# Lecture 24 — Searching

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 27, 2016

Acknowledgments: W.D. Bishop

# Why is Searching Important?

Computers are designed to store & manipulate massive amounts of data, but data is only useful if it can be found quickly.

Consider World Wide Web (WWW) search engines:

Would you use Google if it required 1 hour to answer a query?

How much data does Google search?

They say there are over 60 trillion web pages.

Its index of results is over 100 million gigabytes.

And they try to get results in 1/8 of a second.

# Searching Principles

To talk about searching, we need to establish the basis of a scenario.

We have an array of some type and something we are looking for.

There must be a way to tell if the entry being examined is equal to what we are searching for.

If we find an entry in the array that matches what we are searching for, we return the index of that entry.

If no matching entry is found in the array, we return the index -1.

# Linear Search Algorithm

First idea: linear search.

The algorithm to do a linear search is fairly simple.

1. Start at index 0 (set the current index to 0)
2. Check if the value at the current index is the desired value:
   If yes, return the current index
   If no, increment the current index
3. Check if the current index exceeds the array capacity:
   If yes, return -1
   If no, go to step 2.

# Linear Search Implementation

Take this algorithm and implement to to search for integers:

```
int linear_search( int[] array, int array_length, int searchValue )
{
    for( int index = 0; index < array_length; index++ )
    {
        if( array[index] == searchValue )
        {
            return index;
        }
    }
    return -1;
}
```

# Testing the Linear Search

```
int main( )
{
    int[] data = {1, 3, 4, 5, 6, 7, 8, 10, 11, 13, 14, 15, 20, 36};

    cout << "The index of 1 is: " << linear_search( data, 14,  1 ) << endl;

    cout << "The index of 3 is: " << linear_search( data, 14,  3 ) << endl;

    cout << "The index of 10 is: " << linear_search( data, 14,  10 ) << endl;

    cout << "The index of 11 is: " << linear_search( data, 14,  11 ) << endl;

    cout << "The index of 36 is: " << linear_search( data, 14,  36 ) << endl;

    cout << "The index of 99 is: " << linear_search( data, 14,  99 ) << endl;

    return 0;
}
```

# Linear Search Algorithm Analysis

The linear search algorithm for arrays is very simple:
Easy to write; easy to understand.

As seen in the Google search example, how quickly we find a piece of data is important. We are often interested in the worst case scenario.

The worst case scenario occurs if the desired value is at the end of the array or not in the array at all.

In the worst case, the linear search algorithm checks all $n$ values where $n$ is the capacity of the array.

# Improving on Linear Search

The linear search is an exhaustive search: we check every single entry until we find the answer we want, or reach the end.

Improvement idea: if the list is sorted and we find an entry larger than the one we're searching for, we can stop immediately and return -1.

If we search for 2 in this sorted array and find 3, we can already conclude the answer is -1: `int[] data = {1, 3, 4, 5, 8, 13};`

The worst case (e.g., searching for 99) is still checking *n* entries.
What we need is a better algorithm, not a minor improvement.

# Linear Search Improvement

This improvement works only if the list is sorted ascending.

```
int linear_search( int[] array, int array_length, int searchValue )
{
    for( int index = 0; index < array_length; index++ )
    {
        if( array[index] == searchValue )
        {
            return index;
        }
        else if( array[index] > searchValue )
        {
            return -1;
        }
    }
    return -1;
}
```

# A Better Search Algorithm

Suppose I tell you I am thinking of a number between 1 and 1000. You guess numbers, and I'll tell you "higher", "lower", or "correct".

Linear search is like guessing 1, then 2, then 3...

Better strategy: if you guess 500, there are three possible outcomes:

1. The number is exactly 500. I say "correct" and the game is over.
2. The number is greater than 500. I say "higher", and you know that the answer must be somewhere between 501 and 1000.
3. The number is less than 500. I say "lower", and you know that the answer must be somewhere between 1 and 499.

Assuming the answer wasn't exactly 500, we have reduced the range where the value could be in half!

# A Better Search Algorithm

If I said the number is greater than 500, you won't waste any time guessing numbers between 1 and 500 and can focus on 501 and up.

Then you should repeat the strategy: guess 750.
I'll then tell you if it's in the range 501-749 or 751-1000.

With just two guesses, you have narrowed the values to guess from 1000 possibilities to 250 possibilities.

At each stage, we are dividing the area to search in half.
This is called a binary search.

# Binary Search of Arrays

Let's take this strategy and apply it to a sorted array.

Given an array of length *n*, examine the middle entry. If the middle entry is the one we need, we are done. Otherwise, continue.

As the array is sorted, we can determine in which half the desired value would be in, if it exists in the array at all.

Then repeat the procedure on the half of the array where the desired value could be and do not waste any effort checking the other half.

# Formal Binary Search Algorithm

Important reminder: unlike the linear search, the binary search algorithm will only work if the array is sorted.

1. Set top to index 0 and bottom to index (capacity - 1).
2. While top is less than or equal to bottom:
   a) Set middle to (top + bottom) / 2
   b) Check if value at middle equals the desired value:
      If yes, return middle
      If no, check if value at middle is greater than the desired value:
         If yes, set bottom to middle - 1
         If no, set top to middle + 1
   c) Loop
3. Return -1 if the loop terminates

# Binary Search Implementation

```
int binary_search( int[] array, int array_length, int searchValue )
{
    int top = 0;
    int bottom = array_length - 1;
    int middle;

    while( top <= bottom )
    {
        middle = (top + bottom) / 2;
        if( array[middle] == searchValue )
        {
            return middle;
        }
        else
        {
            if( array[middle] > searchValue )
            {
                bottom = middle - 1;
            }
            else
            {
                top = middle + 1;
            }
        }
    }
    return -1;
}
```

# Testing the Binary Search

```
int main( )
{
    int[] data = {1, 3, 4, 5, 6, 7, 8, 10, 11, 13, 14, 15, 20, 36};

    cout << "The index of 1 is: " << binary_search( data, 14, 1 ) << endl;

    cout << "The index of 3 is: " << binary_search( data, 14, 3 ) << endl;

    cout << "The index of 10 is: " << binary_search( data, 14, 10 ) << endl;

    cout << "The index of 11 is: " << binary_search( data, 14, 11 ) << endl;

    cout << "The index of 36 is: " << binary_search( data, 14, 36 ) << endl;

    cout << "The index of 99 is: " << binary_search( data, 14, 99 ) << endl;

    return 0;

}
```

# Binary Search Output

The output of the binary search test program is identical to the linear search test program.

Will this always be true?

No. If duplicate values exist in the array, the binary search may return a different index position than a linear search.

Imagine this is the array: {1, 19, 177, 285, 285, 385, 712, 714}

A linear search for 285 will return index 3.
A binary search for 285 will return index 4.

# Binary Search Algorithm Analysis

The binary search algorithm relies on sorting to reduce the number of elements of an array to search.

In the worst case, the binary search algorithm searches $\lceil \log_2 n \rceil$ values where n is the length of the array.

Note that $\lceil \log_2 n \rceil$ denotes the ceiling (round up) of $\log_2 n$.

For an array of size 3, it may be necessary to search 2 values (the middle plus one end of the array).

$\lceil \log_2 3 \rceil = 2$

The worst case scenario of the binary search algorithm requires fewer iterations than the worst case scenario of the linear search algorithm.

# Binary Search: Divide And Conquer

The binary search is an example of the divide-and-conquer strategy.

Turn the search problem into a smaller version of the same problem:
"find a value in an array" becomes "find a value in a smaller array".

This suggests that the binary search algorithm is a good candidate for a recursive implementation.

The binary search algorithm presented in the lecture notes is iterative (uses a loop), but as an exercise, try rewriting it in a recursive manner.

# Search Algorithm Comparison

**Linear Search:**

- Works on arrays and linked lists
- Does not require sorted data
- Finds values in small arrays quickly
- Performs poorly on large arrays

**Binary Search:**

- Works on arrays
- Requires sorted data
- Finds values in most positions quickly
- Requires fewer iterations on average (faster)
- Searches large arrays quickly

# Search Algorithm Performance

Is the time required to sort data justified to improve searching?

Yes.

If you will search data multiple times, the sorting time is easily offset by the time saved during searching.

To illustrate this point, consider the following example using a large array of random integers (100,000 elements).

For the example scenario, and a computer running Windows XP, we produced the following results (with clock resolution of about 10 ms):

# Search Comparison Results

Overall result: the binary search outperforms the linear search, even if the time required sorting is considered:

- Sorting requires approximately 100 ms
- 100 000 linear searches of 100 000 elements require 22.12 s
- 100 000 binary searches of 100 000 elements require 20 ms

The binary search is over 1000$\times$ faster than the linear search.

We sorted once and searched many times, making sorting worthwhile.

The performance figures presented depend upon:

- The pattern of memory accesses
- The number of elements in the array
- The search values