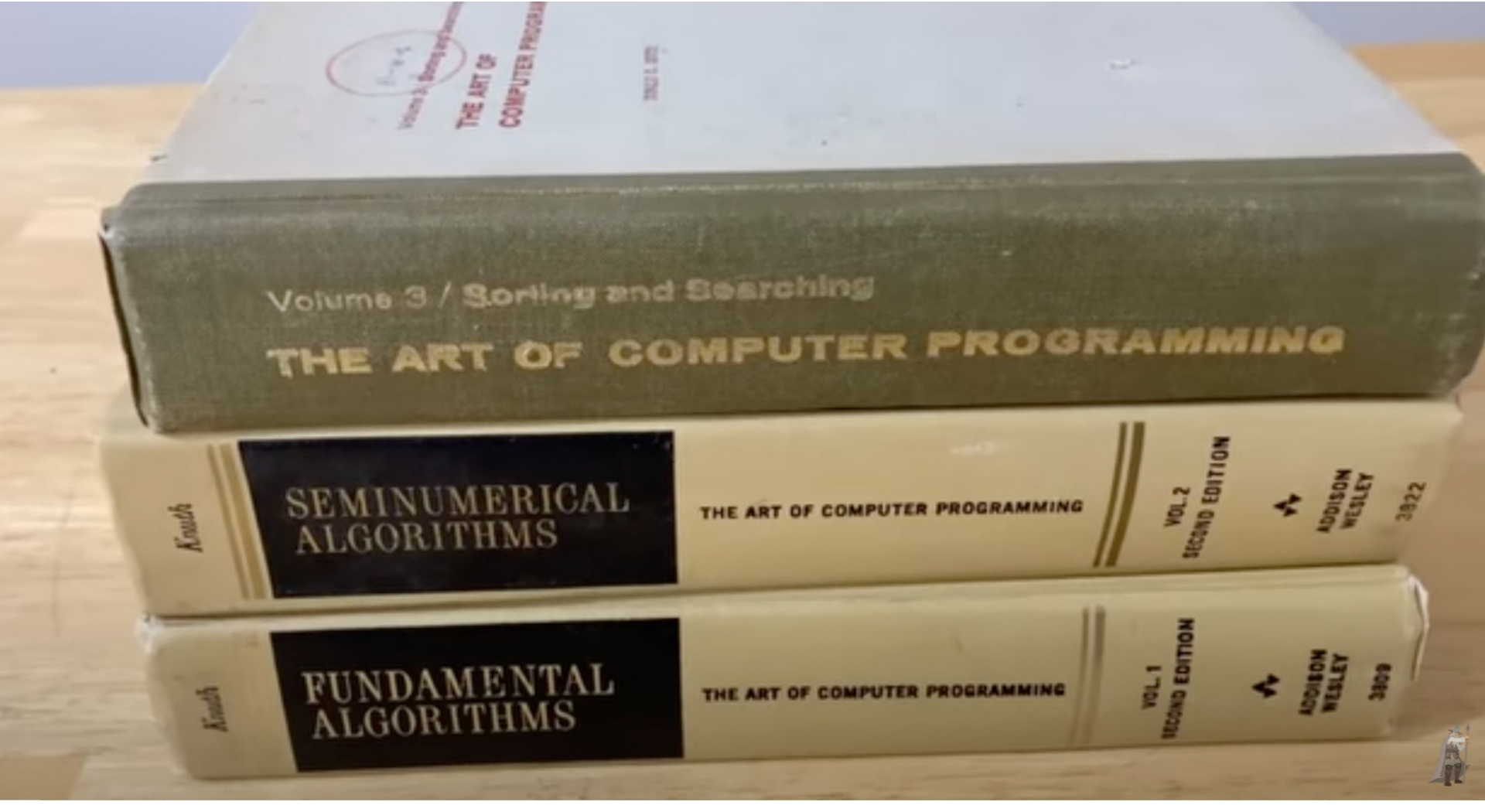
2023-01-11

(Now that I started this ISO8601 thing, I have to commit to the bit.)

7.5
Search Algorithms


THE ART OF COMPUTER PROGRAMMING
Volume 3 / Sorting and Searching
THE ART OF COMPUTER PROGRAMMING
Knuth
SEMINUMERICAL ALGORITHMS
THE ART OF COMPUTER PROGRAMMING
VOL. 2
SECOND EDITION
ADDISON WESLEY
1972
Knuth
FUNDAMENTAL ALGORITHMS
THE ART OF COMPUTER PROGRAMMING
VOL. 1
SECOND EDITION
ADDISON WESLEY
1969

2018

PROFILES IN SCIENCE

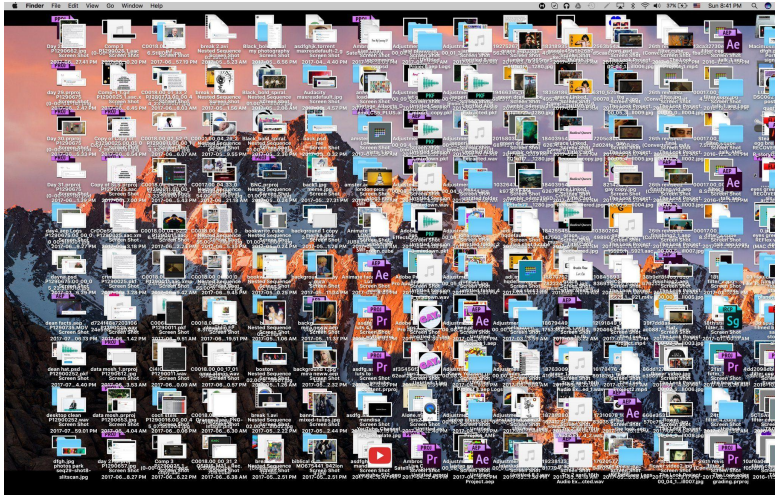
The Yoda of Silicon Valley

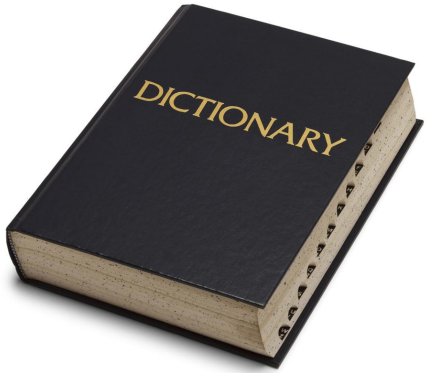
Donald Knuth, master of algorithms, reflects on 50 years of his opus-in-progress, “The Art of Computer Programming.”

Sorting and Searching

- In the next few lectures, we'll introduce algorithms for searching and sorting.
- Computer scientists write code to sort and search on a daily basis – using some of the algorithms that we'll see today.
- Today we will cover **searching**–how to efficiently determine whether an array contains a given element
- Friday and Monday: we will cover our first **sorting** algorithms–ordering an array in an ascending or descending fashion

Searching



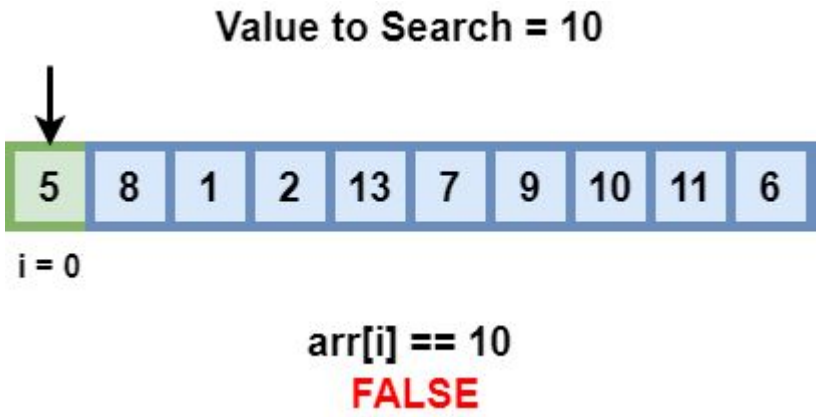


What are some strategies you use to find the item you're looking for?

Sequential Search (aka Linear Search)

Steps:

1. Start at the first item in the list
2. Check if that item is the item you are looking for
    a. If the item is the item you are looking for, you're done!
    b. If it is not the item and there are more items in the array, go to the next item and repeat step 2
3. If you have reached the end of the list, the item is not in the list



Implementing Sequential Search

Sometimes the index of the target item is what's desired.

The `String.indexOf` method we know and love is a sequential search.

```
public int indexOf(ArrayList<String> arrayList, String target) {
  for (int i=0, n=arrayList.size(); i<n; i++) {
    if (arrayList.get(i).equals(target)) {
      return i;
    }
  }
  return -1;
}
```

Implementing Sequential Search

You may be searching for an object that matches some criteria, and you may want the object, and not need the index.

```
public Student findStudentByName(ArrayList<Student> students, String name) {
  for (Student student : students) {
    if (student.getName().equals(name)) {
      return student;
    }
  }
  return null;
}
```

Sequential Search

Pros:

- You can search for an item in any unsorted list and it is guaranteed to be found if it exists in the array

Cons:

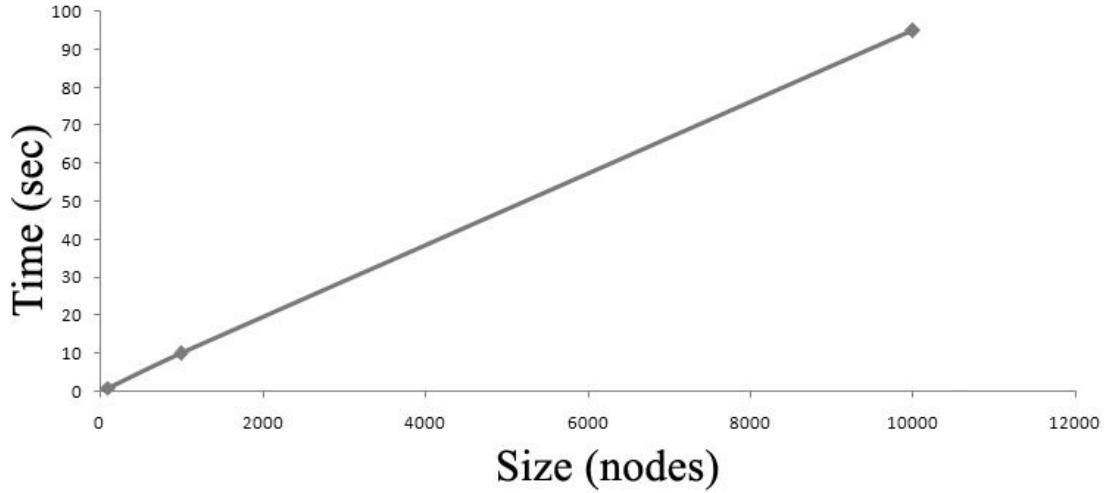
- If you’re searching a long list, this can be a very slow approach
- It can create performance issues even if you're searching a short list very frequently (like a list of monsters every "tick" of a video game)

Sometimes, the data structures that you're using leave you no other option but sequential search.

Sometimes, it's fast enough for the data you're searching. At other times, it may indicate that you need to change your data structures!

Sequential Search

If your list contains N elements, and the target element is at a random position in the list then on average it will take N/2 checks to find your element. When the time an algorithm takes to complete is directly related to the size of the array (N), we call this algorithm **linear** and the notation is O(N).



Can you think of a faster search algorithm?

As of November 2022, there were over 1.14 billion websites on the Internet.

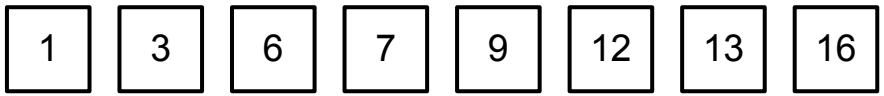
If Google used a linear algorithm, it would take years to find results to your search queries!

Binary Search: Divide and Conquer

Binary search is a much faster algorithm, but requires that the list be **sorted**.

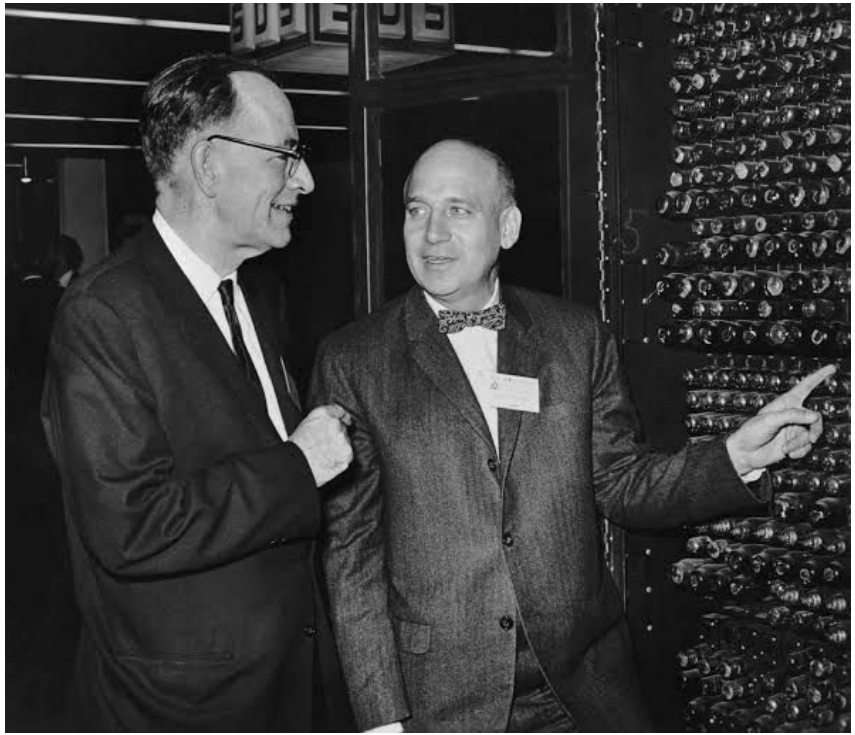
We can use the fact that the list is sorted to reduce the problem to smaller problems.

An algorithm that breaks the problem into smaller sub-problems is called a **divide-and-conquer algorithm**.



Binary Search: OK Boomer

In 1946, John Mauchly made the first mention of binary search as part of the Moore School Lectures, a seminal and foundational college course in computing.[9] In 1957, William Wesley Peterson published the first method for interpolation search.[9][57] Every published binary search algorithm worked only for arrays whose length is one less than a power of two[i] until 1960, when Derrick Henry Lehmer published a binary search algorithm that worked on all arrays.[59]

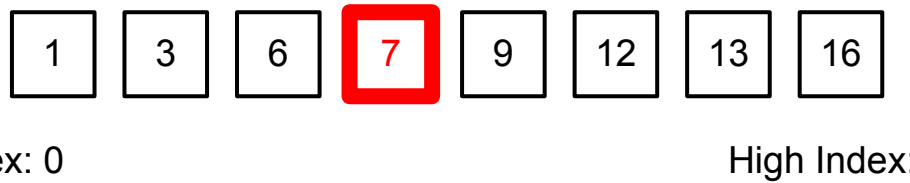


John William Mauchly (August 30, 1907 – January 8, 1980) was an American physicist who, along with J. Presper Eckert, designed ENIAC, the first general-purpose electronic digital computer, as well as EDVAC, BINAC and UNIVAC I, the first commercial computer made in the United States.

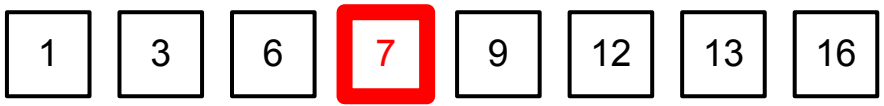
Binary Search: Steps

Find the index of the element 13 in the array if it exists, otherwise return -1

- Step 1: Find the middle

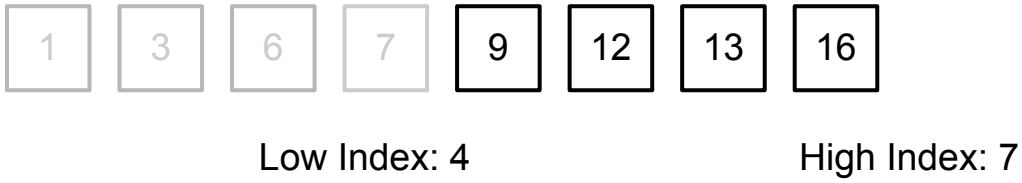


- Step 2: Check if the middle element is equal to the target element



Binary Search: Steps

- Step 3: If the element is greater than the middle element, eliminate all items to the left of the middle element, otherwise eliminate all items to the right



- Repeat Step 1: Find the middle of the remaining elements

Binary Search: Steps

- Repeat Step 2: Check if the middle element is equal to the target element

- Repeat Step 3: If the element is greater than the middle element, eliminate all items to the left of the middle element, otherwise eliminate all items to the right

Low Index: 6 High Index: 7

Binary Search: Steps

- Repeat Step 1: Find the middle of the remaining elements

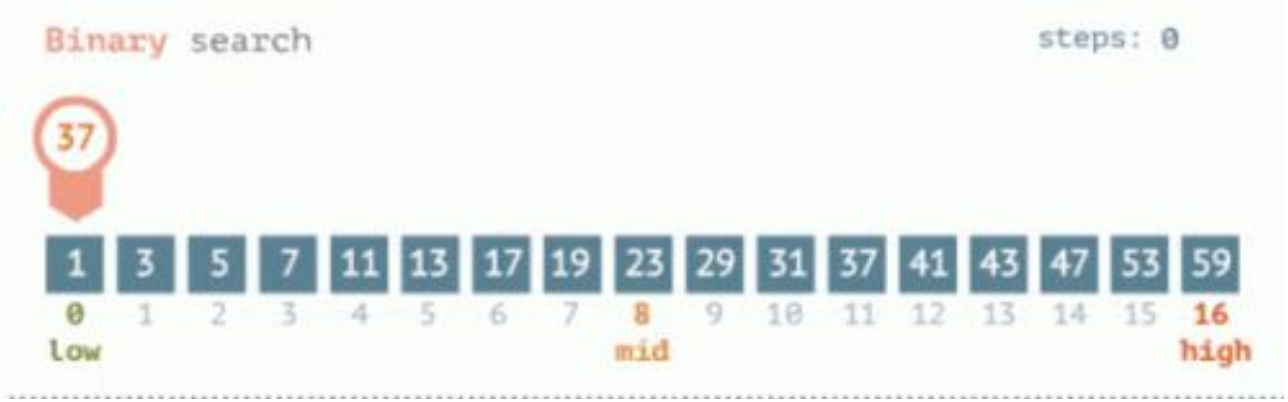
- Repeat Step 2: Check if the middle element is equal to the target element

- Step 4: Return the index of the the target element, or return -1 if the target element is not found

Binary Search

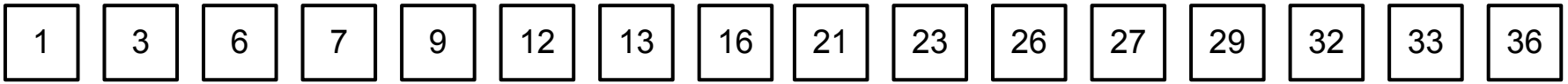
To find 13 in the array, it took 3 comparisons (also known as **probes**)

1	3	6	7	9	12	13	16



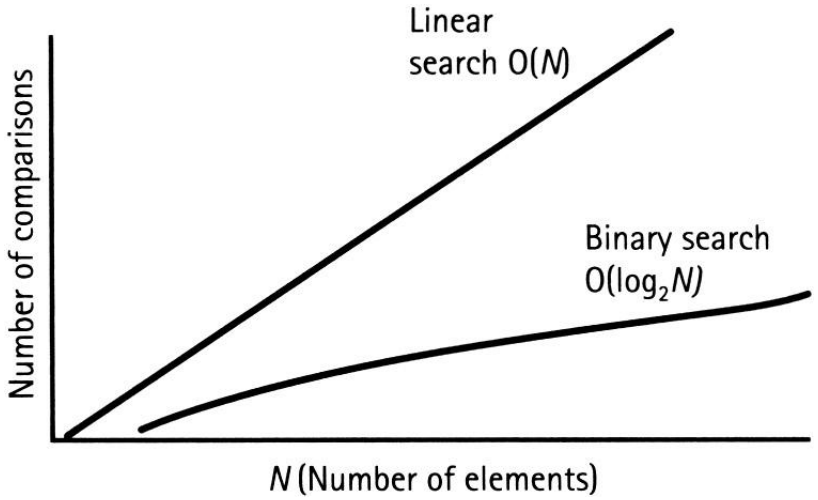
Binary Search

If the number of elements (N) in the array were doubled where N = 16, how many comparisons would you need to find 13?



Binary Search

If your list contains N elements, and the target element is at a random position in the list then on average it will take $\log_2 N$ checks to find your element. To give you a sense of this speedup, the $\log_2 1024 = 10$ (since $2^{10} = 1024$). **This means that binary search runs 10x faster on sorted lists of length ~1K.** The notation for this algorithm is O(log N)



Comparing Sequential Search vs Binary Search (These are "worst case" numbers.)

N	**Sequential Search Comparison Count**	**Binary Search Comparison Count**
2	2	2
4	4	3
8	8	4
16	16	5
100	100	7

Exercise: BinarySearch

- Today, you're going to write your own implementation of binary search.
- Implement the `binarySearch` method in the BinarySearch Replit
- Also, implement an input loop that asks the user for a word, and searches for the word using binarySearch.
- Take care – this algorithm is reaaaaally easy to get mostly right but still wrong...

> Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky
>
> — Donald Knuth[2]

Towards the end of class, we'll review a solution together.

Chris's solution

```
private int binarySearch(ArrayList<String> words, String targetWord, boolean verbose) {
  int lowIdx = 0;
  int highIdx = words.size() - 1;
  while (lowIdx <= highIdx) {
    int middleIdx = ((highIdx - lowIdx) / 2) + lowIdx;
    int middleCompareResult = words.get(middleIdx).compareTo(targetWord);
    if (middleCompareResult == 0) {
      return middleIdx;
    } else if (middleCompareResult > 0) {
      highIdx = middleIdx - 1;
    } else {
      lowIdx = middleIdx + 1;
    }
  }
  return -1;
}
```

Gary's solution

```
// Precondition: The words ArrayList must be in sorted ascending order.
private int binarySearch(ArrayList<String> words, String targetWord, boolean verbose) {
  // TODO Put your binary search implementation here!
  int lowIndex = 0;
  int highIndex = words.size();
  while (lowIndex < highIndex) {
    int middleIndex = (lowIndex + highIndex) / 2;
    if (verbose) {
      System.out.println(middleIndex + ": " + words.get(middleIndex));
    }
    int compareResult = words.get(middleIndex).compareTo(targetWord);
    if (compareResult == 0) {
      return middleIndex;
    } else if (compareResult < 0) {
      lowIndex = middleIndex + 1;
    } else {
      highIndex = middleIndex;
    }
  }
  return -1;
}
```