

Extracted from:

Practical Programming, Third Edition
An Introduction to Computer Science Using Python 3.6

This PDF file contains pages extracted from *Practical Programming, Third Edition*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2017 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf


Raleigh, North Carolina

The
Pragmatic
Programmers

Practical Programming

Third Edition

An Introduction to
Computer Science
Using Python 3.6



Paul Gries
Jennifer Campbell
Jason Montojo
edited by Tammy Coron

Practical Programming, Third Edition
An Introduction to Computer Science Using Python 3.6

Paul Gries
Jennifer Campbell
Jason Montojo

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at <https://pragprog.com>.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Tammy Coron
Indexing: Potomac Indexing
Copy Editor: Liz Welch
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2017 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-13: 978-1-6805026-8-8

Encoded using the finest acid-free high-entropy binary digits.

Book version: P1.0—December 2017

A huge part of computer science involves studying how to organize, store, and retrieve data. There are many ways to organize and process data, and you need to develop an understanding of how to analyze how good an approach is. This chapter introduces you to some tools and concepts that you can use to tell whether a particular approach is faster or slower than another.

As you know, there are many solutions to each programming problem. If a problem involves a large amount of data, a slow algorithm will mean the problem can't be solved in a reasonable amount of time, even with an incredibly powerful computer. This chapter includes several examples of both slower and faster algorithms. Try running them yourself; experiencing just how slow (or fast) something is has a much more profound effect on your understanding than the data we include in this chapter.

Searching and sorting data are fundamental parts of programming. In this chapter, we will develop several algorithms for searching and sorting lists, and then we will use them to explore what it means for one algorithm to be faster than another. As a bonus, this approach will give you another set of examples of how there are many solutions to any problem, and that the approach you take to solving a problem will dictate which solution you come up with.

Searching a List

As you have already seen in [Table 11, List Methods, on page ?](#), Python lists have a method called `index` that searches for a particular item:

```
index(...)
    L.index(value, [start, [stop]]) -> integer -- return first index of value
```

List method `index` starts at the front of the list and examines each item in turn. For reasons that will soon become clear, this technique is called *linear search*. Linear search is used to find an item in an *unsorted* list. If there are duplicate values, our algorithms will find the leftmost one:

```
>>> ['d', 'a', 'b', 'a'].index('a')
1
```

We're going to write several versions of linear search in order to demonstrate how to compare different algorithms that all solve the same problem.

After we do this analysis, we will see that we can search a *sorted* list much faster than we can search an *unsorted* list.

An Overview of Linear Search

Linear search starts at index 0 and looks at each item one by one. At each index, we ask this question: Is the value we are looking for at the current index? We'll show three variations of this. All of them use a loop of some kind, and they are all implementations of this function:

```
from typing import Any
def linear_search(lst: list, value: Any) -> int:
    """Return the index of the first occurrence of value in lst, or return
    -1 if value is not in lst.

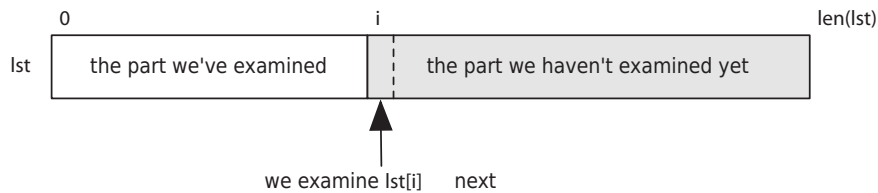
    >>> linear_search([2, 5, 1, -3], 5)
    1
    >>> linear_search([2, 4, 2], 2)
    0
    >>> linear_search([2, 5, 1, -3], 4)
    -1
    >>> linear_search([], 5)
    -1
    """

    # examine the items at each index i in lst, starting at index 0:
    #   is lst[i] the value we are looking for? if so, stop searching.
```

The algorithm in the function body describes what every variation will do to look for the value.

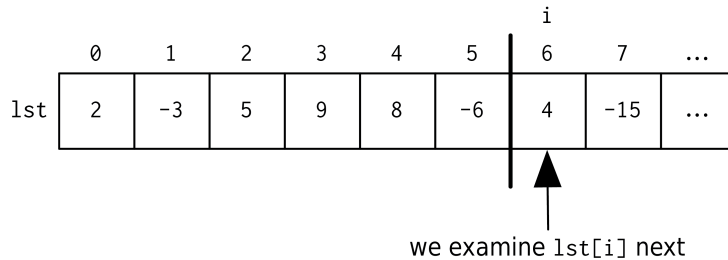
We've found it to be helpful to have a picture of how linear search works. (We will use pictures throughout this chapter for both searching and sorting.)

Because our versions examine index 0 first, then index 1, then 2, and so on, that means that partway through our searching process we have this situation:



There is a part of the list that we've examined and another part that remains to be examined. We use variable `i` to mark the current index.

Here's a concrete example of where we are searching for a value in a list that starts like this: `[2, -3, 5, 9, 8, -6, 4, 15, ...]`. We don't know how long the list is, but let's say that after six iterations we have examined items at indices 0, 1, 2, 3, 4, and 5. Index 6 is the index of the next item to examine:



That vertical line divides the list in two: the part we have examined and the part we haven't. Because we stop when we find the value, we know that the value isn't in the first part:



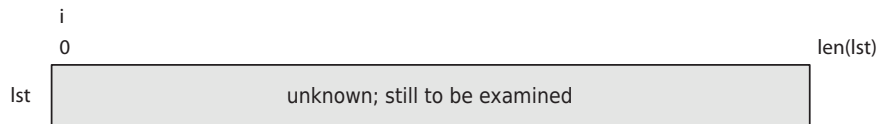
This picture is sometimes called an *invariant* of linear search. An invariant is something that remains unchanged throughout a process. But variable `i` is changing—how can that picture be an invariant?

Here is a text version of the picture:

`lst[0:i]` doesn't contain value, and `0 <= i <= len(lst)`

This word version says that we know that value wasn't found before index i and that i is somewhere between 0 and the length of the list. If our code matches that word version, that word version is an invariant of the code, and so is the picture version.

We can use invariants to come up with the initial values of our variables. For example, with linear search, at the very beginning the entire list is unknown—we haven't examined anything:



Variable i refers to 0 at the beginning, because then the section with the label value not here is empty; further, `lst[0:0]` is an empty list, which is exactly what we want according to the word version of the invariant. So the initial value of i should be 0 in all of our versions of linear search.

The while Loop Version of Linear Search

Let's develop our first version of linear search. We need to refine our comments to get them closer to Python:

```
Examine every index  $i$  in lst, starting at index 0:
    Is lst[i] the value we are looking for? if so, stop searching
```

Here's a refinement:

```
 $i = 0$  # The index of the next item in lst to examine
While the unknown section isn't empty, and lst[i] isn't
the value we are looking for:
    add 1 to  $i$ 
```

That's easier to translate. The unknown section is empty when $i == \text{len}(\text{lst})$, so it isn't empty as long as $i \neq \text{len}(\text{lst})$. Here is the code:

```
from typing import Any
def linear_search(lst: list, value: Any) -> int:
    """Return the index of the first occurrence of value in lst, or return
    -1 if value is not in lst.

    >>> linear_search([2, 5, 1, -3], 5)
    1
    >>> linear_search([2, 4, 2], 2)
    0
    >>> linear_search([2, 5, 1, -3], 4)
```



```

-1
>>> linear_search([], 5)
-1
"""
i = 0 # The index of the next item in lst to examine.
# Keep going until we reach the end of lst or until we find value.
while i != len(lst) and lst[i] != value:
    i = i + 1
# If we fell off the end of the list, we didn't find value.
if i == len(lst):
    return -1
else:
    return i

```

This version uses variable `i` as the current index and marches through the values in `lst`, stopping in one of two situations: when we have run out of values to examine or when we find the value we are looking for.

The first check in the loop condition, `i != len(lst)`, makes sure that we still have values to look at; if we were to omit that check, then if `value` isn't in `lst`, we would end up trying to access `lst[len(lst)]`. This would result in an `IndexError`.

The second check, `lst[i] != value`, causes the loop to exit when we find `value`. The loop body increments `i`; we enter the loop when we haven't reached the end of `lst`, and when `lst[i]` isn't the value we are looking for.

After the loop terminates, if `i == len(lst)` then `value` wasn't in `lst`, so we return `-1`. Otherwise, the loop terminated because we found `value` at index `i`.

The for Loop Version of Linear Search

The first version evaluates two Boolean subexpressions each time through the loop. But the first check, `i != len(lst)`, is almost unnecessary; it evaluates to `True` almost every time through the loop, so the only effect it has is to make sure we don't attempt to index past the end of the list. We can instead exit the function as soon as we find the value:

```

i = 0 # The index of the next item in lst to examine
For each index i in lst:
    If lst[i] is the value we are looking for:
        return i

```

If we get here, `value` was not in `lst`, so we return `-1`

In this version, we use Python's for loop to examine each index.

```
from typing import Any
def linear_search(lst: list, value: Any) -> int:
    """... Exactly the same docstring goes here ...
    """
    for i in range(len(lst)):
        if lst[i] == value:
            return i
    return -1
```

With this version, we no longer need the first check because the for loop controls the number of iterations. This for loop version is significantly faster than our first version; we'll see in a bit how much faster.

Sentinel Search

The last linear search we will study is called *sentinel search*. (A sentinel is a guard whose job it is to stand watch.) Remember that one problem with the while loop linear search is that we check `i != len(lst)` every time through the loop even though it can never evaluate to False except when value is not in lst. So we'll play a trick: we'll add value to the end of lst before we search. That way we are guaranteed to find it! We also need to remove it before the function exits so that the list looks unchanged to whoever called this function:

```
Set up the sentinel: append value to the end of lst
i = 0 # The index of the next item in lst to examine
While lst[i] isn't the value we are looking for:
    Add 1 to i
Remove the sentinel
return i
```

Let's translate that to Python:

```
from typing import Any
def linear_search(lst: list, value: Any) -> int:
    """... Exactly the same docstring goes here ...
    """
    # Add the sentinel.
    lst.append(value)
    i = 0
    # Keep going until we find value.
    while lst[i] != value:
        i = i + 1
```

```

# Remove the sentinel.
lst.pop()

# If we reached the end of the list we didn't find value.
if i == len(lst):
    return -1
else:
    return i

```

All three of our linear search functions are correct. Which one you prefer is largely a matter of taste: some programmers dislike returning in the middle of a loop, so they won't like the second version. Others dislike modifying parameters in any way, so they won't like the third version. Still others will dislike that extra check that happens in the first version.

Timing the Searches

Here is a program that we used to time the three searches on a list with about ten million values:

```

import time
import linear_search_1
import linear_search_2
import linear_search_3

from typing import Callable, Any

def time_it(search: Callable[[list, Any], Any], L: list, v: Any) -> float:
    """Time how long it takes to run function search to find
    value v in list L.
    """

    t1 = time.perf_counter()
    search(L, v)
    t2 = time.perf_counter()
    return (t2 - t1) * 1000.0

def print_times(v: Any, L: list) -> None:
    """Print the number of milliseconds it takes for linear_search(v, L)
    to run for list.index, the while loop linear search, the for loop
    linear search, and sentinel search.
    """

    # Get list.index's running time.
    t1 = time.perf_counter()
    L.index(v)
    t2 = time.perf_counter()
    index_time = (t2 - t1) * 1000.0

    # Get the other three running times.
    while_time = time_it(linear_search_1.linear_search, L, v)
    for_time = time_it(linear_search_2.linear_search, L, v)
    sentinel_time = time_it(linear_search_3.linear_search, L, v)

```

```

print("{0}\t{1:.2f}\t{2:.2f}\t{3:.2f}\t{4:.2f}".format(
    v, while_time, for_time, sentinel_time, index_time))
L = list(range(10000001)) # A list with just over ten million values
print_times(10, L) # How fast is it to search near the beginning?
print_times(5000000, L) # How fast is it to search near the middle?
print_times(10000000, L) # How fast is it to search near the end?

```

This program makes use of function `perf_counter` in built-in module `time`. Function `time_it` will call whichever search function it's given on `v` and `L` and returns how long that search took. Function `print_times` calls `time_it` with the various linear search functions we have been exploring and prints those search times.

Linear Search Running Time

The running times of the three linear searches with that of Python's `list.index` are compared in Table 18. This comparison used a list of 10,000,001 items and three test cases: an item near the front, an item roughly in the middle, and the last item. Except for the first case, where the speeds differ by very little, our `while` loop linear search takes about thirteen times as long as the one built into Python, and the `for` loop search and `sentinel` search take about five and seven times as long, respectively.

Case	<code>while</code>	<code>for</code>	<code>sentinel</code>	<code>list.index</code>
First	0.01	0.01	0.01	0.01
Middle	1261	515	697	106
Last	2673	1029	1394	212

Table 18—Running Times for Linear Search (in milliseconds)

What is more interesting is the way the *running times* of these functions increase with the number of items they have to examine. Roughly speaking, when they have to look through twice as much data, every one of them takes twice as long. This is reasonable because indexing a list, adding 1 to an integer, and evaluating the loop control expression require the computer to do a fixed amount of work. Doubling the number of times the loop has to be executed therefore doubles the total number of operations, which in turn should double the total running time. This is why this kind of search is called *linear*: the time to do it grows linearly with the amount of data being processed.

Binary Search

Consider a list of 1 million *sorted* values. Linear search starts at the beginning of the list and asks, “Is this value what I’m looking for?” If it isn’t, the same is

asked about the second value, and then the third. Up to 1 million questions are asked. This algorithm doesn't take advantage of the list being sorted.

Here's a new algorithm, called *binary search*, that relies on the list being sorted: look at the middle value and ask, "Is this value bigger than or smaller than the one I'm looking for?" With that one question, we can eliminate 500,000 values! That leaves a list of 500,000 values to search. We'll do it again: look at the middle value, ask the same question, and eliminate another 250,000 values. *We have eliminated 3/4 of the list with only two questions!* Asking only 20 questions, we can locate a particular value in a list of 1 million sorted values.

Logarithms

The *logarithm* of a number is how many times that number can be divided until we get to 1. We'll need to know what number we are dividing by—we'll call that the *base*. For binary search, we use base 2, because we divide the list in half each iteration.

The logarithm base 2 of 1, which we'll write as $\log_2 1$, is 0: we don't need to divide 1 at all in order to reach 1.

$\log_2 2$ is 1, because $\frac{2}{2}$ is 1.

$\log_2 4$ is 2: $\frac{4}{2}$ is 2, and $\frac{2}{2}$ is 1, so we divided by 2 twice to reach 1.

$\log_2 8$ is 3: $\frac{8}{2}$ is 4, $\frac{4}{2}$ is 2, and $\frac{2}{2}$ is 1. Every time we double the number, the logarithm base 2 increases by 1.

Here's a table of base 2 logarithms:

N (the # of items)	N as a power of 2	$\log_2 N$
1	2^0	0
2	2^1	1
4	2^2	2
8	2^3	3
16	2^4	4
32	2^5	5
64	2^6	6
128	2^7	7
256	2^8	8
512	2^9	9
1024	2^{10}	10

Table 19—Logarithmic Growth

To figure out how fast it is, we'll think about how big a list we can search with a certain number of questions. With only one question, we can determine whether a list of length 1 contains a value. With two questions, we can search a list of length 2. With three questions, we can search a list of length 4. Four questions, length 8. Five questions, length 16. Every time we get to ask another question, we can search a list twice as big.

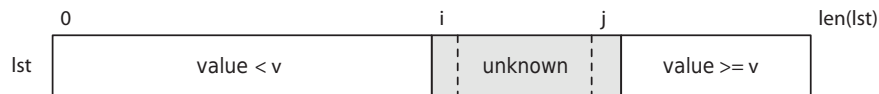
Using logarithmic notation, N sorted values can be searched in $\text{ceiling}(\log_2 N)$ steps, where $\text{ceiling}()$ is the ceiling function that rounds a value up to the nearest integer. As shown in Table 20, this increases much less quickly than the time needed for linear search.

Searching N Items	Worst Case—Linear Search	Worst Case—Binary Search
100	100	7
1000	1000	10
10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20
10,000,000	10,000,000	24

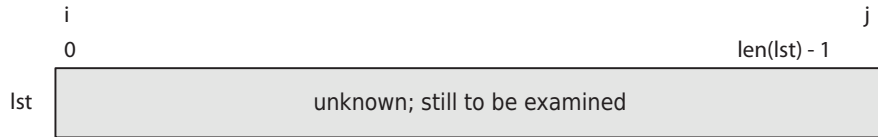
Table 20—Logarithmic Growth

The key to binary search is to keep track of three parts of the list: the left part, which contains values that are smaller than the value we are searching for; the right part, which contains values that are equal to or larger than the value we are searching for; and the middle part, which contains values that we haven't yet examined—the unknown section. If there are duplicate values, we will return the index of the leftmost one, which is why the “equal to” section belongs on the right.

We'll use two variables to keep track of the boundaries: i will mark the index of the first unknown value, and j will mark the index of the last unknown value:



At the beginning of the algorithm, the unknown section makes up the entire list, so we will set i to 0 and j to the length of the list minus one as shown in the [figure on page 15](#).



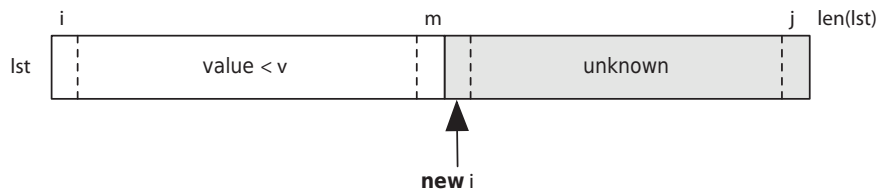
We are done when that unknown section is empty—when we’ve examined every item in the list. This happens when $i == j + 1$ —when the values *cross*. (When $i == j$, there is still one item left in the unknown section.) Here is a picture of what the values are when the unknown section is empty:



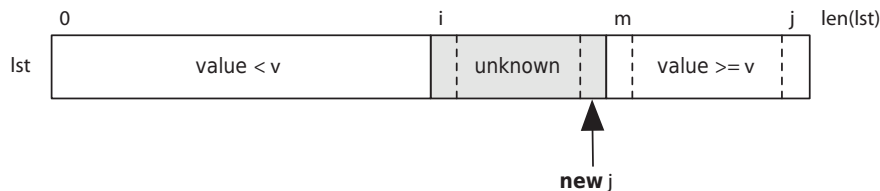
To make progress, we will set either i or j to near the middle of the range between them. Let’s call this index m , which is at $(i + j) // 2$. (Notice the use of integer division: we are calculating an index, so we need an integer.)

Think for a moment about the value at m . If it is less than v , we need to move i up, while if it is greater than v , we should move j down. But where exactly do we move them?

When we move i up, we don’t want to set it to the midpoint exactly, because $lst[m]$ isn’t included in the range; instead, we set it to one past the middle—in other words, to $m + 1$.



Similarly, when we move j down, we move it to $m - 1$:



The completed function is as follows:

```

from typing import Any

def binary_search(L: list, v: Any) -> int:
    """Return the index of the first occurrence of value in L, or return
    -1 if value is not in L.

    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 1)
    0
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 4)
    2
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 5)
    4
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 10)
    7
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], -3)
    -1
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 11)
    -1
    >>> binary_search([1, 3, 4, 4, 5, 7, 9, 10], 2)
    -1
    >>> binary_search([], -3)
    -1
    >>> binary_search([1], 1)
    0
    """

    # Mark the left and right indices of the unknown section.
    i = 0
    j = len(L) - 1

    while i != j + 1:
        m = (i + j) // 2
        if L[m] < v:
            i = m + 1
        else:
            j = m - 1

    if 0 <= i < len(L) and L[i] == v:
        return i
    else:
        return -1

if __name__ == '__main__':
    import doctest
    doctest.testmod()

```

There are a lot of tests because the algorithm is quite complicated and we wanted to test pretty thoroughly. Our tests cover these cases:

- The value is the first item.
- The value occurs twice. We want the index of the first one.
- The value is in the middle of the list.
- The value is the last item.
- The value is smaller than everything in the list.
- The value is larger than everything in the list.
- The value isn't in the list, but it is larger than some and smaller than others.
- The list has no items.
- The list has one item.

In [Chapter 15, *Testing and Debugging*, on page ?](#), you'll learn a different testing framework that allows you to write tests in a separate Python file (thus making docstrings shorter and easier to read; only a couple of examples are necessary), and you'll learn strategies for coming up with your own test cases.

Binary Search Running Time

Binary search is *much* more complicated to write and understand than linear search. Is it fast enough to make the extra effort worthwhile? To find out, we can compare it to `list.index`. As before, we search for the first, middle, and last items in a list with about ten million elements as shown in Table 21.

Case	<code>list.index</code>	<code>binary_search</code>	Ratio
First	0.007	0.02	0.32
Middle	105	0.02	5910
Last	211	0.02 (Wow!)	11661

Table 21—Running Times for Binary Search

The results are impressive. Binary search is up to *several thousand times faster* than its linear counterpart when searching ten million items. Most importantly, if we double the number of items, binary search takes only one more iteration, whereas the time for `list.index` nearly doubles.

Note also that although the time taken for linear search grows in step with the index of the item found, there is no such pattern for binary search. No matter where the item is, it takes the same number of steps.

Built-In Binary Search

The Python standard library's `bisect` module includes binary search functions that are slightly faster than our binary search. Function `bisect_left` returns the index where an item should be inserted in a list to keep it in sorted order, assuming it is sorted to begin with. `insort_left` actually does the insertion.

The word *left* in the name signals that these functions find the leftmost (lowest index) position where they can do their jobs; the complementary functions `bisect_right` and `insort_right` find the rightmost.

There is one minor drawback to binary search: the algorithm assumes that the list is sorted, and sorting is time and memory intensive. We'll look at that next.