The Binary Search

It is possible to take greater advantage of the ordered list if we are clever with our comparisons. In the sequential search, when we compare against the first item, there are at most $n-1$ more items to look through if the first item is not what we are looking for. Instead of searching the list in sequence, a **binary search** will start by examining the middle item. If that item is the one we are searching for, we are done. If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items. If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, must be in the upper half.

We can then repeat the process with the upper half. Start at the middle item and compare it against what we are looking for. Again, we either find it or split the list in half, therefore eliminating another large part of our possible search space. Figure 3 shows how this algorithm can quickly find the value 54. The complete function is shown in CodeLens 3.

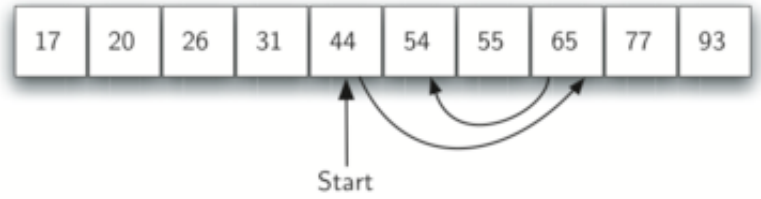

Figure 3: Binary Search of an Ordered List of Integers

quentialSearch.html)

Python 2.7

```
1  def binarySearch(alist, item):
2      first = 0
3      last = len(alist)-1
4      found = False
5
6      while first<=last and not found:
7          midpoint = (first + last)//2
8          if alist[midpoint] == item:
9              found = True
10         else:
11             if item < alist[midpoint]:
12                 last = midpoint-1
13             else:
14                 first = midpoint+1
15
16     return found
17
```

<< First | < Back | Step 1 of 42 | Forward > | Last >>

line that has just executed

next line to execute

Visualized using Online Python Tutor (http://pythontutor.com) by Philip Guo (http://www.pgbovine.net/)

Frames Objects

CodeLens: 1 Binary Search of an Ordered List (search3)

Before we move on to the analysis, we should note that this algorithm is a great example of a divide and conquer strategy. Divide and conquer means that we divide the problem into smaller pieces, solve the smaller pieces in some way, and then reassemble the whole problem to get the result. When we perform a binary search of a list, we first check the middle item. If the item we are searching for is less than the

middle item, we can simply perform a binary search of the left half of the original list. Likewise, if the item is greater, we can perform a binary search of the right half. Either way, this is a recursive call to the binary search function passing a smaller list. CodeLens 4 shows this recursive version.

Python 2.7

```
 1 def binarySearch(alist, item):
 2     if len(alist) == 0:
 3         return False
 4     else:
 5         midpoint = len(alist)//2
 6         if alist[midpoint]==item:
 7           return True
 8         else:
 9           if item<alist[midpoint]:
10             return binarySearch(alist[:midpo
11           else:
12             return binarySearch(alist[midpoi
13
14 testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
15 print(binarySearch(testlist, 3))
16 print(binarySearch(testlist, 13))
```

<< First | < Back | Step 1 of 35 | Forward > | Last >>

line that has just executed

next line to execute

Visualized using Online Python Tutor (http://pythontutor.com) by Philip Guo (http://www.pgbovine.net/)

Frames Objects

CodeLens: 2 A Binary Search--Recursive Version (search4)

Analysis of Binary Search

To analyze the binary search algorithm, we need to recall that each comparison eliminates about half of the remaining items from consideration. What is the maximum number of comparisons this algorithm will require to check the entire list? If we start with *n* items, about $\frac{n}{2}$ items will be left after the first comparison. After the second comparison, there will be about $\frac{n}{4}$. Then $\frac{n}{8}$, $\frac{n}{16}$, and so on. How many times can we split the list? Table 3 helps us to see the answer.

Table 3: Tabular Analysis for a Binary Search

Comparisons	Approximate Number of Items Left
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	
i	$\frac{n}{2^i}$

When we split the list enough times, we end up with a list that has just one item. Either that is the item we are looking for or it is not. Either way, we are done. The number of comparisons necessary to get to this point is *i* where $\frac{n}{2^i} = 1$. Solving for *i* gives us $i = \log n$. The maximum number of comparisons is logarithmic with respect to the number of items in the list. Therefore, the binary search is $O(\log n)$.

One additional analysis issue needs to be addressed. In the recursive solution shown above, the recursive call,

```
binarySearch(alist[:midpoint],item)
```

uses the slice operator to create the left half of the list that is then passed to the next invocation (similarly for the right half as well). The analysis that we did above assumed that the slice operator takes constant time. However, we know that the slice operator in Python is actually O(k). This means that the binary search using slice will not perform in strict logarithmic time. Luckily this can be remedied by passing the list along with the starting and ending indices. The indices can be calculated as we did in Listing 3. We leave this implementation as an exercise.

Even though a binary search is generally better than a sequential search, it is important to note that for small values of *n*, the additional cost of sorting is probably not worth it. In fact, we should always consider whether it is cost effective to take on the extra work of sorting to gain searching benefits. If we can sort once and then search many times, the cost of the sort is not so significant. However, for large lists, sorting even once can be so expensive that simply performing a sequential search from the start may be the best choice.

Self Check

Q-40: Suppose you have the following sorted list [3, 5, 6, 8, 11, 12, 14, 15, 17, 18] and are using the recursive binary search algorithm. Which group of numbers correctly shows the sequence of comparisons used to find the key 8.

- 11, 5, 6, 8
- 12, 6, 11, 8
- 3, 5, 6, 8
- 18, 12, 6, 8

Check Me Compare me

Q-41: Suppose you have the following sorted list [3, 5, 6, 8, 11, 12, 14, 15, 17, 18] and are using the recursive binary search algorithm. Which group of numbers correctly shows the sequence of comoparisons used to search for the key 16?

- 11, 14, 17
- 18, 17, 15
- 14, 17, 15
- 12, 17, 15

Check Me Compare me

❮ (TheSequentialSearch.html) ❯ (Hashing.html)

user not logged in