Introduction

You’ll learn to...

- Recognize the strengths and weaknesses of different search algorithms
- Use the `bisect` module to do a binary search in Python
- Implement a binary search in Python both recursively and iteratively
- Recognize and fix defects in a Python binary search implementation
- Analyze the time-space complexity of the binary search algorithm
- Search even faster than binary search

Introduction

What is a binary search?

Binary search is a method of "searching" a collection for a certain element.

```
x = [1, 2, 4, 6, 9]
```

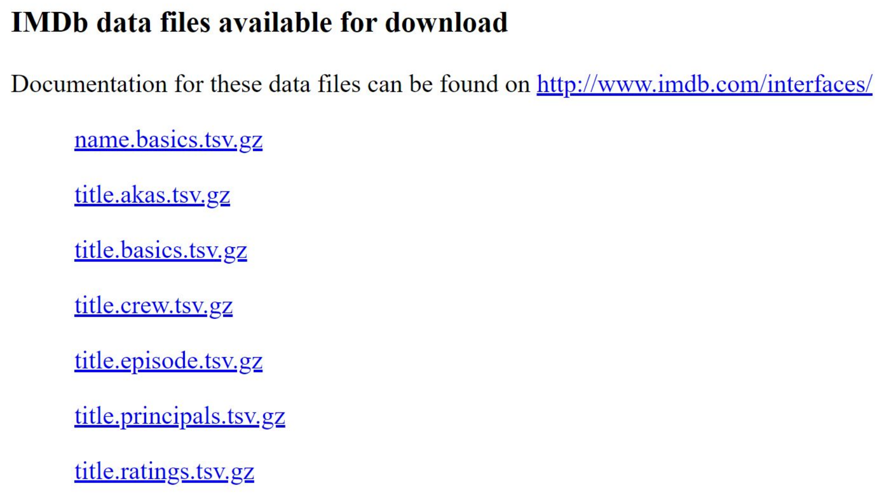
→
```
bin_search(x, 4) = 2
```

Unlike other search algorithms, a binary search requires the collection to be *sorted*, but it leverages this fact to run extremely quickly for all but the largest inputs

TABLE OF CONTENTS

Dataset

- A subset of the IMDb (Internet Movie Database) including millions of actor names.
- Available for non-commercial use at https://datasets.imdbws.com/
- Download and separate the data into `names.txt` and `sorted_names.txt` using the `download_imdb.py` script from the sample code
- Careful! These files will take up almost 1GB of space



IMDb data files available for download

Documentation for these data files can be found on http://www.imdb.com/interfaces/

- name.basics.tsv.gz
- title.akas.tsv.gz
- title.basics.tsv.gz
- title.crew.tsv.gz
- title.episode.tsv.gz
- title.principals.tsv.gz
- title.ratings.tsv.gz

Shell

```
$ python download_imdb.py
Fetching data from IMDb...
Created "names.txt" and "sorted_names.txt"
```

Benchmarking

- Many different ways to measure the performance of your code: time complexity, space complexity, control-flow analysis, and others
- I’ll mostly talk about runtime over the course of this series
    - Python libraries for timing your code include the in-built `time` module, the `timeit` library, and various other libraries
    - The runtimes that I show you were generated by a custom script using the `time.perf_counter_ns` function under the hood
    - Below this course, I’ll provide some links to resources that will show you more information on timing your code

TABLE OF CONTENTS

First idea: random search

- If I asked you to search a full backpack for an eraser, how would you do it?
- You might take something out, check to see if it’s the eraser, and then toss it back in if it’s not and repeat
- That’s random search!

```
from random import randint

def find(items, search_val):

  while True:

    rand_dex = randint(0, len(items)-1)

    rand_el = items[rand_dex]

    if rand_el == search_val:

      return rand_dex
```

Random search: sometimes fast, mostly slow

Search Term	Element Index	Best Time	Average Time	Worst Time
Fred Astaire	0	0.74s	21.69s	43.16s
Alicia Monica	4,500,000	1.02s	26.17s	66.34s
Baoyin Liu	9,500,000	0.11s	17.41s	51.03s
missing	N/A	5m 16s	5m 40s	5m 54s

Better idea: linear search

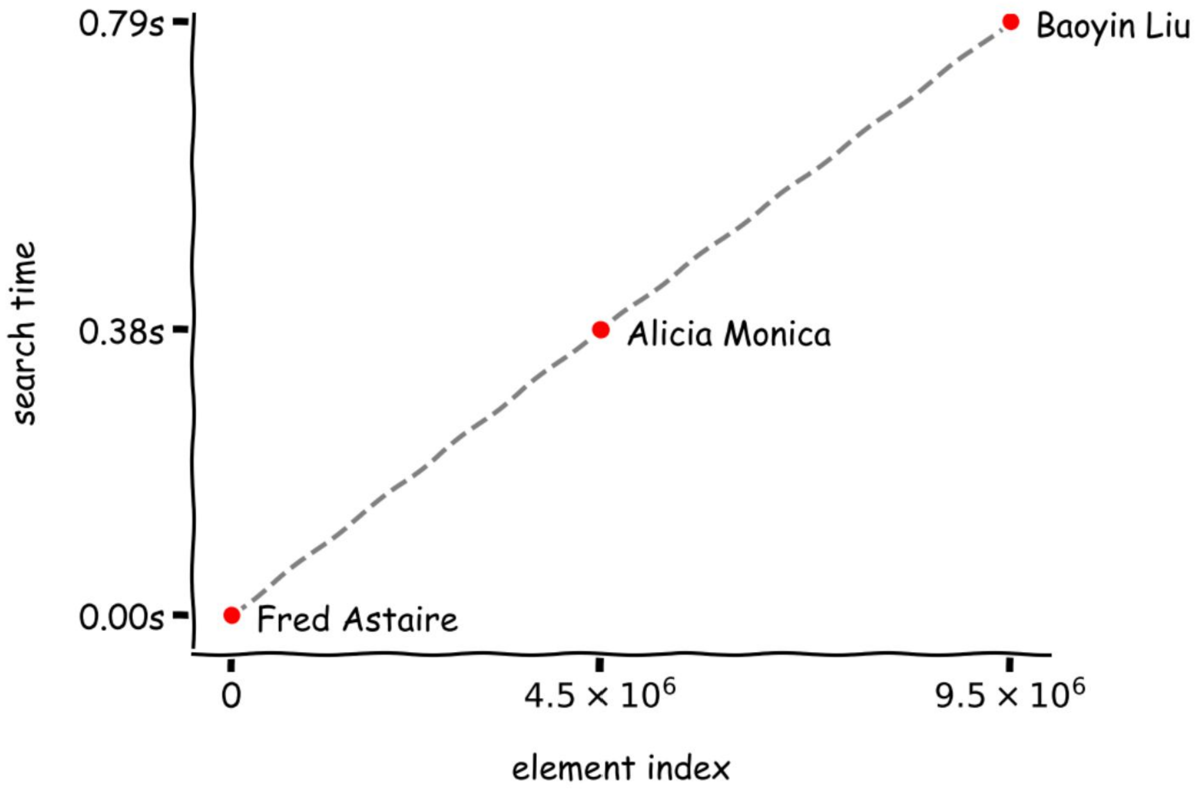
- Going back to the backpack analogy, once you’ve taken a ruler out of your backpack, you shouldn’t have to look at it again
- This is the basic idea of linear search: process the elements in your collection in some order, so that you can’t repeat elements

```python
def find_index(elements, value):
  for dex, el in enumerate(elements):
    if el == value:
      return dex
```

Linear search: faster, and much more consistent

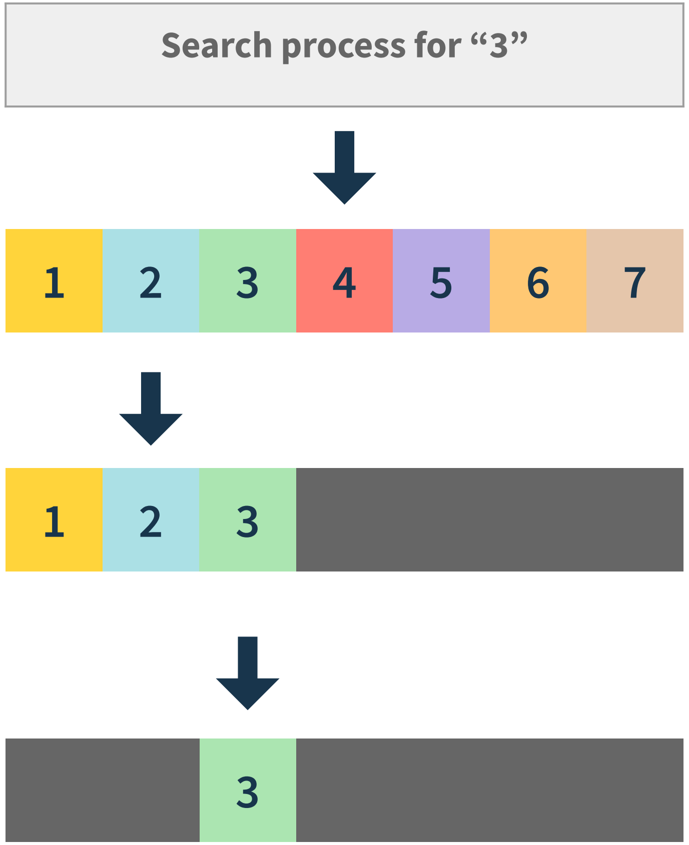
Search Term	Element Index	Best Time	Average Time	Worst Time
Fred Astaire	0	491ns	1.17µs	6.1µs
Alicia Monica	4,500,000	0.37s	0.38s	0.39s
Baoyin Liu	9,500,000	0.77s	0.79s	0.82s
missing	N/A	0.79s	0.81s	0.83s

But...


0.79s
0.38s
0.00s
search time
Baoyin Liu
Alicia Monica
Fred Astaire
0
4.5 × 10^6
9.5 × 10^6
element index

Finally: binary search

- If the elements of your collection are sorted, you can eliminate roughly half of the possibilities with each comparison!
- Imagine flipping through the pages of a book — if your current page is earlier than you want to be, you can discard all earlier pages, and vice versa.



Binary search: amazing speed

Search Term	Element Index	Average Time	Comparisons
(…) Berendse	0	6.52µs	23
Jonathan Samuangte	4,499,997	6.99µs	24
Yorgos Rahmatoulin	9,500,001	6.5µs	23
missing	N/A	7.2µs	23

TABLE OF CONTENTS

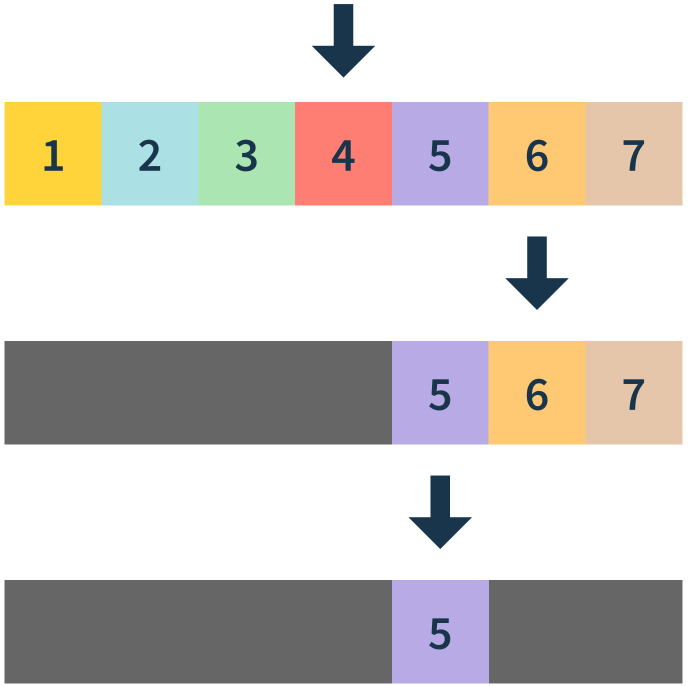
The bisect module: useful functions

- `bisect(lst, item)` aka. `bisect_right(lst, item)`: Finds the index at which `item` could be inserted in `lst` to maintain sorted order. In the case of duplicates, returns the right-most index
- `bisect_left(lst, item)`: same as `bisect`, but returns the left-most possible index—generates the index of existing items
- `insort(lst, item)` aka. `insort_right(lst, item)`: Just like `bisect`, but inserts the item at the index found (rightmost possible)
- `insort_left(lst, item)`: inserts at the left-most possible index

TABLE OF CONTENTS

The dex_mod parameter

- The recursive binary search algorithm can return the wrong index if you’re not careful.
- Remember to adjust for the sub-list indexing problem!



The element 5 has an index of 4 in the original list.
But after the first recursive call, its index is 0 in the new list slice.
On the second (and final) recursive call, 5’s index remains 0.

TABLE OF CONTENTS

Integer overflow

```python
def binary_search(elements, item):

    middle = (left + right) // 2 # May overflow
```

✗

```python
def binary_search(elements, item):

    middle = left + (right - left) // 2 # Can’t overflow
```

✓

Stack overflow

- Python, like most programming languages, has a “recursion depth limit” — the number of recursive calls a function can make before throwing an error
- A good binary search implementation should never cause a stack overflow, but it’s easy to write bugs that do!

```python
def binary_search(els, item, dex_mod):

    ...

    if middle_el > item:

      return

        binary_search(

          els, item, dex_mod

        )

    ...
```

Duplicate elements

- Already discussed: how to write binary search algorithms that return the left- or right-most index of a given item
- But always remember that the basic binary search algorithm is not perfectly stable

```
>>> elements = [1, 2, 2, 4]
>>> binary_search(elements, 2)
1
>>> elements += [5]
>>> elements
[1, 2, 2, 4, 5]
>>> binary_search(elements, 2)
2
```

Floating-point rounding

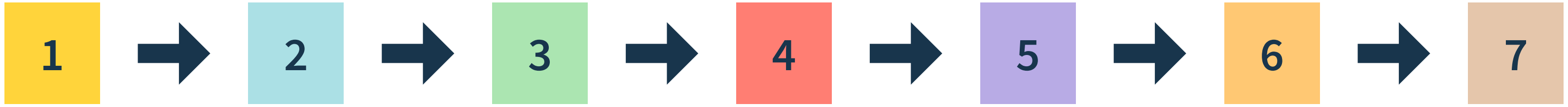
- Floating-point numbers can generate strange comparisons in Python, due to weaknesses in how computers represent floats
- It's often better to decide on the precision you need, then use integers to avoid this issue
- But you can also use `math.isclose()` to generate more accurate comparisons

```
>>> floats = [.1 * i for i in range(10)]

>>> .1 in floats

true

>>> .2 in floats

true

>>> .3 in floats

false

>>> math.isclose(.3, floats[3])

true
```

TABLE OF CONTENTS

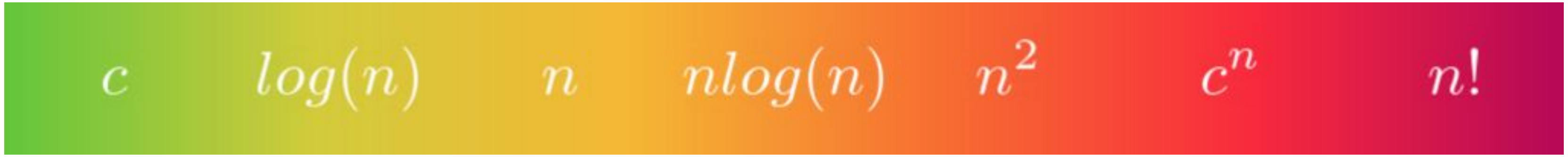
Time-space complexity

- Big O notation: represents the worst case runtime of an algorithm as a function of the size of the input, or *n*.
- Example: linear search is an *O(n)* algorithm because if your input list has length n, and the thing that you're searching for is the last element in the list, it will take n comparison operations to find that thing. Its space complexity, on the other hand, is O(1), or constant, because it requires no space beyond the size of the input to implement the algorithm.



Time-space complexity

Algorithms can be divided into different “classes” based on their worst-case runtime. Of course, certain algorithms in the same classes might be faster or slower than one another in practice, but this is an easy way to classify runtimes generally.

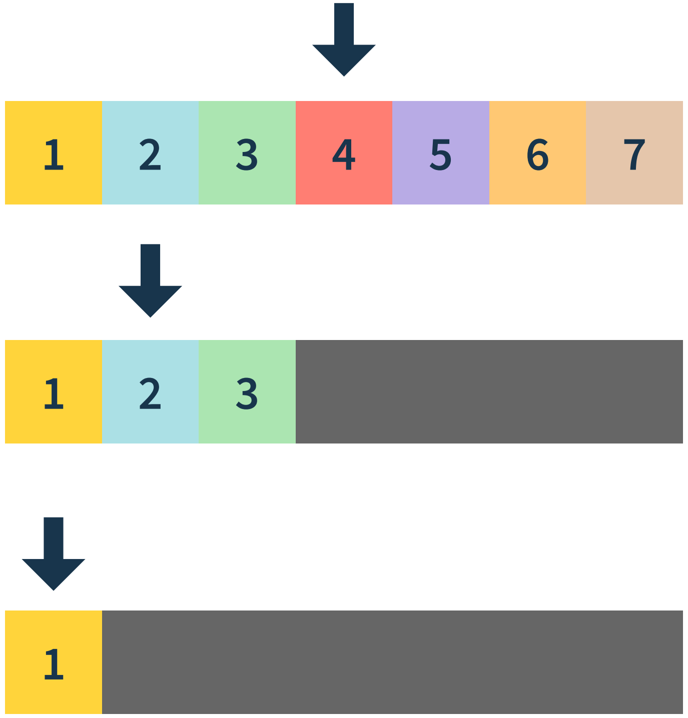


Common runtimes of algorithms

Time-space complexity of binary search

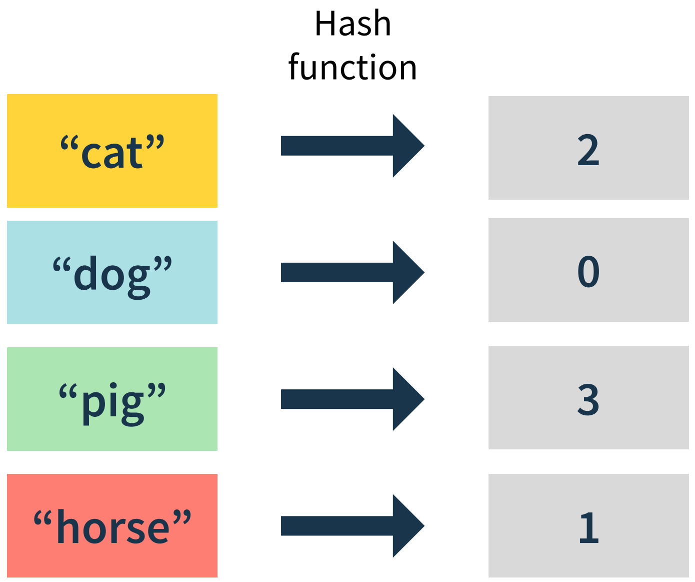
So what are the time and space complexities of binary search? Let’s think about the worst case—when the item you seek is at the beginning or the end of the list.

Comparisons	Elements Left
0	n
1	n/2
2	n/4
…	…
$\log_2(n)$	1



Even faster than binary search

- How can you search even faster than with a binary search?
- The key is by diminishing the search-space even faster. One way to do that is with a hash table or dictionary.
- Hash tables are generally considered to have O(1), or constant-time, lookup speed
- However, they balance this with increased space usage



Value-Index mapping for the python list
`[“dog”, “horse”, “cat”, “pig”]`

Conclusion

You’ve learned about...

- Several different search algorithms and why you might or might not want to use them
- The different ways to perform a binary search in Python
- Some common pitfalls when implementing binary search
- Time-space complexity in binary search and when using hash tables