The Art of Data Structures

Searching

Alan Beadle
CSC 162: The Art of Data Structures
UNIVERSITY of ROCHESTER

Agenda

- To be able to explain and implement sequential and binary search

Searching

Searching

- Searching is the process of looking for a particular value in a collection
- For example, a program that maintains a membership list for a club might need to look up information for a particular member – this involves some sort of search process

Searching

A Simple Searching Problem

- Here is the specification of a simple searching function:

```
def search(x, nums):
    # nums is a list of numbers and x is a number
    # Returns the position in the list where x occurs
     # or -1 if x is not in the list.
```

- Here are some sample interactions:

```
>>> search(4, [3, 1, 4, 2, 5])
2
>>> search(7, [3, 1, 4, 2, 5])
-1
```

Searching

A Simple Searching Problem

- In the first example, the function returns the index where 4 appears in the list
- In the second example, the return value -1 indicates that 7 is not in the list
- Python includes a number of built-in search-related methods!

Searching

A Simple Searching Problem

- We can test to see if a value appears in a sequence using in.

```
if x in nums:
    # do something
```

- If we want to know the position of x in a list, the index method can be used.

```
>>> nums = [3, 1, 4, 2, 5]
>>> nums.index(4)
2
```

Searching

A Simple Searching Problem

- The only difference between our `search` function and `index` is that `index` raises an exception if the target value does not appear in the list
- We could implement `search` using `index` by simply catching the exception and returning `-1` for that case

Searching

A Simple Searching Problem

```
def search(x, nums):
    try:
        return nums.index(x)
    except:
        return -1
```

- Sure, this will work, but we are really interested in the algorithm used to actually search the list in Python!

Sequential Search

Sequential Search

Sequential Search

- Pretend you're the computer, and you were given a page full of randomly ordered numbers and were asked whether 13 was in the list
- How would you do it?
- Would you start at the top of the list, scanning downward, comparing each number to 13? If you saw it, you could tell me it was in the list. If you had scanned the whole list and not seen it, you could tell me it wasn't there.

Sequential Search

Sequential Search

- This strategy is called a linear, or sequential search, where you search through the list of items one by one until the target value is found

Sequential Search

Sequential Search

- The Python `in` and `index` operations both implement linear searching algorithms
- If the collection of data is very large, it makes sense to organize the data somehow so that each data value doesn’t need to be examined

Sequential Search

Sequential Search

- If the data is sorted in ascending order (lowest to highest), we can skip checking some of the data
- As soon as a value is encountered that is greater than the target value, the linear search can be stopped without looking at the rest of the data
- On average, this will save us about half the work

Sequential Search

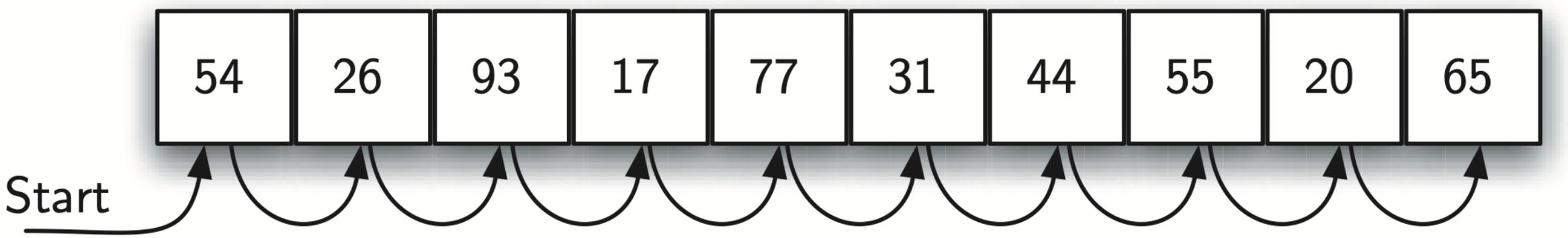
Sequential Search

- This algorithm wasn't hard to develop, and works well for modest-sized lists

```
def search(x, nums):
    for i in range(len(nums)):
        if nums[i] == x:
            # item found, return the index value
            return i
    # loop finished, item was not in list
    return -1
```

Sequential Search

Of a List of Integers



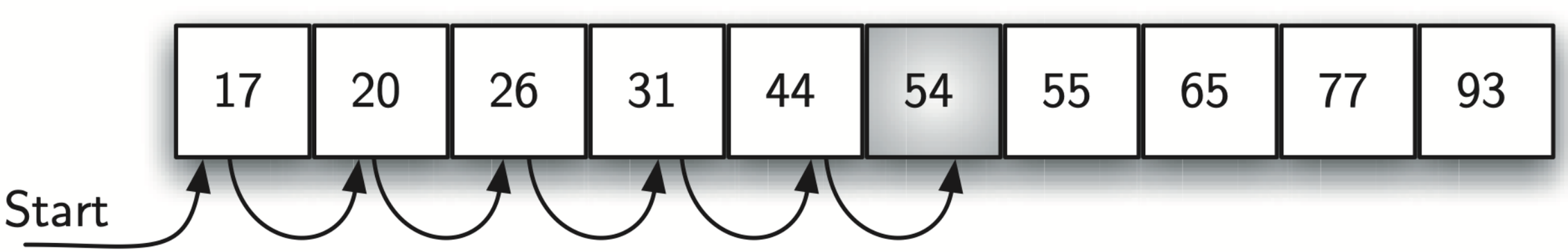
Sequential Search

Of an Unordered List

```
def sequential_search(alist, item):
    pos = 0
    found = False

    while pos < len(alist) and not found:
        if alist[pos] == item:
            found = True
        else:
            pos = pos+1

    return found

testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(sequential_search(testlist, 3))
print(sequential_search(testlist, 13))
```

Sequential Search

Of an Ordered List



Sequential Search

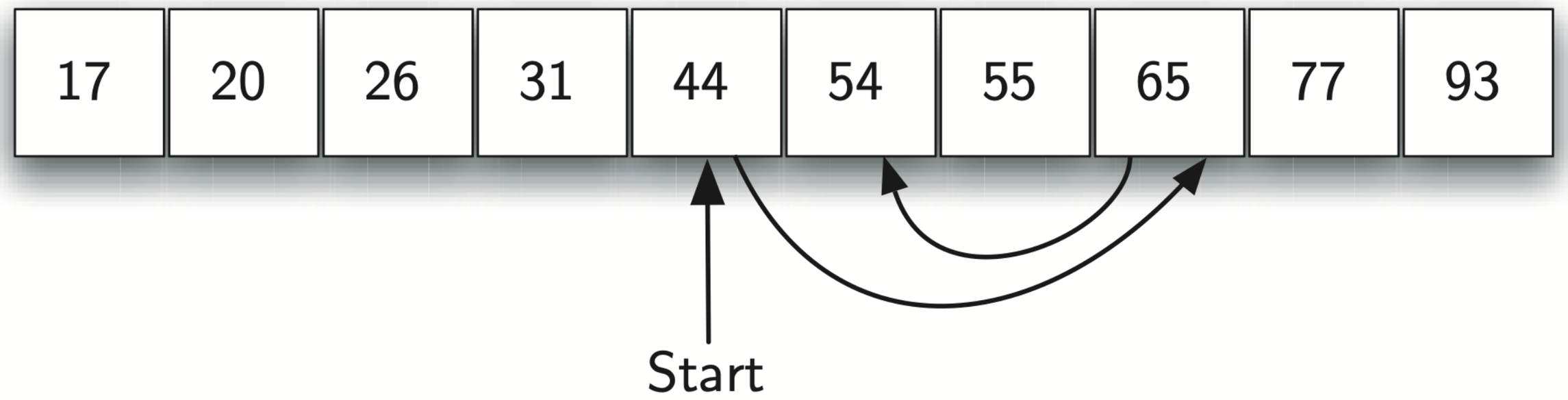
Of an Ordered List

```
def ordered_sequential_search(alist, item):
    pos = 0
    found = False
    stop = False
    while pos < len(alist) and not found and not stop:
        if alist[pos] == item:
            found = True
        else:
            if alist[pos] > item:
                stop = True
            else:
                pos = pos+1

    return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(ordered_sequential_search(testlist, 3))
print(ordered_sequential_search(testlist, 13))
```

Binary Search

Binary Search

Of an Ordered List of Integers



Binary Search

Of an Ordered List

```
def binary_search(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first <= last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        else:
            if item < alist[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1
    return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binary_search(testlist, 3))
print(binary_search(testlist, 13))
```

Binary Search
Of an Ordered List

```
def binary_search_r(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist)//2
        if alist[midpoint] == item:
          return True
        else:
          if item < alist[midpoint]:
            return binary_search_r(alist[:midpoint], item)
          else:
            return binary_search_r(alist[midpoint+1:], item)

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binary_search_r(testlist, 3))
print(binary_search_r(testlist, 13))
```

Analysis

Analysis

Comparing Algorithms

- Which search algorithm is better, linear or binary?
- The linear search is easier to understand and implement
- The binary search is more efficient since it doesn’t need to look at each element in the list
- Intuitively, we might expect the linear search to work better for small lists, and binary search for longer lists. But how can we be sure?

Analysis

Comparing Algorithms

- One way to conduct the test would be to code up the algorithms and try them on varying sized lists, noting the runtime.
  - Linear search is generally faster for lists of length 10 or less
  - There was little difference for lists of 10-1000
  - Binary search is best for 1000+ (for one million list elements, binary search averaged .0003 seconds while linear search averaged 2.5 seconds)

Analysis
Comparing Algorithms

- While interesting, can we guarantee that these empirical results are not dependent on the type of computer they were conducted on, the amount of memory in the computer, the speed of the computer, etc.?
- We could abstractly reason about the algorithms to determine how efficient they are. We can assume that the algorithm with the fewest number of “steps” is more efficient.

Analysis
Comparing Algorithms

- How do we count the number of “steps”?
- Computer scientists attack these problems by analyzing the number of steps that an algorithm will take relative to the size or difficulty of the specific problem instance being solved.

Analysis

Comparing Algorithms

- For searching, the difficulty is determined by the size of the collection – it takes more steps to find a number in a collection of a million numbers than it does in a collection of 10 numbers.
- How many steps are needed to find a value in a list of size n?
- In particular, what happens as n gets very large?

Analysis
Comparing Algorithms

- Let’s consider linear search.
  - For a list of 10 items, the most work we might have to do is to look at each item in turn – looping at most 10 times.
  - For a list twice as large, we would loop at most 20 times.
  - For a list three times as large, we would loop at most 30 times!

Analysis

Comparing Algorithms

- The amount of time required is linearly related to the size of the list, `n`
- This is what computer scientists call a *linear time* algorithm

Analysis
Comparing Algorithms

- Now, let's consider binary search
  - Suppose the list has 16 items. Each time through the loop, half the items are removed. After one loop, 8 items remain
  - After two loops, 4 items remain
  - After three loops, 2 items remain
  - After four loops, 1 item remains

Analysis

Comparing Algorithms

- If a binary search loops `i` times, it can find a single value in a list of size 2^i

Analysis
Comparing Algorithms

- To determine how many items are examined in a list of size `n`, we need to solve:

$$n = 2^i$$

for `i`, or:

$$i = log_2 n$$

- Binary search is an example of a log time algorithm – the amount of time it takes to solve one of these problems grows as the *log* of the problem size

Analysis

Comparing Algorithms

- This logarithmic property can be very powerful!
- Suppose you have the New York City phone book with 12 million names
- You could walk up to a New Yorker and, assuming they are listed in the phone book, make them this proposition: “I’m going to try guessing your name. Each time I guess a name, you tell me if your name comes alphabetically before or after the name I guess.” How many guesses will you need?

Analysis
Comparing Algorithms

- Our analysis shows us the answer to this question is

$$log_2 12000000$$

- We can guess the name of the New Yorker in 24 guesses!
  - By comparison, using the linear search we would need to make, on average, *6,000,000* guesses!

Analysis
Comparing Algorithms

- Earlier, we mentioned that Python uses linear search in its built-in searching methods. Why doesn't it use binary search?
  - Binary search requires the data to be sorted
  - If the data is unsorted, it must be sorted first!

Questions?