6.3. The Sequential Search

When data items are stored in a collection such as a list, we say that they have a linear or sequential relationship. Each data item is stored in a position relative to the others. In Python lists, these relative positions are the index values of the individual items. Since these index values are ordered, it is possible for us to visit them in sequence. This process gives rise to our first searching technique, the **sequential search**.

Figure 1 shows how this search works. Starting at the first item in the list, we simply move from item to item, following the underlying sequential ordering until we either find what we are looking for or run out of items. If we run out of items, we have discovered that the item we were searching for was not present.

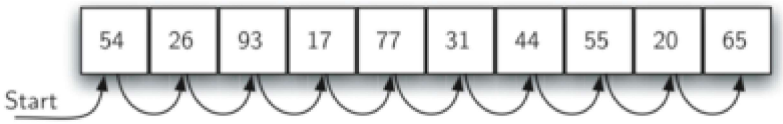

Figure 1: Sequential Search of a List of Integers

The Python implementation for this algorithm is shown in CodeLens 1. The function needs the list and the item we are looking for and returns a boolean value as to whether it is present. The boolean variable `found` is initialized to `False` and is assigned the value `True` if we discover the item in the list.

(searching.html) (TheBinarySearch

```
1  def sequentialSearch(alist, item):
2      pos = 0
3      found = False
4
5      while pos < len(alist) and not fo
6          if alist[pos] == item:
7              found = True
8          else:
9              pos = pos+1
10
11     return found
12
13 testlist = [1, 2, 32, 8, 17, 19, 42,
14 print(sequentialSearch(testlist, 3))
15 print(sequentialSearch(testlist, 13))
```

line that just executed
next line to execute

< Prev Next >

Step 1 of 67

Python Tutor (http://pythontutor.com/) by Philip Guo (http://pgbovine.net/)

Customize visualization (NEW!)

Print output (drag lower right corner to resize)

Frames Objects

Activity: CodeLens Sequential Search of an Unordered List (search1)

6.3.1. Analysis of Sequential Search

To analyze searching algorithms, we need to decide on a basic unit of computation. Recall that this is typically the common step that must be repeated in order to solve the problem. For searching, it makes sense to count the number of comparisons performed. Each comparison may or may not discover the item we are looking for. In addition, we make another assumption here. The list of items is not ordered in any way. The items have been placed randomly into the list. In other words, the probability that the item we are looking for is in any particular position is exactly the same for each position of the list.

If the item is not in the list, the only way to know it is to compare it against every item present. If there are n items, then the sequential search requires n comparisons to discover that the item is not there. In the case where the item is in the list, the analysis is not so straightforward. There are actually three different scenarios that can occur. In the best case we will find the item in the first place we look, at the beginning of the list. We will need only one comparison. In the worst case, we will not discover the item until the very last comparison, the *nth* comparison.

What about the average case? On average, we will find the item about halfway into the list; that is, we will compare against $\frac{n}{2}$ items. Recall, however, that as *n* gets large, the coefficients, no matter what they are, become insignificant in our approximation, so the complexity of the sequential search, is $O(n)$. Table 1 summarizes these results.

Table 1: Comparisons Used in a Sequential Search of an Unordered List

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

We assumed earlier that the items in our collection had been randomly placed so that there is no relative order between the items. What would happen to the sequential search if the items were ordered in some way? Would we be able to gain any efficiency in our search technique?

Assume that the list of items was constructed so that the items were in ascending order, from low to high. If the item we are looking for is present in the list, the chance of it being in any one of the *n* positions is still the same as before. We will still have the same number of comparisons to find the item. However, if the item is not present there is a slight advantage. Figure 2 shows this process as the algorithm looks for the item 50. Notice that items are still compared in sequence until 54. At this point, however, we know something extra. Not only is 54 not the item we are looking for, but no other elements beyond 54 can work either since the list is sorted. In this case, the algorithm does not have to continue looking through all of the items to report that the item was not found. It can stop immediately. CodeLens 2 shows this variation of the sequential search function.

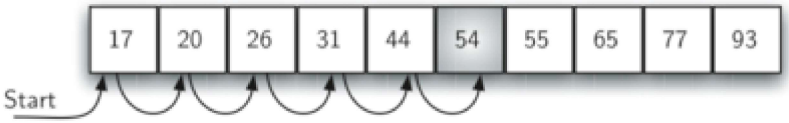

Figure 2: Sequential Search of an Ordered List of Integers

(searching.html) (TheBinarySearch

```
1  def orderedSequentialSearch(alist, it
2      pos = 0
3      found = False
4      stop = False
5      while pos < len(alist) and not fo
6          if alist[pos] == item:
7              found = True
8          else:
9              if alist[pos] > item:
10                 stop = True
11             else:
12                 pos = pos+1
13
14     return found
15
16 testlist = [0, 1, 2, 8, 13, 17, 19, 3
17 print(orderedSequentialSearch(testli
```

line that just executed

next line to execute

< Prev Next >

Step 1 of 53

Python Tutor (http://pythontutor.com/) by Philip Guo (http://pgbovine.net/)

Customize visualization (NEW!)

Print output (drag lower right corner to resize)

Frames Objects

Activity: CodeLens Sequential Search of an Ordered List (search2)

Table 2 summarizes these results. Note that in the best case we might discover that the item is not in the list by looking at only one item. On average, we will know after looking through only $\frac{n}{2}$ items. However, this technique is still $O(n)$. In summary, a sequential search is improved by ordering the list only in the case where we do not find the item.

(unordered.html) (TheBinarySearch

Table 2: Comparisons Used in Sequential Search of an Ordered List

item is present	1	n	$\frac{n}{2}$

item not present	1	n	$\frac{n}{2}$

Self Check

Q-3: Suppose you are doing a sequential search of the list [15, 18, 2, 19, 18, 0, 8, 14, 19, 14]. How many comparisons would you need to do in order to find the key 18?

- **A. 5**
- **B. 10**
- **C. 4**
- **D. 2**

Check Me Compare me

Activity: 6.3.1.2 Multiple Choice (question_SRCH_1)

Q-4: Suppose you are doing a sequential search of the ordered list [3, 5, 6, 8, 11, 12, 14, 15, 17, 18]. How many comparisons would you need to do in order to find the key 13?

- **A. 10**
- **B. 5**
- **C. 7**
- **D. 6**

Check Me Compare me

Activity: 6.3.1.3 Multiple Choice (question_SRCH_2)

user not logged in

(searching.html) (TheBinarySearch