THE UNIVERSITY OF WESTERN AUSTRALIA

Lecture 18
Search algorithms

Objectives

- To know what searching is and understand the algorithms for linear and binary search.
- To understand the basic techniques for analyzing the efficiency of algorithms.

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 a search process.
  - *Range from simple search in a list to search across huge databases, e.g. Google search*

Simple Searching

- 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 None 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])
None
```

Simple Searching Python

- The Boolean list method `in` tests for list membership

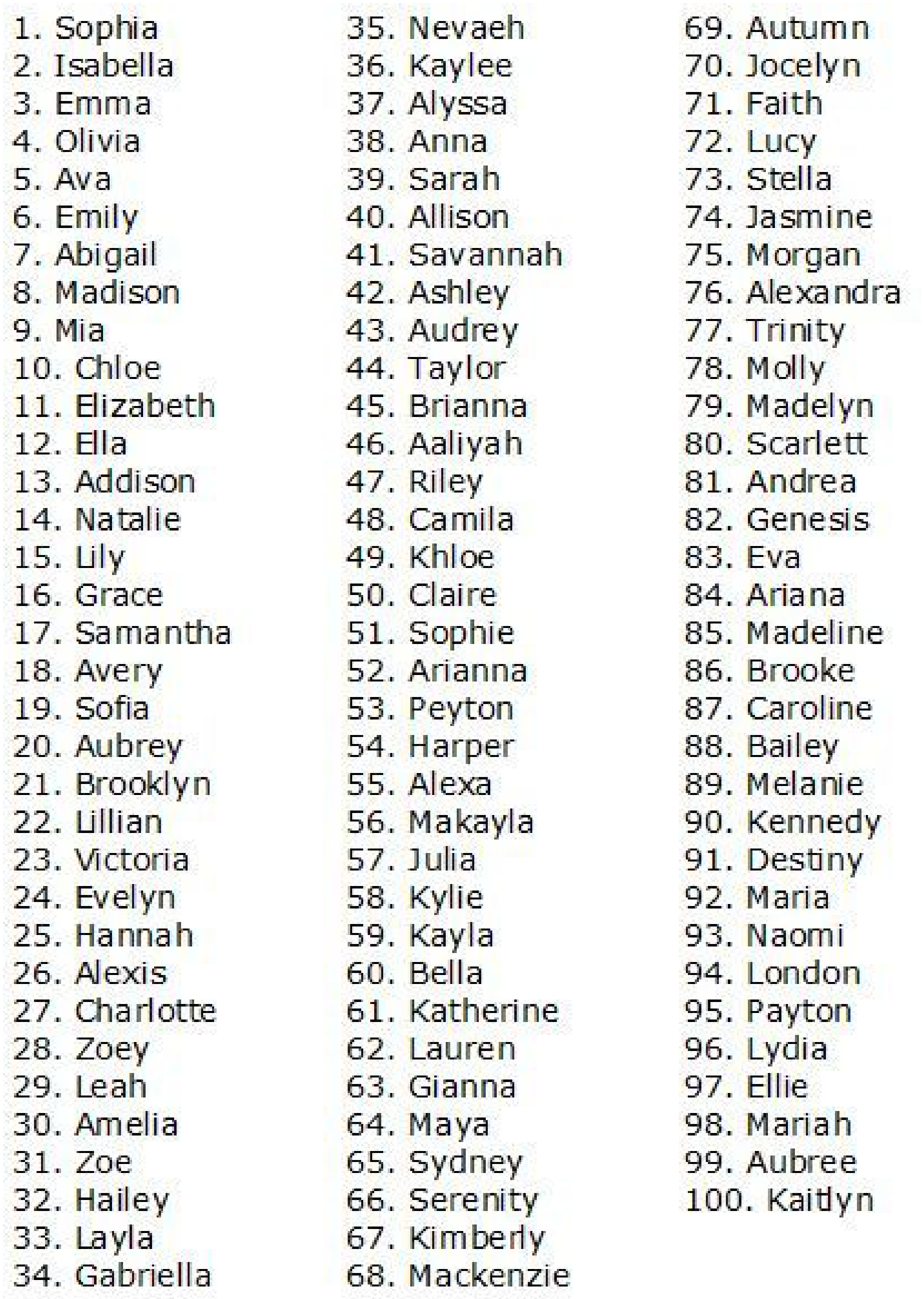
```
 x in nums
```

- Use `index()` to find position of item in a list

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

- Only problem is that the index method raises an exception if the sought item is not present

Example: Find Julia and Mandy



1. Sophia
2. Isabella
3. Emma
4. Olivia
5. Ava
6. Emily
7. Abigail
8. Madison
9. Mia
10. Chloe
11. Elizabeth
12. Ella
13. Addison
14. Natalie
15. Lily
16. Grace
17. Samantha
18. Avery
19. Sofia
20. Aubrey
21. Brooklyn
22. Lillian
23. Victoria
24. Evelyn
25. Hannah
26. Alexis
27. Charlotte
28. Zoey
29. Leah
30. Amelia
31. Zoe
32. Hailey
33. Layla
34. Gabriella
35. Nevaeh
36. Kaylee
37. Alyssa
38. Anna
39. Sarah
40. Allison
41. Savannah
42. Ashley
43. Audrey
44. Taylor
45. Brianna
46. Aaliyah
47. Riley
48. Camila
49. Khloe
50. Claire
51. Sophie
52. Arianna
53. Peyton
54. Harper
55. Alexa
56. Makayla
57. Julia
58. Kylie
59. Kayla
60. Bella
61. Katherine
62. Lauren
63. Gianna
64. Maya
65. Sydney
66. Serenity
67. Kimberly
68. Mackenzie
69. Autumn
70. Jocelyn
71. Faith
72. Lucy
73. Stella
74. Jasmine
75. Morgan
76. Alexandra
77. Trinity
78. Molly
79. Madelyn
80. Scarlett
81. Andrea
82. Genesis
83. Eva
84. Ariana
85. Madeline
86. Brooke
87. Caroline
88. Bailey
89. Melanie
90. Kennedy
91. Destiny
92. Maria
93. Naomi
94. London
95. Payton
96. Lydia
97. Ellie
98. Mariah
99. Aubree
100. Kaitlyn

Linear Search

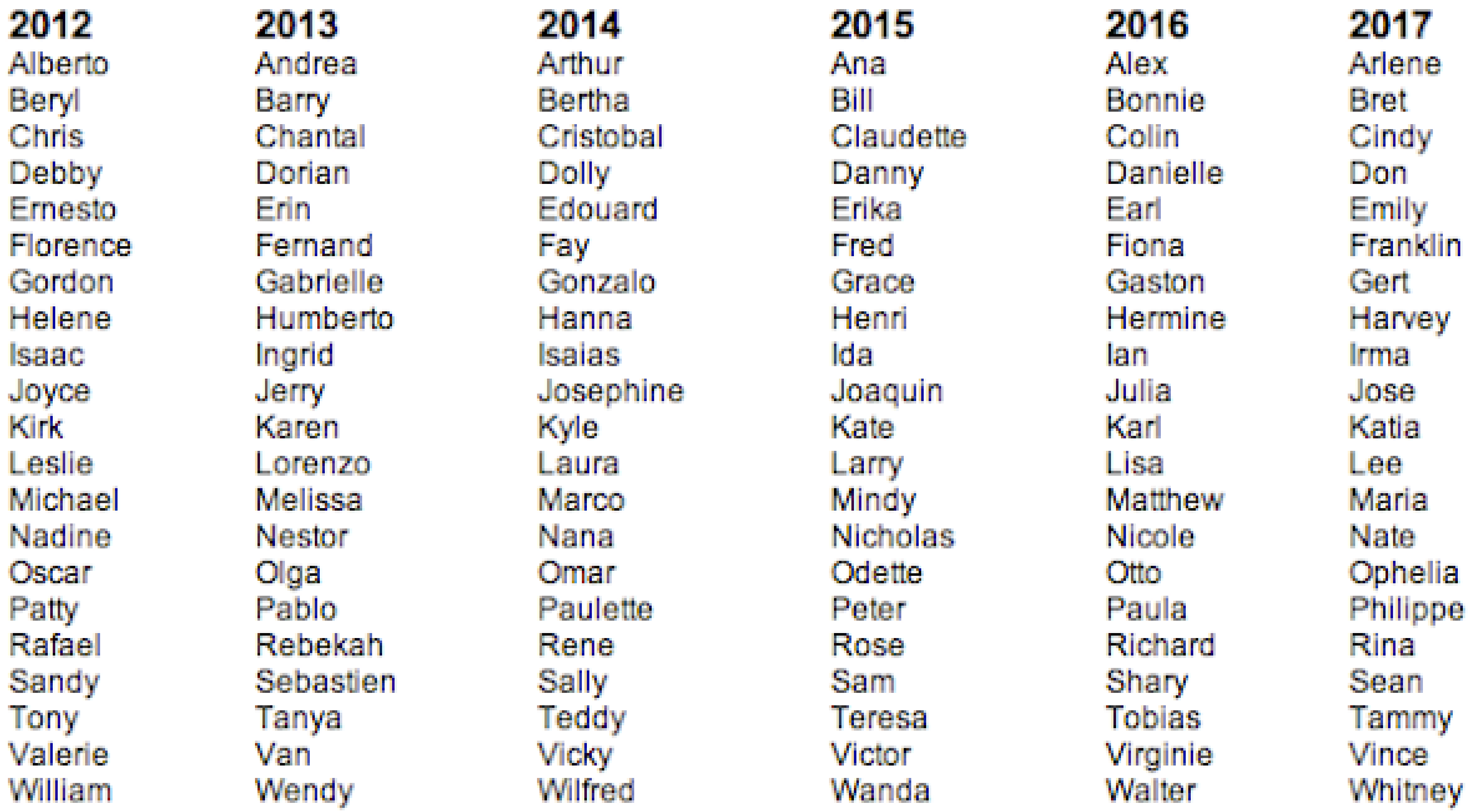
- Say you are given a page full of randomly ordered numbers and are asked whether 13 is in the list.
- You may start at the front of the list, comparing each number to 13
- If you see it, you can say that it is in the list. If you have scanned the whole list and not seen it, you will tell me it isn't there.
- This is called linear search.

Linear Search

```
def search(x, nums):
  for i in range(len(nums)):
    if nums[i] == x: # item found
      return i  #return index value
  return None   #loop finished, item not in list
```

- This algorithm wasn't hard to develop, and works well for modest-sized lists
- 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.
  – *Avoid non-solutions*

Find Omar and Otto



2012	2013	2014	2015	2016	2017
Alberto	Andrea	Arthur	Ana	Alex	Arlene
Beryl	Barry	Bertha	Bill	Bonnie	Bret
Chris	Chantal	Cristobal	Claudette	Colin	Cindy
Debby	Dorian	Dolly	Danny	Danielle	Don
Ernesto	Erin	Edouard	Erika	Earl	Emily
Florence	Fernand	Fay	Fred	Fiona	Franklin
Gordon	Gabrielle	Gonzalo	Grace	Gaston	Gert
Helene	Humberto	Hanna	Henri	Hermine	Harvey
Isaac	Ingrid	Isaias	Ida	Ian	Irma
Joyce	Jerry	Josephine	Joaquin	Julia	Jose
Kirk	Karen	Kyle	Kate	Karl	Katia
Leslie	Lorenzo	Laura	Larry	Lisa	Lee
Michael	Melissa	Marco	Mindy	Matthew	Maria
Nadine	Nestor	Nana	Nicholas	Nicole	Nate
Oscar	Olga	Omar	Odette	Otto	Ophelia
Patty	Pablo	Paulette	Peter	Paula	Philippe
Rafael	Rebekah	Rene	Rose	Richard	Rina
Sandy	Sebastien	Sally	Sam	Shary	Sean
Tony	Tanya	Teddy	Teresa	Tobias	Tammy
Valerie	Van	Vicky	Victor	Virginie	Vince
William	Wendy	Wilfred	Wanda	Walter	Whitney

Strategy 1: Linear 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.

Strategy 2: Binary Search

- If the data is sorted, there is an even better searching strategy – one you probably already know!
- Have you ever played the number guessing game, where I pick a number between 1 and 100 and you try to guess it? Each time you guess, I'll tell you whether your guess is correct, too high, or too low. What strategy do you use? How many maximum number of guesses are required?
- Each time we guess the middle of the remaining numbers to try to narrow down the range.
- This strategy is called ***binary search***, because at each step we are dividing the remaining group of numbers into two parts.

Strategy 2: Binary Search

- We can use the same approach in our binary search algorithm! We can use two variables to keep track of the endpoints of the range in the sorted list.
- Since the target could be anywhere in the list, initially `low` is set to the first location in the list, and `high` is set to the last.
- The heart of the algorithm is a loop that looks at the middle element of the range, comparing it to the value `x`.
- If `x` is smaller than the middle item, `high` is moved so that the search is confined to the lower half.
- If `x` is larger than the middle item, `low` is moved to narrow the search to the upper half.

Strategy 2: Binary Search

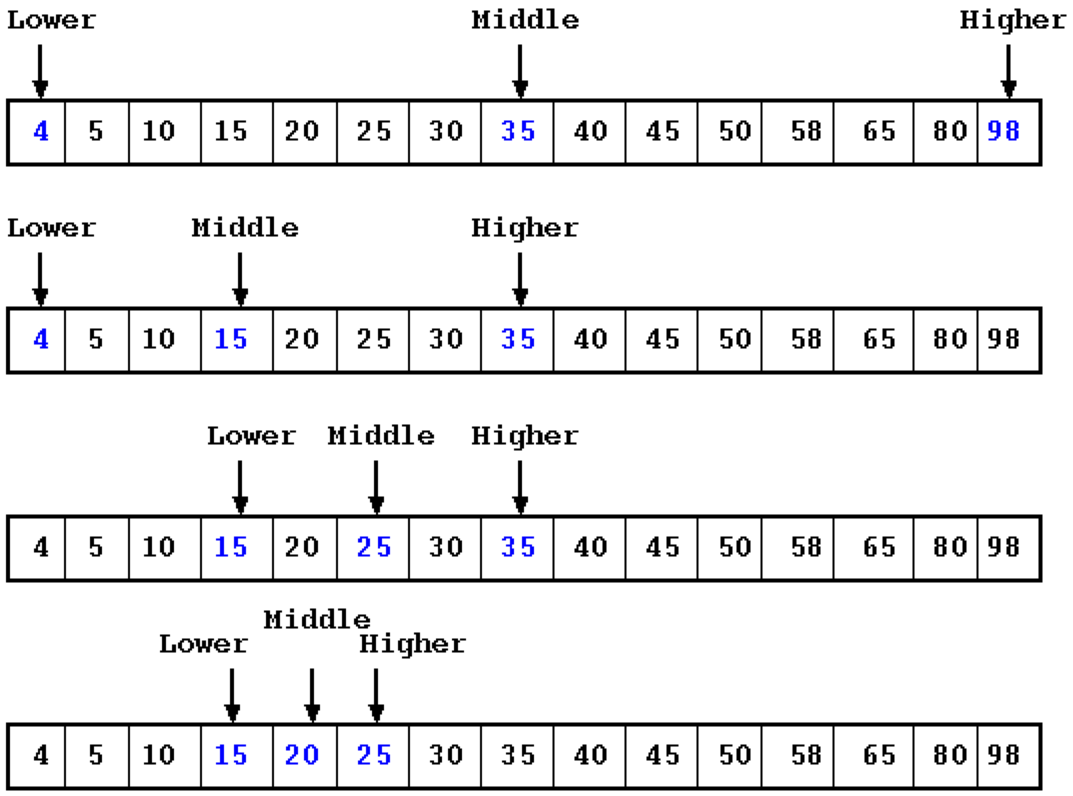
- We can use the same approach in our binary search algorithm! We can use two variables to keep track of the endpoints of the range in the sorted list.
- Since the target could be anywhere in the list, initially `low` is set to the first location in the list, and `high` is set to the last.
- The heart of the algorithm is a loop that looks at the middle element of the range, comparing it to the value `x`.
- If `x` is smaller than the middle item, `high` is moved so that the search is confined to the lower half.
- If `x` is larger than the middle item, `low` is moved to narrow the search to the upper half.

Strategy 2: Binary Search

- The loop terminates when either
  - *x is found*
  - *There are no more places to look* (`low > high`)

Example: Search for 20 in this list of numbers.

What happens if you search for 7?



Strategy 2: Binary Search

```
def search(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:          # There is still a range to search
        mid = (low + high)//2 # Position of middle item
        item = nums[mid]
        if x == item:           # Found it! Return the index
            return mid
        if x < item:            # x is in lower half of range
            high = mid - 1      #  move top marker down
        else:                   # x is in upper half of range
            low = mid + 1       #  move bottom marker up
    return -1                   # No range left to search,
                                # x is not there
```

Linear vs Binary Search

- 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?
  - *Experiment*

Linear vs Binary Search

- Test program searches 100 times for a integer in a list of 1,000,000 integers
  - *Using linear and binary search*
  - *When randomly chosen integer present, and when random integer not present in list*

```
Search when item is in list
Linear search
User: 10.35 Sys: 0.01
Binary search
User: 0.00 Sys: 0.00
```

```
Search for item not in list
Linear search
User: 15.52 Sys: 0.02
Binary search
User: 0.00 Sys: 0.00
```

Comparing Algorithms

- Empirical results are 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.
- How do we count the number of "steps"?
- Computer scientists attack these problems by analyzing the number of steps (very approximately) that an algorithm will take relative to the size or difficulty of the specific problem instance being solved.

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?

Comparing Algorithms

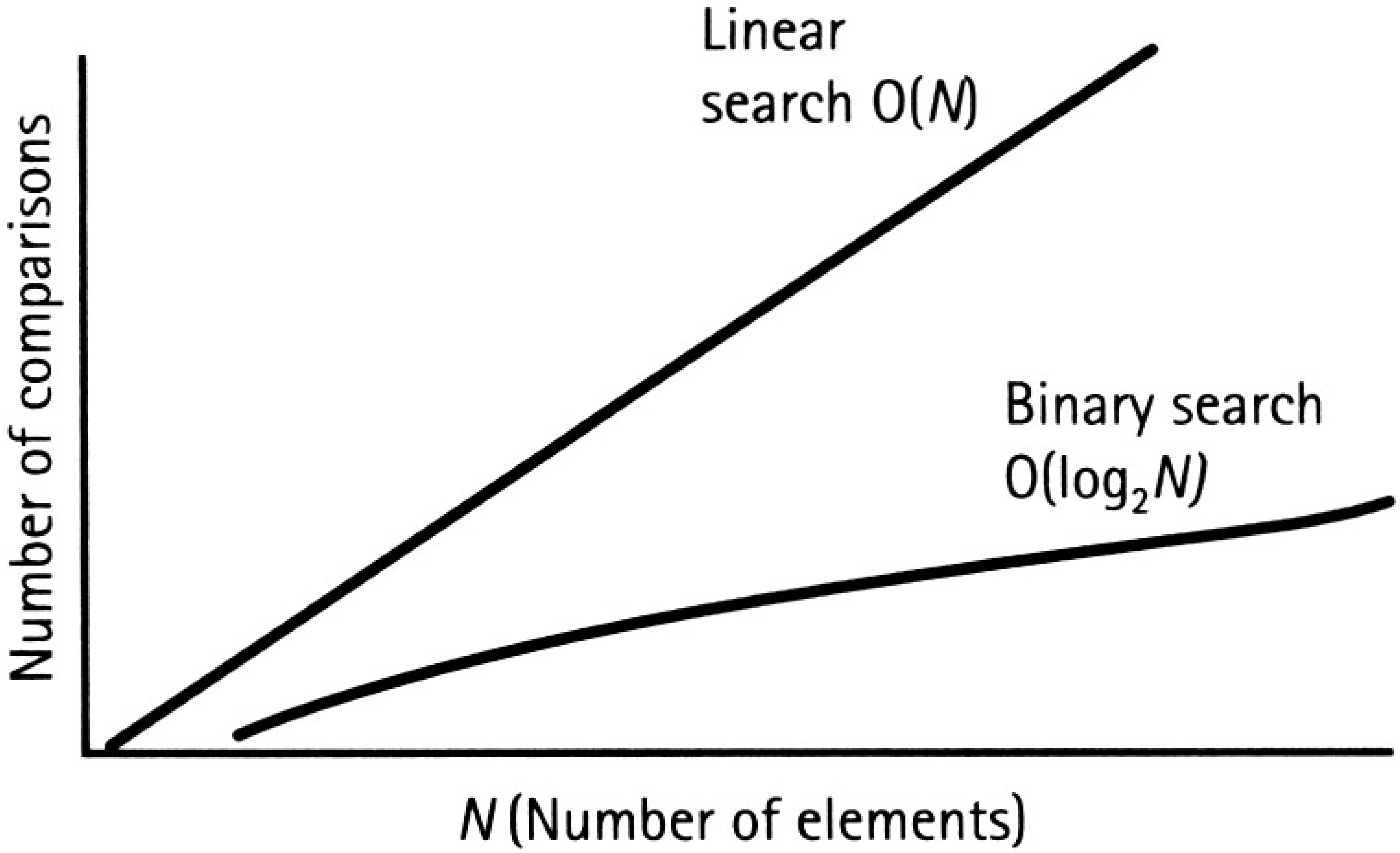
- Let's consider linear search.
  - *For a list of 10 items, the most work we might have to do is to look at each item in turn – looping at most 10 times.*
  - *For a list twice as large, we would loop at most 20 times.*
  - *For a list three times as large, we would loop at most 30 times!*
- The amount of time required is linearly related to the size of the list, *n*. This is what computer scientists call a *linear time* algorithm.
  - *Notation used is O(N) – order notation*

Comparing Algorithms

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

- If a binary search loops *i* times, it can find a single value in a list of size 2^i.
  - *Put another way, if the list has size N, i =* $log_2(N)$ *loops will be required. O(log(N)) time*
  - *Approx 20 loops for 1,000,000 item list*

Comparing Algorithms



https://www.techtud.com/computer-science-and-information-technology/algorithms/searching/binary-search

Comparing Algorithms

- Earlier, I 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!*
    - You will learn higher level programming courses that sorting takes $O(N \times \log(N))$ so not worth the trouble

Binary Search overview

- The basic idea between the binary search algorithm was to successfully divide the problem in half.
- This technique is known as a ***divide and conquer*** approach.
- Divide and conquer divides the original problem into sub-problems that are smaller versions of the original problem.
- In the binary search, the initial range is the entire list. We look at the middle element… if it is the target, we're done. Otherwise, we continue by performing a binary search on either the top half or bottom half of the list.

Summary

- Understood and coded Linear search
- Understood and coded Binary search
- Comparing efficiency of algorithms