ACTIVITY **3.4.2**

Investigating Search

INTRODUCTION

As with sorting, search algorithms are a critical requirement for modern computing. From banking to medicine to the entire Web itself, almost all businesses and organizations require fast and efficient computational searching. Can you predict the number of searches performed every day, worldwide, by Google? Conduct a search to find out how many Google searches are performed daily and you might be surprised. Searching quickly and accurately is one of the great applications of computers today. In this activity, you will learn about some of the best search algorithms.

Materials

- Computer with Android™ Studio
- Android™ tablet and a USB cable, or device emulator

RESOURCES

Free Response Question: HorseBarn
Resources available online

Procedure

Part I: Search Algorithms

1 Observe the following algorithm:

```
public static someMethod(int a)
{
    int numbers[] = {4,8,17,2,9,13,6,21,7,1,8,2,3,5,8};
    int b = 0;

    for (int i = 0; i <= numbers.length - 1; i++)
    {
            if (numbers[i] == a)
            {
                b++;
            }
        }
    }
```

Describe in detail what you think this code does and what the variable `b` represents.

2. Obtain and import the app *`3.4.2WhiteboardAppSort`* as directed by your teacher.
3. Explore the app and its search classes, `LinearSearch` and `BinarySearch`.
4. Add the following algorithm to the `onCreate(…)` method:

```
ArrayList<Integer> myList = new ArrayList<>();
int numElements = 100;
TextView textOut = (TextView)findViewById(R.id.textView);

int prevNum = 0;
for (int i = 0; i < numElements; i++) {
    int nextNum = (int)(Math.random() * 3 + 1);
    myList.add(prevNum + nextNum);
    prevNum = prevNum + nextNum;
    textOut.append(prevNum + ":");
}
```

5. Test your app. Describe the list of numbers that were generated.
6. Investigate the line that generates random numbers by removing the parentheses around the computational portion so the line reads `int nextNum = (int) Math.random() * 3 + 1;`.
7. Run the app. Why is the `nextNum` always 1?
8. Replace the parentheses and re-test your app to ensure random increments.
9. Use `LinearSearch.search(…)` to search for a value in your list and report the index number, if it was found.

Part II: Improve the Algorithm

Due to the randomized way that you added numbers to your list, your search may not always return a result when you test it.

10. To validate the algorithm through testing, one way to guarantee that a number can be found is to search for a value that you know is already in the list.
    a. Use `Math.random()` to generate a random index in the appropriate range.
    b. Store the value in the array list at that index as your search value.
11. Test to see whether your code works with a `BinarySearch.search(…)`.

You may have noticed that the algorithm has a bit of overkill: an ArrayList is not required and neither are Integer objects.

12 Convert your app to use an array of integers. You will need variable size arrays in subsequent steps, so use the variable `numElements` to declare your array.

13 Test both the binary and linear search methods.

If your search value happens to be near the beginning (or the end) of the list, consider how easy (or difficult) it is to validate your algorithm. As one part of testing, engineers often supply contrived (artificial or manufactured) test data to make validation easier:

a. Modify your algorithm to choose a `searchValue` near the beginning of the list that is easy to validate, then validate the results for both searches.

b. Modify your `searchValue` to be the *last* value in the list and validate the results for both searches.

Part III: Analyze the Performance

Now that you have validated your app, you can confidently analyze the search performance. To do this, you need to know the number of iterations for each search performed.

14 Modify your app to display the number of iterations rather than the index of where the element is found.

15 To test the "worst-case" performance, consider what values you could use for each search. You may have considered that the last element for a linear search and the worst-case value for a binary search may be difficult to predict. But the worst-case search scenario is actually an element that cannot not be found.

a. Modify the search algorithms to return the execution count even when an element is not found.

b. Modify your work area code to use a `searchValue` that does not exist in the list. Comment out code that is not necessary for this worst-case scenario.

16 To get a range of performance data, add an outer loop to gather search data for arrays containing 1000, 2000, 3000, ... up to 10000 integers.

- This is similar to how you changed your performance analysis code for sorting.
- Do not display all of the values in the list, just the execution counts.

Because a search executes faster than a sort, you can collect a wider range of performance values.

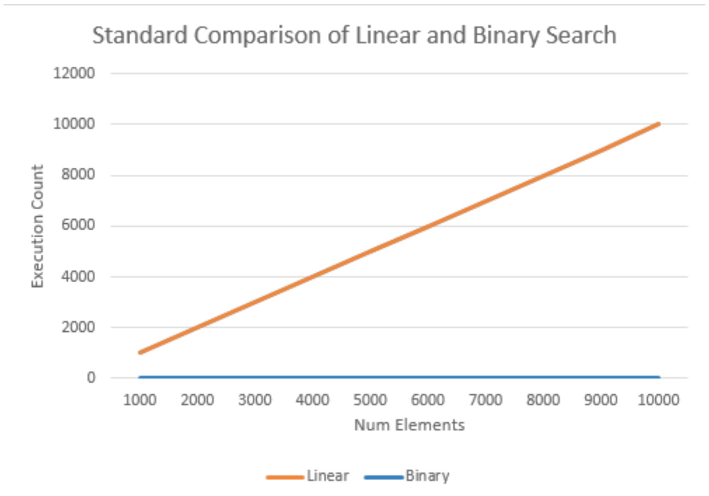
17 Collect the following performance data:

Element	Linear Search Statement Execution Count	Binary Search Statement Execution Count
1000		
2000		
3000		
4000		
5000		
6000		
7000		
8000		
9000		
10000		

How do the performance values compare between a linear search and a binary search?

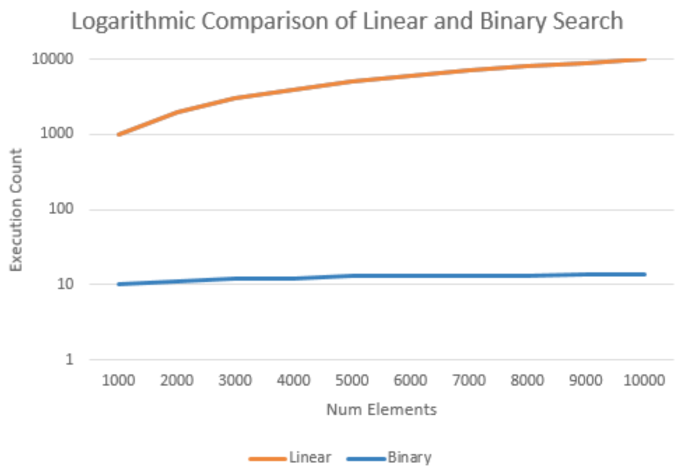
How do the differences between the two search algorithms compare to the differences between the sort algorithms?

The following graph shows the execution counts of linear and binary searches. You can easily see the increase in execution counts for a linear search, but due to the scale of the graph and the large difference between the linear and binary searches, you cannot tell much about the binary search.



18 Using the graph above, predict approximate worst-case execution count for a linear search of 20,000 elements:

Instead of a standard chart, a logarithmic chart also shows the performance data for a binary search:



The same data has been graphed, but notice the y-axis has log10 values (10^0, 10^1, 10^2, etc.). You can now see that the binary search counts change, but never by a great amount.

19 Using the logarithmic chart, predict an approximate worst-case execution count for a binary search of 20,000 elements.

AP Focus: In addition to sorting and searching, you can practice an algorithm to remove null entries in an array. Complete **FRQ: HorseBarn**.

CHALLENGE

Analyze two online search algorithms— **linear and binary**—in terms of search times.

a. Use a high Animation Speed.

b. Choose a worst-case search value, in other words, a value that is not in the list, but if it were in the list, would occur near the end of the sorted list.

c. Record search times using an online stopwatch. Record the results of a few performance runs.

   What performance conclusions can you make about the two search methods?

CONCLUSION

1. Do both linear and binary search methods require sorted data? Why or why not?
2. Do either of the search algorithms resemble one (or more) sort algorithms? In what ways are they similar or different?