OzoBlockly Lesson: Evo Demonstrates Linear and Binary Searches

By Richard Born
Associate Professor Emeritus
Northern Illinois University
rborn2@niu.edu

Introduction

Searching tables for needed information is a common activity in everyday life. For example, as April 15 approaches each year, millions of American citizens search for their tax based upon taxable income and marital status. If you are in the market for a new TV, you might begin your search by going online to an electronics store and search for TV's based upon type, size, and price. You have a paper catalog that you have received from a toy seller and go to the alphabetical index to search for a specific toy of interest. With an upcoming mother's day, you go to the telephone book to search for a local florist. Perhaps you are playing a game with a friend to guess a number between 1 and 100. One of the newest search techniques is found in the airline industry, where passengers are allowed to board a plane by computer facial recognition. In these examples and countless others, table searching is used in one form or another.

The exact technique used in a search depends upon a variety of things including how the data has been stored, whether or not it is indexed, and if it has been sorted. Two of the most commonly used search algorithms are linear search and binary search. Their importance is great enough that they are often included in introductory courses in computer science.

A ***linear search*** is a very basic and simple search technique. In a linear search, the items in a list are searched sequentially beginning with the first item and continuing until the list item matches the search key or until the item is not found. The list need not be initially sorted. Linear search works better if the list is not lengthy. This technique would not be particularly advisable when trying to guess a number between 1 and 100 that a friend has in mind.

A ***binary search*** is somewhat more complicated and does require a sorted list. The search key is first compared to the value in the middle of the list. If there is a match, then the desired data is retrieved. If the search key is less than middle element in the list, then it must be in the lower half of the list or not be in the list at all. If the search key is greater than the middle element, then it must be in the upper half of the list or not be in the list at all. This process continues with the lower or upper half of the list until the search key is found or it is determined that the search key is not in the list at all. This technique would be quite efficient when trying to guess a number between 1 and 100 that a friend has in mind.

In this lesson, provided OzoBlockly programs that simulate linear and binary searches are loaded into Evo. Students then use provided OzoMaps on which Evo searches for values interactively inputted by the students. Evo moves his way through the list until a match is found or until it is determined that the search value is not in the list. Evo's motion clearly demonstrates the difference between the two search techniques.

Evo's Search Problem

A common problem when involved with inventory control in a warehouse is to search for the ***quantity on hand*** (QOH) given a specific ***item number***. For example, such a search may be necessary when a customer calls requesting to order 12 of item number "such and such". If the item number does not exist, the customer can be told to double check the item number being requested. If the item number does exist and there are at least 12 on hand, the order can be placed. If there are only 9 on hand, the customer can advise how to proceed with the order. If the QOH is zero, the customer can decide whether or not to place a backorder.

Figure 1 shows reduced size copies of the linear search and binary search OzoMaps for this inventory search. Larger copies, suitable for duplicating for student use with Evo can be found on the last two pages of this document. The upper left corner of each OzoMap shows the start location for Evo, where the student enters the search key, *i.e.*, the item number for which the QOH is desired. After entering the search key, Evo will speak the search key and then start moving to search for the key and corresponding QOH. There are 28 items in the inventory, each with a unique item number between 1 and 99. For the linear search, you can see that the items are not ordered. However, they are ordered from smallest to largest item number in the binary search. The indexes range from 0 to 27, as the 28 item numbers and their corresponding QOH are stored in arrays in OzoBlockly.

While moving on the OzoMap, Evo will beep each time that he compares the search key to the item number at the location where he has stopped. This gives the student an audio indication of the efficiency of the algorithm, as more comparisons mean less efficiency. If the key is found, Evo will speak the QOH for the item number, show a green top LED, and then speak the total number of comparisons (beeps). If Evo doesn't find the search key, he shows a red top LED and speaks only the total number of comparisons.

Some Discussion Points

Keep in mind that with the binary search it takes Evo some time to move from one comparison location to the next. This "move time" for Evo is not required by a computer executing the binary search, as an array is a direct access memory—it takes about the same amount of time to get to one location as another, regardless of array size. It is the number of comparisons required for a search that eats up computer cycles. Therefore, the ***time*** required by ***Evo*** to find a match is ***not*** related to the efficiency of the two search techniques. Only the number of comparisons (beeps) is a measure of the efficiency.

It should also be noted that sort algorithms are slower than a linear search. So it is not particularly advantageous to sort data if you are just going to do a single binary search at some point in the future. If you plan on performing many binary searches, it could be worthwhile to sort the data first. You also need to ask if the quicker binary search is worth the associated cost of keeping the data sorted. This could become an issue if there are numerous insert and delete operations but only an occasional binary search.

With a linear search, the time require to search a list grows at the same rate as the list itself. Computer scientists say that such a sort has a complexity of order n, denoted O(n), where n is the is the size of the list. With a binary search the number of comparison grows more slowly than the list size, $O(\log_2 n)$.

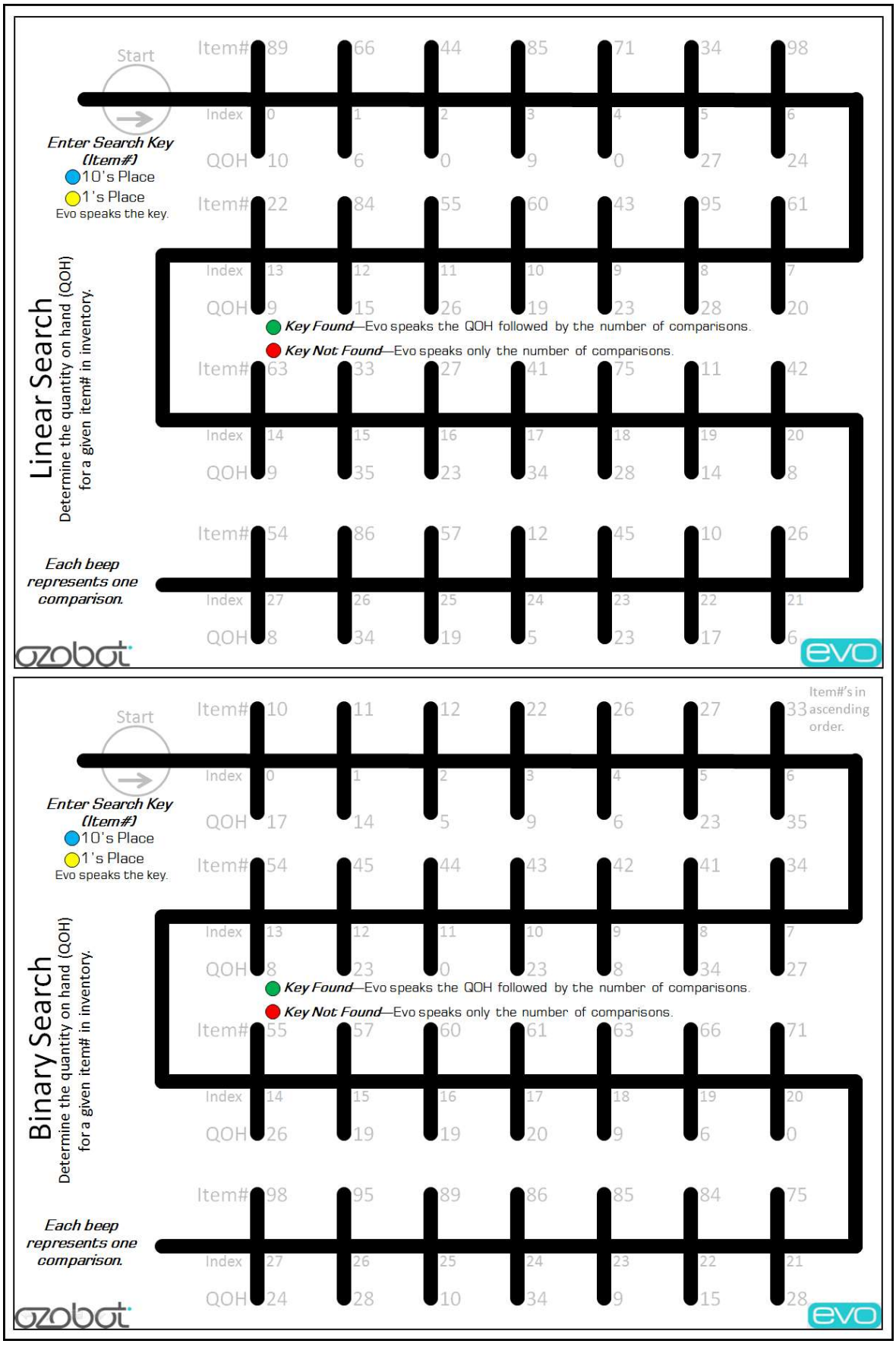

Start
Enter Search Key (Item#)
10's Place
1's Place
Evo speaks the key.
Linear Search
Determine the quantity on hand (QOH) for a given item# in inventory.
Key Found—Evo speaks the QOH followed by the number of comparisons.
Key Not Found—Evo speaks only the number of comparisons.
Each beep represents one comparison.
Item# 89 66 44 85 71 34 98
Index 0 1 2 3 4 5 6
QOH 10 6 0 9 0 27 24
Item# 22 84 55 60 43 95 61
Index 13 12 11 10 9 8 7
QOH 9 15 26 19 23 28 20
Item# 63 33 27 41 75 11 42
Index 14 15 16 17 18 19 20
QOH 9 35 23 34 28 14 8
Item# 54 86 57 12 45 10 26
Index 27 26 25 24 23 22 21
QOH 8 34 19 5 23 17 6
ozobot
evo
Start
Enter Search Key (Item#)
10's Place
1's Place
Evo speaks the key.
Binary Search
Determine the quantity on hand (QOH) for a given item# in inventory.
Item#'s in ascending order.
Key Found—Evo speaks the QOH followed by the number of comparisons.
Key Not Found—Evo speaks only the number of comparisons.
Each beep represents one comparison.
Item# 10 11 12 22 26 27 33
Index 0 1 2 3 4 5 6
QOH 17 14 5 9 6 23 35
Item# 54 45 44 43 42 41 34
Index 13 12 11 10 9 8 7
QOH 8 23 0 23 8 34 27
Item# 55 57 60 61 63 66 71
Index 14 15 16 17 18 19 20
QOH 26 19 19 20 9 6 0
Item# 98 95 89 86 85 84 75
Index 27 26 25 24 23 22 21
QOH 24 28 10 34 9 15 28
ozobot
evo

Figure 1

Running the OzoBlockly Search Programs

1. Open either the provided *LinearSearch.ozocode* file or the *BinarySearch.ozocode* file while at the OzoBlockly web site. Then load the program onto Evo and use a printed copy of the appropriate Ozomap from the last two pages of this document.
2. To run the loaded program on Evo, begin with Evo powered down. Place Evo centered on the "Start" gray circle on the Ozomap and facing the direction shown by the gray arrow.
3. Press Evo's button once to turn Evo on. Then when the front lights show light blue, quickly double-press the button. The OzoBlockly program has started running.
4. The top light will show white for a couple of seconds and then turn blue. Let's suppose that you are searching for the QOH for item number 63. While the top light is blue, you have 8 seconds to press Evo's button a total of 6 times, representing the 6 (10's place) is 63. After each button press, Evo's top light will quickly flash white to let you know that it recognized the button press. Make sure that you see the flash before pushing the button again.
5. When the light turns yellow you have another 8 seconds to press the button 3 times, representing the 3 (1's place) in 63.
6. While Evo is still setting at the start location, he will then speak the number you entered—in this case, the number 63.
7. Evo will then begin moving, searching for the item number 63. Each time he ***compares*** the number 63 to the item number at the location where he has momentarily stopped, he will ***beep***.
8. When he finds the item number 63, he will speak the corresponding QOH (9), show a green top LED, and then speak the total number of comparisons required for the search.
9. If he does not find the search key, then he will show a red top LED and speak only the total number of comparisons required for the search.
10. Evo will then power down.
11. Try searching for the QOH for item number 51. What do you notice regarding differences in behavior for the linear and binary searches?

The OzoBlockly Linear Search Main Program

This section is optional and intended for teachers interested in discussing a few of the details of the OzoBlockly linear search program with their students. The entire program is quite lengthy with several subroutines that are fairly complex. But the main program, which contains the linear search, is well worth discussion and is shown in Figure 2.

One array contains the item numbers and another array contains the corresponding QOH values. Both of length 28, indexes vary from 0 through 27. Just before starting the linear search, the *Input Search Value* subroutine captures the item number from the student button presses to be searched by Evo. The variable *n* is given the value of the length of the two arrays. Then a *count with* loop varies the *index* into the array from 0 to the highest array index *n-1*. Inside the loop, Evo moves to the next position in sequence on the line and beeps upon reaching the next position. Evo then checks to see if the search value equals *ItemNumber[index]*. If it does, Evo speaks the corresponding *QuantityOnHand[index],* displays a green top light to indicate a

successful search, speaks the total number of comparisons required (*index+1*), and then turns off. If not, the count continues with the next iteration of the loop. In the event that the loop completes all *n* iterations without finding the search value, the blocks immediately following the loop are executed. Evo displays a red top LED to indicate an unsuccessful search, speaks the total number of iterations, beeps for a second, and then turns off.

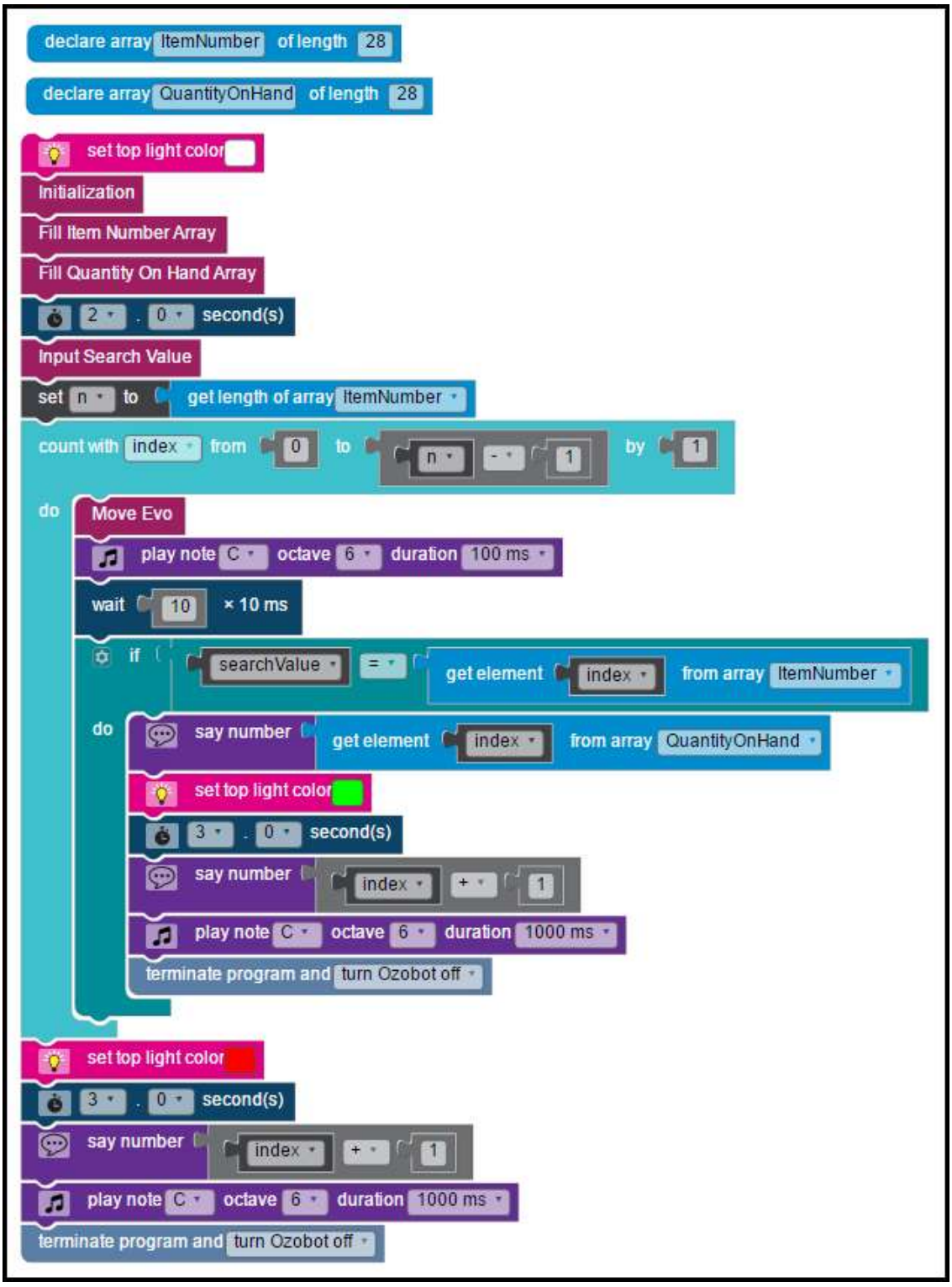


Figure 2

The OzoBlockly Binary Search Main Program

As with the previous section, this section is optional and intended for teachers interested in discussing some of the details of the OzoBlockly binary search program with their students. The entire program is quite lengthy with several subroutines that are fairly complex. But the main program, which contains the linear search, is well worth discussion and much of it is shown in Figure 3.

The binary search process requires keeping track of the left and right end indexes of the remaining search region of the array. The variable *n* is set to the length of the array. The initial search region is the entire array, so *left* is set to the index 0 and *right* is set to the index *n*-1. Then a loop repeats until *left* > right, in which case the search value is not in the array, or until the search value is found. With each iteration of the loop, one comparison is made, so the *numberOfComparisons* is incremented by 1. The variable middle is set to the average of *left* and *right*, and Evo moves to the array index *middle.*

Next, if the array value at *middle* is less than the search value, then *left* is set to *middle*+1. If the array value at *middle* is greater than the search value, then *right* is set to *middle*-1. In either case, the next iteration of the loop is executed. If the array value at *middle* is equal to the search value, then the search value has been found in the array. The top LED shows green to indicate a successful search, Evo speaks the corresponding QOH and total number of comparisons, and then turns off.

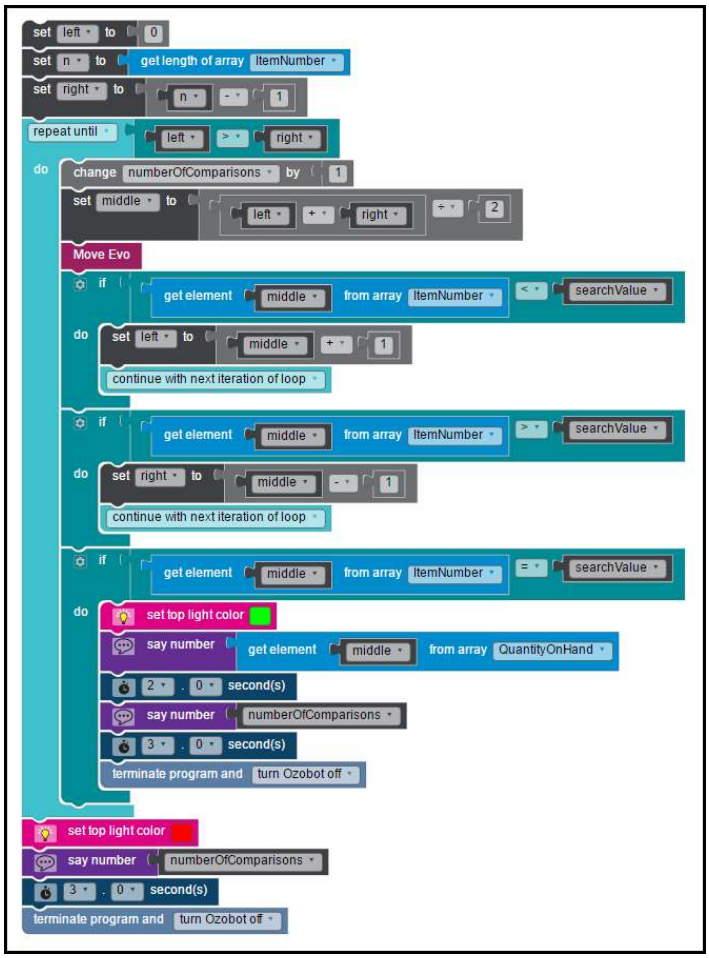


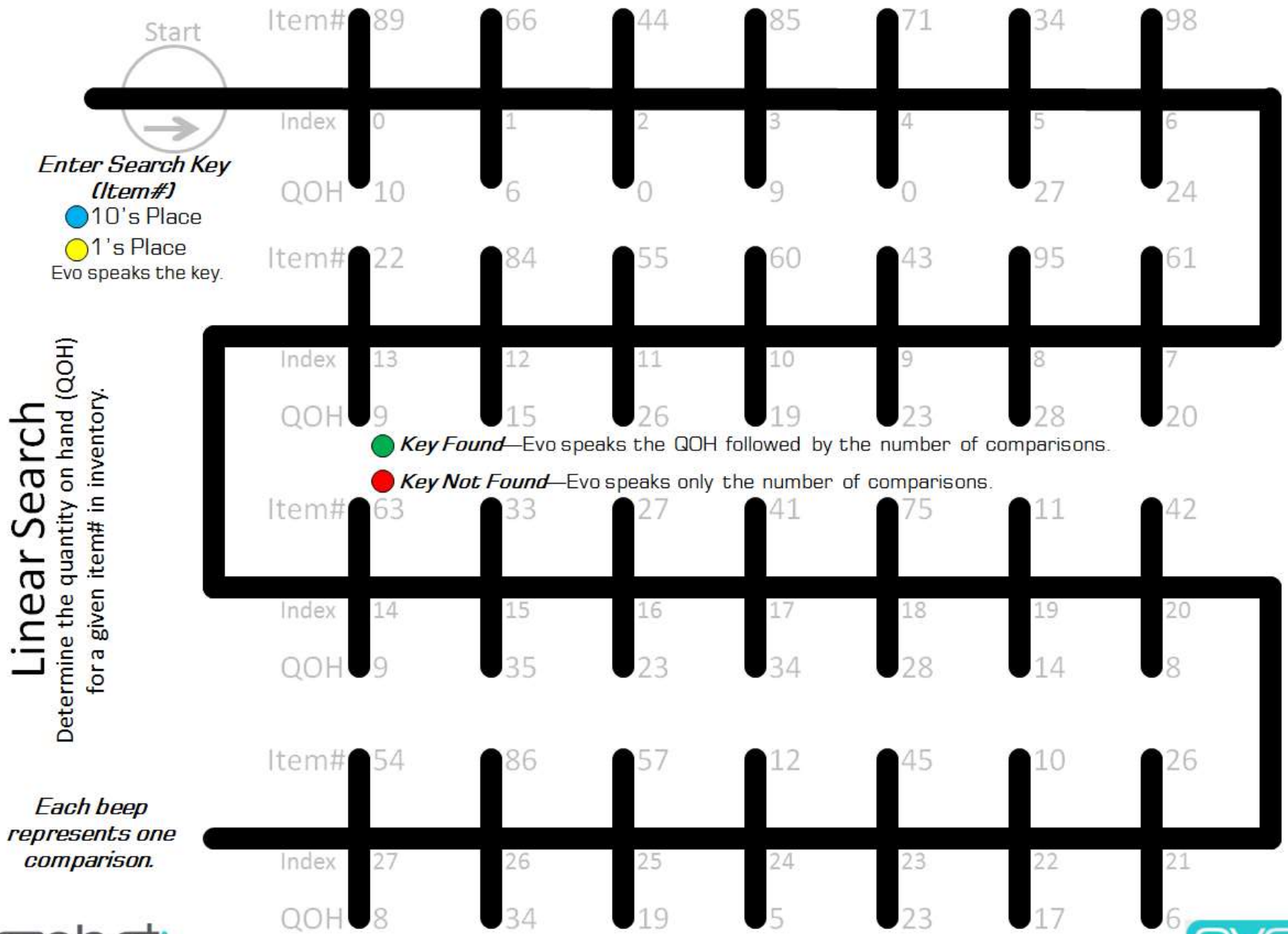
Figure 3

Exercises

1. Have Evo search for the QOH when the item number is 60. What does Evo come up with for the QOH? How many comparisons are required for the linear search? For the binary search?
2. Have Evo search for the QOH when the item number is 89. What does Evo come up with for the QOH? How many comparisons are required for the linear search? For the binary search?
3. What is always the array index for the first comparison in Evo's linear search? In Evo's binary search?
4. Suppose that an array has a length of 33 elements, rather than the 28 elements, in Evo's binary search. What will be the index for the first comparison?
5. Have Evo search for the QOH when the item number is 9. What does Evo determine and how many comparisons are needed for the linear search? For the binary search?
6. Assuming that any search value is equally likely in a linear search with an array having *n* elements, what would be the maximum number of comparisons required? The minimum? The average?
7. Let's concentrate on Evo's binary search to gain a real feel for how the binary search works. Carefully observe how Evo halves the size of the search region after each comparison. How many comparisons are required by Evo for each of the following item number searches: 54, 33, 42, 84, 61? What appears to be the minimum number of comparisons? The maximum?
8. Computer scientists tell us that the maximum number of comparisons required for a binary search in an array of *n* elements is the greatest integer not greater than $log_2 n + 1$. How does this compare to your guess in exercise 7?
9. Suppose that you have a sorted array with 1000 elements. What is the maximum number of comparisons required in a binary search of this array?
10. An algorithm's ***time complexity*** relates the length of its input to the number of steps it takes. The ***order of complexity*** of an algorithm relates to the limiting behavior when the input ***n*** *tends toward infinity*. Some common orders for algorithms are described as O(log n), O(n), O(n log n), $O(n^2)$, $O(2^n)$, and O(n!)—from very good to terrible.
    a. Which of the above **O's** best describes the worst case scenario for the linear search?
    b. Which of the above **O's** best describes the worst case scenario for the binary search?

Linear Search

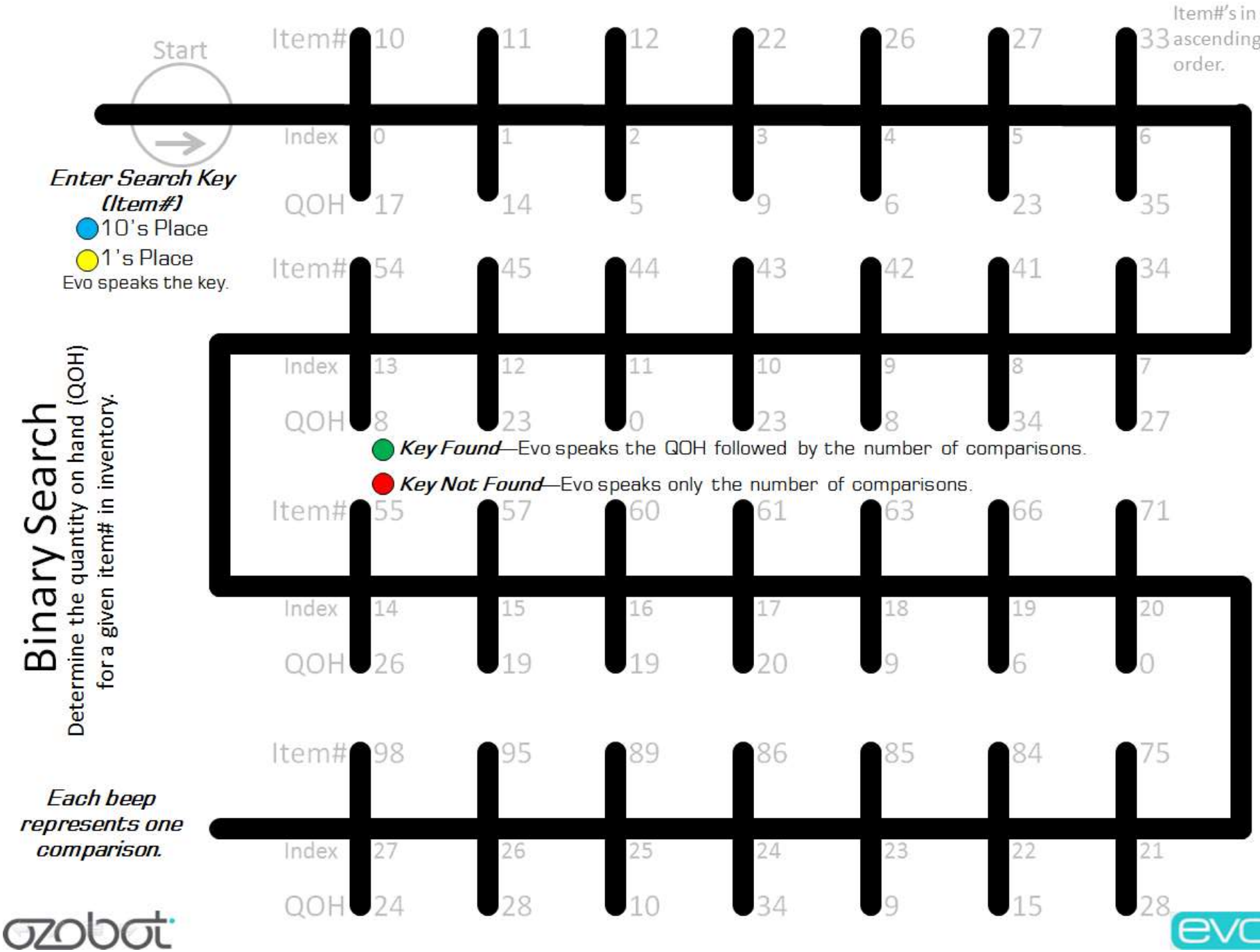
Determine the quantity on hand (QOH) for a given item# in inventory.



Each beep represents one comparison.

Binary Search

Determine the quantity on hand (QOH) for a given item# in inventory.



Enter Search Key (Item#)

10's Place

1's Place

Evo speaks the key.

Key Found—Evo speaks the QOH followed by the number of comparisons.

Key Not Found—Evo speaks only the number of comparisons.

Each beep represents one comparison.