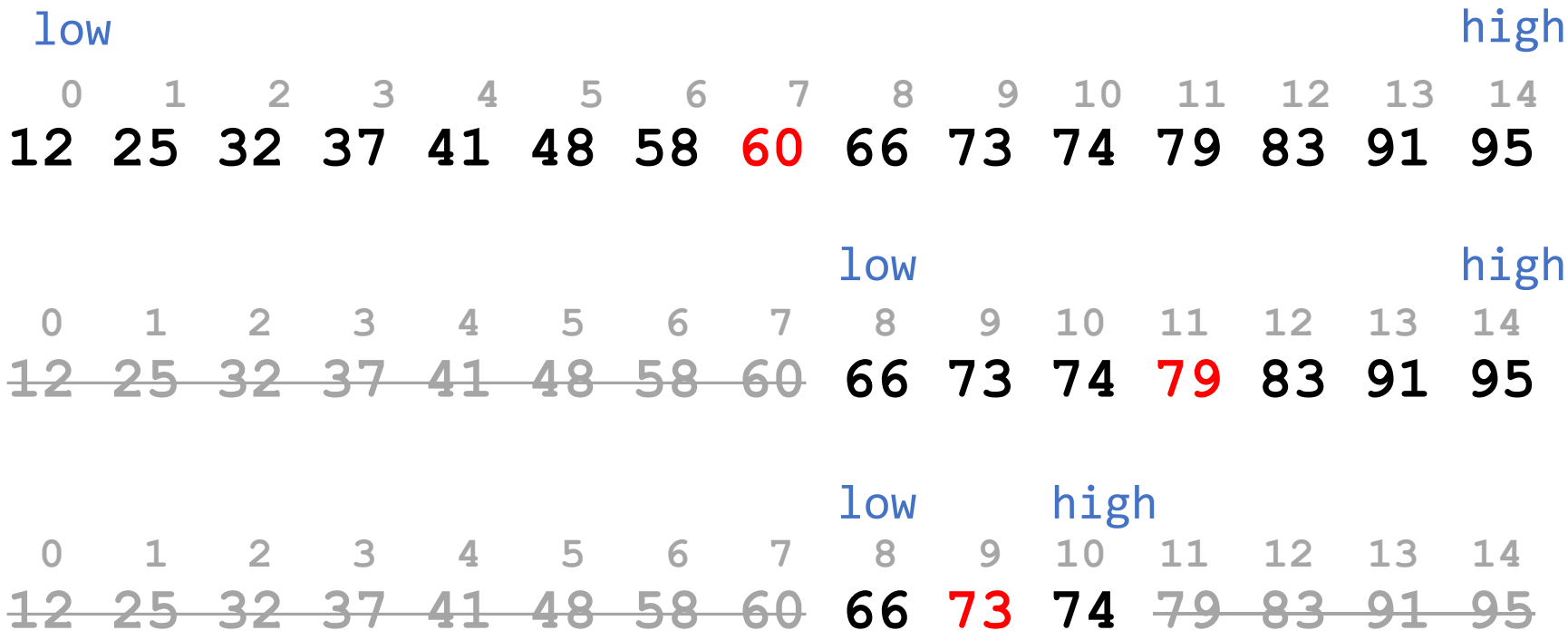
Binary Search – Iterative

1. Initialize two variables, `low` and `high`, marking the range of the list to still search
2. Loop while the range is valid `low <= high`:
   a) Compute the middle index
   b) Compare the middle element to `target`:
      i. If they're equal, the target is found, return `True`
      ii. If target is smaller, the target must be in the left half of the list thus we narrow the range by setting `high = mid - 1`.
      iii. Otherwise, target is larger and it must be in the right half of the list, so we set `low = mid + 1`.

What if there are an even number of elements? We'll break ties to the left.

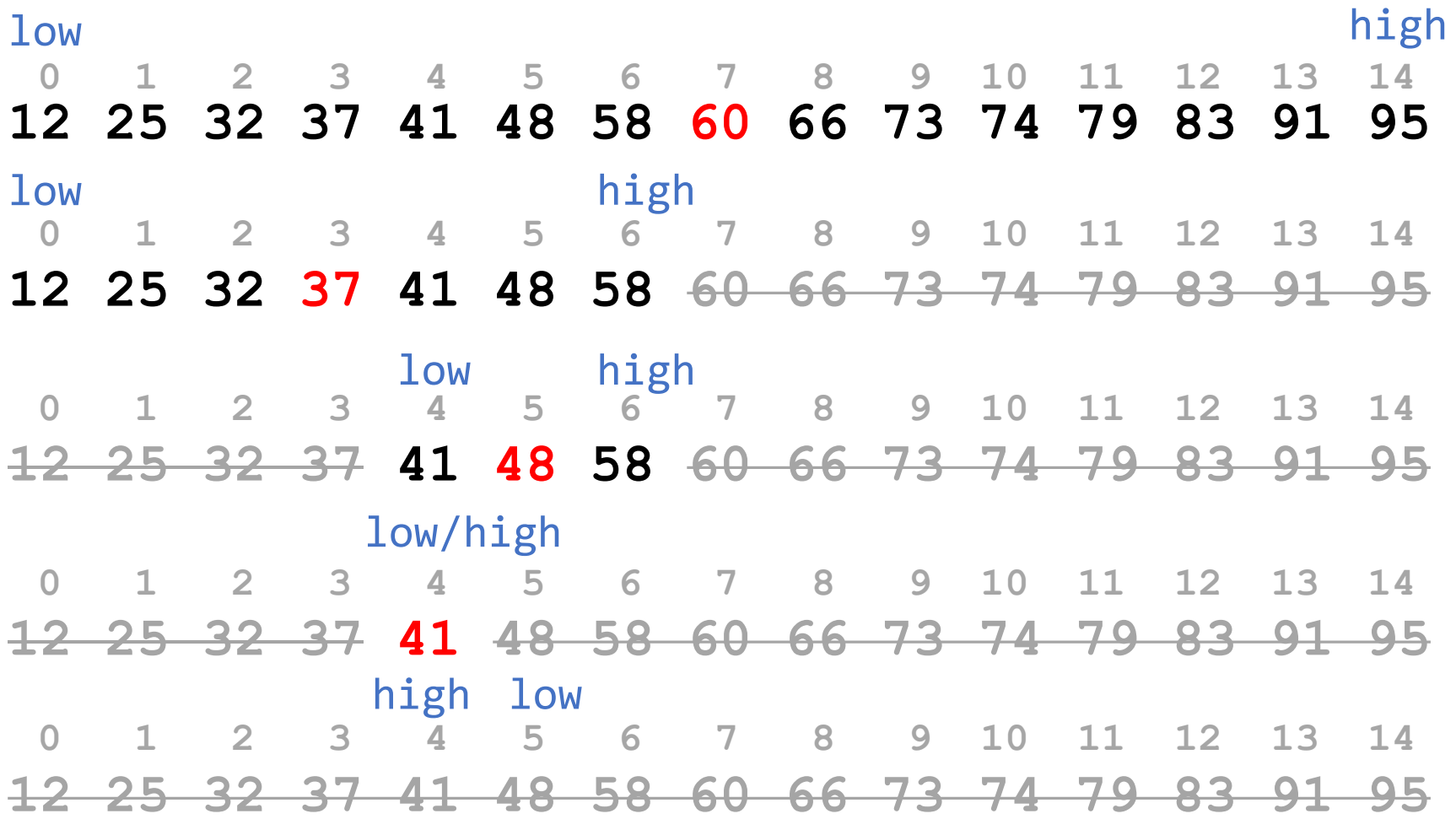
```
def binarySearch(lst, target):
    low = 0
    high = len(lst) - 1
    while low <= high:
        mid = (low + high) // 2
        if lst[mid] == target:
            return True
        elif target < lst[mid]:
            high = mid - 1
        else:
            low = mid + 1
    return False
```

Example 1: Search for 73



Found: return True

Example 2: Search for 42



Not found: return False

Activity: Trace Binary Search Using Code

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

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

```
def binarySearch(lst, target):
    low = 0
    high = len(lst) - 1
    while low <= high:
        mid = (low + high) // 2
        if lst[mid] == target:
            return True
        elif target < lst[mid]:
            high = mid - 1
        else:
            low = mid + 1
    return False
```

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 (as long as the list is already sorted).

This makes sense intuitively, but we don't yet have a way to **prove** that binary search is more efficient. We'll introduce a way to do this soon.

Can We Do Even Better?

Binary search is faster than linear search, but can we do even better?

We can often increase the efficiency of an algorithm by **thinking about the problem in a different way**. Try using a different data structure or an entirely different algorithmic approach to solve the problem. Or try putting new **constraints** on the problem to speed the process up.

New goal: can we add more constraints to design the **fastest possible** search algorithm?

Optimizing Search: Constraints

Search in Real Life – Post Boxes

Consider how you receive mail. Your mail is sent to the post boxes at the lower level of the UC. Do you have to check every box to find your mail?



No - you just check the box **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. No matter how many students go to the UC, finding your mail will always **take the same amount of time**.

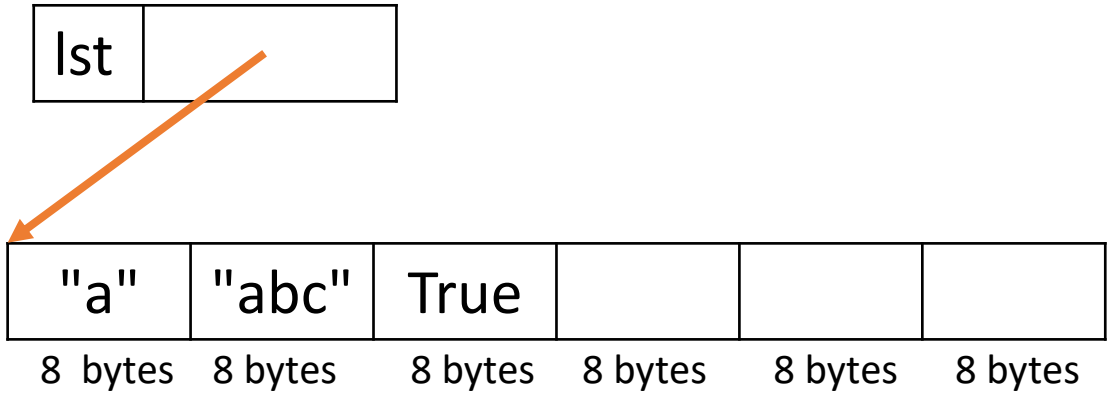
Compare this to picking up a package. Everyone picks up packages at the same window, so you must wait in line. The more students there are, **the longer it takes** for you to get the package.

Search in Programming – List Indexes

We can't search a list for an item that quickly, because we don't know where the item will be. But we **can** look up an item based on its index very quickly!

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**.

We can calculate exactly where an index is located in memory with a single equation; no repeated search is required.



Combine the Concepts

To implement super-fast search, we want to combine the ideas of post boxes and list index lookup. We want to determine which index a value should be stored in **based on the value itself**.

If we can calculate the index based on the value, we can look for the value really quickly without needing to check other indexes.

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:

- The function should be **deterministic**. Given a specific value `x`, `hash(x)` must **always** return the same output `i`.
- The function should produce **different outputs**. 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, because Python already has one!

```
x = "abc"
hash(x) # some giant number
```

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

Optimizing Search: Hashtables

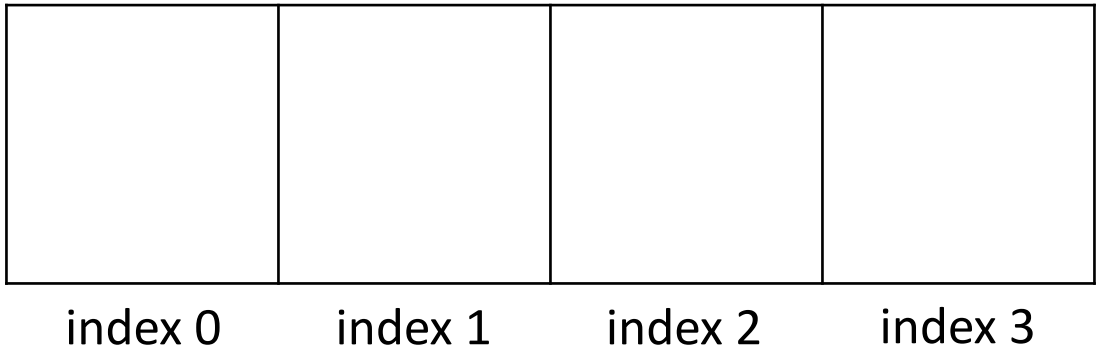
Hashtables Organize Values

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

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

We often call these indexes 'buckets'. For example, the hashtable to the right has four buckets.

Important: actual hashtables are **huge** and have far more buckets than this!



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.

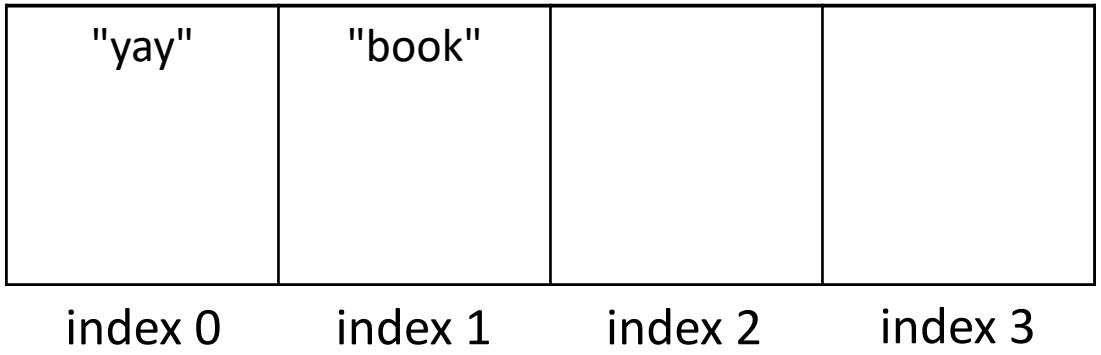
First, add `"book"` to the table. `hash("book")` is `1`, so we'll put the value in bucket 1.

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.

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. And our example hashtable is far too small!

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
def hash(s):
    return alphabet.index(s[0])
```

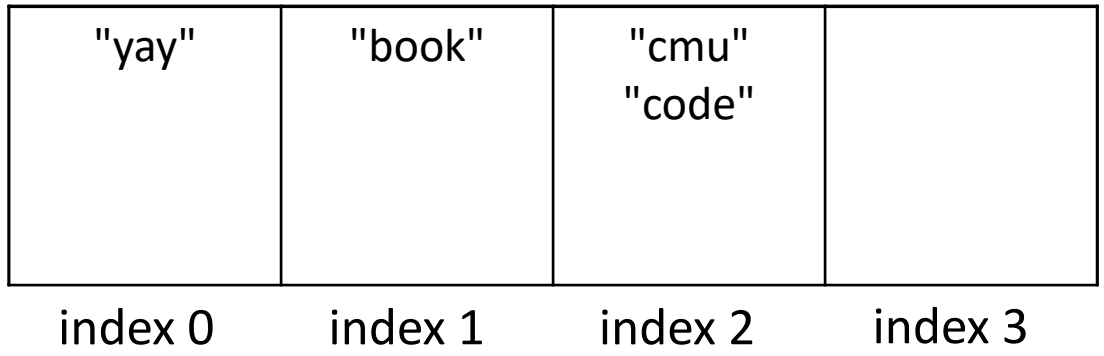


Dealing with Collisions

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 `"code"` to our table, they will collide.

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
def hash(s):
    return alphabet.index(s[0])
```

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 usually hold a small number of values.

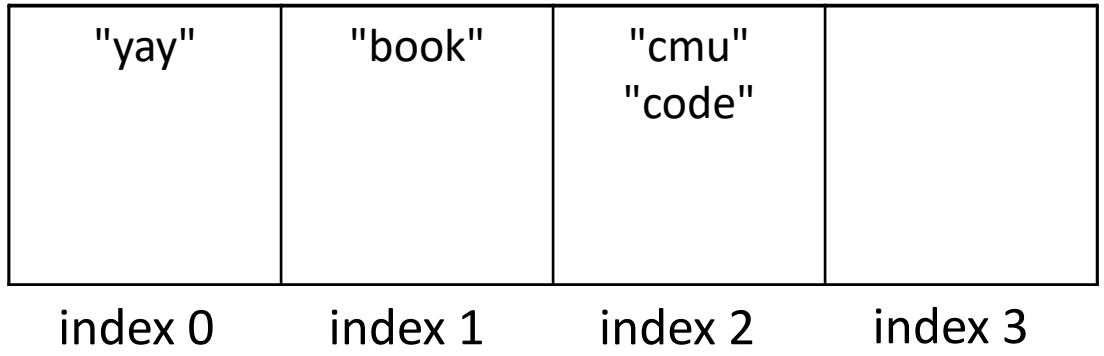


You Do: Search a Hashtable

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

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
def hash(s):
    return alphabet.index(s[0])
```

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



Searching a Hashtable is Fast!

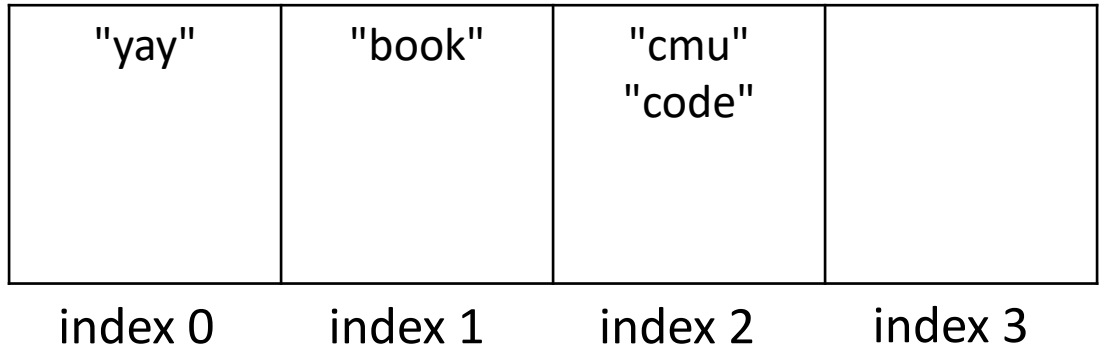
To search for a value, use the same algorithm you would use to insert that value. **The index produced is the only index you need to check!**

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 `"global"` 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 fast and does not require repeated search, **even if the hashtable is huge**. Wow!

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
def hash(s):
    return alphabet.index(s[0])
```



Caveat: Don't Hash Mutable Values!

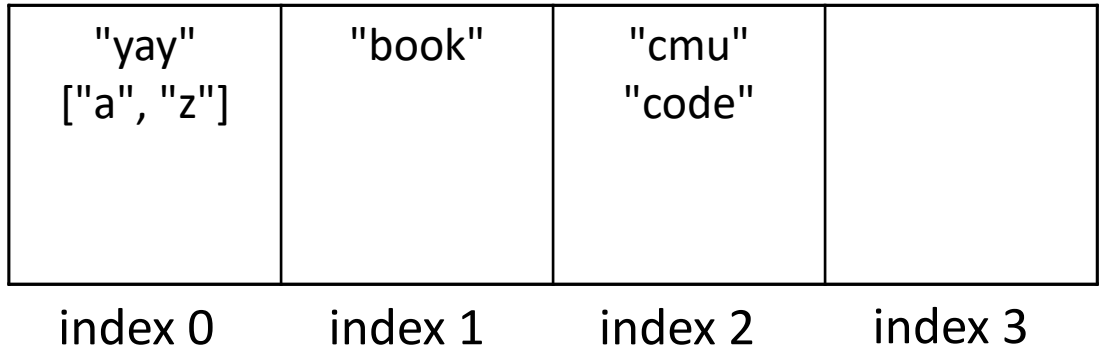
What happens if you try to put a list in a hashtable? Let's set `lst = ["a", "z"]` and use the given hash to add `lst`.

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.

```
alphabet = "abcdefghijklmnopqrstuvwxyz"
def hash(s):
    return alphabet.index(s[0])
```



Search Algorithms Strengths and Weaknesses

	Linear Search	**Binary Search**	**Hash Tables**
Works on unsorted data	Yes	No	Yes
Setup cost	None	Needs sorted data	Needs good hash function
Can find “next item in order”	Yes (slowly)	Yes	No, no sense of order

You do: Search Algorithm Selection

For each of the following scenarios, decide whether **linear search**, **binary search** or **hashed search** should be used.

- Looking up the translation of "cat" in an alphabetized English-Albanian print dictionary
- Finding your exam on a jumbled stack of returned exams
- Checking if a username is already taken when a new student signs up for a CMU Andrew ID
- Looking for your exam grade in a list of names and grades, sorted by grade

Learning Objectives

- Recognize and use **linear search** and **binary search** to search for items in given lists
- Recognize the requirements for building a good **hash function** and a good **hashtable** that lead to **constant-time search**
- Recognize whether or not two algorithms implement the same **functionality**.
- Identify any **restrictions** placed on the inputs of an algorithm to ensure it functions correctly.
- Compare similar algorithms to determine which is **more efficient**.